UNIVERSITY
of
GLASGOW

An Architecture for the Compilation of Persistent Polymorphic Reflective Higher-order Languages

João António Correia Lopes

Department of Computing Science

Submitted for the degree of

Doctor of Philosophy

Thesis
10818
Copy 1

An Architecture for the Compilation of Persistent Polymorphic Reflective Higher-order Languages

by

João António Correia Lopes

Submitted to the Department of Computing Science
UNIVERSITY OF GLASGOW
for the degree of
Doctor of Philosophy
February 1997

Abstract

Persistent Application Systems are potentially very large and long-lived application systems which use information technology: computers, communications, networks, software and databases. They are vital to the organisations that depend on them and have to be adaptable to organisational and technological changes and evolvable without serious interruption of service.

Persistent Programming Languages are a promising technology that facilitate the task of incrementally building and maintaining persistent application systems. This thesis identifies a number of technical challenges in making persistent programming languages scalable, with adequate performance and sufficient longevity and in amortising costs by providing general services.

A new architecture to support the compilation of long-lived, large-scale applications is proposed. This architecture comprises an intermediate language to be used by front-ends, high-level and machine independent optimisers, low-level optimisers and code generators of target machine code.

The intermediate target language, TPL, has been designed to allow compiler writers to utilise common technology for several different orthogonally persistent higher-order reflective languages. The goal is to reuse optimisation and code-generation or interpretation technology with a variety of front-ends. A subsidiary goal is to provide an experimental framework for those investigating optimisation and code generation. TPL has a simple, clean type system and will support orthogonally persistent, reflective, higher-order, polymorphic languages. TPL allows code generation and the abstraction over details of the underlying software and hardware layers.

An experiment to build a prototype of the proposed architecture was designed, developed and evaluated. The experimental work includes a language processor and examples of its use are presented in this dissertation. The design space was covered by describing the implications of the goals of supporting the class of languages anticipated while ensuring long-term persistence of data and programs, and sufficient efficiency. For each of the goals, the design decisions were evaluated in face of the results.

Thesis Supervisor: Professor Malcolm Atkinson

Acknowledgements

This work would not have been possible without the help and encouragement of many people and the support of various organisations.

I am most indebted to my supervisor, Malcolm Atkinson, for proposing the research project that leaded to this thesis, for his guidance and enthusiasm, his many ideas and all the suggestions that gave shape to it. His comments on how to do research were greatly appreciated and his example of how to combine teaching and research proved to me that it is possible and worthwhile to pursue this career. Professor Atkinson is a model researcher and teacher who knows how to balance theory with engineering work. I would like to thank him again for the many comments, suggestions and corrections that greatly improved this dissertation.

Peter Dickman in his rôle as second supervisor during the final part of the research, helped in focusing the implementation and we had many discussions that helped clarify the research. His encouragement and cheerful approach were a great help. He made several helpful suggestions and corrections which greatly improved this dissertation.

Paul Philbrow was a patient listener to my half-baked ideas and gave comments and suggestions that helped to debug them. He was always ready to help me with my misunderstandings related to the usage of the Napier88 language and the associated technology. He also supported the computer programs that form the persistent programming environment used in the implementation. He is a model of consistency in following sound principles in life that most of us do not have enough energy to pursue.

Ron Morrison and the persistent research group at the University of St Andrews provided the Napier88 persistent programming environment used in the implementation and useful discussions.

I had the pleasure to discuss functional programming and other matters with Phil Trinder and Gebreselassie Baraki. Both of them always demonstrated their friendship and introduced me to the difficult problems of cricket rules, local politics and the idiosyncrasies of the English language.

Discussions with Paul Wilson, Dag Sjøberg, Graham Kirby, Quintin Cutts and Laurent Daynès helped to clarify issues related to this research.

Anne Philbrow did gud job in proof-reading this dissertation.

Gordon Russell provided the software to cross-reference citations.

My PhD colleagues and all the staff of the Computing Science Department made my stay in the Department a pleasant time.

Hermano Moura and the rest of the "Portuguese speaking community" provided the intense social life that gave the right balance to the hard and, not always immediately rewarding, research work.

My colleagues in Porto, Cristina Ribeiro, João Canas Ferreira, João Pascoal de Faria and Gabriel David, were patient listeners to my grumbles during the writing of this dissertation.

JNICT, National Board of Science and Technological Research in Portugal, provided extensive financial support for my stay in Glasgow, through "Programa Ciência", scholarship *BD/1310/91-IA*, and "Programa PRAXIS XXI", scholarship *BD/3173/94*.

The FIDE$_2$ ESPRIT BRA, project number 6309, provided part of the research environment and introduced me to the world leading research groups in the area.

And last but not least, I would like to thank my wife, Ana Ventura and all my family, for their love, support and encouragement at all times.

João Correia Lopes

"You should be glad that bridge fell down.
I was planning to build thirteen more to that same design"

Isambard Kingdom Brunel

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is an increasing demand for capacity to store, process and distribute information. This demand is being further increased by the rapid deployment of applications using international digital networks, such as the World Wide Web [Berners-Lee *et al.*, 1992]. Recently there has been a growing interest in programming representations independent of the software and hardware platform [Gosling, 1995].

Despite the day to day hardware performance improvements, Software Engineering research is still striving to reduce the large amount of effort needed to produce and maintain the information systems [Ramamoorthy *et al.*, 1984]. It is possible to identify a class of these information systems that are of general importance. These, which involve both complex data and sophisticated software, are invariably long-lived in response to organisational needs. These persistent application systems are built and evolve using a disparate mix of technologies: database systems, communication systems, user interface systems, operating systems, compilers, etc. Persistence is an active area of research in the operating system, programming language and database communities.

To deal with persistent application systems, many technologies and methodologies have been proposed: Database Management Systems, Database Programming Languages, Persistent Programming Languages, 4GLs and other CASE tools, Structured Analysis and Design, Object-Oriented methodologies and Object-Oriented Database Management Systems. This thesis is concerned with one promising technology, Persistent Programming Languages, that facilitate the task of incrementally building and maintaining persistent application systems. This dissertation identifies a number of technical challenges: making Persistent Programming Languages scalable, with adequate performance, with sufficient longevity and in amortising costs by providing general services. These challenges are met by a proposal for a new architecture that accommodates architecture-independent optimisations, evolution of supporting technology and interoperability between different languages. The viability of this approach is demonstrated and the following specific issues are investigated: a new architecture for compilation with a context for high-level machine independent optimisations and code generation, the design of an intermediate language to support the class of languages of interest, the use of the continuation-passing style program transformation, the use of C as a portable target language and the interaction with a persistent object store.

This introductory chapter proceeds by a definition of the kind of applications of concern to this thesis, a survey of some approaches tried in the last decades to solve the problem of building and maintaining those applications and the introduction to the new and promising approach known as orthogonal persistence.

1.1 Persistent Application Systems

The class of information systems that concern this work are defined and their attributes characterised in this section. Persistent Application Systems (PAS) [Atkinson and Morrison, 1995] are potentially very large, long-lived application systems which use information technology: computers, communications, networks, software and databases. Examples of such **PAS** are integrated information systems in organisations such as government and public administration or hospitals, geographical information systems, CAD/CAM systems, office automation systems and CASE tools.

PAS are characterised by having a size that may range from a small personal database to large amount of data in organisation's information systems. PAS are usually required for long periods of time to support the human and organisational time scales. They are frequently used to support people in cooperative tasks such as caring for a patient. The cooperation spans time (the patient's life) and space (hospitals, surgeries, laboratories, etc.). A particular PAS may need to scale in order to cope with new organisational needs or merely as a consequence of the longevity of data it maintains.

These PAS are, most of the time, vital for the survival of the organisation that depends on them. Being so crucial, these long-lived systems have to adapt to organisational changes and evolve without serious interruption of service. This is the well known and pressing problem of *maintenance*. Maintenance is absorbing most of the effort in the software industry [Ramamoorthy *et al.*, 1984]. Improvements in the development and maintainability of PAS will have economic impacts. Technical limitations mean that: PAS are difficult to build, are too expensive to maintain and adapt and are, as yet, unachievable to the desired standards. Consequently it is worthwhile seeking improved technology to support them.

1.2 Historical Background

In the past, several approaches have been tried to deal with the problem of building and maintaining persistent application systems. The technologies and methodologies proposed are reviewed here.

During the last two decades many **technologies** aimed at reducing software engineering costs have been proposed:

1. Data Base Management Systems (DBMS) assuring the storage of data in a physical layer and providing a high level view (Logical Model) [Codasyl Committee on Data

System Languages, 1971, Taylor and Frank, 1976]. These DBMS provide data independence together with a Data Definition Language, a Data Manipulation Language, a Query Language and a host language interface (C, PASCAL or COBOL) [Ullman, 1988].

2. Logical Data Models [Tsichritzis and Lochovsky, 1982] with better characteristics, supporting implementations with better performance and providing easier to use query languages, e.g. the Relational Model [Codd, 1970].

3. Semantic Models providing a higher-level description (conceptual model), independent of the implementation and aiming at supporting the usage of a design methodology for software development; the entity-relationship model [Chen, 1976], RM/T [Codd, 1979], TAXIS [Mylopoulos *et al.*, 1980], SDM [Hammer and McLeod, 1981], and IFO [Abiteboul and Hull, 1987].

4. The ODMG model based on objects, more complex and powerful than the Relational Model [Cattell, 1994]. This norm comprises the data model, a data definition language (ODL), a query language (OQL), a mapping to C++ and Smalltalk and leaves unspecified the data manipulation language (OML).

5. Other data models [Brodie, 1984] based on different data manipulation paradigms; functional models, e.g. Daplex [Shipman, 1981] or logical, e.g. LDL [Beeri *et al.*, 1987].

6. Database Programming Languages which integrate a data model in an existent programming language, e.g. Pascal/R [Schmidt, 1977].

7. Persistent Programming Languages, e.g. PS-algol [Atkinson *et al.*, 1982], or Napier88 [Morrison *et al.*, 1989].

8. Extensions to Relational DBMS, to incorporate objects, versions, historical data, procedures and a powerful extended relational query language, as in POSTGRES [Stonebraker and Rowe, 1986] or to allow user-defined extensions and a query language that extends the relational algebra, as in STARBUST [Schwartz *et al.*, 1986].

9. DBMS generators like EXODUS [Carey *et al.*, 1988].

10. Object-Oriented Data Base Management Systems (ODBMS) [Cattell, 1991a, Cattell, 1991b], concerned with more complex data structures that arise in CAD, CASE or Office Systems: O_2 [Bancilhon *et al.*, 1988], ONTOS [Andrews *et al.*, 1989], GemStone [Maier and Stein, 1987], ObjectStore [Lamb *et al.*, 1991], IRIS [Fishman *et al.*, 1987].

11. Higher-level languages and packages usually focus on specific application domains: 4GLs [Carson, 1989], form-oriented tools oriented to business database applications [Bor, 1990], or products targeted at simplifying development of user interfaces in window systems [Mic, 1992].

Over the same period, several **methodologies** have been proposed to tackle the difficulties encountered in building and maintaining persistent application systems. These make effective use of both data models to design the database and programming languages to code the processes dealing with data. The following methodologies have been widely used:

- Relational Normalisation [Codd, 1970, Codd, 1972].
- Entity Relationship Modelling [Chen, 1976].
- Structured Analysis [DeMarco, 1978].
- Structured Design [Yourdon and Constantine, 1978].
- Structured Systems Design [Gane and Sarson, 1982].
- Jackson System Development (JSD) [Jackson, 1983].
- Object-Oriented methodologies [Booch, 1991, Coad and Yourdon, 1990, Rumbaugh, 1991].

CASE (computer-aided software engineering) tools are used to support the use of methodologies and assist in the development and maintenance of application systems [Davelaar and van Kooten, 1996]. These tools provide support for: specification, design, development, maintenance, project coordination, multiple version handling and support for simultaneous access.

The most recent proposals involve object-orientation concepts. The intention of these approaches is that both the design and implementation depend on the same concepts: encapsulation, inheritance, information hiding, modularity, etc. This identification of design and implementation concepts should facilitate the software production. Indeed, a great deal of effort has been put into providing Object-Oriented Database Management Systems (ODBMS) with the sort of theoretical framework [Atkinson *et al.*, 1989], capabilities and performance that the Relational technology has reached [Lécluse *et al.*, 1990, Benzaken and Delobel, 1990, Delobel *et al.*, 1995]. ODBMS are supported by Object Stores[1].

One last approach to minimise the difficulties encountered in building and maintaining persistent application systems, lies in the field of formal specifications. Formal specifications and automatic code generation from specifications have been subject to extensive research and several languages exist: Z [Spivey, 1989, Dilles, 1990], VDM [Jones, 1990], or LOTOS [Bolognesi and Brinksma, 1989]. As yet, these formal approaches do not appear to scale up to large persistent application systems. In any case, the use of formal specifications can still benefit from improvements in their target technology.

Until now none of these approaches is sufficient to effectively support the development and maintenance of Persistent Application Systems.

1.3 Persistence

This section provides background information in persistence for the benefit of readers that are not familiar with the subject. The work on persistent languages was initiated by Malcolm Atkinson in 1978 [Atkinson, 1978]. The text presented here draws heavily on the work of the research groups from the University of Glasgow and the University of St Andrews in Scotland.

[1]Object stores are also used to support persistent programming languages introduced in Section 3.2.

1.3.1 Definition of Persistence

Persistence of a data object is defined as the length of time that the object exists and is usable ([Atkinson *et al.*, 1983a, Atkinson and Morrison, 1985]). A spectrum of persistent values exist [Atkinson and Morrison, 1995]:

- transient results in expression evaluation;
- local variables in procedure activations;
- global variables, heap items;
- data that lasts a whole execution of a program;
- data that lasts for several executions of several programs;
- data that lasts during the life of a program;
- data that outlives a version of a program;
- data that outlives versions of the persistent system.

Traditionally, programming languages have supported short-lived data and file systems or DBMSs have been used to support the other categories of data.

1.3.2 The Traditional Approach to Persistence

The traditional approach to the provision of persistence of data is to store it in operating system files or databases. Procedures are held in libraries and can be reused by linking them to programs. Traditionally, a programming language is used to manipulate transient values and a DBMS or file system is used to manipulate persistent values. DBMSs usually have an interface to an embedded programming language that overcomes the lack of computational completeness that characterises the DBMS's data manipulation languages. To build user interfaces and perform complex calculations, programmers usually had to use this programming language interface.

Data used inside programs are usually organised in some structured way (e.g. lists or trees) that must be flattened and explicitly transferred to some secondary storage. To be reused, these data are reread into memory and the structure must be rebuilt. With procedures in libraries, type information such as the signatures of the procedures does not usually go with the procedure and, therefore, types cannot be verified on each procedure's usage. In the traditional approach to the provision of persistence the mappings represented in Figure 1.1 must be maintained.

Within this approach an "**impedance mismatch**" exists between data in memory when the program is running and the same data made persistent; the fact that there are two views of data has some important disadvantages [Atkinson *et al.*, 1983a]:

- programming is considerably more difficult because the programmer has to maintain the three mappings between the database model, the programming language model and the real world model;

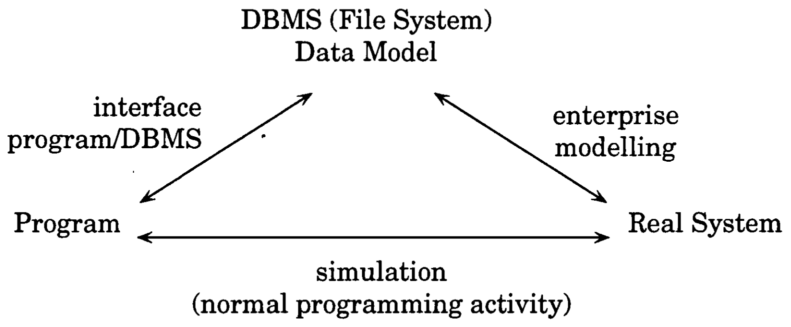


Figure 1.1: Conceptual Mapping in the Traditional Approach to the Provision of Persistence

- usually 30% of the code deals with the transfer of data to and from files or DBMS[IBM, 1978];
- data protection offered by programming language's type systems is lost across this mapping;
- referential integrity of objects may be lost across store operations; and
- computational costs may be increased as the programming language runtime system system, operating system and DBMS vie for common resources.

To avoid these unnecessary complications which are illustrated by Figure 1.1, a different approach to the provision of persistence must be used.

1.3.3 Orthogonal Persistence

The need for orthogonal persistence was first identified in [Atkinson, 1978]. A single model for data of all ranges of persistence was proposed [Atkinson *et al.*, 1982, Atkinson *et al.*, 1983a]. That model applies to data with the full spectrum of persistence: from data that only lives during a program activation (or even a block in the program) to data that outlives the program. Using the model, the simplification represented in Figure 1.2 is achieved.

Such a language is said to be a **persistent programming language** (PPL) if the programmer does not need to explicitly order the movement of data to or from a persistent store. If values of all the types of the language have the right to persist then the language displays **orthogonal persistence** [Atkinson and Buneman, 1987].

It has been observed that parsimony of concepts allied with the use of powerful composition rules could achieve expressive programming languages: "power through simplicity and simplicity through generality". Strachey and Tennent quantified these ideas in principles that should guide the design of programming languages: the principle of correspondence, the principle of abstraction and the principle of data type completeness [Tennent, 1977]. The last principle states that when a type may be used in a constructor, any type is legal without

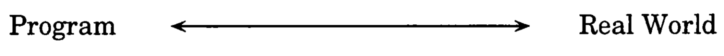


Figure 1.2: Conceptual Mapping Simplification by Using a PPL

exception, that is, every type has the same "civil rights" () in the language [Tennent, 1977, Morrison, 1979]. Languages obeying these principles are more powerful and less complex as they have few defining rules allowing no exceptions.

These general principles lead to specific principles identified in [Atkinson *et al.*, 1983a] as yielding **orthogonal persistence** and stated in [Atkinson and Morrison, 1995] as:

- **principle of persistence independence** — the form of a program is independent of the longevity of the data that it manipulates, that is, programs look exactly the same whether they are manipulating short-term or long-term data;
- **principle of data type orthogonality** — all data values should be allowed the full range of persistence irrespective of their type; and
- **principle of persistence identification** — the choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system, that is, the mechanism for identifying persistent objects is not related to the type system.

The first principle requires, for example, that a procedure may be applied with persistent or transient parameters. One important consequence of the second principle is that it allows programs to be incrementally developed and simplifies maintenance by component replacement. Languages which conform with these principles avoid the impedance mismatch problem and its associated disadvantages and greatly simplify the work of programmers in coding PAS systems. The advantages of orthogonal persistence are described in [Morrison and Atkinson, 1990]. The use of orthogonal persistence and flexible binding mechanisms were identified as contribution to the possibility of software reuse and system evolution [Connor, 1991]. Different methodologies for system composition are possible [Dearle, 1988, Connor, 1991, Cutts, 1993, Sjøberg, 1993].

Based on the hypothesis that the provision of persistence should be independent of all the other language design aspects, one would expect to find persistent languages arising from all programming paradigms. That is indeed the case and some known persistent languages are enumerated in Table 1.1.

Persistent programming languages and **database programming languages** (DBPL) are aimed at dealing with large amounts of long-lived data. While PPLs start from a language and use its type system to provide a data model, a DBPL starts from a data model and aims to provide a general-purpose algebra over it. Recent research in type systems has led to a repertoire of constructs that "would appear to offer similar descriptive power to that in data models" and the correspondence presented in Table 1.2 may then be drawn between programming languages and databases [Atkinson, 1992a].

Due to its advantages in coding and maintaining PAS, languages which display orthogonal persistence are identified as the most promising approach to be followed in seeking improved technology for PAS support. As pointed out in [Carey and DeWitt, 1996], this technology failed to emerge in commercial products so far despite the fact that research in the area generated a number of interesting results.

Language	Paradigm
Pascal-R [Schmidt, 1977]	imperative, relational
PS-algol [Atkinson *et al.*, 1982]	imperative, Algol types
Napier88 [Morrison *et al.*, 1989]	imperative, polymorphic types
DBPL [Matthes and Schmidt, 1989]	imperative, relational
Daplex [Shipman, 1981]	applicative
Poly [Matthews, 1985]	applicative
Amber [Cardelli, 1986]	applicative, parametric and inclusion polym.
Staple [Davie and McNally, 1990b]	applicative
P-Quest [Matthes, 1991]	applicative
Tycoon [Matthes *et al.*, 1994]	applicative
Fibonacci [Albano *et al.*, 1994]	applicative
Galileo [Albano *et al.*, 1985]	object, inclusion polymorphism
Leibniz [Evered, 1985]	object
Persistent Smalltalk [Hosking *et al.*, 1990]	object
O_2 [Bancilhon *et al.*, 1988]	object
Shore [Carey *et al.*, 1994]	object
Theta [Liskov *et al.*, 1994]	object
E [Richardson, 1989]	C++ based
ObjectStore [Lamb *et al.*, 1991]	C++ based
ONTOS [Ontologic Inc., 1991]	C++ based
GemStone [Maier and Stein, 1987]	Smalltalk based
ORION [Kim *et al.*, 1988]	LISP based
Persistent PROLOG [Gray *et al.*, 1988]	logic
LDL [Tsur and Zaniolo, 1986]	logic
TAXIS [Mylopoulos *et al.*, 1980]	semantic
χ [Hurst and Sajeev, 1989]	capability

Table 1.1: Examples of Persistent Languages and Systems

Programming languages	Databases
Type system	Data model
Type	Schema
Variable	Database
Value	Instantaneous DB extent

Table 1.2: Vocabulary of Equivalences between Programming Languages and Databases

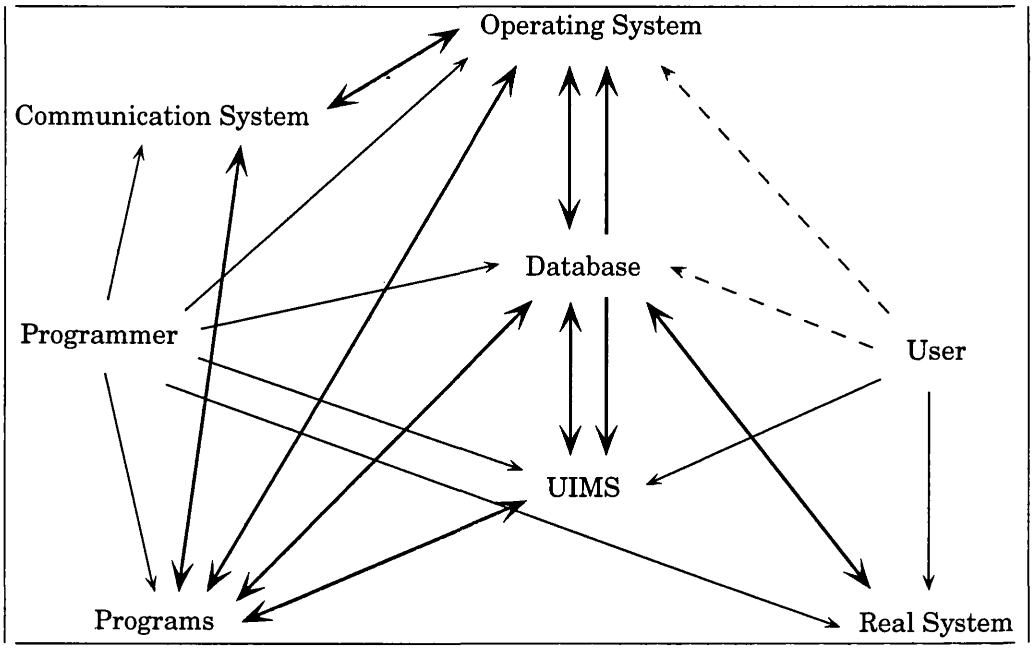


Figure 1.3: Complex Mappings to be Maintained in a Typical PAS

1.4 The Need for New Architectures

In order to construct PAS, programmers have to use a multitude of different construction components such as: operating systems, user interface management systems, DBMS, programming languages, communication systems, etc. Instead of the triangle of Figure 1.1, a more realistic representation is shown in Figure 1.3 [Atkinson, 1992a]. Heavy arrows denote mappings that have to be maintained by programmers and the light arrows denote the components each class of person has to understand. The dashed arrows denote undesirable awareness by users of construction components. Keeping all these mappings consistent is a difficult task and erroneous behaviour may occur in the PAS functioning, due to differences in semantics of the different views over common concepts.

Facilities like: persistence[2], stability[3], recovery[4], concurrency[5], etc. are provided simultaneously by the operating system and DBMS but not always with a consistent model and sometimes conflicting with each other in those tasks. It was pointed out in [Atkinson, 1992a] that:

> "It is marginal differences in the behaviour of subcomponents that purport to provide the same service that cause the problems when systems are under stress, whereas differences in the special part of each construction component are precisely those that are useful."

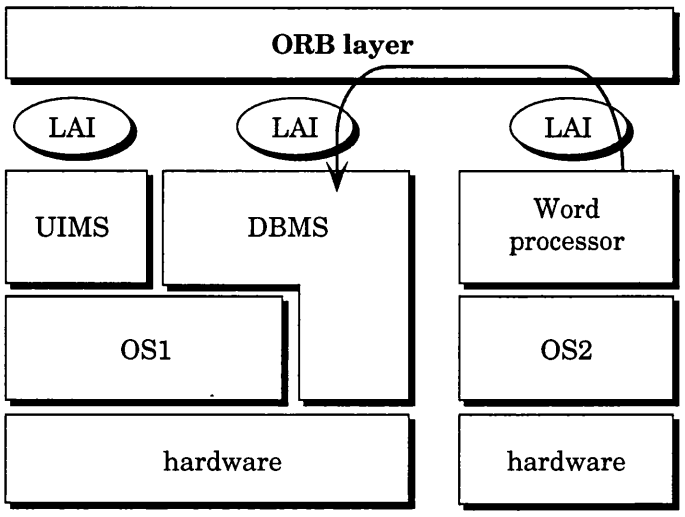
A widely accepted approach to enable interoperability is to build a standard interface between heterogeneous sub-systems. This approach is surveyed in next section.

[2]The support for data values during their full life times.

[3]Being conceptually failure free.

[4]The ability to recover from transaction, system or media failures to a consistent state.

[5]The ability to have more than one program or different version of the same program running simultaneously.



LAI – local application interface

Figure 1.4: Common Object Request Broker Architecture

1.4.1 Common Object Request Broker Architecture

The “Common Object Request Broker Architecture” (CORBA), a proposal by the Object Management Group (OMG), is aimed at achieving interoperability between standard components using standard protocols [Schaffert, 1992]. As depicted in Figure 1.4, a layer is overlaid on top of the different service providers in order to hide inconsistencies, thereby enabling interoperability between heterogeneous environments and allowing the integration of a wide variety of object systems.

An architecture is presented in [OMG, 1991], by specifying a concrete object model and an Interface Definition Language (IDL), that can be used to describe the interfaces that client objects call and object implementations provide. The **Object Request Broker** (ORB) layer provides message passing (an object request) between objects and clients, as represented in Figure 1.5. In this way, objects may be implemented using different languages and then mapped to IDL with the aid of stub generators.

Using this approach some uniformity of behaviour can be obtained with relatively little

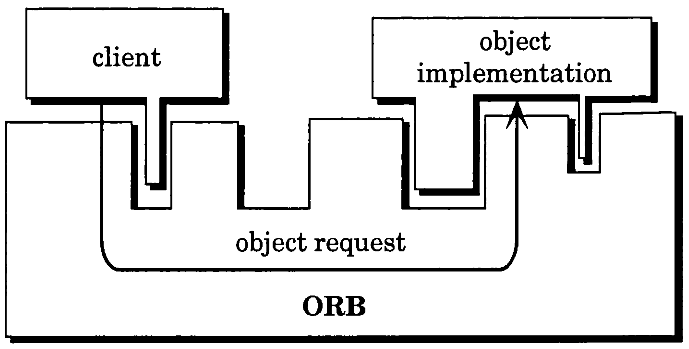


Figure 1.5: Interface Definition Language

effort. As applications are fully responsible for the management of their CORBA objects, application performance and the programmer's productivity may be affected. The impedance mismatch between persistent and transient data referred to in Section 1.3.2 may again be present. Although the objects themselves may be mapped to a common model, as yet, failure behaviour, recovery, resource management, etc. cannot. To achieve this end, the semantics of an acceptable common model will first have to be developed and validated.

CORBA does not solve the problem of keeping the mappings of Figure 1.3 consistent.

1.5 Contributions of This Work

The research presented in this dissertation concerns the support of persistent higher-order and reflective languages. These languages are used in coding and maintaining long-lived and potentially large application systems.

The technical challenges in making persistent programming languages scalable, with adequate performance and sufficient longevity and in amortising costs by providing general services are identified and an architecture is proposed. As will be demonstrated later, some of the crucial components of the architecture are: the use of a persistent object store and a means to identify the longevity of data items, an incremental binding mechanism to allow existing data and new data to be combined, an identity mechanism stable for long-lived data, management of closures in order to provide a form of block retention and management of space in order to find pointers during garbage collection. The architecture must provide adequate constructs to support uniform polymorphism, a type-checking mechanism working for data of all spectra of persistence and a naming mechanism oriented to incremental construction and change.

The novelty of the approach presented in this dissertation resides in the use of an independent representation for programs written in persistent higher-order and reflective languages, and the introduction of machine independent optimisations and code generation into the context of the support for this class of languages.

The following contributions are made in the field of support for persistent applications and languages:

1. the proposal of a new architecture for compilation, comprising three-stages, with a context for high-level machine independent optimisations, machine dependent optimisations and code generation;

2. the identification of the constructs needed to support persistent higher-order and reflective languages;

3. the design of an intermediate language incorporating those constructs and which can serve as a target for parsers for different high-level languages;

4. the study of high-level and machine independent optimisations in the persistent programming language context which can be accomplished by transforming the internal representation proposed;

5. an investigation of the usage of the continuation-passing style transformation in this context, as a means to achieve performance and to simplify the runtime system;
6. an investigation of time and space efficient management of closures in the context of persistent programming;
7. an investigation of the use of C as a portable representation of programs expressed in the intermediate representation; and
8. a compilation framework which can be used to experiment further in the context of the support for long-lived and potentially large application systems.

1.6 Thesis Structure

The remainder of this dissertation comprises eleven further chapters. Chapters 2 to 5 propose a new architecture for compilation of the persistent programming languages of interest for this work. After the identification of the needed constructs, for each component of the architecture the possible techniques to be used are described and compared. Chapter 6 presents the experiment to be conducted in order to help in identifying the technical challenges involved in the task of making support for persistent languages of adequate performance and of sufficient longevity, and in the task of providing for reuse of components. Chapters 7 to 11 present and evaluate the results of this experiment and chapter 12 draws the conclusions for the work presented in this dissertation.

The remainder of this chapter presents a brief description for each chapter of this dissertation.

Part I: Introduction

This chapter introduces the problem of building and maintaining persistent application systems and indicates traditional and new solutions to the provision of persistence. The use of orthogonally persistent languages is recognised as a promising technology. This chapter also identifies the need for new architectures in order to simplify the usage of the multitude of different construction mechanisms which are in use today.

Part II: An Architecture for Compilation

Chapter 2 concentrates on an intermediate language which can support the compilation and execution of programs written in persistent higher-order and reflective languages. The motives to investigate the facilities to be provided by this language are enumerated. A comparison of interpretation and machine code generation or combinations of both is presented. This leads to a new architecture for compilation. This architecture has two intermediate representations: the first representation is an internal language at a higher-level and the second is a target language closer to the hardware machine.

Chapter 3 describes the characteristics of the source languages which the proposed architecture needs to support and identifies the constructs which must be included in the internal language. These persistent higher-order reflective languages (PHOLs) are recognised to facilitate the task of incrementally building and maintaining persistent application systems. The requirements for the underlying layers of the architecture in order to support persistence and stability are also identified.

Chapter 4 surveys possible technologies which can be used to achieve a concrete high-level intermediate representation to be used by front-ends for the high-level languages anticipated. Existing examples of possible techniques are compared with respect to space efficiency, simplicity of optimisation, simplicity of code generation and generality with respect to the high-level language and the hardware machine. The chapter concludes by choosing a representation suitable to be used in an experiment that builds a prototype of the proposed architecture.

Chapter 5 surveys possible techniques for target languages needed at the lower-level end of the proposed architecture. These candidate technologies are compared with respect to their adequacy for store management; their support for persistence, stability, recovery and concurrency; their support for dynamic binding and linking; their independence of the target machine. For each target language, the quality of the generated code, in terms of volume and execution speed, and the compilation speed are also discussed.

Part III: Design of an Experiment

In Chapter 6 an experiment to build a prototype for the proposed architecture which can prove the thesis, prove the architecture feasible and worthwhile and lead to the identification and validation of its crucial features, is presented. The components of the prototype language framework to be built are also enumerated. A suitable PHOL to be used in the experiment is described and the strategies which can be used to transform the high-level internal representation into the other internal representations at a lower-level are enumerated and compared. Finally, this chapter concludes by choosing and justifying the enabling technology and describing briefly the internal data structures which can be used to support the high-level internal representation and the transformations performed.

Part IV: Implementation and Evaluation

Chapter 7 presents the language design of the internal representation intended to be used by all front-ends of the language compilation framework. The characteristics of this language (called TPL) are enumerated, together with the complete set of data types and corresponding instructions. A concrete syntax is presented in order to be used later to illustrate the use of this language.

Chapter 8 continues the description of the experiment by showing how TPL can be used by front-ends in compiling the PHOL anticipated. This is demonstrated by presenting examples for relevant language constructs extracted from the compilation of complete programs. For each construct, the translation rules involved in the process are enumerated. Finally, the front-end used in the prototype is briefly described.

Chapter 9 illustrates the support for high-level and machine independent optimisations on TPL internal representations of programs. The transformations described in this chapter include partial evaluation techniques, such as constant folding and constant propagation; redundancy elimination techniques, such as unreachable-code elimination, useless-code elimination, common-subexpression elimination; and procedure call transformations, such as inlining, procedures called only once, dropping unused arguments and tail recursion. The components of the language framework which implement some of these transformations are described. This chapter finishes by describing the use of continuations as a vehicle for optimisation, the implementation of this transformation in TPL and the properties of TPL changed by this transformation.

Chapter 10 presents the design of a low-level abstract machine and discusses how it can be used to support TPL. The transformations which must be performed in TPL in order to achieve a representation suitable for execution are described and illustrated by fragments of programs. The runtime system, which supports object creation and access and the interaction with the underlying layers, is presented. Finally, the use of a garbage-collected object store to achieve persistence, target machine code generation and program execution with dynamic binding are discussed.

Chapter 11 evaluates the architecture proposed, presents the findings from the experiment conducted and concludes by presenting design changes. Together with the achievements, the limitations of the prototype and of the experimental work are presented. The design space is covered by describing the implications of the goals of supporting this class of languages while ensuring longer-term persistence of data and sufficient efficiency. For each of the goals, the design decisions are evaluated in the face of the results.

Part V: Conclusions

Chapter 12 presents a survey of the conclusions drawn in Chapter 11 and presents proposals for future work. It concludes that the architecture presented in this thesis proved to be appropriate in the construction of supporting technology for persistence.

Chapter 2

An Architecture for Compilation

The previous chapter identified the need to support persistent application systems with a coherent set of construction components. This chapter refers to the SPF architecture as the solution to this problem and concentrates in its interface low-level language. This intermediate language can support the compilation and execution of programs written in persistent higher-order and reflective languages. The motives to investigate the facilities to be provided by this language are enumerated. A comparison of interpretation and machine code generation, or combinations of both, is presented. These techniques can be used to achieve the above-mentioned compilation. Finally, a new architecture for compilation is presented. This architecture has two intermediate representations: the first representation is an internal language at a higher-level and the second is a target language closer to the hardware machine.

2.1 Introduction

The **motivation** for starting this research on architectures to support persistent programming languages (PPL) was due to the observation that the current technology did not perform adequately. The currently available persistent environments are comparatively slow in response time and sometimes greedy in space. More efficient implementations are needed.

As there is not an agreed interface to object stores, a lot of research is going on in duplicating store implementation. The cost of building stores could be amortised if the same store can be used in all applications that use a common architecture. This possibility introduces greater flexibility as applications can move unchanged to new stores as their load evolves, by the use of a program designed to perform that task.

The provision of an intermediate representation stable over changes in the underlying machine architecture, may allow future language implementations to take advantage of the underlying features, with only a small effort in porting the back-end to generate the internal representation. On the other hand, the dramatic changes that are taking place in the

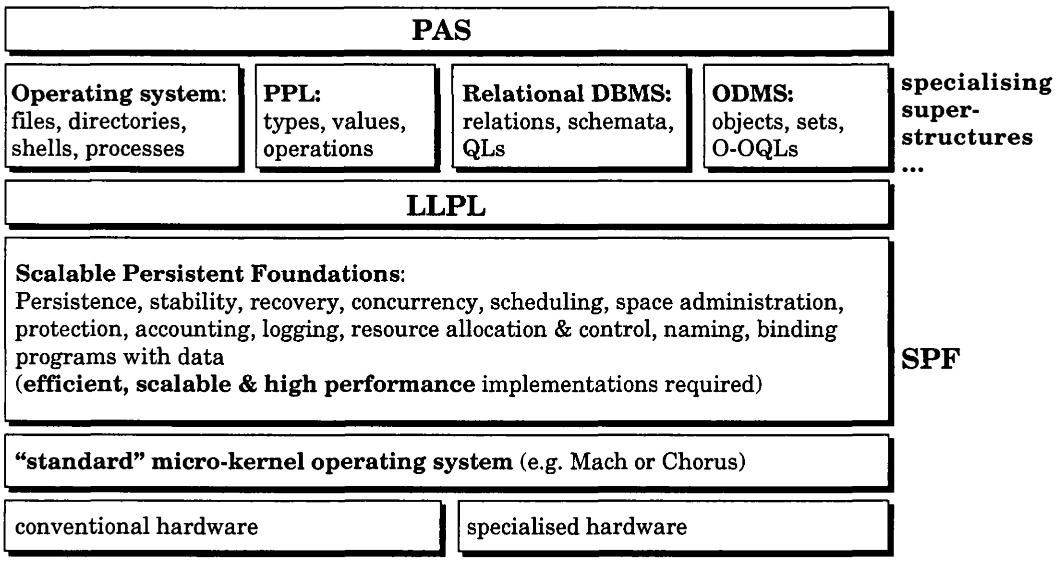

PAS
Operating system: files, directories, shells, processes
PPL: types, values, operations
Relational DBMS: relations, schemata, QLs
ODMS: objects, sets, O-OQLs
specialising super-structures ...
LLPL
Scalable Persistent Foundations: Persistence, stability, recovery, concurrency, scheduling, space administration, protection, accounting, logging, resource allocation & control, naming, binding programs with data (**efficient, scalable & high performance** implementations required)
SPF
"standard" micro-kernel operating system (e.g. Mach or Chorus)
conventional hardware
specialised hardware

Figure 2.1: Scalable Persistent Foundation Architecture

support technology (e.g. hardware or store implementations) will be isolated from the applications that use this technology[1]. In this way, the introduction of this technology will have an impact on the efficient use of languages and systems incorporating persistent programming principles and on the adaptation to hardware changes. That efficiency will later be reflected in PAS development and maintenance.

2.2 New Architectures to Support PAS

The need for new architectures which can provide construction components with consistent semantics was identified in the introductory chapter. The CORBA approach was referred to as a common proposal to solve the problem. It was concluded that this approach does not solve it satisfactorily.

2.2.1 Scalable Persistent Foundations

A different approach to achieve interoperability has been proposed in [Atkinson, 1992a, Gruber and Valduriez, 1994]. These proposals advocate a two level architecture. Such an architecture is represented in Figure 2.1; it is called a **Scalable Persistent Foundation** (SPF) in [Atkinson, 1991]. A common substrate provides the most critical functionality of construction components not normally used directly by application programmers. This common substrate includes the support technology, SPF, and its interface language, called LLPL (Low-Level Programming Language). LLPL is a stable, or at least easily evolved, interface.

[1]The introduction of this intermediate representation may introduce inefficiencies which will be dealt by the use of program analysis and transformations in order to achieve more efficient representations and by generating target machine code.

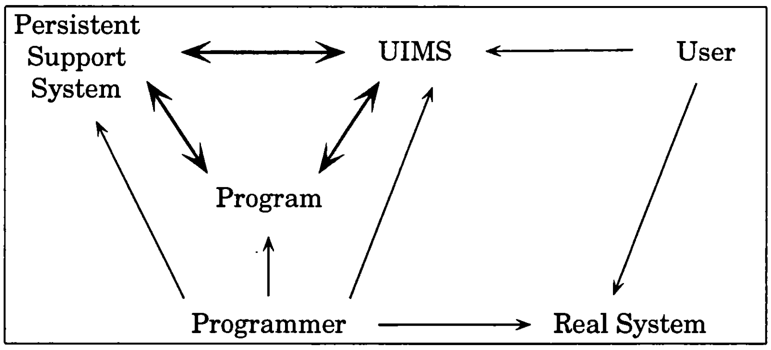


Figure 2.2: The Goal of a Simpler PAS

Using LLPL, specialising superstructures can be built taking advantage of the common foundation and concentrating on the differences. These special parts include facilities traditionally provided by operating systems, persistent programming languages or DBMS. The cost of building the support technology and its interface will be amortised over all specialising superstructures and PASs.

The SPF architecture would:

1. give a common model to all such components including their behaviour under stress;
2. provide economy of scale and reuse;
3. reduce the code required to support individual components like PPLs and DBMSs;
4. provide efficient scalability.

This architecture could potentially achieve high performance because it could, for example, make direct use of the memory management hardware for data movement, protection and stability.

Taking the simplification in the computational context referred to in Section 1.3.3 further, the research experiment proposed in [Atkinson, 1992a] is aimed at enabling the construction, maintenance and operation of PAS and presenting to users and programmers of the supported PASs a simpler set of mappings, as depicted in Figure 2.2.

Research is needed to identify the facilities to be provided by SPF via LLPL and to ensure that these facilities are sufficiently independent of particular technologies that they can be kept operational for many decades.

2.2.2 Other Approaches

In order to obtain consistent behaviour and efficiency there are other approaches under investigation.

Researchers developing persistent operating systems are also approaching the delivery of similar support functionality starting from a conventional hardware platform [Dearle *et al.*, 1994]. They are designing an operating system that directly supports orthogonal persistence and a capability-based protection mechanism. Within this operating system, processes are integrated with the object space. On top of these operating systems all languages will achieve persistence automatically.

A design study of a hardware architecture to support object addressing at instruction level was undertaken in DAIS [Russell *et al.*, 1994] as part of a proposed object-oriented persistent environment [Russell, 1995]. The DAIS approach uses a cache structure based directly on object descriptors and offsets aiming at providing both security and speed. The virtual memory architecture allows for position and media independence of data. DAIS achieves efficiency by providing a RISC-like architecture with only a minimum of object-access instructions. Earlier experiments on hardware support for persistence were made in the development of the MONADS architecture [Rosenberg and Keedy, 1987].

These operating systems or hardware architectures may substitute the lower layers of SPF but they still need a sort of LLPL as well in order to enable interoperability.

2.3 Target Persistent Language

This work presents an instance of the LLPL, the interface language to SPF and a workbench for future experiments towards a SPF. This intermediate representation is called TPL, which stands for Target Persistent Language and is pronounced "tipple" [2]. Some of LLPL's intended features will be covered by TPL, namely the facilities related with support for more than one language and the use of possible different object stores to achieve persistence, stability and recovery. Inter-language interoperability and object store independence are depicted in Figure 2.3. TPL resembles the idea introduced in the UNCOL (UNiversal COmpiler-oriented Language) [Strong *et al.*, 1958] but with similar high-level languages above it and similar object stores below it. UNCOL was proposed as a universal internal representation enabling the construction of compilers for l programming languages producing code for m target machines by using l front-ends plus m code generators, as opposed to the otherwise lxm distinct compilers needed. UNCOL was an ambitious effort that failed because it was too general (as it would be applied to all the languages), because the machines at that time had insufficient capacity and because language and compiler technology were not yet mature[3].

The new architecture should take account of recent proposed improvements in compiler technology such as: advances in functional programming language implementations, new intermediate representations, new classes of optimisations and new code generation techniques. This research investigates the application of these new techniques in order to achieve a **high performance target language for persistent systems**.

This target language is an intermediate representation intended to:

1. be general purpose, allowing:

   (a) a means of isolating the work of system writers (e.g. compiler writers) from the underlying object store implementations;

[2]Take the habit of taking alcoholic drinks specially in small quantities; alcoholic drink; device to help unload trucks [Makins, 1991].

[3]In more recent times, ANDF was developed with similar intentions and JAVA bytecodes are now offering again the idea of universal code portability.

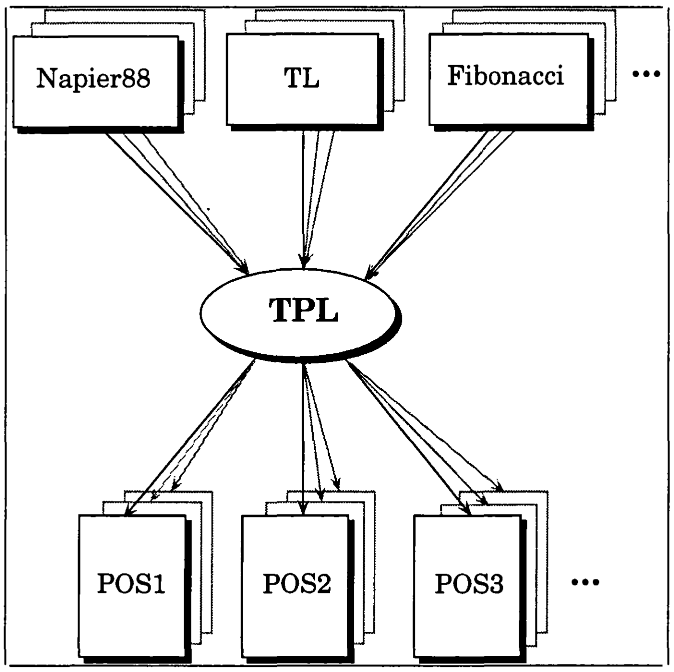


Figure 2.3: Interoperability in the Context on an SPF

   (b) an easy way of experimenting with stores, languages and other systems (e.g. DBMS);

2. provide adequate support to high-level languages with first-class procedures, polymorphism and reflection;
3. provide inter-language interoperability (enabling protection and distribution across different machines);
4. provide longer-term persistence of data which in this context always includes code as in the representation of procedure closures, abstract data types and methods;
5. enable high performance implementations; and
6. ultimately, the full range of facilities needed for building PASs.

The goals to be achieved in the long run by this intermediate representation will be detailed in the following sections. The work described in this dissertation focuses on the support of persistent higher-order and reflective languages and the ability to perform optimisations and code generation in this context.

2.3.1 Generality

TPL will allow writers of compilers for persistent versions of languages like C, C++, Pascal, ML, or PPLs like Napier88, Fibonacci and TL to experiment with using it as a target language (see Figure 2.3). To achieve this generality, the TPL language processor[4] will accept a

[4]The term language processor is used to encompass the software and hardware combination which handles all aspects of compilation and runtime management.

compiled form of TPL. Store writers may also use this language as a way of experimenting and tuning their implementations to a wider range of languages. For that, they may provide a "Store Library", a set of procedures covering the store functionality. This way, TPL may establish a **standard interface** between persistent object stores and all their users, allowing for higher-level systems and object store reuse and independent evolution of stores and languages.

A "sugared" version of TPL would permit its direct usage by other support system implementers, such as DBMS and operating systems providers. In such a way, its users could benefit from the common features provided, such as: persistence, stability or recovery. This human readable form of TPL, intended to be used by system application writers, referred to in [Atkinson, 1992b], constitutes complementary research.

2.3.2 Language Features

This research is directed to supporting high-level programming languages which exhibit: orthogonal persistence, reflection, higher-order procedures and polymorphism. Orthogonal persistence was introduced in Section 1.3.3, and the other language features will be introduced in Section 3.2 and followed by an identification of the constructs TPL must have in order to support this class of languages.

2.3.3 Interoperability

The provision of an intermediate representation, stable over changes in the underlying machine architecture, will enable the movement of objects (which include code) between machines and their distribution across the network. Interoperability among different languages can be supported as one value may be created by one program coded in one language and used by another program coded in a different language. This is a recognised hard task which is an ongoing research area [Kato and Ohori, 1992]. As the intention is that the only route by which data may be accessed is via TPL, its low-level type-system may be used to enforce protection of both transient and permanent data [Morrison *et al.*, 1990].

2.3.4 Longer-term Persistence

It is believed that Persistent Programming Systems will only be successful if users trust the longevity of their data. To guarantee the longevity of data and programs one must guarantee that **their meaning remains the same whenever they are reused**.

Let us look at the simple example of an integer represented as a 32-bit quantity. Even if we only consider a change in size to 64-bit representations, the semantics of integer operations will change (e.g., overflow). This change in semantics may be acceptable in some situations in which case it would only be necessary to generate a new representation from the canonical value representation. On the other hand, it may be necessary to preserve all the semantics and a type must be provided (e.g. *INT32*) together with its operations in order to keep the intended semantics. It is not obvious where to draw the stability line and further research is needed. It is also not obvious when changes are innocuous, so users of the proposed architecture are requested to make a conscious decision to obtain changed semantics. The owner of the data may decide to preserve exactly the same semantics. Therefore in this thesis unchanged semantics is assumed.

To provide longer-term persistence of data, it is then necessary to guarantee that it will be possible to run programs against data either of which were made persistent many years before. In other words, the semantics and bindings of data items in the persistent store must be preserved over the changes in the underlying architecture (e.g. changes in hardware platforms, evolution of compilers, languages and systems). Ideally the abstraction over the format in which data is stored (e.g. length of words, position of bytes inside words, etc.) will be gained through the use of TPL as an architecture independent representation. An architecture that guarantees, at any time, the transformation from that intermediate representation to highly efficient machine code is sought. TPL may itself evolve in a way which keeps old TPL programs with unchanged semantics.

In the kind of languages addressed by this work, with first-class procedures and orthogonal persistence, a store includes procedures bound to other data and data bound to procedures in an intricate graph[5]. Moreover, some of the data is only reachable from the programs (code) that created it (e.g. compilers) and with proper encapsulation that data is only accessible through that code (for example, non-global free-variables).

Old solutions to the problem of migrating data to new architectures by translation [Shu *et al.*, 1977] do not work anymore, as humans are not aware of the formats in use, nor even of all the data to be translated. For example, encapsulated data in free-variables, abstract data types and objects are not directly accessible to them and the format of these data is only known to the compiler writers. The specification of formats is needed to perform the mapping from the old to the new formats. The architecture to be proposed is intended to ensure that the necessary information is captured during the compilation process.

2.3.5 Efficiency

It has been observed in practical applications that there is a real need for efficiency improvements in some of the PPL implementations. For example, the project taken by the author as a warm-up exercise [Lopes, 1992], demonstrated that the overall performance of Napier88 was below that which is required for user interfaces [Lopes, 1993, Sjøberg *et al.*, 1993].

One of the aims of this work is to demonstrate that using stock hardware it is possible to build persistent programming environments that are more efficient, both in time and in space usage, when compared with the currently available implementations.

2.4 Interpretation or Machine Code Generation

The need was identified to support a class of high-level languages by using an intermediate representation which provides generality, efficiency and longer-term persistence. The search

[5]This graph may be seen with nodes representing data objects and arcs representing bindings.

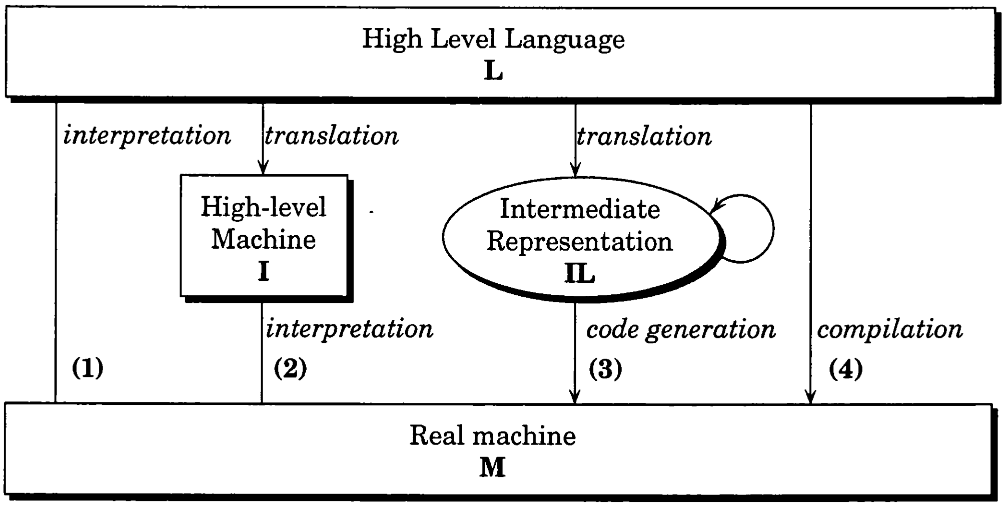


Figure 2.4: Interpretation Versus Machine Code Generation

for adequate architecture continues with a comparison of interpretation and code generation which can support the execution of programs.

When a program is written in a high-level language L, then a way of having it to execute on a machine M must be provided (see Figure 2.4). Implementation of high-level languages may be based on interpretation or compilation or combinations of these techniques. In pure interpretative systems like some BASIC implementations, case (1) of Figure 2.4, a process running on machine M directly implements the high-level language L by fetching, decoding and executing language L instructions. These systems are very inefficient, constrain the language to be simple and have the undesirable property that syntax errors can only be detected at runtime. The other extreme is case (4) where code for machine M is generated and directly executed.

Another approach is to translate the high-level language L into code for a virtual abstract machine I, case (2), which is then interpreted as in case (1). There exist several examples of implementations of high-level languages that follow this approach. For persistent programming languages, the PAM [Brown *et al.*, 1988, Connor *et al.*, 1989] and the PQM [Matthes *et al.*, 1992] are examples of such abstract machines. They use a stack for operands, results and local variables, and execute *bytecode* instructions.

Case (3) represents the last possible approach where language L is first translated to an intermediate representation IL, transformations on that representation can be made to improve it and finally code for machine M is generated. This approach was followed, for example, in the SML/NJ compiler [Appel and MacQueen, 1987]. The compiler to the dynamically typed object-oriented language SELF [Ungar and Smith, 1991] employs a technique called dynamic compilation to achieve better runtime performance than an interpreter similarly to the dynamic translation in the Deutsch-Schiffman Smalltalk system [Deutsch and Schiffman, 1984]. The source program is translated to a simple byte-coded intermediate representation and later when a method is invoked, the compiler is called and the resultant object code is cached for future use. Chambers and Ungar claimed in [Chambers and Ungar, 1991] that SELF compiles about as fast as an optimised C compiler and runs at over half the speed of optimised C. Similarly, Franz's compiler to the Oberon language performs load-time target machine code generation with a modest overhead and reasonable code quality [Franz, 1995].

In the following paragraphs, the most promising routes for language implementation, cases (2) and (3), will be compared; case (1) is too inefficient and case (4) does not accommodate the needs to cope with longevity, unless the source code is saved and compiled every time the support hardware architecture changes.

2.4.1 Compiler Complexity

Having an intermediate abstract machine *I*, or an intermediate language *IL*, greatly simplifies the task of writing a compiler for that language as it divides the total compilation task into two more manageable tasks:

1. the front-end translation from source to *IL*, and
2. code generation from *IL* to machine code for *M* or interpretation of code for *I*.

In principle, each of these steps may be further divided, e.g. by separating optimisations. There is a trade-off between the cost of interfacing the steps and the improved manageability. Persistence, if used in the implementation, shifts this trade-off reducing the interface cost; because structural information is retained between phases, extra steps can be contemplated.

In terms of the complexity of the programs to be written and maintained, it is then better to have an intermediate machine (as in case (2)) or an intermediate representation (case(3)). The complexity of the interpreter program and of the code generation program are in the same order of magnitude. The interpreter is simpler as it doesn't need to deal directly with register allocation or scheduling of machine instructions.

2.4.2 Quality of Code

In terms of the quality of code produced, code generation wins, but the code that the interpreter executes for each instruction of the abstract machine may be very similar to what expansion to machine code would produce. Implementations following code generation as in case (3) or compilation to machine code as in case (4), allow for efficient execution of high-level language programs since they are transformed to a form which can be interpreted directly by hardware. In terms of **efficiency**, interpretation and code generation must be compared with respect to space efficiency and time efficiency. It may be argued that interpretation produces more compact code if the operand locations are implicit and thus is better in space efficiency. Code generation wins over interpretation in terms of runtime program efficiency because of better machine resources usage, e.g. caches, registers and the full range of addressing modes available. In order to achieve good execution speed figures, code generators must be well designed to reduce memory accesses through good register allocation policies and to choose the right sequence of instructions and addressing modes from the huge repertoire of CISC machines; even if instruction selection is easier, as in RISC machines, there are still the non-trivial *instruction scheduling* problem to be solved in order to minimise the *stalls* in pipeline machines [Patterson and Hennessy, 1990].

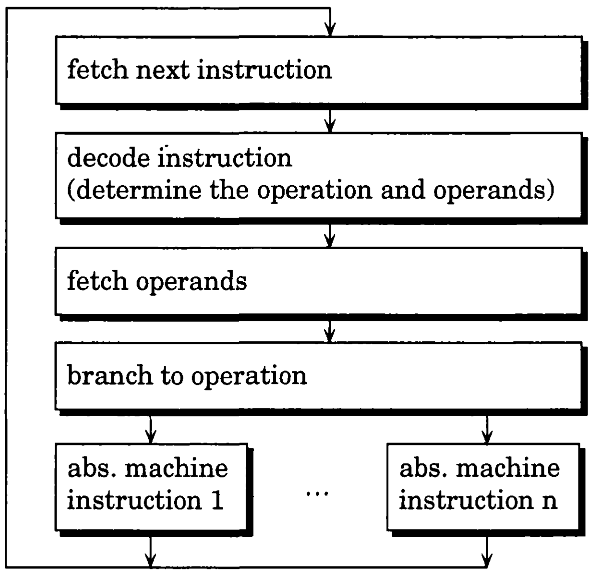

fetch next instruction
decode instruction (determine the operation and operands)
fetch operands
branch to operation
abs. machine instruction 1
...
abs. machine instruction n

Figure 2.5: Program interpretation

An important difference between this two approaches is the fact that in case (2) no low-level optimisations can be done (or they are very difficult [Leroy, 1990]) and in case (3) peephole optimisations, register allocation and instruction scheduling is done, leading to more efficient program execution.

2.4.3 Interpretative Overhead

Despite the fact that the quality of code is similar, there are still reasons to the so called "interpretative overhead" [Leroy, 1990]. As the interpreter is written in a high-level language, additional encodings may be necessary to manipulate bytecode objects, e.g. it may be difficult to store abstract machine registers in actual registers of the hardware machine[6]. Also additional computations may be needed in interpretation as the abstract machine instructions are generic, i.e. they must cope with all possible values of its operands. In some situations the code generator can use more specialised instructions available in the host machine. One example of such a situation occurs with the *add* instruction with the second operand *1*; this operation can be substituted by a cheaper increment operation.

The most important **overhead in interpretation** has to do with fetching and decoding the abstract machine instructions from the bytecode stream (see Figure 2.5); instructions are executed in hardware or microcode in the case when machine code is generated and thus take less time in these situations. Several techniques were developed to try to minimise the cost of instruction fetching, decoding and branching to the part of the interpreter that executes the instruction. In order to reduce the overhead in decoding opcodes, an implicit location for operands can be used instead of addressing modes. This is the reason why stack-based

[6]This disadvantage may be minimised by using a language with full access to the internals of the host machine, such as C.

machines are used in the interpretation of bytecode instead of register-based machines. Another improvement called *threaded* code interpretation[7] can be made by replacing the opcode used to index a table with the address of the routines that implement the operations by the address itself, and so speed up this task. This stack-based abstract machine introduces inefficiencies as values end-up coming on and off the stack many times, leading to extra memory references.

The branching to the runtime routine, which implements the abstract machine instructions, implies runtime execution of target machine code with worst locality when compared with the code obtained by code generation. This may in turn imply more cache misses leading to more inefficient execution.

2.4.4 Portability

In terms of portability, i.e. how easy is it to have a new implementation of the high-level language running in a new target processor, the use of an abstract machine as in case (2) is obviously a good solution. As the abstract machine does not depend on the hardware, the only thing that needs to be done is to recompile the program implementing the interpreter for the new architecture. If a language like C is used to implement the interpreter and it has been carefully written to plan for portability, only a few changes will be needed. On the other hand, retargetting a code generator to a new architecture may involve a lot of effort in order to have good quality code. Code generation is normally machine specific and involves skills based on knowledge of the hardware implementation; as these skills are not widely available, if case (3) is chosen then the architecture has to be planned to simplify code generation.

2.4.5 Conclusions

Interpretation introduces several inefficiencies in the translation process: extra memory references, cache misses, less available machine registers and little changes for optimisation of separated instructions. On the other hand, code generation may improve efficiency but it is a more complex task.

A case for introducing an additional phase in the compilation process of case (3) arises when the costs of achieving high-quality code generators for each architecture is considered. As the goals of this work include achieving generality (as stated in Section 2.3.1) and longevity (see Section 2.3.4), code generators for all platforms that need ever to be supported would need to be written. The introduction of this other phase will reduce the highly skilled labour involved and will avoid repeating work. The intermediate language level *IL* may be divided into two separate internal program representations: one at a higher level intended to make easier the task of front-ends and to allow machine independent optimisations and the second and support for longevity, intended to support for longevity and simplify the construction of code generators which ensure efficient generation of target machine code.

By code generating to a target like C (or more usually a subset of C), the same portability as in the case where interpretation is used may be achieved and it is easier to obtain target

[7]Threaded code was used the first time in the implementation of the Forth language.

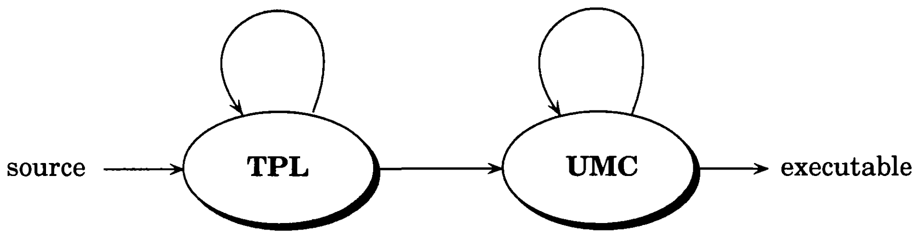


Figure 2.6: An Architecture for Compilation

machine code for different platforms. As always in engineering, there are corresponding costs. For example, in the case where C is used as a **portable assembly language** then exact formats of generated code are not known and consequently the store management system will be compromised unless the information it requires can be rediscovered.

In terms of **longevity**, i.e. keeping the semantics of data for longer periods, a portable, well designed abstract machine interpreter, *I*, would be equivalent to generating code for each new architecture from a sufficiently annotated *IL*. The instructions in the abstract machine would then be more complex and that would then impact on the efficiency of the interpreter.

Both longevity and efficiency reasons seem to suggest that code generation approaches should at least be investigated.

2.5 An Architecture for Compilation

To meet the goals of longevity and efficiency, a three-stage architecture is proposed (as sketched in Figure 2.6). The loops in this figure illustrate the two places where optimisations can be performed in the path from source to executable code. This work is specially oriented to the investigation of what can be done at the intermediate language level TPL. The second level of optimisations, at UMC level, is well studied and implemented for imperative languages[8].

This architecture with two intermediate representations is intended to improve efficiency by introducing room for optimisations in the compilation path. The optimisations at the TPL level are intended to be both language and machine independent. On the other hand, they are also intended to allow easy generation of machine code and at the same time, enable machine dependent optimisations. Ultimately, the intermediate representation should have an abstract model that has clean semantic properties, in order to allow reasoning and proof of the correctness of transformations. It is first necessary to establish the correct level of abstraction at which to place TPL (as it divides two classes of optimisations) before investing effort in its precise definition. This is an engineering issue.

For TPL, there are examples of languages or abstract machines[9] which exist at different levels: the persistent abstract machine (PAM) [Brown *et al.*, 1988, Connor *et al.*, 1989], the functional abstract machine (FAM) [Cardelli, 1983], the P-Quest machine (PQM) [Matthes *et al.*, 1992] (derived from the QUEST machine), Appel's continuation-passing style, closure conversion style (CPS) [Appel, 1992, Appel, 1990], the spineless tagless G-machine (STG)

[8]E.g., backpatching, jump size optimisation, etc. [Aho *et al.*, 1986].

[9]As a language L implicitly defines a virtual machine, namely the virtual machine whose language is L.

[Peyton-Jones, 1992], the CASE [Davie and McNally, 1990a] and PCASE machines [Davie and McNally, 1992, McNally, 1993], Tycoon's TML [Gawecki and Matthes, 1994] and the Persistent Hierarchical Abstract Machine (PHAM) used to support the Fibonacci language [Albano *et al.*, 1995] (derived from the FAM). These possible approaches will be discussed in Chapter 4.

In Figure 2.6, UMC stands for Universal Machine Code and is a low level language close to assembly language. It could take the form of TDF [Defence Research Agency, 1991], RTL [Stallman, 1992], or a subset of the C programming language. The use of C, or a subset of C, was the approach followed in the work of Tarditti *et al.* in SML/NJ to C (SML2C) [Tarditi *et al.*, 1992], Bartlett in scheme-to-C [Bartlett, 1989], the work of Peyton-Jones *et al.* in the Glasgow Haskell compiler [Peyton-Jones, 1992], Feldman *et al.* with Fortran to C [Feldman *et al.*, 1990], Gillespie in Pascal to C (p2c) [Gillespie, 1989], in the implementation of Modula-3 [Chase, 1990], and others [Serrano, 1994]. In general, C has been useful to prototype extensions to itself (C++) or as an intermediate representation of other languages: Ada, Cedar, Eiffel, Fortran, Modula-3, Pascal, Sather, Scheme, SML and Objective-C. These existing UMCs are discussed in Chapter 5.

Any method of initial analysis, optimisation and code generation is in principle possible. This makes experimental verification of the architecture difficult because it may be sensitive to these choices.

2.5.1 The Level of TPL

The level at which the intermediate language TPL will reside is debatable as it may be close to the hardware machine (e.g. SPARC processor) or to the level of the high-level language (e.g. Napier88).

The language features that are intended to be supported lead to constraints in TPL design (see Chapter 3 for a complete description). First-class functions combined with nested scope are better supported by an internal representation which exhibits these characteristics as it frees the front-end from closure analysis (providing access for free-variables) and allows for optimisations and possibly the use of different strategies to represent closures. Polymorphism imposes constraints on the formats, as different values must be accommodated at the same location. To support orthogonal persistence a Stable Store may be used and a *stabilise* operation must be provided together with dynamic binding. To support reflection (see Section 3.3.3, dynamic linking must be achieved together with dynamic type-checking in order to incorporate the results of the execution of the generated code.

Being at a higher-level, very different from machine level, imposes constraints on efficiency. Efficiency will be dealt with the use of carefully designed optimisations and code generation. An internal representation only with efficiency in mind could be much closer to the machine. The relaxation of some of the high-level language constraints would permit more efficient support technologies. For example, to support only a class of persistent safe languages without first-class functions or polymorphism (call it Safe Persistent C) would allow significant efficiency gains.

Generality also leads to constraints in TPL design. The class of high-level languages aimed to be supported exhibit different formats for values, different polymorphisms, different equalities. TPL must be designed to have the minimum functionality that will support all services required by these languages.

2.6 Conclusions

In order to achieve longer-term persistence together with efficiency, an architecture for compilation was proposed. This code generation language processing framework has the advantages referred to in the previous section: less complexity than direct machine code generation, easy portability of code to a new architecture, generation of good quality code leading to more efficient runtime execution of programs and support for an architecture independent representation of programs suitable for longer-term persistence.

The intermediate representation to be generated by high-level language front-ends, called TPL (for Target Persistent Language), must be designed to:

1. be independent of both the high-level language and of the underlying hardware platform;
2. provide all the constructs needed to support the high-level languages (as identified in Sections 3.5, 3.6 and 3.7);
3. capture sufficient information for the underlying support system (as established in Section 3.8);
4. be easily extensible as for longer-term persistence new types may be introduced in the language while retaining the existing types;
5. be suitable to support high-level machine independent optimisations like inlining and constant folding;
6. be suitable for data-flow and control-flow analysis; and
7. allow easy code generation of the low-level language.

Machine independent optimisations manipulate the TPL internal representation and, as this representation is intended to constitute the support for longer-term persistence, will be done once and for all. Having done those optimisations, the low-level representation UMC (for Universal Machine Code) may be generated. This program representation may be later optimised on demand and this way the application is "tuned" to a new architecture. The UMC program representation must be designed to be:

- close enough to the hardware machine to allow easy code generation;
- convenient to perform target specific optimisations like register allocation, instruction scheduling or peephole optimisations;
- a generic store interface language encapsulating the functionality of different implementations of persistent object stores;

- a convenient representation for procedure closures.

The three-stage architecture with two intermediate forms is anticipated as feasible because persistence allows efficient and reliable communication between stages. Nevertheless, there will be costs in language processing time which we anticipate will be recouped in terms of confidence regarding longevity and in PAS operational efficiency.

The following chapters will identify the constructs needed to support the class of high-level languages of interest and will search for suitable representations at the two internal levels of the proposed architecture.

2.7 Thesis Statement

An architecture with a high-level intermediate representation is appropriate in the construction of supporting technology for persistence, it can enable high-level optimisations and easy code generation and it can then effectively support **persistent reflective higher-order polymorphic languages**, ensuring longevity, safety and persistence.

This is demonstrated by presenting an initial design of the architecture and the intermediate language and then identifying and validating its crucial features by prototyping.

Chapter 3

Persistent Programming Languages

The persistent higher-order reflective languages are recognised as facilitating the task of incrementally building and maintaining persistent application systems. This chapter describes the characteristics of the source languages intended to be supported by the proposed architecture and identifies the constructs which must be included in the internal language. The requirements for the lower layers of the architecture in order to support persistence and stability are also identified.

3.1 Introduction

The class of languages displaying orthogonal persistence, as was introduced in Section 1.3.3, was identified as a promising technology to implement and maintain persistent application systems. Two such languages are PS-algol and Napier88.

The language **PS-algol** proved that it is feasible to implement the orthogonal persistence abstraction [Atkinson *et al.*, 1983c, Persistent Programming Research Group, 1987]. PS-algol respects the principles of persistence and the principle of data type completeness referred to in Section 1.3.3. PS-algol supports a persistent transactional stable store [Atkinson *et al.*, 1983b] and an associative store (a table implemented initially as a B-tree). It provided an experimental platform on which following developments took place: pictures [Morrison, 1982], images [Morrison *et al.*, 1986], higher-order persistent procedures [Atkinson and Morrison, 1985], reflection [Cooper *et al.*, 1987], concurrency [Krablin, 1988, Krablin, 1987] and distribution [Wai, 1989].

The language **Napier88** was conceived to carry forward the best features of PS-algol and also to allow experimentation with new type systems for protection and description [Atkinson and Morrison, 1988, Atkinson and Morrison, 1987, Atkinson and Morrison, 1990]. Napier88

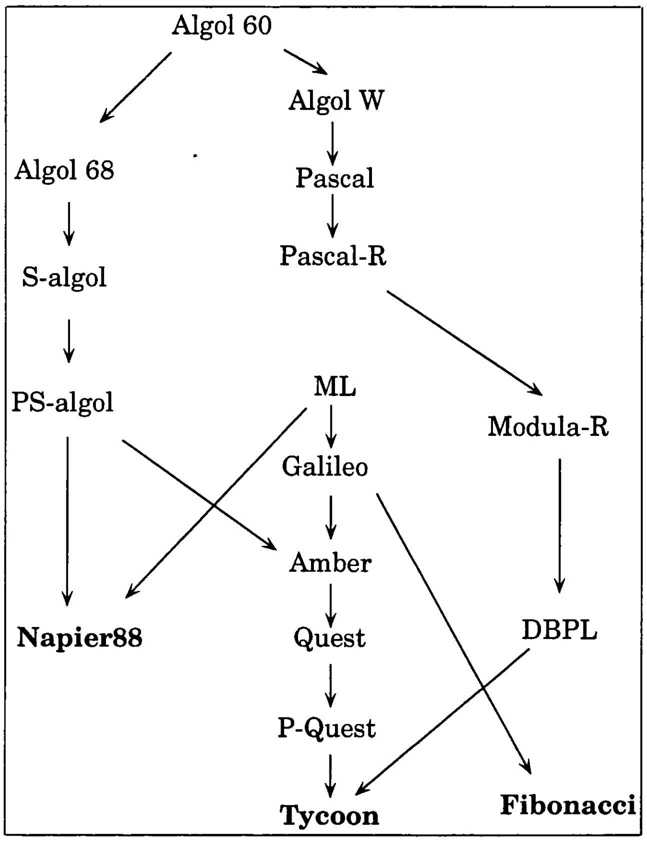


Figure 3.1: PHOL Family Tree

is an orthogonal persistent language in the Algol tradition that obeys the principles of correspondence, abstraction and type completeness [Morrison *et al.*, 1989, Morrison *et al.*, 1994], as was the case of PS-algol.

3.2 Persistent Higher-order Reflective Languages

Figure 3.1 displays the family tree for the subset of persistent languages with higher-order functions, polymorphism and reflection. These languages are called **persistent higher-order reflective languages** (PHOL) and examples of such languages are: Napier88, Fibonacci and TL (the programming language for the Tycoon system). Orthogonal persistence was introduced in Section 1.3.3 and the other language features will be described in the following sections with a reference to its relevance in building and maintaining PAS.

3.3 Persistence

Within a persistent programming language, there is a need to have a uniform means to handle persistent data in the same way as temporary data. There are different models of persistence: persistence by reachability as in PS-algol or Napier88, persistence by type as in ONTOS or allocation-based persistence (also called explicit persistence) as in ObjectStore.

```
let a= 10
let p= proc( i: int -> proc( -> int ) )
     begin
       let b:= 0
       proc( -> int )
       begin
          b:= a*i+b
          b
       end
     end
let q= p(2)
let n= q()
```

Figure 3.2: Escaping Procedure

Another way persistent systems may be created is by embedding the runtime system in a persistent address space [Vaughan, 1994].

In **persistence by type** only certain types of data may become persistent. In the Amber language [Cardelli, 1986] a special data type, *dynamic*, can be used for persistent data together with *export* and *import* commands to save and restore data values to and from files. In **allocation-based persistence** there is an area of storage chosen to store persistent objects and a call to a function (e.g., *new(obj)*) which places persistent objects in that area. In persistent systems built around languages with inheritance (like C++), a data object inherits from a certain class the methods which allow it to move to and from stable storage. For example in E [Richardson, 1989], "an object is persistent only if it is created as such, either by being declared a *persistent* variable or by being created within a persistent collection" [Richardson *et al.*, 1993].

In **persistence by reachability** a data item is persistent and will outlive a program execution, whenever it is in the transitive closure of one or more distinguished roots of persistence (i.e., it is reachable from those roots). In PS-algol, persistence is identified by reachability from distinguished roots in a persistence store and is modelled by a set of standard procedures (e.g. *create.database*, *open.database*, *commit*). Persistence determined by reachability frees the programmers from thinking about persistence but may imply more overhead in determining which objects are persistent and in object copying to and from main memory, when compared with the other models. Its most significant gain is that it guarantees referential integrity and ensures there are no dangling persistent references which might otherwise corrupt the store. Because of these characteristics, the model of persistence by reachability was chosen in this thesis to decide which objects are persistent.

3.3.1 Higher-order Procedures

A procedure is called higher-order if either its arguments or its results are themselves procedures. In the example presented in Figure 3.2 written in Napier88 syntax, procedure *p* when executed generates another procedure as a result. Several programming languages treat procedures as "first-class citizens". In these languages, procedures are "first-class values" of

```
rec type List[ T ] is variant( empty: null; full: Cell[ T ] )
&       Cell[ T ] is structure( hd: T; tl: List[ T ] )

let l_length:= proc[T]( l: List[T] -> int )
begin
   let noe:= 0
   while l isnt empty do
   begin
      noe:= noe + 1
      l:= l'full( tl )
   end
   noe
end
```

Figure 3.3: Parametric Polymorphic Procedure

the language, that is, procedures have the same "civil rights" as any other data object in the language, such as being:

- assignable;
- the result of expressions or of other procedures; and
- elements of structures or vectors.

The first language with first-class procedures (called functions) was LISP [McCarthy and others, 1962] and applicative languages developed around this concept. Functional languages, e.g. ML [Milner, 1983], and some persistent languages, e.g. Napier88 [Morrison *et al.*, 1989] also have first-class procedures. In [Atkinson and Morrison, 1985] the authors show how higher-order procedures and orthogonal persistence may be used to implement abstract data types (ADT), modules, separate compilation, views, and data protection. First-class procedures and orthogonal persistence can constitute the basis for incremental construction, and in the same way as modules in other languages (e.g. Modula-2 [Wirth, 1983]), can serve as the unit of specification, compilation, testing and assembly.

3.3.2 Polymorphism

A language is monomorphic if every value and variable can be interpreted to be of one and only one type, as in Pascal [Jensen and Wirth, 1975]). Polymorphic types may be defined as types whose operations are applicable to operands of more than one type. From all forms of polymorphism described in [Cardelli and Wegner, 1985], the two forms of universal or true polymorphism, parametric and inclusion, are described here. The end goal is for TPL to have the potential to support both forms of polymorphism.

In **parametric universal polymorphism**, a procedure will work on an infinite number of types. A parametric polymorphic procedure has one or more type parameters (possibly implicit) which determine the type of the arguments or result. Procedure *l_length* of Figure 3.3 will work for any type T on values of type $List[T]$ giving back the length of the list passed as argument. As it is the case with this example, this form of polymorphism works when values have a common structure.

In **inclusion polymorphism** a procedure will work on a set of types related by an implicit or explicit ordering. Inclusion polymorphism is introduced into the language by subtyping [Cardelli and Wegner, 1985]. A value of a subtype can be accepted anywhere a value of a supertype is expected. A type T_1 is a subtype of type T_2 ($T_1 \sqsubseteq T_2$) if all operations allowed on T_2 are also allowed on T_1. This subtype relation defines a partial ordering[1] of the types which therefore form a lattice.

A special case of inclusion polymorphism, which is of great interest, is subtyping over record types. This form of polymorphism can be found in Object-Oriented Languages and in the DBPL Galileo [Albano *et al.*, 1985]. A record type TR_1 is defined to be a subtype of type TR_2 ($TR_1 \sqsubseteq TR_2$) if all the fields of TR_2 are also present in TR_1[2]. Moreover, the common fields satisfy the subtyping relation. Evolution in the Object-Oriented paradigm relies on subtyping over records achieved by the mechanism of inheritance, whereby a new type is defined by extending (explicitly or implicitly) an already defined type. Inheritance models *is-a* relationships between entities. Objects are incrementally defined and descendants inherit properties — data and procedures fields — from their ancestors following a single or multiple inheritance chain. Subtypes may have their own properties added and may redefine ancestor properties (overriding). When a property is redefined its name is associated with the most specialised type to which the object belongs (late binding).

3.3.3 Reflection

"A computational system is said to be reflective if it incorporates causally connected data representing (aspects of) itself" [Maes, 1987]. Reflection is the process of reasoning about or acting upon oneself and changing behaviour in the course of one evaluation. It can be achieved by changing the way that programs are evaluated in the systems or by changing a program's own structures. In **behavioural reflection** a program can alter its own meaning by manipulating its own evaluator[3], or as in object-oriented languages, to provide a meta-object for every object in the system which specifies aspects of its behaviour. An object can modify aspects of its behaviour (e.g. how it inherits from super-classes) by sending messages to the meta-object. In **linguistic reflection** programs can change themselves directly, e.g. by generating new data structures to be interpreted or new code to be executed [Kirby, 1993].

Programs can change themselves directly at compile-time or at runtime. PS-algol and Napier88, both support runtime linguistic reflection by allowing for programs to be constructed, compiled and integrated into the current computation [Stemple *et al.*, 1992]. Figure 3.4 presents an example of linguistic reflection coded in Napier88. This program constructs the source for another Napier88 program with a procedure declaration, calls the compiler passing it the source and executes the result of that compilation. Normally, the program to be compiled at runtime depends on input from the user, or in other cases, the algorithm it

[1]It is antisymmetric as $T_2 \sqsubseteq T_1$ does not hold.
[2]And maybe some more fields.
[3]E.g., an interpreter, leading to different actions in the process of interpretation.

```
!* put names in scope
use PS() with comp, IO: env in
use IO with writeString: proc(string); readString(-> string) in
use comp with compile: proc(string -> any) in
begin
   !* generate a program as a string with user input
   writeString("Please write an integer expression involving i 'n")
   let theProgram = "proc(i: int -> int); " ++ readString()
   !* compile the program
   let theResult = compile(theProgram)
   !* bind the variable newProc to the resultant procedure
   project theResult as R onto
      proc(int -> int): let newProc= R()
      string         : writeString("Compilation errors: " ++ R)
   default: {} !* it should never happen
   !* now, newProc may be used in this program
end
```

Figure 3.4: Runtime Linguistic Reflection in Napier88

performs depends on types know only at runtime; in this situation, it is generated after these types are known.

3.3.4 Conclusions

Higher-order procedures and polymorphism allow the programmer to express algorithms in a more abstract and general way and languages exhibiting these features allow greater reuse. Both the different forms of polymorphism and reflective programming allow abstraction over details and the construction of general programs and thus they promote reuse. Polymorphism can be used when a computation does not depend on the types of the operands, allowing the type to be abstracted as in parametric polymorphism, or when it depends only partially on the types, as in inclusion polymorphism. Using type-safe linguistic reflection, programs can be written that depend to an arbitrary extent on the types of the data they manipulate. Examples in which reflection is useful include [Cooper *et al.*, 1987, Kirby, 1993]: natural join in a language without built-in support for relations, traversal functions over recursive data types, implementing data models and implementing browsers. In an evolving system where new values and types may be incrementally created without the need to recompile or relink existing programs, these programs need to operate over values whose type is not known in advance. Reflective systems provide support for applications able to adapt themselves avoiding the need to reimplement applications as data evolves.

3.4 Language Features to be Supported

The architecture to be designed and demonstrated is aimed at supporting strongly typed languages which display orthogonal persistence, first-class procedures, polymorphism and reflection. The languages to be supported will have some of the following features:

- orthogonal persistence;
- reflection;
- parametric polymorphism;
- inclusion polymorphism;
- first-class procedures;
- graphical data types;
- bulk data types;
- collections of bindings; and
- incremental binding.

Therefore these features must be supported by the intermediate language and the specific challenges they present are considered in this thesis. Of course traditional features such as: aggregate types (vectors, records or tuples[4], images), union types[5] and type *any*[6] will be needed, but they do not pose major problems. Similarly, a variety of control structures will appear: if-then-else, case, repeat-until, while, loop and for, but they are, again, straightforward. Versions, scoping structures (such as blocks, modules, objects and classes, ADT and procedures), exceptions and various binding models may also be required. Concurrency (threads and semaphores) and transaction models may also vary between languages. This thesis does not consider all of these, but assumes they are orthogonal to the design issues that are considered.

As this work is concerned with the design of an intermediate language, it must be emphasised that the syntactic issues are not important as they will be dealt with by the front-end. Given first-class higher-order persistent procedures it is then possible to provide modules and abstract data types [Atkinson and Morrison, 1985]. Therefore it suffices to show that first-class higher-order procedures can be supported, to show that these derivable constructs can be supported. Initially, the focus will be on supporting first-class higher-order procedures. Later it may be appropriate to review whether there are efficiency gains in supporting these derived constructs directly.

Among the data values to be supported, there are several that need to be allocated in a dynamically managed area of storage. For example in data complete languages, vectors enjoy the same civil rights as any other data object including assignment or being fields of other vectors. In a language with variable size vectors it is impossible to copy the vectors on assignment as the space needed in the stack cannot be predicted [Davie and Morrison, 1981]. Therefore the vector is represented as a pointer in the stack with its elements in a heap object. In order to approximate a conceptually unbound store, garbage collection is necessary. Garbage collection will be discussed in Section 3.8.

The following sections will describe and compare different ways of supporting the language features of interest to this work.

[4]Labelled Cartesian products.
[5]Labelled disjoint sums or discriminated unions.
[6]Infinite union of all types.

3.5 Constructs for First-class Procedures

The **closure of a procedure** includes all the information necessary to execute the procedure correctly: code plus its environment consisting of local and free-variables. Traditionally, procedure local variables and intermediate values of expressions are stored in an activation record allocated in a stack at procedure's call time. In the programming languages of interest to this work, a procedure can be returned as a result (or stored in structures) and a non-global free variable may be used after it has left the scope. Therefore, a stack to store activation records and a display to access non-local variables is insufficient because the lifetime of variables may exceed the activation record of the procedure where they were declared. In the example of Figure 3.2, the variable *b* is used in line 11 when *p* is applied and then in the following line when *q* is applied but this time after leaving the scope.

In block-structured languages (e.g. Algol-like languages) identifiers are declared inside a block and blocks can be nested[7]. The scope of an identifier is the rest of the block where it is declared unless the same name is used in an inner block in the declaration of a new identifier. In this situation, the outer identifier is hidden by the other declaration for the rest of the inner block. Languages exhibit **true block scope** if procedures can be freely declared in any scope so they can be nested and contain free-variables that are local variables of enclosing procedures. It is the combination of first-class procedures with true block scope which makes the support of first-class procedures more difficult to deal with; languages like C [Kernighan and Ritchie, 1988], where functions cannot be defined inside other functions, greatly simplify this problem. With C, a free-variable is a global variable, and is thus visible in all situations.

A stack may be used to support procedure activations in languages like Pascal where nested procedures can only be passed as parameters, but never be results of functions, blocks or expressions,but to support first-class procedures, some form of block retention is therefore needed [Johnston, 1971]. There are different techniques to achieve this.

In the PAM a block retention mechanism, as represented in Figure 3.5, is used; that figure corresponds to the code of Figure 3.2. This mechanism consists on having, at runtime, heap objects (frames) for every block of scope and pointers to maintain access to procedure free-variables [Connor *et al.*, 1989]. The procedure literal value returned by *p* is copied to the local variable *q*. The corresponding closure includes a pointer to the code and a pointer to the environment. Free-variable *b* is reached through this second pointer. The same mechanism is also used to implement polymorphism and ADT [Morrison *et al.*, 1991]. One disadvantage of this solution is that to retain free-variables all the frame is retained, leading to inefficient space usage. Using this method to access non-locals, a static chain is maintained. The usage of a static chain implies that the access to non-locals must contain a block level (or a block

[7]A *variable* is a data object that contains a value and has a *name* which, when interpreted is some context, constitute an *identifier*. It is possible for the same name to serve as identifier for two variables when in different contexts. An identifier is associated with the entity it denotes by a *binding*. The value of a variable may be used and may be updated by *assignment* several times in a program. These updateable variables are different from the mathematical variables which stand for a fixed but unknown value (the same meaning of functional programming variables). There are other data objects which cannot change value, called *constants*. A binding is, therefore, a 4-tuple of the form:

`{name,value,type,constancy}`

The term variable is used in the text to include both updateable and constant value data objects, when the distinction is not important.

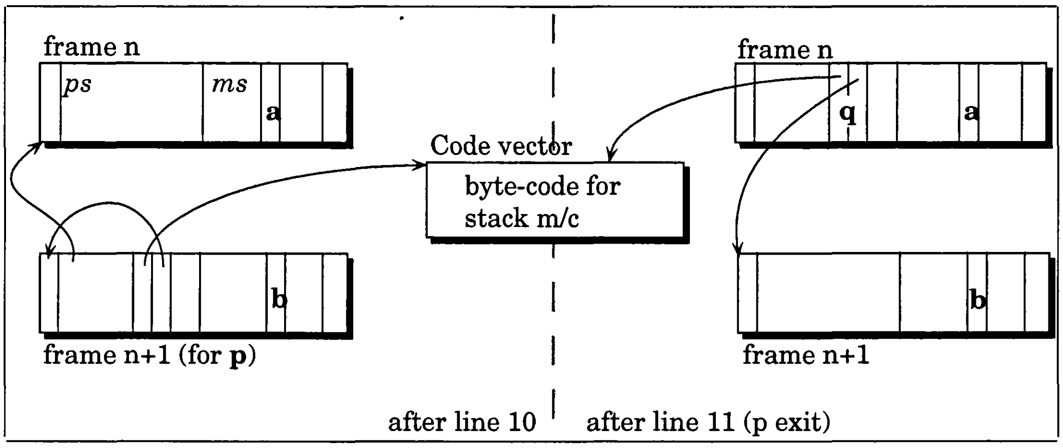


Figure 3.5: Block Retention Mechanism in the PAM

difference) plus an offset and incurs in the overhead of maintaining the chain in procedure entry and exit.

Another way of implementing higher-order procedures is based on the technique described in [Davie, 1979] and consists of having direct references to, or copies of, all the free-variables in the record representing the closure of the procedure. This way, activation records may be stored in the stack and closures are simply heap items. The construction of the closure and loading (access to) all its free-variables takes place only once and will be shared by all its invocations. In the PAM, an object must be created for each procedure invocation to hold the activation record. In the persistent environment, the trade off, time which takes to create the closure, versus, time which takes to call the procedure, is more favourable to this technique, as a persistent closure is supposed to be called by other programs. This method was also used by Cardelli in the FAM abstract machine [Cardelli, 1983] conceived for the implementation of an ML compiler. Several other abstract machines follow this approach; among those are the PCASE and PQM abstract machines and also in the PAMCASE designed to support the language Napier88 [Cutts *et al.*, 1997]. The spineless tagless G-machine (STG), which is intended to support the compilation of strongly-typed, higher-order, non-strict, purely functional languages such as LML or Haskell [Peyton-Jones, 1992], also uses the same approach. Appel's work in implementing the SML/NJ system [Appel and MacQueen, 1987, Appel and MacQueen, 1991], comprising a compiler and runtime environment for the standard ML language, is another example of the use of closures similar to Davie's via a technique called closure conversion [Appel and Jim, 1989]. This work will be referred to as CPS (for continuation-passing style) and is discussed further in Section 4.3.

Therefore, the intermediate language has to accommodate at least one means of representing closures. The particular challenges are space efficient representations balanced against the costs of calling and the identification of pointers, particularly in free-variables on stacks and implicit in code (e.g. calls of other closures).

3.6 Constructs for Polymorphism

This section surveys the different ways of supporting both forms of polymorphism introduced in Section 3.3.2.

3.6.1 Parametric Universal Polymorphism

The most common way of implementing parametric universal polymorphism, followed for example in FAM, PCASE, PQM and CPS, is to make every value of the same size when they are assigned to variables and passed as parameters. Some means of identifying which values are pointers is then needed and either the low-order bit is used as a tag, or pointers are restricted to certain locations. This is called a **uniform polymorphism** implementation as all data has a uniform representation and the same code is executed at machine level, regardless of the type of data being manipulated. One of the advantages of this approach is that it is easy to implement since the compiler generates uniform code. The other advantage is that it is efficient in space usage as there is only one copy of the machine code for each of the polymorphic forms. One disadvantage is that this uniform representation precludes efficient representation of all values as even the monomorphic values pay the price of being boxed into a heap object. The creation of new data involves the creation of a heap object and, moreover, all data have to be addressed indirectly through the pointer to the heap object. The tagging scheme also makes arithmetic operations more expensive. This technique has been successfully used to implement functional languages where data are transient but it is less suitable to implement persistent languages as the extra indirection may involve an object fault in the access to the object store [Connor, 1991].

Another way of implementing uniform polymorphism is by using the retention mechanism needed to implement first-class procedures — it is called an **ad hoc** implementation of polymorphism in [Morrison *et al.*, 1991]. With this method, the type parameters provided when the procedure is specialised reside in the closure of the procedure and a conversion of arguments to a uniform representation and of the result back from that uniform representation is performed during procedure activation and return. This method works well for polymorphic procedure values and for implementations that used non-uniform formats. Recently, the same method has been used to support ML in the work described in [Tarditi *et al.*, 1996].

Apart from uniform polymorphism implementations, there are also tagged and textual implementations. In the **tagged implementation** of polymorphism every object is tagged with its type and the code examines this tag to determine how to execute. Machine code is uniform but store representations are non-uniform and so the code must be aware of the different formats. In the **textual implementation** of polymorphism different machine code may be generated for each instantiation type, leading to non-uniform machine code. For programming languages with first-class polymorphic values or orthogonal persistence this technique is unsuitable because the compiler is unable to perform any statical analysis of the uses of a procedure [Connor, 1991][8]. However, different versions of optimal instruction

[8]Either all calls to polymorphic procedures must dynamically invoke the compiler or all possible procedures must be available.

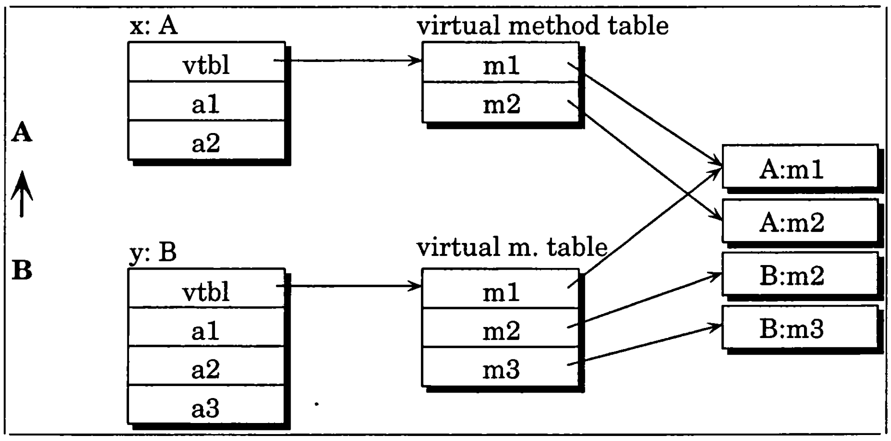


Figure 3.6: Implementation of Subtype Inheritance on Static Typed Languages

sequences for different instantiation types may still be used and constitute an optimisation similar to the work in SELF described in [Hölzle *et al.*, 1991].

Because the ad hoc implementation works well with persistent polymorphic procedure values and is more efficient that both uniform polymorphism and tagged polymorphism, it is anticipated it will constitute the choice of TPL users. If front-ends are permitted to use uniform representations, both for uniform and for ad hoc implementations, then a means of accommodating all possible TPL values is needed.

3.6.2 Inclusion Polymorphism

Implementations of inclusion polymorphism in Object-Oriented Languages depend on the amount of information the compiler has available to construct the structures to support inheritance and late binding. Object-Oriented Languages may be statically typed as in C++ ([Stroustrup, 1986]) and Eiffel ([Meyer, 1988]) or dynamically typed as in Smalltalk ([Goldberg and Robson, 1983]) and Objective-C [Cox, 1984]. Data attributes in an object can be allocated at fixed locations relative to the beginning of the object. Objects belonging to more specialised types can only extend the memory layout representation of objects belonging to their ancestor's type and the offsets of common data fields are equal in the two of them. Procedure fields (methods) could be implemented the same way. This scheme would be time efficient to access a method but not space efficient, as methods are common to all instances of objects from one type and would be replicated. It would be inefficient as well because it would take longer to initialise every new object. The usual solution is to have a single table for each type with pointers to all procedures and call procedures via one indirection through this table. This table is called the **virtual methods table**. Figure 3.6 represents the data structures that may support subtype inheritance for two types A and B, where $A \sqsubseteq B$. Objects of type B inherit attributes a_1, a_2 and m_1; override m_2 and extend A with the new attributes a_3 and m_3. All this structures can be decided statically by the compiler.

In dynamically typed languages the structures to support late binding can only be decided at runtime. Associated with each type is a table, known as the **dispatch table**, with entries with the name and code for all procedures that it implements. Each dispatch table also has

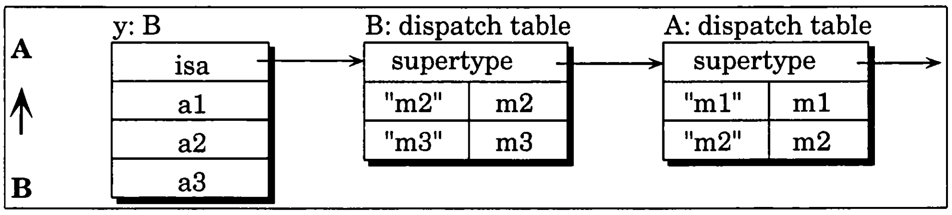


Figure 3.7: Implementation of Subtype Inheritance on Dynamic Typed Languages

a pointer to the table of its ancestor type and each object has a pointer to the corresponding dispatch table as represented in Figure 3.7. To find the code to apply it is then necessary to start at the dispatch table of the type pointed by the object that invokes the procedure and search for a match in the name of the procedure, following the pointers to its ancestors if not found. If the root class is reached without a match, a runtime error will be generated. Although economical in space, this scheme is inefficient in time, which is proportional to the depth of the inheritance. A system wide cache is usually used to speed-up the process of binding code to a procedure call.

Multiple inheritance complicates the access to fields and the late binding of procedures by the usage of virtual tables or dispatch tables. An implementation of multiple inheritance for the language C++ is sketched in [Ellis and Stroustrup, 1990].

Providing records in the intermediate representation enables front-ends of statically typed languages to construct the structures to support inclusion polymorphism. Accessing data and procedures is then done by field addressing (i.e. retrieving the value associated with a label in a record). Alternatively, it may be decided to provide for more specialised and low-level constructs and operations. In this case, the study done by Connor in [Connor, 1991] may help; uniform, tagged and partially-tagged field-addressing implementations are described. For front-ends of dynamically typed languages, it would be helpful to to have a type *Table* with the usual operations of *insert*, *test*, *delete* and *scan*, which could be used to map names to methods.

3.7 Constructs for Reflection

From the different kinds of reflection introduced in Section 3.3.3, our work focuses on type-safe linguistic runtime reflection. Compile-time reflection through the use of macros (e.g. in Scheme [Rees and Clinger, 1986]) is not our concern as it must be solved by the front-end of the compiler and thus it does not pose any constraint on the intermediate representation or the back-end.

Interactive LISP implementations achieve reflection by having a selection of meta-constructs to manipulate aspects of its evaluator [Maes, 1987]: *eval* and *apply* for programs given as data; *catch* and *throw* for the runtime stack; and *boundp* for the runtime environment.

Type safe runtime linguistic reflection [Stemple *et al.*, 1992] is achieved in the Napier88 programming environment by allowing a program to generate code that is then integrated into the program's own execution. This is simply accomplished by having the compiler as a persistent procedure that can be called inside any program. The same approach was previously followed in the reflective language PS-algol [Persistent Programming Research Group, 1987]. Run-time linguistic reflection requires a dynamic linking mechanism [Kirby, 1993]; in PS-algol it involves dereferencing a pointer and in Napier88 a projection out of the infinite union type (type *any* in Napier88). In order to allow for execution, the compiler must also be able to produce procedure closures. If procedure closures are not used then some mechanism must exist at the language level to denote executable programs as values [Kirby, 1993]. In Napier88, the compiler procedure is called during the execution of the program, the string representation of the source is translated to PAM code and encapsulated in an *any* value (see Figure 3.4). The value produced is a string if an error occurred; otherwise the compiler produces a procedure that, when applied, will execute the program provided as a string. In the example of Figure 3.4 the dynamic type-checking and binding[9] is achieved in the *project* clause and the variable *newProc* will contain code obtained during the execution of the program.

To support runtime linguistic reflection this way, it is then necessary to be able to call the compiler and to achieve dynamic linking and type-checking in order to link the result of the generated code into the running program execution. If the higher-order language uses an approach similar to Napier88, then it is necessary to be able to represent a value of type *any* and dynamic binding and type-checking will be achieved by projection from that type.

3.8 Constructs for Stable Store Management

The dominant method to implement orthogonal persistence is by software where a **Persistent Object Store** (POS) is built using the existing facilities of the operating system plus hardware. Further to the support of persistence by reachability, POS provides stability, recovery and concurrent accesses. A largely used POS was built by Brown as is described in [Brown, 1989]. Brown POS is used by versions of the following abstract machines [Brown *et al.*, 1992]: persistent abstract machine (PAM) [Brown *et al.*, 1988] for Napier88, PCASE machine [Davie and McNally, 1992, McNally, 1993] for Staple; PQM machine [Matthes *et al.*, 1992] for P-Quest, a persistent version of the Quest language [Cardelli, 1989]. Other POS with similar functionality exist: e.g., Mneme [Moss, 1989, Moss, 1990], the O_2 object manager [Bancilhon *et al.*, 1992, Velez *et al.*, 1989], EOS [Gruber *et al.*, 1992] and Texas Persistent Store [Singhal *et al.*, 1992]. Other approaches to the provision of persistence are being investigated in work on persistent operating systems supported by conventional hardware and in specialised hardware support, have been referred to in Section 2.2.2.

The architectures that use a POS have a runtime heap of objects, where objects are created and garbage collected if the space is exhausted, and has a system that manages the storage and retrieval of persistent data between the runtime heap and the persistent object

[9]In static binding the association between name and object can be determined by statically analysing the program. In contrast, with dynamic binding the association is made during the dynamic evaluation of the program by the runtime system using dynamic scope for names [Atkinson and Morrison, 1988]. Strongly typed languages will perform a dynamic type-check by having the runtime system to execute code that ensures that the value is of correct type. Dynamic binding of procedures requires the runtime system to perform dynamic linking of new code into the executing program.

store. Such a persistent management system is POMS ([Cockshott *et al.*, 1984, Brown, 1989]). POMS uses a table of address translations from the virtual memory address to the address in the POS, the persistent identifier, and the translation is done in software (pointer swizzling). The POMS allows a value to be made persistent or a persistent value to be accessed. In persistence by reachability, all values in the transitive closure of the root or roots of persistence are retained in the POS over garbage collections. The same property of values may be used to decide which values must be moved to the POS when a checkpoint is performed. The function of the garbage collector is to identify the data objects that are not any more in use and make their space available for reuse. Data object formats must therefore be recognised by the garbage collector to distinguish pointers from scalars as it needs to follow pointers to decide which objects are *alive*. In order to perform a **garbage collection**, store management must be able to identify all the pointers and know the size of all the objects stored. There are several algorithms to implement garbage collectors; [Wilson, 1992] contains a comprehensive survey. Store management must provide for efficient space usage, checkpointing and recovery.

Further to the POS and a means to identify the longevity of data items, other components of a persistent architecture are needed: an incremental binding mechanism to allow existing data and new data to be combined, an identity mechanism stable for long-lived data (pointers cannot be fabricated), a type-checking mechanism working for data of all spectra of persistence and a naming mechanism (at the language level) oriented to incremental construction and change (or at least the persistent root name for persistence by reachability). All of these needs must be supported by the proposed architecture. In addition, persistent object stores (or the runtime heap) require the intermediate language to agree on certain conventions and to provide certain information. For example, the POS must be able to identify pointers from non-pointers and to know the size of objects.

3.9 Summary

In order to support the class of persistent, higher-order and reflective languages, the proposed architecture must provide solutions to the following issues:

1. management of closures, in order to perform the mapping from higher-order languages to flat representations, providing a form of block retention;
2. management of dynamic binding and dynamic type-checking, in order to support incremental binding of new and persistent values, and to support runtime linguistic reflection;
3. management of space, in order to be able to find all the pointers and know the size of objects for garbage collection;
4. management of identity, in order to be able to identify persistent values; and
5. provision of a union type, to include all possible values in order to support a uniform implementation of universal parametric polymorphism.

These issues will be considered in the following chapters, namely in the TPL design and in the description of the prototype implementation.

Chapter 4

Intermediate Representations

This chapter surveys possible technologies which can be used to achieve a concrete intermediate representation to be used by front-ends for the high-level languages of interest to this work. This level of the proposed architecture constitutes the TPL intermediate representation. Existing examples of possible techniques are compared with respect to space efficiency, simplicity of optimisation, simplicity code generation and generality with respect to the high-level language and the hardware machine. The chapter concludes by choosing a representation suitable to be used in an experiment to build a prototype of the proposed architecture.

4.1 Introduction

Intermediate representations differ in the way they represent the operations, the control-flow and the data-flow. The level of operations can range from using high-level language operations to using target machine operations. Sub-machine language operations to a very primitive machine can also be included [Brandis, 1995]. The control-flow can be modelled using source language structures, or alternatively the branching structure of the program can be explicitly exposed. With data-flow, multi-assignment or single-assignment to the variables can be used. Some techniques can also combine control and data-flow in the same representation.

The following sections present a brief description of the attributes of some of the known representations which are relevant in the context of PHOL support. When a concrete example for a technique can be found, it will also be briefly characterised and assessed with respect to easy usage to perform high-level machine independent optimisations and code generation, and in supporting longer-term persistence of data. These forms can be used as a starting point to the design of TPL.

4.2 Three-address Code

This internal representation is described in [Aho *et al.*, 1986]. Three-address code is a sequence of statements of the general form

$$x := y\ op\ z$$

where *op* stands for operators, like integer arithmetic operators or logical operators, and x, y and z are names, constants or compiler-generated temporaries. Three-address statements may be implemented as quadruples, triples or indirect triples [Aho *et al.*, 1986]. For example, quadruples are records with four fields, *op*, *arg*1, *arg*2 and *result*. For the expression

$$\{a + b * 10\}$$

the corresponding three-address code representation is:

$$\begin{aligned} t1 &:= b * 10 \\ t2 &:= a + t1 \end{aligned}$$

and the quadruples representation is:

op	arg1	arg2	result
*	*b*	10	*t*1
+	*a*	*t*1	*t*2

Because of its simplicity — no complicated expressions or nested flow or control statements — this intermediate representation is well suited to localised optimisation and target code generation. To minimise space usage, the operand and result types are encoded in the operators. The operator set must be designed so that it is enough to implement the operations of the high-level languages in order to allow for good quality code to be generated; otherwise long sequences of instructions have to be generated by front-ends which in turn lead to more complex optimisers and code generators. The semantics of three-address code is low-level, sometimes machine specific and is specified in an *ad hoc* manner.

It is clear that this representation is both independent of the source and also of the target machines and that longer-term persistence may be achieved by providing operators and constants with the intended semantics.. For evolution, the operator set must be easily extensible.

The intermediate representation used in [Aho *et al.*, 1986] to illustrate intermediate code generation, code optimisation and target machine code generation, is a concrete form of 3-address code. This internal representation includes the following types of 3-address statements:

1. *x* := *y op z* — binary arithmetic or logical operation
2. *x* := *op y* — unary arithmetic or logical operation
3. *x* := *y* — assignment

4. *goto L* — unconditional jump

5. *if x relop y goto L* — conditional jump

6. *param X1, ..., param Xn; call p,n* — procedure call (procedure calls can be nested)

7. *x := y[i]* — indexing

8. *x := &y; x := *y* — pointer assignment and addressing

It is recognised in [Aho *et al.*, 1986] that "the choice of allowable operators is an important issue in the design of an internal form". A small set of operators is easier to implement on a new target machine but may force the front-end to generate long sequences of statements which creates difficulty for the optimiser and code generator. The operator set must be rich enough to implement the operations of the source language.

Because it is a simple representation close to the target machine, code generation from it is a simple task. The control-flow is well represented but the data-flow representation is poor. In order to perform high-level optimisations, it is necessary to do some analysis on top of this representation. Optimisations can be performed by extensive data-flow analysis done on top of the flow graph which corresponds to this intermediate representation.

4.3 Continuation-passing Style

Continuation-passing style (CPS) is an internal representation used for several higher-order languages in order to simplify the compilation process: RABBIT for Scheme [Steele Jr., 1978], ORBIT for Scheme [Kranz *et al.*, 1987], SML/NJ for Standard ML [Appel and MacQueen, 1987] and others [Teodosiu, 1991, Gawecki and Matthes, 1994].

The source language is translated into CPS by adding a *continuation* to every user procedure to represent the remaining execution of a program. When a procedure computes its result, instead of returning, it calls the continuation with the result as the argument [Appel and Jim, 1989]. CPS has simple and clean semantics (based on the λ-calculus) and also matches the execution model of a von Neumann register machine, which makes code generation easily. In CPS, each actual parameter of a procedure is *atomic* — a constant or a variable. Representations with this property are referred to as linear, as the flow of control is decided. The operands of arithmetic operators are also atomic and the result of the operation is bound to a new variable. The expression

$$\{a + b * 10\}$$

is represented is CPS as

$$\begin{array}{llll} (\; * & b & 10 & (\; \lambda r1. \\ \;\;\; + & a & r1 & (\; \lambda r2. \\ & & & \quad \cdots) \;) \;) \end{array}$$

One of the advantages of CPS lies in its appeal to formal semantics which makes it possible to reason about the representation and to formally prove the correctness of transformations. Another advantage is CPS's simplicity, as all control and environment structures are represented by λ-expressions and their application [Shivers, 1988]. The continuation represents the control point to which control will be transfered after the execution of the function. CPS has some of the advantages of the three-address representation, namely easy code generation for different hardware machines, as it is basically a stylised assembly language and, as all intermediate compiler-generated values are named, data-flow (as well as control-flow) information is explicit. Other advantages are reported in [Appel, 1992], namely:

1. as control is explicitly represented by continuations, it makes it easy to implement exceptions and first-class continuations;
2. procedure calls can be considered as "gotos" with arguments and tail-recursion elimination achieved automatically; and
3. a compiler can perform more transformations on the intermediate representation than on the source language [Flanagan *et al.*, 1993].

With respect to longer-term persistence support and generality this representation and the three-address representation have similar properties.

4.3.1 Appel's Continuation-passing Style (CPS)

CPS is defined in [Appel, 1992] as an ML datatype that represents CPS expression trees. In CPS every function has a name and there is a syntactic operator to define mutually recursive functions and n-tuple operators to model records and closures. CPS code is linear in the sense that arguments to a function (including primitive operators) are atomic (i.e., variables or constants and never other function applications). This property is of great value for code generation as machine code operations expect atomic operands as well.

CPS includes expressions to:

1. capture information for store management:
   - *RECORD([VAR a,INT 2,VAR c],w,E)*, builds a n-tuple in the heap initialised with the given values, binds the result to w and continues with expression E.
   - *SELECT(i,v,w,E)*, select the i^{th} field of the record v, bind the result to w and continues with expression E.
   - *OFFSET(j,v,w,E)*, adjust the pointer v, that is pointing to the i^{th} field of a record to point to field $(i+j)$, bind the result to w and continues with expression E.
2. capture control-flow:
   - *SWITCH(VAR i,[E_0,E_1,...,E_n])*, is a indexed branch; the continuation expression E_i is evaluated depending on the value of i.
3. capture higher-order procedures and control-flow:

- *APP(f,[a_1,...,a_k])*, calls the function *f*. The body of *f* is evaluated with the actual parameters *[a_1,...,a_k]* substituted for the formal parameters of *f*. As this is a tail-call there is no continuation expression[1].
- *FIX([(f_1,[a_{11},...,a_{1m}],B_1),..., (f_n,[a_{n1},...,a_{nm}],B_n)],E)*, defines zero or more recursive functions f_i and continues with expression E. The scope of each f_i includes all bodies B_i and expression E.

4. capture primitive operations:

- *PRIMPOP(+,[VAR a,INT 1],[u],E)*, binds to the variable u, the result of performing the primitive operation + with arguments given by variable *a* and constant *1* and continues with expression E.

Continuation-passing Style is an intermediate representation which makes explicit program's control-flow as well as data-flow, in the sense that every intermediate value has a name. However, it must be noted that it is still necessary to do data-flow analysis to determine the definition-use chains or live-variable analysis used in some code-improving transformations as described in [Aho *et al.*, 1986]. As CPS is closely related to Church's λ-calculus, it has a well defined and well understood semantics making proofs of correctness possible, at least in principle. The CPS representation proved to be useful in the compilation of languages like SML and Scheme, which allow for side-effects and have a precise evaluation order.

4.3.2 Tycoon Machine Language

Tycoon Machine Language (TML) is used as a persistent intermediate program representation within the Tycoon system, an "open persistent polymorphic programming environment" [Matthes *et al.*, 1994]. TML is based on CPS and its abstract syntax is defined in [Gawecki and Matthes, 1994]. TML values include literal constants, references to complex objects in the store (object identifiers), variables and λ-abstractions. Values are bound to variables in applications and there are some predefined primitive procedures to be used.

TML semantics are based on λ-calculus and well formed TML programs must satisfy a number of additional constraints that are statically enforced by the compiler front-end and never violated by the transformations performed in TML. Among these restrictions, identifiers are constrained not to be bound more than once (that is, all identifiers are different even when declared in different scopes) and continuations are not first-class objects (e.g., it is not possible to store continuations in data structures and subsequently apply them). User-level procedures always take two continuation parameters: one for the "normal continuation" which receives the continuation value, and one for the "exception continuation" which is invoked if a runtime exception occurs [Gawecki and Matthes, 1994]. Some of the primitive procedures provided to support source program compilation include primitives to create a mutable array holding object references, create an immutable array, the Y combinator to define mutually recursive function bindings, a call of C language functions and functions to

[1]Because no function returns, there is no need for the runtime system to maintain a runtime stack of return addresses and local variables.

install, remove and raise exceptions. As TML does not allow λ-variables to be modified after they are bound, mutable variables are boxed and modification of values is translated to explicit store manipulation.

A valuable property for longer-term persistence in TML is that it is easily extensible. To extend TML with a new primitive, it is enough to provide at back-end compile-time the following [Gawecki and Matthes, 1994]:

1. a function to generate machine-code;
2. a meta-evaluation function to perform optimisations on TML nodes representing calls to this primitive procedure;
3. a function to estimate the runtime cost of a given call (for inlining purposes);
4. a collection of attributes useful for the optimiser;

Before code generation, the Tycoon compiler removes exception continuations and calls to appropriate primitive procedures (*pushHandler* and *popHandler*) are inserted. Finally the compiler generates C code where procedures return instead of calling its continuation and continuation calls are transformed into *goto*s to labels in the current C scope[2].

TML has the same advantages of CPS, namely, it is formally described (call-by-value λ-calculus with store semantics), it has a small number of constructs (λ-abstraction and application) which simplifies the optimiser and it includes exception handling.

4.4 λ-calculus

The λ-calculus without explicit continuations is used in several compilers for functional languages [Peyton-Jones, 1987]. Intermediate representations based on the λ-calculus have the same advantages as CPS with respect to its formal semantics, namely reasoning and proof of correctness for transformations. According to Appel in [Appel, 1992], the λ-calculus "does not appear to be well suited to dataflow analysis" though.

4.4.1 Spineless Tagless G-machine

The STG language is a "very austere purely-functional language" which is the machine code for the Spineless Tagless G-machine [Peyton-Jones, 1992]. Apart from the usual λ-calculus denotational semantics which makes proofs of correctness possible, in [Peyton-Jones, 1992] STG was given a direct operational semantics using a state transition system. This semantics explains how it is intended to be executed.

A STG program is a collection of bindings of the form

$$\begin{aligned} bind &\rightarrow var = lf \\ lf &\rightarrow vars_f \ \backslash\pi \ vars_a \texttt{->} \ expr \end{aligned}$$

[2]In practical terms the CPS transformation is reversed.

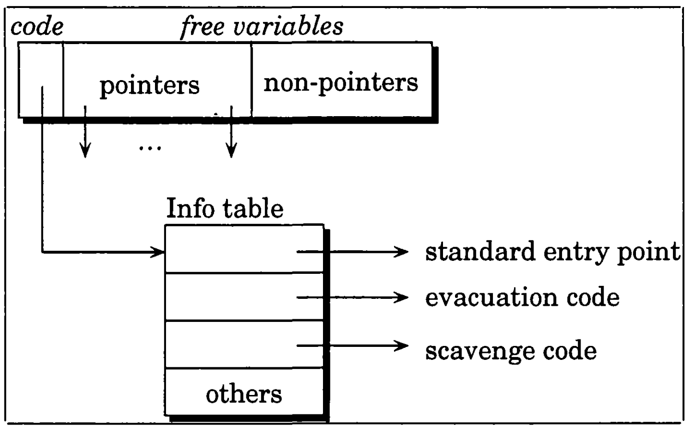


Figure 4.1: Closure Representation in STG

where each λ-form lf consists of a list of free-variables $vars_f$, an update flag $\backslash\pi$, a list of function arguments $vars_a$ and the function body $expr$. For example, the following binding

$$f = \{v_1, \ldots, v_n\} \ \backslash u \ \{x_1, \ldots, x_m\} \text{ -> } e$$

from a denotational point of view binds f to the function $(\lambda x_1 \ldots x_m.e)$; from an operational point of view, the function f is bound to a heap-allocated closure (to represent the function $(\lambda x_1 \ldots x_m.e)$ containing a code pointer and (pointers to) the free-variables $\{v_1, \ldots, v_n\}$ (see Figure 4.1); $\backslash u$ is a value of the update flag ($\backslash\pi$) which informs when the closure needs to be evaluated more than once (in this example it has the value $\backslash u$ which means update).

A salient constraint of STG is that each constructor or function can only have as arguments simple variables and constants and not, for example, other function applications. STG supports unboxed values which makes arithmetic easier and functions may have free-variables and that way λ-lifting need not be done. This concrete λ-calculus representation is similar to CPS, it is close to the high-level language and it has nested scope.

4.5 A-normal Forms

By studying the CPS transformation in [Sabry and Felleisen, 1992] the authors observed that since naïve CPS transformations considerably increase the size of programs, CPS compilers like ORBIT or RABBIT perform reductions to produce a more compact intermediate representation. They also observed that their code generators treat continuations specially in order to achieve a better memory usage and so better performance. While realistic CPS compilers, in a way, undo the CPS transformation, they claim to achieve the same results with a single source-level transformation called A-reduction. Known advantages of CPS transformation can be achieved using A-reductions:

1. an intermediate representation with a formal definition where optimisations can be performed, and
2. easy code generation.

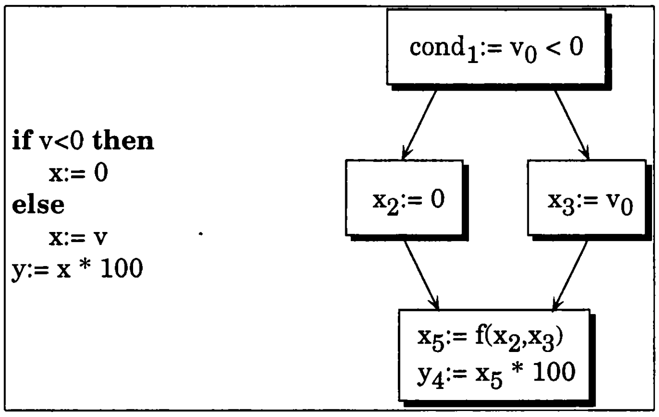


Figure 4.2: SSA Internal Representation

Because of that, CPS transformation is called a "red herring" in [Peyton-Jones, 1994]. A-normal forms appear to be the substitute of the CPS transformation in the future, but there is still work which needs to be done to prove its usefulness.

4.6 Static Single Assignment Forms

Static single assignment forms (SSA) is a program representation designed for the efficient implementation of certain data-flow algorithms. In [Cytron *et al.*, 1989] a new algorithm is presented that efficiently computes a SSA data structure for arbitrary control-flow graphs.

This data-flow representation has the useful property that each variable is assigned exactly once, and special statements called *ϕ-function* are inserted to distinguish values of variables transmitted on distinct incoming control-flow edges (see Figure 4.2). SSA is the more appropriate representation for dataflow analysis, and so to support elaborated transformations and have similar properties to CPS or 3-address representations for code generation and support for longer-term persistence.

4.7 Program Dependence Graph

Program dependence graph (PDG) is an intermediate program representation that makes explicit both the data and control dependences for each operation in a program, as described in [Ferrante *et al.*, 1987]. The PDG representation provides explicit representation of the definition-use relationships implicitly present in a source program and also the essential control relationships as presented in the control flow graph. The definition-use relationships are those which are explicitly presented in data dependence graphs. The nodes of the graph represent statements and predicate expressions that occur in the program and edges representing either a data dependence or a control dependence among program components.

Many traditional optimisations operate more efficiently on the PDG, as a single walk over the dependences explicit in the graph is sometimes enough. Since both data and control dependences are present in a single form, transformations can treat them uniformly, and if

```
IF (v<0) THEN a:= v*-1 ELSE a:= v END

(1) greg:
    (2) cmp v,0
    (3) if-less: (2)
        (4) a:= mul v,-1
    (5) if-gteq: (2)
        (6) a:= id v
    (7) i-merge: (3),(5)
    (8) a:= gate (7),(4),(6)
```

Figure 4.3: GSA Representation

transformations require interaction between the two, they can easily be handled in the PDG (see [Ferrante *et al.*, 1987]).

PDG has been used in inter-procedural slicing [Horwitz *et al.*, 1990], a technique useful in program debugging, automatic parallelisation and in program integration. The PDG is a more elaborated representation than the control graph present in CPS or the 3-address representation (which helps in data-flow analysis in optimising compilers) but retains the properties concerning easy code generation (see [Norris and Pollock, 1994] for register allocation using the program PDG) or support for longer-term persistence.

4.7.1 Guarded Single-Assignment Forms

Guarded Single-Assignment Forms (GSA) is the intermediate representation used in the optimising Oberon compiler OOC2 [Brandis, 1995]. An algorithm able to produce GSA in one pass for structured languages is presented in [Brandis and Mössenböck, 1994]. These programs contain assignments and structured statements (such as *if*, *repeat*, *while*) but no *goto*s. This intermediate representation combines in one graph both a high-level representation of the control-flow, by "guarded instructions with instruction lists at the machine level" [Brandis, 1995], and a static data-flow graph. This representation was inspired by the *Program Dependence Web* which is a refinement of the PDG.

Figure 4.3 shows a simple Oberon statement and its GSA representation. GSA instructions consist of an opcode followed by a list of operands which can be constants, variables, types or results of other instructions. An instruction result is referred to by parenthesising the corresponding instruction number. Control structures are represented by guards and merge instructions. Guards (e.g. *if-less: (2)*) take the result of a comparison (a condition code, e.g. *if-less*) and determine whether the condition holds. Guards control the execution of the following statements. Merge instructions combine predicates by or-ing them and they provide a list of predicates to gates determining which predicate is selected in the gate (e.g. in the instruction (8), operand (4) is returned by the gate if in instruction (7), path (3) holds).

Because they integrate both control-flow and data-flow in a single representation, GSA enable more powerful and simpler optimisation algorithms, and as its level is close to machine code, code generation from GSA is a simple task. In his PhD thesis, Brandis presents measurements that favourably compare the size and execution speed of the code generated by his compiler using GSA with other optimising compilers. These advantages of GSA make this representation well suited to constituting the basis of TPL.

4.8 Other Approaches

This section briefly describes some other concrete approaches which do not match any of the previous techniques. P-code, FAM, PAM, DIANA and PAIL are intermediate representations designed to support particular languages. Nevertheless, they have characteristics worth analysing in this section.

4.8.1 P-code

The use of an intermediate representation of a program as code for an abstract machine that is to be expanded into real target code was introduced in the UNCOL proposal. A much more successful implementation of the UNCOL approach was the usage of P-code as an internal representation for Pascal compilers [Nori *et al.*, 1981]. These compilers generate P-code for an abstract stack machine which is later interpreted or translated into target machine code. Moving the compiler to a new architecture involves only the construction of the translator to machine code or the construction of an interpreter for the abstract machine. This is a simple task, as the machine was kept simple. This approach made Pascal available on almost every platform and thus contributed decisively to its popularity. The disadvantages are concerned with the execution speed which is reported to be roughly four times slower than compiled code [Fischer and Leblanc, Jr., 1988].

The P-code experience shows that, as long as the high-level languages are similar, the introduction of an intermediate representation improves portability and therefore generality with respect to the target machine and reduces the complexity of the compiler.

4.8.2 FAM

The Functional Abstract Machine (FAM) was designed to support strongly-typed, statically-scoped functional languages and implemented to support the language ML in a VAX architecture [Cardelli, 1983]. The FAM is a stack machine that uses three stacks and a heap. The Argument Stack is for arguments and results and also for local and temporary values, the Return Stack is where the Program Counter and Frame Pointer are saved, and the Trap Stack is used to deal with exceptions. All objects in the stack have the same size as all are boxed. The FAM instructions are intended to be an intermediate representation which will be translated into native machine code language rather than being interpreted. The FAM abstract machine influenced implementations of ML and similar languages which possess first-class procedures and parametric polymorphism.

Operand shape	Prefix to instruction
one integer word	w
two integer words	dw
one pointer	p
two pointers	dp
one integer word and one pointer	wp
two integer words and two pointers	dwdp

Table 4.1: Types of Objects Supported by the PAM

4.8.3 PAM

The Persistent Abstract Machine (PAM) is part of the layered architecture that supports Napier88. PAM was designed to support only this language [Brown *et al.*, 1988] and is closely based in the PS-algol abstract machine [Persistent Programming Research Group, 1985] which in turn evolved from the S-algol abstract machine [Bailey *et al.*, 1980]. In [Connor *et al.*, 1989] it is stated that PAM, due to the modularity of its design and implementation, may be used to support any language with no more than: persistence, subtype inheritance, first-class procedures, ADT and block structure.

The PAM is a single-heap based storage architecture without the usual stack found in block structured languages. The piece of stack required to implement each block or procedure execution of the source language is kept in a stack frame which is a (normal) heap object. The low-level type system has two levels, one to describe the object shape, i.e. location and size (see Table 4.1) and the other at a higher level to describe some semantic knowledge of the object. Operations that depend on the semantics of the objects, like comparisons for example, are separated for each of the high-level types: *integer*, *boolean*, *pixel*, *real*, *string*, *structure*, *vector*, *image*, *file*, *ADT*, *procedure*, *variant* and *any*. *ADT*, *variant* and *any* will have a dynamic tag to qualify the value at runtime. PAM provides operations with operands of all these types, jumps and stack load and assignment. It also provides functions to create and destroy objects, to garbage collect and to initialise the local heap of objects.

The PAM code is a persistent stack-based intermediate representation which supports important concepts to this work like first-class procedures or polymorphism. It proved not too adequate for optimisations and for code generation, though. PAM bytecodes are interpreted by an interpreter written in C.

4.8.4 DIANA

DIANA is an intermediate form of ADA programs suitable for communication between the front-end and back-end of ADA compilers, described in [Goos *et al.*, 1983]. DIANA consists of an ADT defining a set of operations that provide the only way in which instances of the type can be examined or modified, and, this way, is representation independent. The design of DIANA was based on ADA's formal definition with the aim of being efficiently implemented and to support the programming environment. DIANA is also suitable for other programming support tools as it retains the structure of the original ADA program.

4.8.5 PAIL

The persistent architecture intermediate language (PAIL) is an intermediate representation intended to represent any valid PS-algol or Napier88 program [Dearle, 1988]. Later PAIL was extended by Hurst & Sajeev in order to be used as an intermediate language target for compilation of the χ language [Sajeev and Hurst, 1992]. PAIL was designed to be a canonical form of the abstract data graphs representing a program in a Persistent Information Space Architecture (PISA) [Atkinson *et al.*, 1987] manipulated by compilers, optimisers, diagnostic and utility programs. One design aim was to be able to reproduce the original source from the intermediate representation in the form of PAIL in order to produce good diagnostic information by the compiler or the runtime system.

PAIL does not have a concrete textual linear syntax. A valid PAIL program is a graph structure containing instances of classes representing all aspects of computations supported by PISA. Each node in the PAIL tree contains encoded type information, the abstract code (a subgraph containing an arbitrary piece of PAIL code) and a reference to the node immediately above in order to provide contextual information. The classes in PAIL represent:

- assignments;
- control (sequencing, choice, repetition and exceptions);
- store allocation (structures, vectors, ADT and images) and declarations (associating the links stored in symbol tables with the corresponding PAIL expression code);
- symbol table entries (called links);
- indexing (structure, vector, pixel, image, string);
- scoping (to introduce a new block or a new procedure literal); and also
- aliasing, store to store operations, literals, application, comments and optimisations.

To illustrate PAIL, Figure 4.4 represents the PAIL graph [Dearle, 1988] for the following source code:

```
proc(E1, E2 -> E3); E4
```

It is reported that PAIL is used to do optimisations both at compile time and in the underlying abstract machine code [Dearle, 1988]. An example of compile-time optimisations that can be done is constant folding whereby parts of a PAIL program can be dropped because they are never reached. A PAIL class links the optimisation with the original PAIL in order to allow good diagnostics. The trade-off here is larger space usage, more complexity, and less efficiency on traversals as the original node has to be traversed as well. As PAIL programs retain all the information contained in the original program and also the information added by code generators, it is possible to tell the user more about what happened and why, in the event of a runtime error. Code generators traverse PAIL trees and produce executable code for a low-level machine and decorate the PAIL tree with additional information, such as addresses for program variables.

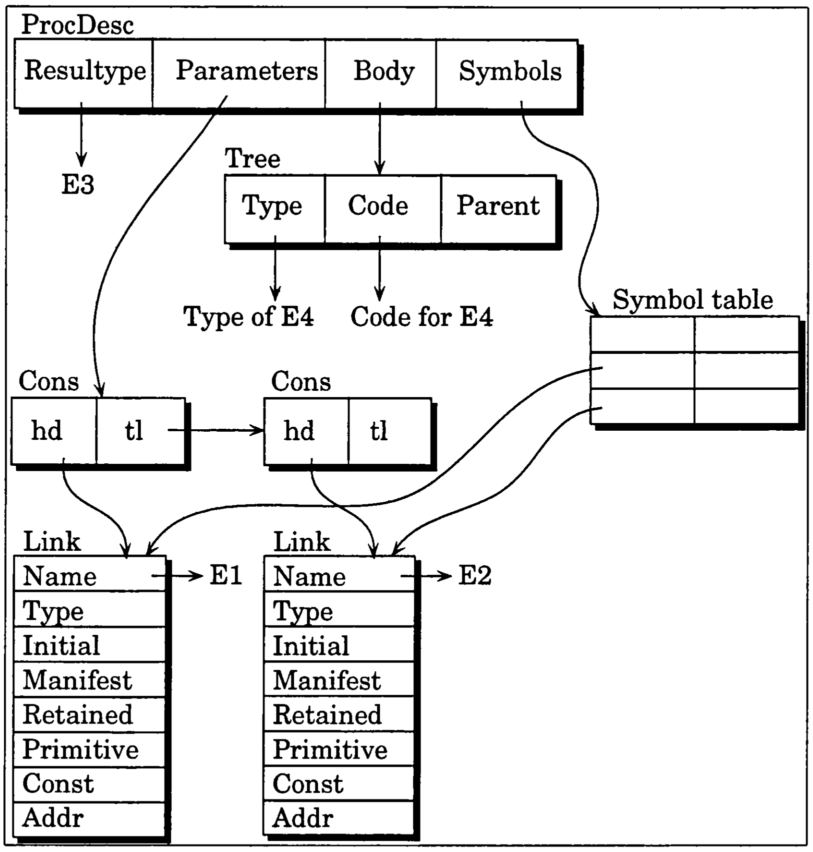


Figure 4.4: Procedure Definition in PAIL

The goal of multiple uses for PAIL and its consequent accumulation of data leads to large and complex structures that militate against efficiency. When compared with 3-address code further to its complexity, it can also be noted that operands have types attached instead of being encoded in the operators. This impacts negatively in space usage and in efficiency. Because PAIL failed as an intermediate representation, the lesson to be learnt seems to be that an intermediate representation cannot be too complex as that impacts negatively on efficiency.

4.9 Discussion of Intermediate Representations

Because of its simplicity and proximity to the machine, CPS and 3-address program representations are better suited for code generation. Inlining optimisation is easier to perform in CPS and 3-address representations because the arguments to function calls are atomic (i.e. code is linear in these representations). Optimisations that need a detailed data-flow analysis are better supported by the more elaborated representations like SSA or PDG. Because control-flow and data-flow are represented together, PDG internal representations have the additional advantages of having only one data structure to be maintained and offering support for algorithms that can combine both control-flow and data-flow dependencies. All of these representations are independent of the source language and also of the target machine

attribute	FAM	PAM	DIANA	3-add.	STG	CPS	PAIL	GSA
form	interp.	interp.	ADT	3-add.	λ-calc.	CPS	graph	PDG
persistent	no	yes	no	no	no	no	yes	no
linear	yes	yes		yes	yes	yes	yes	yes
interm. named	no	no		yes	yes	yes	no	no
nested scope	no	no	no	no	yes	yes	no	no
reasoning	no	no	no	no	yes	yes	no	yes

Table 4.2: Attributes of Intermediate Representations

and may therefore constitute a convenient vehicle to achieve generality. Longer-term persistence can be achieved by annotating the usual representations with information describing machine dependent characteristics and by extending the internal language with new types and the corresponding operations.

Table 4.2 presents a summary of attributes of the concrete intermediate representations described. TML and P-code are not represented. TML is a persistent version of CPS, and P-code is interpreted and has similar properties to the FAM. Figure 4.5 compares them with respect to their level, from close to hardware (0) to close to the high-level language (1); their complexity, which is a way of major how easy it is for front-ends to use it; data-flow and control-flow attributes represent the level of descriptions achieved in each representation; optimisation and code-generation represent how easy it is to perform those tasks; and debugging represent the amount of information available to enable good diagnostics. For each one of these attributes, value 0 corresponds to the worst case and 1 to the best.

The closer to the source language the intermediate representation (IR) is positioned, the easier it is to be used by the front-end, and at the same time, the less general it is. On the other hand, an IR that depends on any machine language characteristic does not serve as a proper representation as far as longevity is concerned. Further to the level of the IR, its complexity impacts negatively on its usage. The existence of nested block scope is a benefit because the IR is intended to support a high-level language with nested block scope. An IR closer to machine level simplifies code generation, but more important to the simplification of this task, is the quality of having every intermediate compiler value explicitly named which leads to a representation where operands are ready to be used by primitive operations. Having explicit control-flow and data-flow also helps in optimisations and code generation. The ability to be able to reason about an IR and to prove the correctness of optimisations and other transformations is also important in an optimising compiler.

4.10 Conclusions

TPL is achieved by having the constructs identified in Sections 3.5 to 3.8 in the representation and by adapting the representation to the constraints imposed by generality, efficiency and longer-term persistence.

CPS and 3-address representations are closer to the machine and also simpler than the other representations, and their characteristics make them well suited to target machine

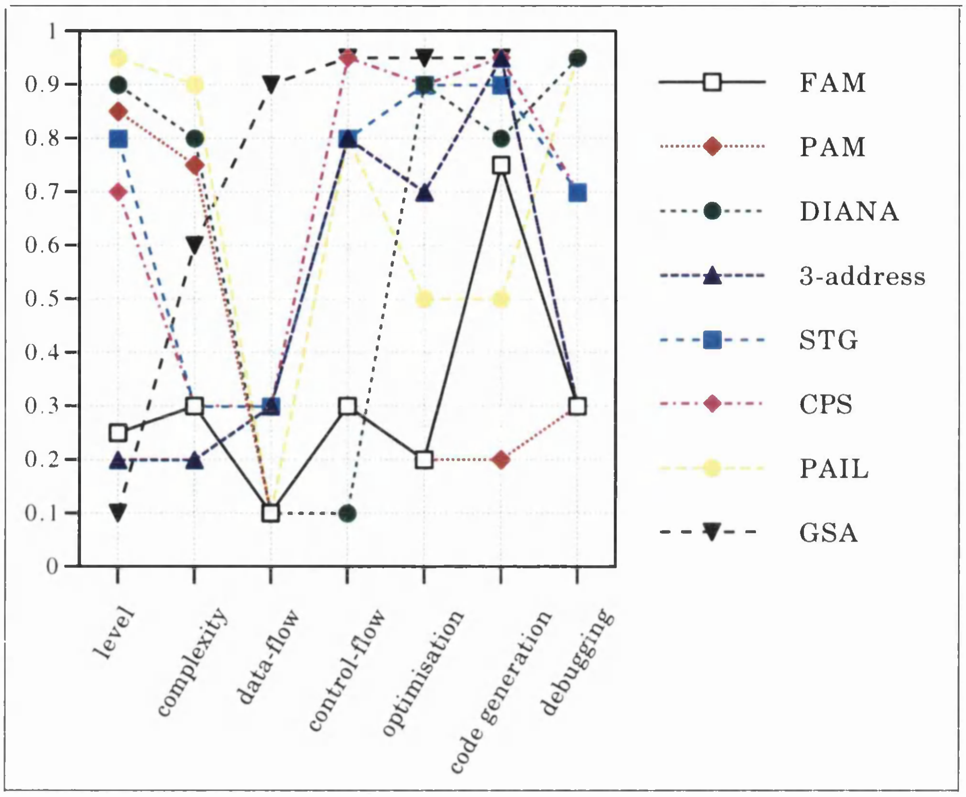


Figure 4.5: Comparing Different Internal Representations

code generation as well as low-level optimisations. PDG and SSA are more appropriate representations to be used by more powerful algorithms but are more complex and therefore more expensive to generate by front-ends. The aim is to prove optimisations possible and not in doing research on program transformations. Because of their complexity when compared with 3-address or CPS representations, PDG or SSA are not going to be considered in this experiment. CPS properties help in performing high-level optimisations. Some CPS properties can also be introduced in a 3-address modified form; code can be made linear, with nested scope, and all intermediates can be named. As compilation is intended and because they are closer to the target machine, modified 3-address code or modified CPS seem to be the more adequate representations from which to evolve TPL. The complete TPL design will be presented later in this dissertation on Chapter 7.

Chapter 5

Target Languages

This chapter surveys possible technologies which can be used to achieve the target language needed at the lower-level end of the proposed architecture. These candidate technologies are compared with respect to their adequacy for store management; their support for persistence, stability, recovery and concurrency; their support for dynamic binding and linking; and their independence of the hardware machine which simplifies retargetting the representation to a different architecture. For each target language, the quality of the generated code in terms of volume and execution speed, and the compilation speed are also discussed.

5.1 Introduction

Having the goal of being able to have high-quality code generators for several architectures (portability), some of them even unknown (longer-term persistence), is forcing the architecture proposed in Section 2.5 to have two levels: a machine independent representation, TPL, and another level designed to fit a particular class of architectures. This second representation can be used by different code generators which generate target machine code for concrete architectures. A way to generate machine code "on-the-fly" is then needed to support reflection on heterogeneous components from different "era". Whenever the Object Store is moved to a different architecture, a new **era** is initiated. The work associated with the initialisation of a new era involves the translation of all values to the new physical formats (according to new byte-orderings, etc.) and the invalidation of target machine code. Object code for the new architecture needs to be generated, bound and loaded on demand and cached for future use.

Some optimisations do not depend on the underlying architecture; examples of such optimisations are constant folding, constant propagation or dead code elimination. On the other hand, optimisations like register allocation and instruction scheduling are dependent on the architecture and cannot be done at the same level as the former optimisations.

In addition to the language at the TPL level, leading to machine independent optimisations (among other advantages already discussed), the proposed architecture includes a

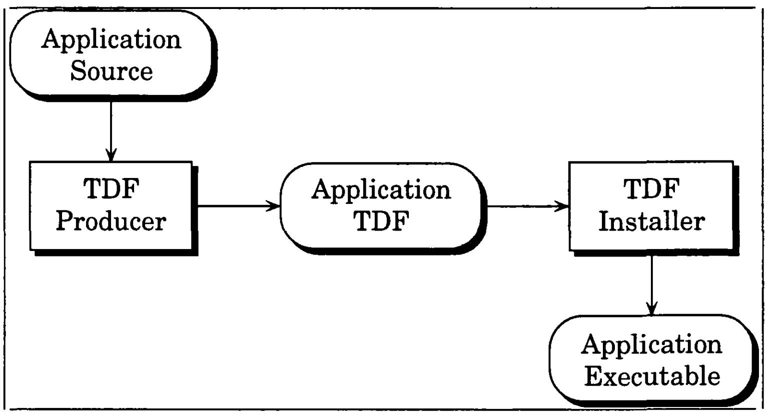


Figure 5.1: Distribution of Applications Using TDF

second level. This level introduces room to machine dependent optimisations. Further to its fundamental rôle in terms of longevity with respect to the underlying architecture (portability), the language at this level, UMC, must fulfil the requirements of easy code generation and generality in the sense that it must encapsulate the object store and the hardware machine functionality.

Possible candidate technologies are described, and compared and a choice made of the representation to be used in the experiment to be conducted.

5.2 TenDRA Distribution Format (TDF)

TDF is a tree structured intermediate language designed with the aim of retaining all the information needed for code optimisation techniques and to serve as an architecture independent intermediate format for distribution of software applications [Defence Research Agency, 1991]. It is intended that TDF may be produced from a variety of programming languages and installed on a very wide range of architectures. TDF was designed to support ANSI C, C++, FORTRAN 77, COBOL, Pascal, ADA, Modula-2, Common Lisp and Standard ML. The process of distribution of applications via TDF is illustrated in Figure 5.1. The software vendor writes the application in any familiar high-level language and then uses a package (called PRODUCER in TDF parlance) to obtain a single version of the application in TDF. This version is shipped and converted to the machine code of any target computer owned by the purchaser. The conversion from TDF to executable code is performed by a package called INSTALLER. At installation time, the portability interface specified in the TDF code is substituted by efficient architecture specific pieces with the same semantics, and architecture dependent matters are handled, this way.

TDF is not pseudo-code for any abstract machine; instead, it is a tree-structured intermediate language containing abstractions for common programming languages concepts [Defence Research Agency, 1994]. For distribution, TDF trees are "flattened" and encoded into a stream of bits. DRA asserts that this encoded stream of bits is space efficient and extensible enough to allow upwards compatibility for any future enhancements to TDF definition [Defence Research Agency, 1991]. At the level of TDF, optimisations may be performed by

TDF-to-TDF transformations. These optimisations are performed by software that is portable and can be included in any INSTALLER for a new architecture. Reference [Defence Research Agency, 1992] shows figures for C language where the TDF compiler's performance achieves a 0.97 to 1.38 factor when compared with "native compilers" for VAX, MIPS, 80X86, SPARC and 68040 platforms. The compile time is affected by a factor that varies from 0.68 (in the VAX) to 1.31 (in the SPARC). TDF is less compact than object code by a factor of two for CISC machines and on RISC machines is 1.4 times the size of the corresponding binary produced by a native compiler.

A subset of TDF was chosen in June 1991 by the OSF to serve as their Architecture Neutral Distribution Format (ANDF). ANDF was aimed at providing the technology to support the development of portable code and to allow its distribution in an architecture-neutral format. The design goals of ANDF are similar to TDF goals and are enumerated in [Macrakis, 1993]:

1. architecture neutrality;
2. language neutrality;
3. easy extension to any given API (Application Programming Interface);
4. protection from reverse engineering;
5. efficient code, comparable to native compilers;
6. small size, comparable to usual executables; and
7. openness to future evolution and innovation in software, hardware and APIs.

TDF/ANDF compiler technology seems to help in solving the portability of applications, as long as the source code conforms to some API and the platform specific libraries used are present in the installation environment. Producers check code for portability, optimisations can be done in the intermediate representation, and the overall process achieves good quality of code. This technology does not intend to support data portability as it does not provide any support for hiding the byte ordering of the platform [Macrakis, 1993]. TDF/ANDF illustrates a good solution to the generality and low-level optimisation goals of this work. It is more ambitious though, as it goes in the UNCOL direction by supporting high-level languages as different as C or Scheme. The way trees are flattened before distribution may constitute a good solution to transport TPL code to a new architecture.

5.3 Code-generator Generators

There are several code-generator generators using tree pattern matching and dynamic programming [Aho *et al.*, 1986]. The input to these code-generator generators are tree patterns with associated costs and semantic actions (e.g., allocate registers or emit code). They produce tree matchers from the grammar specifying the intended behaviour. These code generators perform two passes over the tree: the first pass is bottom-up to determine a set of patterns

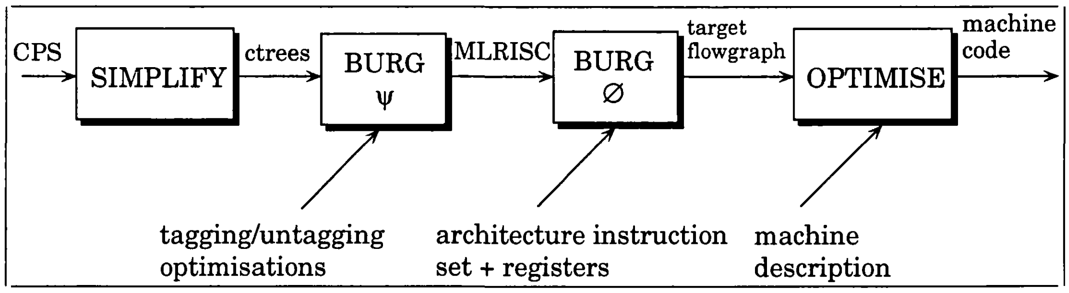


Figure 5.2: Code-generator Generator Using BURG

that cover the tree with minimum cost; the second pass executes the semantic actions associated with the minimum-cost patterns at the nodes they matched [Fraser *et al.*, 1992].

BEG and Twig code-generator generators use dynamic programming at compile time to identify a minimum cost cover. BURG code-generator generator uses the bottom-up rewrite system (BURS) theory to move dynamic programming to compiler-compile time. They produce matchers that generate optimal code in constant time per node, but the estimated execution costs must be constant, while systems that use dynamic programming at compile-time permit costs to involve arbitrary computations. IBURG [Fraser *et al.*, 1992] reads BURG specifications and produces a matcher that does dynamic programming at compile time, allowing dynamic cost computations.

In [George *et al.*, 1994] the authors propose a back-end to the SML/NJ compiler that uses BURS techniques to generate code. The architecture they propose is depicted in Figure 5.2. MLRISC is a language intended to represent the simplest and most basic operations implementable in hardware that makes no assumptions about addressing modes. A BURG grammar defining the instruction set and the associated semantic actions is used to translate MLRISC to target machine code. In this way, a good sequence of instructions is generated that makes full use of the capabilities of a particular architecture. At a second stage, target specific optimisations are performed, e.g. liveness analysis or graph colouring and register allocation. The overall price to pay is a slower back-end. The authors argue that having a SML/NJ compiler to a new architecture requires substantially less effort than porting the existing abstract machine, where instructions are macro expanded into target machine instructions and optimisations specific to a new architecture are manually coded.

This technique can be used to achieve a code generator for every new architecture. These code generators translate TPL code where representation decisions were made, to target machine code.

5.4 GNU Register Transfer Language (RTL)

The Free Software Foundation compilers consist of a front-end, a back-end and Register Transfer Language (RTL) intermediate code; documentation to the gnu C compiler (GCC) is available as [Stallman, 1992]. Operations in RTL are low-level and each RTL statement has almost a one-to-one mapping to a machine instruction. RTL uses five kinds of values: expressions, integers, wide integers, strings and vectors. For expressions, RTL provides a typical set of numerical operations, operations to load a register from memory and to store a register in memory, and control-flow operations. Each RTL expression has room for a machine mode that describes the size and representation used for the data objects involved (a four-byte integer is an example of a machine mode). For example,

```
(plus:M X Y)
```

represents the sum of the values represented by X and Y carried out in machine mode M.

The front-end processes source code and produces fragments of a syntax tree (TREE), "not-quite-syntax-tree data structure" [Stallman, 1993] defined in the GCC file *tree.h*. In a single pass, the front-end builds a tree for each function or statement at a time. The trees are translated into RTL by the front-end, using a set of procedures that look for machine dependent information contained in configuration files defining the target machine. Therefore the RTL representation is machine dependent, and the trees are the only (almost) independent machine representation that may be used to adapt GCC to a new language by redoing its front-end. The back-end processes RTL, and by looking at code generation patterns for the target machine defined for the RTL instructions, generates machine code.

In the Modula-3 compiler, the front-end was modified to process a whole compilation unit and build a tree for it [Moss, 1993]. This front-end performs multi-passes through the tree to resolve definitions, perform static checks, and in the end, walks through the tree to produce RTL. This is an interesting experience for the work described in this dissertation, as the Modula-3 compiler needs to cover exception handling, garbage collection and persistence. The lesson to be learnt seems to be that using GCC needs the understanding of the internals of the front-end and its modification together with the extension of the internal syntax tree and that involves a lot of work.

To support a new source language one needs to work on TREE plus RTL and the tree representation is poorly documented (e.g. the functions to be called). Moreover, the GCC compiler is a large system to be manipulated, involving effort to learn and change. Another disadvantage of this approach is that GCC is a batch compiler oriented to produce static *.o* files rather than code that can reside in an object store and be dynamically loaded. This needs to be done in this work.

5.5 Assembly C

Due to its low-level facilities, a rich set of operations, the absence of restrictions and the fact that a compiler for C is available in almost all architectures, this language has been used as a portable assembler in compilers for several languages: Scheme [Bartlett, 1989], SML [Tarditi *et al.*, 1992], Haskell [Peyton-Jones, 1992], Napier88 [Bushell *et al.*, 1994] and many others, as referred to in Section 2.5. Usually, the internal representation constructed by the front-end is flattened to a file by writing a textual representation in C source code, the C compiler is called and the object code produced is then used as target machine code to be executed.

It seems advisable to treat C as an assembly language and avoid its type system as much as possible in order to minimise the problems introduced by optimisations [Chase, 1990] and the mismatch with the high-level language being compiled. At what level should C be used

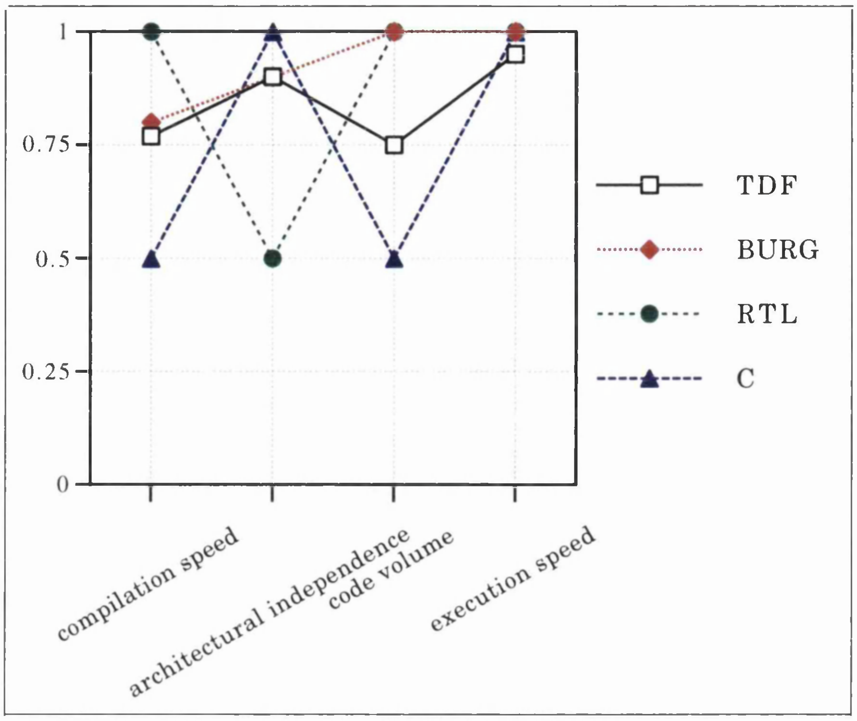


Figure 5.3: Comparison of UMC Languages

is a matter of debate: by using the C stack the target code can benefit from optimisations done by the C compiler but, on the other hand, that complicates the solution to issues such as garbage collection, first-class procedures or exception handling.

There are solutions to the problem of mapping languages with first-class procedures to C but no satisfactory solution has been proposed as yet to the problem of finding the pointers in the C stack. Garbage collection has been done by a conservative trace through activation records or by "registering" roots of garbage collection as they appear. Using these methods, garbage collection may collect reachable objects or relocate them without updating the pointers when in the presence of legal C compiler optimisations that disguise pointers, as described in [Boehm, 1991]. The problem is that in C there is no way of recognising a pointer until it is used to dereference an object. In the Modula-3 project the GCC back-end was enhanced to keep track of pointers by generating tables at possible garbage collection points that allow the garbage collector to find the registers which contain live pointers to objects and also stack locations that contain pointers to objects [Diwan *et al.*, 1992]. Using the return address stored in frames it is then possible to consult the tables for previous frames and in that way "decode" all of the stack. The authors report that the compiler-generated tables consumed 16% to 45% of the optimised code size and so impacts negatively on space efficiency.

5.6 Discussion of Target Languages

Figure 5.3 shows a comparison of the possible UMC described. The value 0 corresponds to the worst case and 1 to the best.

To achieve the goals of portability and easy code generation, TDF is an attractive possibility as it guarantees an architecture independent representation and low-level optimisations plus a way of generating code for any target machine by providing an INSTALLER. To achieve the goals enumerated, it would suffice to change the INSTALLER to accommodate persistence and reflection. TDF is a sort of extended C with the advantage of being unambiguous[1]. C has additional disadvantages of not having safe-pointers, exceptions or nested scope. The intermediate C code can be large and pose problems to C compilers because it is machine-generated (e.g. the use of "unusual" identifiers). It is easier to generate tree structures than human readable streams of characters and moreover, to flatten a tree to a C source and have a C compiler to parse it and construct a new tree again is not efficient. An existing intermediate language like TDF or an easy way of retargetting the compiler to a new architecture seems preferable than using C. The use of the tree data structure inside GCC offers the possibility of reusing compiler technology but it would involve great effort in understanding it. For practical reasons it is anticipated that C may be used with success to prototype the proposed architecture.

5.7 Conclusions

Because of the required generality with respect to the underlying architecture including the use of different object stores for persistent objects, the need for portability of the code generator and the goal of enabling low-level optimisations, another low-level intermediate representation was introduced at the UMC level. This representation is still independent of the target machine but has all representation decisions made (e.g. how closures are represented) and it must be potentially easy to generate target machine code from it. Among the values that will reside in the object store, is code inside procedure closures. This poses portability problems and introduces the need for dynamic binding and loading. When such an object needs to "move" to another architecture, new machine code must be generated from UMC. It is when target machine code is generated that low-level machine dependent optimisations are performed. These include instruction scheduling or peephole optimisations.

As discussed in Section 5.6, TDF or an easy way of retargetting the back-end of the compiler to a new architecture using modified BURG technology guarantees architecture independence without losing efficiency. BURG technology is intended to simplify the process of achieving a new back-end for a new architecture and, because of that, compares favourably with the task of building a new TDF INSTALLER for the same architecture. Changing the internal structures of the GCC compiler together with its back-end does not seem very attractive, due to the amount of work needed and the lack of documentation available. As described in Section 2.5, carefully chosen subsets of C have been used as a UMC for other languages.

Using C as a UMC has the disadvantages of an increase in code volume, slower compilation, and the absence of safe-pointers, exceptions and nested scope. Even considering these disadvantages using C is attractive for practical reasons. Using C, it is easier to achieve portability as a compiler for C exists on (almost) every platform.

[1]C has machine dependent semantics (e.g. integer or floating-point precision).

There are decisions related to the usage of C still to be made:

- will using the C stack for local variables (enabling the full set of optimisations usually performed by compilers) justify the cost of having to track pointers on this stack?
- will C function calls be used to implement TPL calls?

These and other decisions will be presented when the strategies for compilation are enumerated later in this dissertation.

This completes the search for possible technologies or techniques which can be used in the implementation of the three-stage architecture proposed. An experiment to be conducted in order to build a prototype of such an architecture will be described next.

Chapter 6

Experimental Design

This chapter presents the design of an experiment to build a prototype of a language compilation framework for the proposed architecture, which can prove the thesis by proving the architecture feasible and worthwhile and lead to the identification and validation of its crucial features. The components of the prototype language framework to be built are enumerated and will be detailed in the following chapters. A suitable source language to be used in the experiment is described, and the strategies which can be used to implement the compilation framework are enumerated and compared. Finally, this chapter concludes by describing briefly the enabling technology and the internal data structures which can be used to support the high-level internal representation and the transformations performed.

6.1 Introduction

The experiment conducted was designed to prove the thesis and to identify and validate the proposed architecture's crucial features and prove the architecture feasible and worthwhile. In the design of the high-level internal representation TPL, 3-address code and CPS were used as starting points (as was chosen in Section 4.10) to support the compilation of PHOL and implement high-level optimisations. To support interoperability and longevity, C was used as an assembly language (as was chosen in Section 5.7). Persistence was achieved by devising an object store interface and implementing a minimal memory mapped store.

In order to meet the goals enumerated in Section 2.3, a three stage architecture was proposed in Section 2.5. This architecture includes two languages to be designed, TPL and UMC, and several processors that will change the representation of programs from source code to machine code, as represented in Figure 6.1. The front-end parses and type-checks source representations of programs written in one PHOL and generates the intermediate representation TPL. High-level machine independent optimisations can be performed in order to improve code quality. By defining a complete particular abstract machine (e.g. deciding which parameter passing strategy or how to represent closures) a new internal representation is

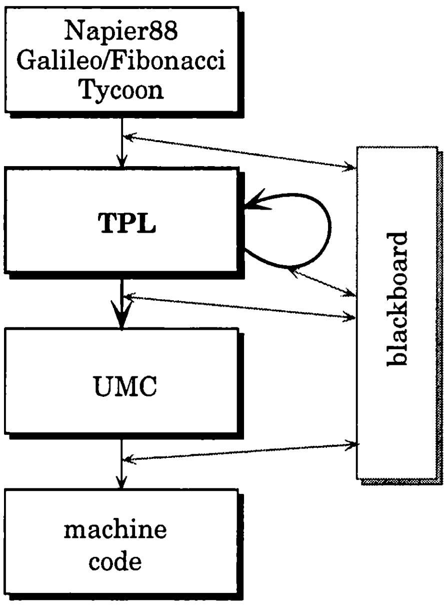


Figure 6.1: The Three-stage Architecture

obtained that is still independent of actual target machines. The third stage of Figure 6.1 consists of a back-end that generates machine code and performs machine dependent optimisations. A *blackboard* technology can be used to pass information between stages.

This work is mainly concerned with the design of TPL (so it can effectively support PHOL and high-level optimisations) and with the translation to UMC and the interface with an Object Store by a runtime system supporting the requirements of persistence, reflection, polymorphism and the higher-order property. Ideally, an existing front-end would be used and slightly modified at one end, and the same could be done to an existing back-end plus object store, on the other end of the internal representations. For the sake of the experiment, prototypes of all components of the intended architecture were built.

Chapter 4 contains a survey of existing intermediate representations that can be used at the TPL level, and Chapter 5 contains a survey of representations for the UMC level. At this stage, some decisions must be made about the forms both languages will take. The two languages will be defined in chapters 7 and 10, respectively.

6.2 Language Framework

The language framework to be built is represented in Figure 6.2 and includes the components, shown as rectangles, and the languages, shown as ellipses. COREL is the PHOL to be compiled and cTPL (or its equivalent representation C--) the UMC, as described previously. TPLk is TPL in the continuation-passing style and exec is executable target machine code.

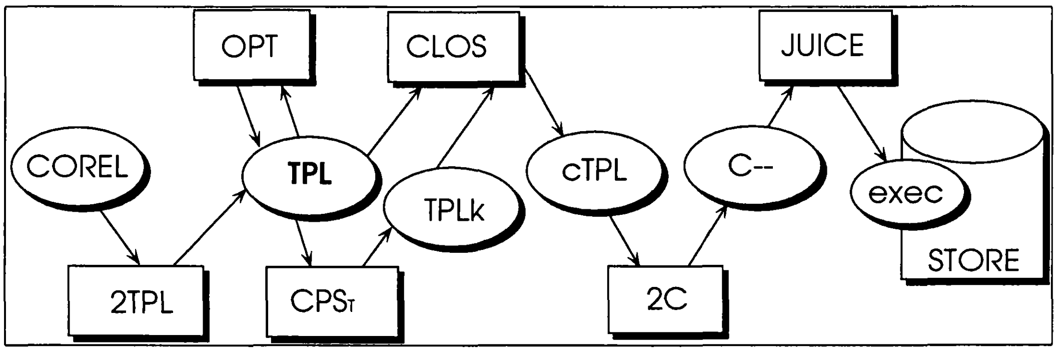


Figure 6.2: TPL Language Framework

6.3 Components to Build

The language framework to be built includes the following components, which will be detailed in subsequent chapters:

2TPL — a translator from COREL to TPL. It is the front-end of a compiler parsing and type-checking COREL programs and generating an internal representation in TPL.

LABEL — annotates the TPL internal representation and collects blackboard information.

OPT — a TPL optimiser that works by transforming the program represented in TPL into a (hopefully) better representation. OPT is in fact made out of a collection of components, each one performing a simple transformation which can be composed in any order. The included components implementing TPL transformations are:

UNUSED — useless-code and dead-variable elimination;

FOLD — constant propagation, constant folding, copy propagation and algebraic simplifications;

COMPAR — unreachable-code elimination or comparison folding;

INLINE — inline procedure calls; and

NOPS — *NOP* instruction removal.

CPSt — continuation-passing style (CPS) transformation[1].

CLOSE — closure analysis, choice of a particular parameter passing convention and commitment to a particular abstract machine architecture by emitting cTPL.

2C — a translator from cTPL to C-- programs.

JUICE — links the main C-- program with the runtime system in order to generate a UNIX executable.

STORE — a persistent object store to achieve persistence and stability.

PP — a pretty printer for all forms of TPL programs.

[1]TPLk could also be obtained directly from COREL. The decision of starting from TPL has to do with the desire of having only one parser to maintain in the prototype implementation.

```
1    let a1= 10
2    let p1='proc(i: int -> proc(int->int))
3    begin
4      let a1:= i
5      proc(x:int->int); { a1:= x+a1; a1 }
6    end
7    let v1= vector 0 to 1 of p1(0)
8    let a3= v1(1)(5)
9    use PS() with writeInt: proc(int) in
10     writeInt(a3)
```

Figure 6.3: An Example of a COREL Program

6.4 Core Language (COREL)

In the experiment designed, a core language (COREL) was chosen as the language to represent the class of languages targeted by this research: PHOL. It is a language heavily based on Napier88 that displays orthogonal persistence, first-class procedures, nested scopes, name spaces and a minimum set of types (*int*, *bool*, *string*, *proc*, *structure*, *vector* and *env*). The reason for choosing this limited set of features concerns the intention of minimising the amount of implementation work, in order to make the prototype a feasible 2 person-years project. The complete definition of the COREL language is presented in Appendix B.

It must be said that the other missing features, like polymorphism and reflection, can be proved to be achievable as long as this minimal set is supported. Subsequent chapters will deal with this.

Figure 6.3 shows a COREL program containing a first-class procedure (the result of *p1* applied to *0* at line 7) with a free-variable (*a1* at line 4) used after leaving its scope. This example uses a persistent value from the Object Store (the use clause put *writeInt* in scope). *v1* is a vector with two procedures that sum its parameter to a hidden accumulator (its free-variable *a1*) initialised with *0*.

6.5 Possible Compilation Strategies

With respect to the intended goals, the more promising choices for forming the basis for TPL were identified in Section 4.10 as being a modification of 3-address instructions (**3-address***) or a modification of CPS (**CPS***). At the output end of the proposed architecture, for the experimental prototype, the generation of low-level C code (**C--**) was identified in Section 5.7 as a convenient UMC. Choosing an internal representation with high-level **optimisations** in mind, then it must be decided which set of optimisations to supported (e.g. constant folding and propagation, copy propagation, unreachable and useless-code elimination, code lifting and inlining), which is the most profitable sequence of transformations and when to stop applying. Taking CPS* or 3-address* and a set of high-level optimisations, there are still several decisions to be made in order to build an abstract machine cTPL at the UMC level. The set of possible choices must be enumerated and the possible strategies to be pursued require experimental investigation and synthesis in one framework. The issues to be considered at each level of the compilation framework are introduced in turn.

6.5.1 Allocation of Activation Records

The abstract machine needs a **heap** to support procedure closures when the procedure's corresponding activation records have an indefinite extent. As stated in Section 3.5, activation records do not follow a LIFO policy in block retention languages. One of the solutions to the problem of free-variables of some procedures being used after leaving the scope, is to provide access to them through pointers in the procedure closure, as described in Section 3.5. This technique was referred to be more favourable in a persistent environment then other approaches. In this case, a stack may be used to store procedure activation records, as the closure in the heap provides access to all free-variables. The heap is also needed for data objects that need dynamically allocated storage, e.g. first-class vectors with variable size [Davie and Morrison, 1981].

In a **heap allocation** strategy, the heap is also used to store activation records allocating storage for local and intermediate variables at declaration time. The other possible strategies for program locals and intermediates are to store them in an effectively infinite number of **registers** or in **stacks**. It can be done with one stack whose layout is known or with two stacks: one for pointers and the other for non-pointers. The allocation of procedure activation records in the heap or in the stack, has been discussed in the literature [Bobrow and Wegbreit, 1973, Appel, 1987, Moss, 1987, Hieb *et al.*, 1990, Appel and Shao, 1994]. Stacks allow rapid allocation and deallocation of activation records and efficient linkage on call and return, together with better locality of memory operations, than heap allocation. Appel made the radical claim that heap allocation is more efficient than stack allocation in a large memory. If more than seven times the memory required for the computation is available, then garbage collection is essentially free [Appel, 1987]. An argument against this claim is that, although memory may be large, modern hierarchical memory systems that use caches, penalise programs that use large amounts of memory without a high degree of locality [Hieb *et al.*, 1990]. A study of the performance of memory sub-systems [Diwan *et al.*, 1995], concludes that "a stack is not needed to achieve good memory-system performance [...] and heap allocation of activation records can also have good memory-system performance". An argument in favour of the use of the heap to allocate procedure activation records in a linked list, is its simplicity in supporting first-class continuations, exceptions and threads. If a stack is used, extensive copying from the stack to the heap is needed when a continuation is created or a more complex stack/heap implementation must be used [Hieb *et al.*, 1990]. The idea here is to have the stack represented as a linked list of stack segments and heap records that mark continuations. When a continuation is reinstalled, the content of the stack segment of the continuation is copied into the current stack segment. In systems with multiple threads, each thread must have its own stack which makes stack management more difficult than using heap allocation. A technique of using a stack as if it were a heap, by providing links among activation records, has been described [Bobrow and Wegbreit, 1973]; this technique is usually referred to as "spaghetti stacks". Yet another technique is to allocate a fixed-size stack whenever a new thread of control is spawned ("cactus-stacks"). It should be noted that when stacks are used to store activation records with local and intermediate variables, free-variables used after leaving the scope need to be boxed inside a heap object and it is the pointer to this object which is kept in the stack. Procedure closures cannot point into stacks, as the activation record where one free-variable was declared may not still be in the stack when required, because first-class procedure closures have an indefinite extent. Because of persistence, the closure may already be in the store, created by another program, and therefore the same considerations applies.

6.5.2 Parameter Passing

As parameters are a subset of the local variables (constants initialised when the call occurs), the same set of strategies may also be employed to pass parameters to procedure calls. The most common strategy is to pass **parameters on the stack** by using the same stack as that used for local and intermediate variables. Therefore, if stacks are used for activation records, it seems reasonable to use them for parameters as well. Registers can also be used to pass parameters (**parameters in registers**) and this strategy can be combined with the usage of registers for locals and intermediates. Finally, another strategy that can be used is to accommodate the parameters in a heap object and to pass a pointer to that object to the procedure to be called (**parameters in a heap object**).

6.5.3 Mapping Locals and Parameters to a UMC

When generating target machine code, if a strategy of using an infinite supply of registers was followed and the intention is to use only the machine registers [Appel, 1992], then proper register allocation must be performed with spills to heap objects when needed. On the other hand, in the generated C--, registers can be accommodated inside a **C array** of convenient size as in the SML2C work described in [Tarditi *et al.*, 1992]. Alternatively, **C locals** may be used to represent the registers used in cTPL. This approach has the advantage of enabling the C compilers to apply their full set of optimisations to C-- procedures. The disadvantage is that pointers are difficult to find in the C stack when needed, e.g. during garbage collections and stabilisations. Strategies for locating these pointers will be discussed later (see Section 11.6.4).

These earlier decisions have implications for the C code generated. In any of these approaches, abstract machine registers should be stored in machine registers for the sake of efficiency. If heap allocation is used for local variables or parameters, they are accessed through a pointer to the object that represents the activation record (**indirect access**). If a stack is used, it can be simulated in C-- in a **C array** or managed inside a heap object of convenient size. Procedure calls in cTPL may be represented in C-- by corresponding **C calls** or by **gotos**. When a C array is used to store registers after register allocation, or a stack with local and intermediate variables and parameters is stored in a C array, there is no need to use the C stack, and gotos (longjumps in C parlance) may be used.

TPL	cTPL locals	cTPL param.	C-- locals	C-- param.	C-- calls
3-address*	heap	heap	indirect access	indirect access	gotos
CPS*	stack	stack	C arrays	C arrays	C calls
	registers	registers	C locals	C locals	

Table 6.1: Possible Choices in the Compilation Strategies

6.5.4 Identifying the Decision Space

Table 6.1 presents the set of possible exclusive choices (within different rows) for each decision that needs to be made in the proposed architecture (in columns). Because of the dependencies on previous choices presented in the text, not all choices may be considered at each decision point. After an analysis of the compatibility between these choices, the possible compilation strategies can be presented in the decision tree of Figure 6.4. The compilation strategies which can be followed, obtained by taking a compatible set of decisions, are represented by paths from the root to the leaves of the tree. In the experiment to be conducted, only the most promising strategies are going to be investigated; therefore, the tree needs to be pruned.

The strategy of having a C array for locals or parameters does not fit in the three-stage architecture proposed, as it does not permit low-level optimisations in UMC. After taking the corresponding paths from Figure 6.4, the four enumerated strategies are left. If register allocation is done in cTPL, then it seems better to also do instruction scheduling optimisations while generating native object code and there would be no need for the UMC level. On the other hand, heap allocation or the use of stacks, implies one more level of indirection and so, less efficient code. Therefore, cases 1 and 2 are not going to be investigated. These considerations suggest that the use of the C stack to store locals, intermediates and parameters should be investigated in this experiment.

In summary, the more promising strategies (at least for validating the compilation architecture) are a combination of the following choices:

- 3-address*, optimisations, locals and temporaries in registers, parameters in a heap object, C locals and C calls without parameters;
- 3-address*, optimisations, locals and temporaries in registers, parameters in registers, C locals and C calls with parameters;
- CPS*, optimisations, locals and temporaries in registers, parameters in a heap object, C locals and C calls without parameters;
- CPS*, optimisations, locals and temporaries in registers, parameters in registers, C locals and C calls without parameters.

The first strategy, corresponding to case 3 (in bold face in Figure 6.4) will be investigated. Both a modified 3-address representation and a modified CPS representation for TPL will be used in order to compare these two approaches. The parameters in registers can be accommodated easily into C function parameters and the results compared with the use of a heap object and indirect access for parameters.

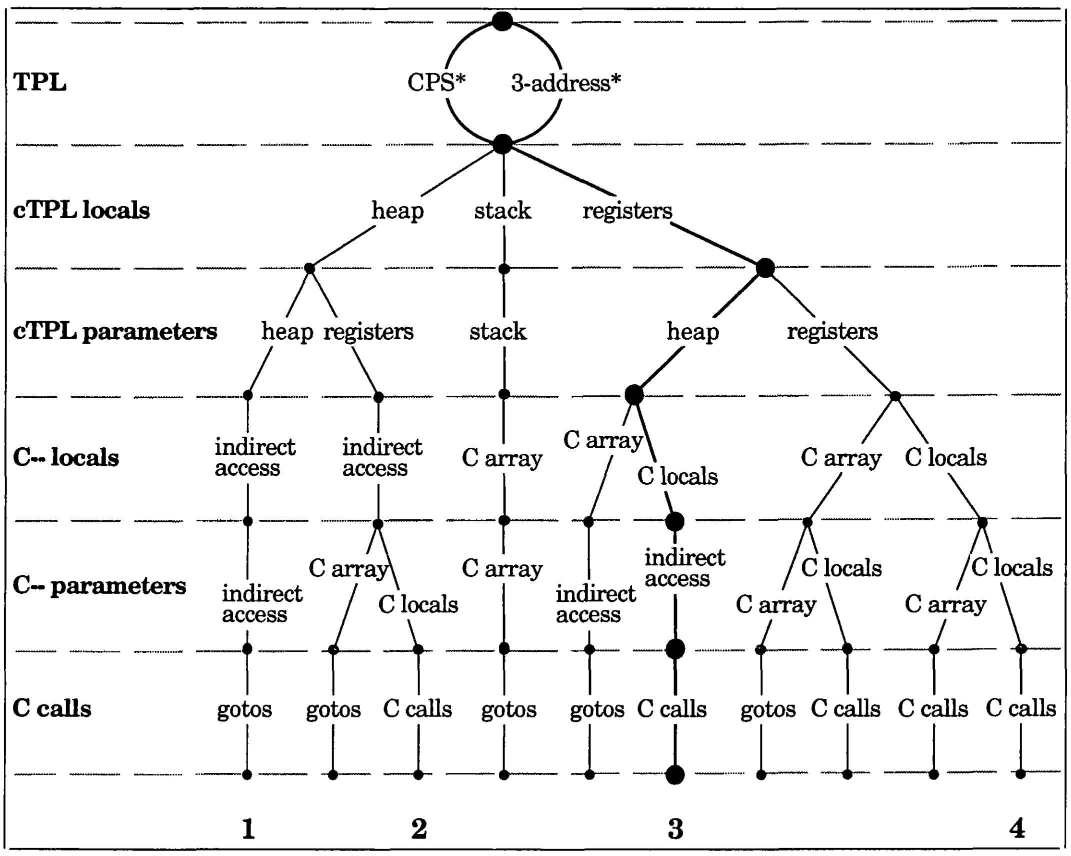


Figure 6.4: Possible Compilation Strategies

6.6 Enabling Technology

In order to implement the components referred to in Section 6.3 and obtain the language framework prototype which proves the architecture feasible, a programming environment has to be used. The programming languages used in this experiment were Napier88 and ANSI C in the UNIX operating system. A brief description of the relevant features of Napier88 to this work, together with a justification of the choice of these two languages is presented.

In Napier88, type-checking applies to all values, including the persistent values. Collections of bindings (environments) can be dynamically created (using the data type *env*) and bindings can be incrementally added to these environments, including bindings to other environments [Dearle, 1988]. Environments are first class citizens, which can be used to structure the store. As was the case with PS-algol, linguistic support exists in Napier88 for the construction of user interfaces. In Napier88, this support takes the form of two graphical data types: pictures constructed as line drawings in two dimensions of an infinite space and arrays of pixels. Types are considered as sets of values [Cardelli and Wegner, 1985] and type equivalence checking uses a structural model [Connor, 1991].

Napier88 was chosen to implement the prototype due to its characteristics: it is type complete, orthogonally persistent, polymorphic, higher-order and strongly-typed. Orthogonal persistence makes available to the implementation environment all the advantages referred

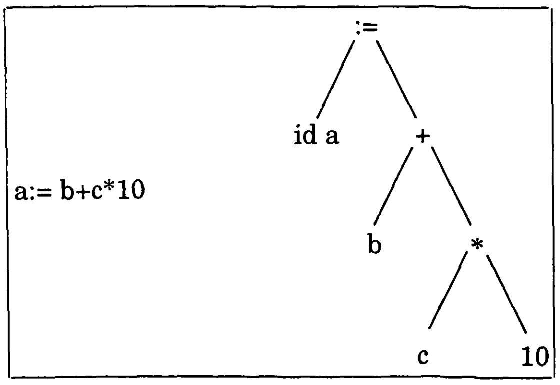


Figure 6.5: TPL Representation as a Tree

to in Section 1.3.3. An specially important consequence of using a persistent language in the compilation framework, is the fact that the interface between compilation phases (as well as passes) is a persistent data structure. This enables the construction of a "plug and play" framework, where components can be added to perform a single analysis or transformation and update the internal data representation accordingly. In a non-persistent context, one would need to use files and this process could become too slow. The other two important advantages associated with the usage of a persistent programming language are the maintenance of referential integrity and the possibility of keeping a strongly-typed environment at all times. Re-use and expressiveness are facilitated by polymorphism and first-class procedures. Environments constitute a convenient way of imposing structure on programs and their use makes incremental construction of applications possible. Finally, many errors in programs are discovered as soon as possible because the language is strongly (mostly statically) typed.

C is the obvious choice for the implementation of the object store and runtime environment, due to its low-level characteristics. These characteristics constitute an advantage in systems' construction as they give access to the internals of the host machine. As the prototype is implemented in machines with the UNIX operating system, C is also the better choice because it is tightly integrated with this operating system.

6.7 Internal Data Structures

The TPL representation in internal data structures of the programming language must be chosen in order to facilitate the work of the components that are going to generate it (e.g. 2TPL) and the work of the components that manipulate this representation to perform transformations in TPL (e.g. OPT, CLOSE, 2C). On the other hand, the representation must be easily extensible, so new TPL instructions, and the corresponding optimisations, may be added with minimum impact on existing programs.

The TPL abstract syntax tree can be represented by a tree or by quadruples, if 3-address instructions are used [Aho *et al.*, 1986]. In the tree representation, each node is represented by a record which contains an operation and fields pointing to its children, as shown in Figure 6.5. In order to make the representation more compact, a directed acyclic graph (DAG)

		op	arg1	arg2	result
*t1:= c*10*	(0)	*	c	10	t1
t2:= b+t1	(1)	+	b	t1	t2
a:= t2	(2)	:=	t2		a

Figure 6.6: TPL Representation as Quadruples

can be used. This representation, taking the form of a tree or a DAG, makes code generation more difficult, as for one instruction, other tree nodes need to be followed in order to discover the locations of operands. The representation as quadruples is a linearisation of the abstract syntax tree, where explicit names (intermediates) correspond to the interior nodes of the graph, as in Figure 6.6. A quadruple is followed implicitly by the quadruple immediately after, except in conditional or unconditional jumps, where a different label is explicitly named.

Some compilers for functional languages following a CPS strategy ([Appel, 1992, Gawecki and Matthes, 1994]) use a tree internal representation with a small number of different nodes for the CPS expressions referred to in Section 4.3. In this representation, the records representing nodes include fields that explicitly indicate the next expression to be evaluated (the continuation). In PAIL (see Section 4.8.5) instructions are implemented as a double-linked list which may help when performing transformations. Because PAIL representation has a lot of data for each instruction, it becomes inefficient when used to generate code. TPL is more compact as it does not include symbol tables, or versions of code modified by transformations as happens in PAIL. Of course the debugging information is not as good in TPL as in PAIL.

Because TPL will be based on modified 3-address code, TPL's internal representation is similar to quadruples. A graph with a different node for each TPL instruction will be maintained in the implementation in Napier88. Each node is a Napier88 *variant* which contains the information needed for the instruction it implements. For example, the conditional jump

```
BRA(R3,tpl-block-1,JUMP(L7),tpl-block-2,JUMP(L21))
```

has a boolean argument *R3*, a pointer to the block of TPL code to execute in the case the condition holds, a pointer to the TPL instruction to execute after that block and similarly for the case when the condition does not hold. Figure 6.7 presents the correspondence between

```
                                        LABEL(L7)
(7)   AND    R1   R2   R3          (7)  R3:= AND.BOOL(R1,R2)
(8)   COND   R3        (9)         (8)  BRA(R3,
(9)   ...                          (9)    TPL-block-1,
(14)  JUMP             (7)                JUMP(L7),
(15)  ...                          (15)     TPL-block-2,
(20)  JUMP             (21)                 JUMP(L21))
(21)  MULT   12   24   R4               LABEL(L21)
(22)  ...                          (21) R4:= MULT.INT(12,24)
```

Figure 6.7: 3-address and TPL Syntaxes

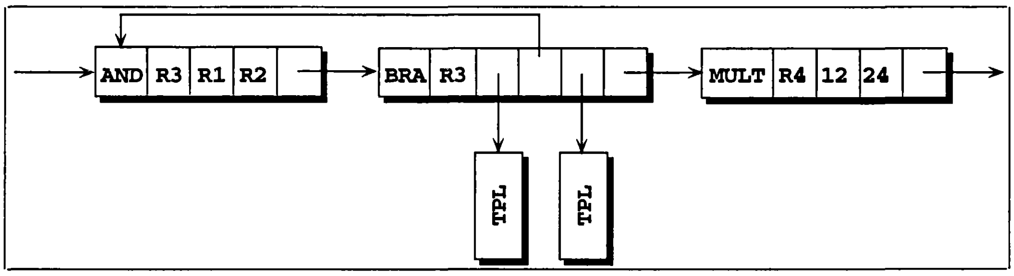


Figure 6.8: TPL Internal Representation

the syntax used to represent 3-address instructions [Aho *et al.*, 1986] and the syntax used in TPL. Figure 6.8 represents the graph data structure used to implement the TPL conditional branch.

By choosing the PPL Napier88 to implement the prototype, all the components of this framework will share the same internal data structures. These persistent data structures implement the abstract syntax tree, generated by the front-end and subsequently changed by each transformation. Because the intermediate representation is persistent, it constitutes a convenient way of experimenting with several compositions of the TPL to TPL transformations, by adding new components implementing new optimisations to the language framework.

6.8 Summary

The experiment to be conducted will investigate the use of modified 3-address forms and of modified CPS, to achieve a high-level intermediate representation for PHOL programs. The architecture independent optimisations, which can be easily supported by these representations, will be investigated. Locals and parameters in the heap or in registers, and the use as much as possible of the C stack in the C code generated to achieve low-level optimisations, will also be investigated. The following chapters will present a definition of the languages and components of the architecture involved in the prototype constructed to prove the architecture feasible. The implementation follows the strategies chosen in this chapter.

Chapter 7

Target Persistent Language

This chapter presents the language design of TPL, the intermediate language to be used by front-ends of the language compilation framework. In the design of TPL several aspects were considered, namely, how close to the source language TPL should be, what set of types and type-checking policy should be present, what set of primitive procedures should be provided, what mutability policy for locations, etc. This chapter presents the answers to these questions. First the TPL language characteristics are presented, followed by the set of TPL set of data types and corresponding instructions. A concrete syntax is presented in order to be used later in illustrations of the use of this language.

7.1 Introduction

TPL is a low-level programming language designed to support persistent reflective higher-order polymorphic languages, to easily support code generation, to abstract over details of the underlying software and hardware layers, to allow easy experimentation of abstract machine design (namely, different parameter passing strategies or different activation record representations), and finally, to provide support for typical multimedia values (like images and sound). TPL programs preserve the structure of the source language program and enough type information is kept in TPL programs to enable full inter-procedural analysis [Horwitz *et al.*, 1990].

Procedures are first-class values in the language and may have free-variables. TPL is linear as calls are ordered and arguments to procedures, user-defined or predefined, are atomic (constants or variables). In TPL all names are explicit and the control flow is also explicit. The set of TPL instructions include *constructors*, which bind a value to a new identifier, *primitive operations*, like *MULT* or *PLUS*, provided as predefined procedures which operate on constants or variables and bind the result to a newly created variable, and other instructions. These include an update instruction, which isolates all assigns to identifiers, procedure application, a generic branch and instructions to deal with persistent values. Low-level operations, like *MULT* or *PLUS*, are provided as primitive (or predefined) procedures which operate on constants or variables and bind the result to a newly created variable.

TPL production rules presented in this chapter use the Extended Backus-Naur Form notation as described in Appendix A. The TPL complete abstract syntax is presented in Appendix A as well and TPL will be pretty-printed as shown in Figure 6.7. In the TPL syntax, the operation is followed by its arguments using the notation usual for procedure calls, and the result is bound to a new variable, as in:

```
R121 := PLUS.INT(R120,10)
```

which computes the addition of the integer content of variable *R120* with integer *10* and binds the result to location *R121*. In TPL there is no overloading of operations and no automatic coercion of operands; the type of the result of the operation (*<op>*) is implicit and the types of the operands explicit (*<op-type>*), following the notation:

```
<op>.<op-type>()
```

The *<op>* part is omitted when the operation is the construction of a new value of the type. In coercion operations the types of both the in and out types are specified:

```
<out-type>.<op-type>()
```

and there is also a lookup operation which is represented by:

```
LOOKUP.<out-type>()
```

The following sections will detail the properties of TPL, and describe its types and their associated operations, the TPL statements for control-flow, assignment, type conversions, among others.

7.2 TPL Programs and Scope

A program is a block enclosed by the initialisation and shutdown of the runtime environment:

```
<pgm> ::= INIT() <tpl-block> CLOSE()
```

and a block is a sequence of instructions enclosed by *START* and *END*:

```
<tpl-block> ::= START {<tpl>}* END
```

A procedure is constructed by *PROC()*, provided with a string of characters specifying the types of the procedure arguments and the type of the result, a list of parameter identifiers and the body:

```
<id> := PROC(<string>,<id-list>,<tpl-block>PROCEND)
```

as in:

```
R1 := PROC("INT->INT",[A1],
      START
         R0:= MULT.INT(A1,2)
      END
      PROCEND)
```

Addressing mode	Example	Meaning
immediate	`x`	immediate value x, e.g. *100*, *TRUE*, *String25*
register	`Rn`	value is in location *Rn*
register offset	`Rn!off`	value is in the offset *off* from *RECORD* location *Rn*
register index	`Rn@idx`	value is in the index *idx* from *VECTOR* location *Rn*

Table 7.1: Addressing Modes in TPL

Procedures are first-class values in the language, can be nested and may access variables bound in external blocks which are therefore free in the procedure body. Arguments for user-defined or primitive procedures must be values of the required type; a literal value or a location holding a value:

```
<value> ::= <literal> | <loc>
```

A value can be bound to a new location by any of the TPL primitive-procedures and a value in a location can be updated with a new value of the same type. A location is an identifier (variable), an offset from an identifier of type *RECORD* or an index taken from an identifier of type *VECTOR*:

```
<loc> ::= <id> | <offset> | <index>
<offset> ::= <id>!<slot>
<index> ::= <id>@<slot>
<slot> ::= <id> | int
```

Access to values in TPL can be interpreted as described in Table 7.1. These addressing modes can be used in all TPL instructions which expect a value.

TPL is a block-structured language with true block **scope** in the Algol tradition. Identifiers are in scope immediately after the TPL instruction that binds them to a value and until the end of the block, identified by the first unmatched *END*. Identifiers are visible in enclosed inner blocks as well, but if the same name is bound again in an inner block, the new identifier will be in scope instead. The scope of the formal parameters of a procedure is all of the procedure's body. The scope is lexical or static, that is, the declaration that corresponds to an identifier is discovered by examining the program text. Nevertheless, in order to simplify the analysis of TPL programs by always knowing which identifier is being referred to, different names are used in all scopes. This unique-binding property is maintained by 2TPL, or if not, could be enforced by a new component of the language framework (CHECK) that would check TPL and rename all identifiers before any further processing.

7.3 Constancy

All locations are mutable. The value in a location can only be changed to another value of the same type by an *UPDATE()* instruction. All constancy checks for locations must be done by the high-level language front-end which must also make provision to verify the mutability of fields of structures or elements of arrays by appropriate runtime checks.

7.4 Equality

The equality operators provided in TPL follow a policy of equality meaning **identity**. Atomic values, e.g. integers, are tested for equality and aggregate values, e.g. records, are tested for identity. Other equalities are possible, like the structural equality, whereby two values are equal if they have the same structure and its constituents are equal. The equality is *deep* if the constituents are structurally equal and *shallow* if they are equal by identity. Should another policy be needed and the corresponding operator must be introduced into the TPL framework. Alternatively, the front-end can generate the structural equality tests. The equality by identity suits PHOL like Napier88 or Tycoon, where equality is also by identity but that is not the case of Galileo and Fibonacci.

7.5 Persistence

Persistence in TPL is implemented **by reachability**. All values in the transitive closure of a distinguished root of persistence, will persist. The root of persistence can be discovered using the instruction *PROOT()*, as in:

```
<id> := PROOT()
```

and the underlying object store will be checkpointed by:

```
STABLE()
```

which brings the store to a stable consistent state, thus allowing the recovery to this state after a media or software failure. *STABLE* is explicitly applied before shutdown of the runtime to bring the object store to a consistent state as well. The runtime will restart from the last checkpoint. To rollback to a previous stable state discarding all changes made, another TPL instruction is provided:

```
RESTART()
```

To support more sophisticated and flexible stability and recovering policies (e.g. nested transactions) the needed TPL instructions must be introduced. This subject is outside the scope of this work.

7.6 Type System

Type systems are intended to offer both modelling and protection. The low-level type system present in TPL is designed to offer protection and minimum modelling primitives as it is not intended to be used directly by humans, but instead by computer programs. Modelling will be done in the high-level source language.

Types of all identifiers are inferred from the TPL instruction that creates the initial value (constructors). The only place where types have to be explicit is to describe procedure parameters so they can be used in the body.

The language has an infinite number of data types defined recursively by the rules:

1. the base types are *INT*, *DOUBLE*, *BOOL*, *BITS*, *PIXEL* and *CHARS*;

2. *RECORD(T1,...,Tn)* is the type of a labelled cross product with implicit *1* to *n* labels and fields of type *T1* to *Tn*;
3. *VECTOR(T)* is the type of a vector with values of type *T*,
4. *PROC(T1,...,Tn->T)* is the type of a procedure with parameters of types *T1,...,Tn* and result of type *T*;
5. *INF* is the type of the package of a value of any TPL type together with its type and an integer tag;
6. *MAP* is the type of a map from strings to typed values; and
7. *VOID* denotes the absence of a type in a TPL instruction.

TPL instructions apply only to values of the same type (that is, operations are not polymorphic[1]) and there is no automatic coercion between values of different types when they appear as operands.

Type equivalence in TPL follows the structural rule, that is:

1. two base types are equivalent if they result from applications of the same constructor;
2. two *RECORD* types are equivalent if they result from the application of the *RECORD()* constructor over equivalent types and in the same order;
3. two *VECTOR* types are equivalent if they result from the application of the *VECTOR()* constructor over equivalent types;
4. two *PROC* types are equivalent if they result from the application of the *PROC()* constructor over equivalent types in the same order;
5. two *INF* types are always equivalent; and
6. two *MAP* types are always equivalent.

A new type can be introduced into the TPL language framework by specifying its literals and the set of primitive procedures (operations) that accept that type and the constructors for that type. For each operation the type of the result must be identified.

7.7 First-Class Citizenship

Following the Principle of Data Type Completeness, all data types may be used in any combination. Values of all TPL types have first-class citizenship with the following additional rights:

1. the right to be bound to an identifier;
2. the right to be assigned a new value;

[1]In fact the only exception is the assignment instruction *UPDATE()*.

3. the right to have equality (and inequality) defined over them; and
4. the right to persist.

Therefore, for each new TPL type, further to a type constructor, it is then necessary to specify the instructions corresponding to these rights.

7.8 TPL Types and Operations

In order to describe the high-level intermediate representation proposed, for each TPL type, in this section the literals are identified, constructors that bind a new location to a value of that type are presented and finally the primitive operations that accept arguments of the type are enumerated together with the type of its result.

7.8.1 Universal Operations

To fulfil the civil rights of every TPL data value, there is a set of universal operations provided:

```
<id> := EQ.<mc-type>(<value>,<value>)     % equality
<id> := NEQ.<mc-type>(<value>,<value>)    % inequality
<id> := MOVE.<mc-type>(<value>)           % shallow copy
INSERT.<mc-type>(<id>,<loc>,<value>)      % insert into MAP
LOOKUP:<mc-type>(<id>,<loc>,<value>)      % lookup from MAP
```

These instructions may be implemented differently for each machine-type and will be described together with the other instructions of each type.

7.8.2 Integer Operations

Included in the set of *INT* values are integer numbers that can be represented by 32 bits[2]; therefore they are in the range -2^{31} to $2^{31} - 1$.

Literals

An *INT* literal is one or more digits preceded by an optional sign:

```
<int> ::= {<add-op>}<digit><digit>*
<add-op> ::= + | -
```

Constructors

To bind a value to a new identifier *id* of type *INT* the *INT()* constructor is used with an argument of type *INT*:

```
<id> := INT(<value>)
```

as, for example, in:

```
R1 := INT(100)
```

[2]Similarly, types *BYTE*, *SHORT* and *LONG* may be defined to be represented by 8, 16 and 64 bits, respectively.

Primitive operations

Primitive operations on values of type *INT* have the following syntax[3]:

```
<id> := <prim-int>(<value>,<value>)
```

as, for example, to bind the addition of *100* to the value of location *R1*:

```
R2 := PLUS.INT(R1,100)
```

The primitive operations (see *<prim-int>* in Appendix A) have the usual semantics and bind a value of type *INT*:

arithmetic: `PLUS.INT, MINUS.INT, MULT.INT, DIV.INT, REM.INT, NEG.INT`

bitwise: `BAND.INT, BOR.INT, BXOR.INT, BNOT.INT, BSHIFTR.INT, BSHIFTL.INT`

and the following operations bind a value of type *BOOL*:

relational: `EQ.INT, NEQ.INT, LT.INT, LTE.INT, GT.INT, GTE.INT`[4]

DIV.INT() generates a runtime error when the second operand is *0*[5] The result of *DIV.INT()* or *REM.INT()* is negative if and only if only one operand is negative.

7.8.3 Real Operations

The set of *DOUBLE* values are floating-point numbers that can be represented by 64 bits using the IEEE 754 standard[6]; therefore they are in the range ± *4.94E-324* to *1.798E308*.

Literals

A *DOUBLE* literal is an integer followed by . (dot) followed by zero or more digits optionally followed by *E* and an integer:

```
<real> ::= <int>.<digit>*{E<int>}
```

For example, *2.71E3* means 2.71 times 10 to the power 3, i.e. the number *2710*.

Constructors

To bind a value to a new identifier *id* of type *DOUBLE*, the *DOUBLE()* constructor is used with an argument of type *DOUBLE*:

```
<id> := DOUBLE(<value>)
```

as, for example, in:

```
R3 := DOUBLE(2.71E3)
```

[3]With the exception of *BNOT.INT()* and *ABS.INT()* that take only one argument of type *INT*.

[4]It should be noted that some of the operations in this list are redundant. This set of operations can be obtained by taking the logical *AND*, *OR* and *NOT* and, for example, the relational operations *EQ* (equal) and *LT* (less than). A possible compromise is to omit one instruction for each complementary pair, such as *LTE* (less than or equal) because it is complimentary to *GT* (greater then).

[5]Run-time errors will raise an exception when exceptions are introduced in TPL. The source language name of the corresponding identifier is important to give meaningful error messages and must therefore be available at runtime.

[6]Similarly, type *SINGLE* may be defined for floating-point numbers to be represented by 32 bits.

Primitive operations

Primitive operations on values of type *DOUBLE* have a syntax analogous to the syntax of the corresponding operations on *INT* values:

```
<id> := <prim-real>(<value>,<value>)
```

as, for example, in:

```
R4 := TIMES.DOUBLE(R3,3.14)
```

The primitive operations (*<prim-real>*) that bind a value of type *DOUBLE* are the following:

> **arithmetic:** `PLUS.DOUBLE`, `MINUS.DOUBLE`, `MULT.DOUBLE`, `DIV.DOUBLE`, `NEG.DOUBLE`, `ABS.DOUBLE`

and the following operations bind a value of type *BOOL*:

> **relational:** `EQ.DOUBLE`, `NEQ.DOUBLE`, `LT.DOUBLE`, `LTE.DOUBLE`, `GT.DOUBLE`, `GTE.DOUBLE`

DIV.DOUBLE() generates a runtime error when the second operand is *0*.

7.8.4 Boolean Operations

Type *BOOL* has only the truth values true and false.

Literals

The *BOOL* literal values are introduced by uppercase strings of characters:

```
<bool> ::= TRUE | FALSE
```

Constructors

To bind a value to a new identifier *id* of type *BOOL*, the *BOOL()* constructor is used with an argument of type *BOOL*:

```
<id> := BOOL(<value>)
```

as, for example, in:

```
R5 := BOOL(FALSE)
```

Primitive operations

Primitive operations on values of type *BOOL* have the following syntax[7]:

```
<id> := <prim-bool>(<value>,<value>)
```

as, for example, to bind to *R6* the logical *AND* of *R5* with the literal *TRUE*[8]:

```
R6 := AND.BOOL(R5,TRUE)
```

[7]With the exception of *NOT.BOOL()* which takes only one argument of type *BOOL*.

[8]It should be noted that this operation is a candidate for optimisation.

The primitive operations (*<prim-bool>*) that bind a value of type *BOOL* are the following:

boolean: `AND.BOOL`, `OR.BOOL`, `NOT.BOOL`

The following operations bind a value of type *BOOL*:

relational: `EQ.BOOL`, `NEQ.BOOL`

7.8.5 Operations on Bits

Values of type *BITS* are bitmaps, i.e. sequences of zero or more bits with values *0* or *1*.

Literals

The *BITS* literal values are introduced by # followed by a sequence of zeros and ones or *X* followed by the equivalent values coded in hexadecimal:

```
<bits> ::= #<bit>* | X<byte><byte>*
```

Constructors

To bind a value to a new identifier *id* of type *BITS* the *BITS()* constructor is used with an argument of type *BITS*:

```
<id> := BITS(<value>)
```

as, for example, in:

```
R7 := BITS(#0010101001)
```

Primitive operations

The primitive operations (*<prim-bits>*) which bind a value of type *BITS* are the following:

concatenation: `<id> := CAT.BITS(<value>,<value>)`, binds to *id* a new value of type *BITS* formed by the concatenation of the first and second arguments;

sub-part: `<id> := SUB.BITS(<value>,<value>,<value>)`, binds to *id* a new value of type *BITS* formed by the part of the first argument in the range specified by the other arguments, that is, starting at the bit specified by the second argument and with length specified by the third argument;

bitwise: `BAND.BITS`, `BOR.BITS`, `BXOR.BITS`, `BNOT.BITS`, `BSHIFTR.BITS`, `BSHIFTL.BITS`

The following operations are performed *in situ* and therefore do not bind a value[9]:

not-part: `NOT.BITS(<value>,<value>,<value>)`, *in situ* inverts the value of bits of the first argument in the range specified by the other arguments;

set-part: `SET.BITS(<value>,<value>,<value>)`, *in situ* set to *1* the bits of the first argument in the range specified by the other arguments;

[9]Version of these operations could be provided to bind a *BITS* value to a new identifier.

> **clear-part:** `CLEAR.BITS(<value>,<value>,<value>)`, *in situ* clear to *0* the bits of the first argument in the range specified by the other arguments;

and the following operation binds a value of type *INT*:

> **length:** `<id> := LEN.BITS(<value>)`, binds to *id* a new value of type *INT* equal to the number of bits (length) of the argument;

For example, the following instruction binds a copy of part of *R7* bitmap starting at bit *1* and of length *5*:

```
R8 := SUB.BITS(R7,1,5)
```

The following operations bind a value of type *BOOL*:

> **relational:** `EQ.BITS`, `NEQ.BITS`, `LT.BITS`, `LTE.BITS`, `GT.BITS`, `GTE.BITS`

Two values of type *BITS* are equal if they are of the same length and all corresponding bits are equal. In comparing values of type *BITS* a null sequence is less than any other non-null value and the values are compared by taking corresponding bits from each value, one at a time and, considering *0* to be less than *1*, comparing these bits until one value is found to be less than the other.

7.8.6 Pixel Operations

Values of type *PIXEL* represent pixels of up to 24 planes in depth and each plane has a bit value of 1 (ON) or 0 (OFF)[10].

Literals

The *PIXEL* literal values are introduced by a sequence of 1 to 24 zeros or ones preceded by *#*:

```
<pixel> ::= #<bit><bit>*
```

Constructors

To bind a value to a new identifier *id* of type *PIXEL* the *PIXEL()* constructor is used with literal of type *PIXEL* as an argument:

```
<id> := PIXEL(<value>)
```

as, for example, in:

```
R55 := PIXEL(#11010)
```

[10]It should be noted that values of the *PIXEL* type could be accommodated in *BITS*. The introduction of *PIXEL* gives access to more specialised operations.

Primitive operations

The primitive operations (*<prim-pixel>*) that bind a value of type *PIXEL* are the following:

> **concatenation:** `<id> := CAT.PIXEL(<value>,<value>)`, binds to *id* a new value of type *PIXEL* formed by the concatenation of the first and second arguments;
>
> **sub-part:** `<id> := SUB.PIXEL(<value>,<value>,<value>)`, binds to *id* a new value of type *PIXEL* formed by the planes of the first argument in the range specified by the other arguments, that is, starting at the plane specified by the second argument and selecting a sequence of planes with depth specified by the third argument;

The following operation binds a value of type *INT*:

> **length:** `<id> := LEN.PIXEL(<value>)`, binds to *id* a new value of type *INT* equal to the number of planes (length) of the argument;

And the following operations bind a value of type *BOOL*:

> **relational:** `EQ.PIXEL`, `NEQ.PIXEL`

Two *PIXEL* values are equal if they have the same number of planes (depth) and the corresponding planes have the same value. When non-existing planes are specified in *SUB.PIXEL()* or the result of *CAT.PIXEL()* would exceed 24 planes a runtime error is generated.

7.8.7 Operations on Strings of Characters

Values of type *CHARS* are sequences of unsigned bytes (eight bits); therefore they are values in the range *0 – 255*.

Literals

The literal values of type *CHARS* are introduced by a sequence of hexadecimal values preceded by *X* or else a sequence of printable characters following the ISO-8859-1 standard[11]:

```
<chars> ::= X<byte><byte>* | "<print-char>*"
```

Constructors

To bind a value of type *CHARS* to a new identifier, the *CHARS()* constructor is used with an argument of type *CHARS*:

```
<id> := CHARS(<value>)
```

as, for example, to bind to *R9* the ISO-8859-1 representation of the string of characters *João*:

```
R9 := CHARS(X4A6FE36F)
```

or, equivalently:

```
R9 := CHARS("João")
```

[11]All eight bits characters coded in the ISO-8859-1 (Latin-1) standard that are possible to enter and display may be used. This gives more characters than the seven bits subset of ASCII. Incidentally, a type *UNICODE* may be defined together with the usual operations to concatenate two strings and take a part of a string. This type would correspond to the more general ISO-10646 (UNICODE) where each character uses two bytes.

Primitive operations

The primitive operations (*<prim-bytes>*) that bind a value of type *CHARS*, are the following:

concatenation: `<id> := CAT.CHARS(<value>,<value>)`, binds to *id* a new value of type *CHARS* formed by the concatenation of the first and second arguments;

sub-part: `<id> := SUB.CHARS(<value>,<value>,<value>)`, binds to *id* a new value of type *CHARS* formed by the part of the first argument starting at the byte specified by the second argument and with length specified by the third argument;

and the following operation that binds a value of type *INT*:

length: `<id> := LEN.CHARS(<value>)`, binds to *id* a new value of type *INT* equal to the number of bytes (length) of the argument.

For example, the following instruction binds to *R11* the string *João Lopes*, the concatenation of the string *João* with the string *␣Lopes*:

```
R10 := CHARS(" Lopes")
R11 := CAT.CHARS(R9,R10)
```

The following operations bind a value of type *BOOL*:

relational: `EQ.CHARS`, `NEQ.CHARS`, `LT.CHARS`, `LTE.CHARS`, `GT.CHARS`, `GTE.CHARS`

Two values of type *CHARS* are equal if they are of the same length and all corresponding bytes are equal. Individual bytes are ordered according to the ISO-8859-1 standard. In comparing values of type *CHARS* a null sequence is less than any other non-null value and the values are compared by taking corresponding bytes from each value, one at a time and comparing these bytes until one value is found to be less than the other.

This type is intended to support the subset of strings of characters with its usual operations. For example, creating a string and taking a sub-string with the first 4 (length of *R9*) characters of *R11*, is:

```
R12 := LEN.CHARS(R9)
R13 := SUB.CHARS(R11,1,R12)
```

7.8.8 Operations on Records

Values of type *RECORD* are aggregate objects composed by other TPL values possibly of different types.

Constructors

To bind a value of type *RECORD* to a new identifier *id*, the *RECORD()* constructor is used with a list of values enclosed by rectangular parentheses as an argument:

```
<id> := RECORD(<value-list>)
```

For example, to construct a value of type *RECORD* with a field with the *CHARS* value *João* and another field with the *INT* value *4*:

```
R14 := CHARS("João")
R15 := RECORD([R14,4])
```

Primitive operations

The only primitive operations on values of type *RECORD* are the boolean comparison that bind a value of type *BOOL*:

> **relational:** `EQ.RECORD`, `NEQ.RECORD`.

Fields of values of type *RECORD* can be accessed by offsetting using a label equal to the implicit position of the field. For example, to use the first field of *R15* in the construction of a new value to be bound to *R16*:

```
R16 := CHARS(R15!1)
```

7.8.9 Operations on Vectors

Values of type *VECTOR* are aggregate objects composed by several TPL values of the same type.

Constructors

To bind a value of type *VECTOR* to a new identifier *id*, the *VECTOR()* constructor is used with the vector lower bound and upper bound limits and the initialising value as arguments:

```
<id> := VECTOR(<value>,<value>,<value>)
```

For example, to construct a vector starting at *0*, with 12 integers all initialised with *0*:

```
R17 := VECTOR(0,12,0)
```

Primitive operations

The primitive operations on values of type *VECTOR* that bind a value of type *INT*, which check the bounds of the vector are:

> **bound-check:** `LWB.VECTOR`, `UPB.VECTOR`.

and the following operations on values of type *VECTOR* bind a value of type *BOOL*:

> **relational:** `EQ.VECTOR`, `NEQ.VECTOR`.

To access an element of a vector, indexing is used by specifying a value of type *INT*. If the index is out of the vector bounds, a runtime error is generated. For example, to use the second element of *R17* in the construction of a new value to be bound to *R18*:

```
R18 := INT(R17@2)
```

7.8.10 Operations on Procedures

TPL has support for procedures with values of type *PROC*.

Constructors

To bind a value of type *PROC* to a new identifier *id*, the *PROC()* constructor may be used. The *PROC()* constructor takes a string of characters representing the formals' types and the result-type, a list of corresponding procedure formals' names and the body:

```
<id> := PROC("<type-list>-><type>",<id-list>,<tpl-block>PROCEND)
```

An example of the construction of a value of type *PROC* was presented on Section 7.2. Another example is the following procedure which writes an integer to the standard output:

```
R19 := PROC("INT->VOID",[A1],
       START
         VOID:= CALLCC("printf",[" %d ",A1],"")
       END
       PROCEND)
```

Procedures are first-class values in the language and may have free-variables.

Primitive operations

The operations on values of type *PROC* that bind a value of type *BOOL*, are:

relational: EQ.PROC, NEQ.PROC

A procedure may be applied by using the *CALL()* instruction with a list of actual parameters with types respectively equivalent to the formals. For example, to write the integer *10* to the standard output, the procedure *R19* may be used:

```
VOID := CALL(R19,[10])
```

It must be noted that this procedure does not return a value to be bound. This situation is signalled by the *VOID* identifier used in the place of the identifier to bind the result.

To apply an external C function and bind the result to a TPL identifier, the *CALLCC()* may be used. It takes a string of characters with the name to be called, a list of actual parameters and a string with the name of the library to be linked, or an empty string if it is a function from the C standard library:

```
<id> := CALLCC(<string>,<value-list>,<string>)
```

For example, the *printf* function is applied with:

```
VOID := CALLCC("printf",[" %d ",10],"")
```

The library will be dynamically linked with the executable. If the library cannot be found or the C function name remains unknown after the dynamic link-editing operation, a runtime error will be generated. This way, the semantics of the *CALLCC()* may vary depending on the external C library.

7.8.11 Operations on *INF*

Values of any TPL type can be packaged together with an integer tag and its internal TPL type into a value of type *INF*[12].

[12]The *INF* type is a low-level infinite union type intended to support high-level constructs such as variants, *any* and polymorphism, as will be shown in Chapter 8.

Constructors

To bind a value of type *INF* to a new identifier *id*, the *INF()* constructor may be used. This constructor takes a type encoding, a value of that type and an integer value tag:

```
<id> := INF(<value>,<mc-type>,<value>)
```

TPL types are encoded in integers and a constant for each base type and type constructor is available to be used by the front-end: *INT*, *DOUBLE*, *BOOL*, *BITS*, *CHARS*, *RECORD*, *VECTOR*, *PROC* and *MAP*. For example, to have the *TRUE* value with tag *1*:

```
R20 := INF(TRUE,BOOL,1)
```

Primitive operations

To project the value out of an *INF*, the *PROJ()* operation may be used.

projection: `<id> := PROJ(<value>,<mc-type>,<value>)`

The first argument specifies the value to be projected, the second the internal type encoding and the third its tag. If the value is not of the required tag or type, a runtime error will be generated.

The operations on values of type *INF* that bind a value of type *INT*, are:

get-tag: `<id> := TAG(<value>)`

get-type: `<id> := TYPE(<value>)`

and the operations on values of type *INF* that bind a value of type *BOOL*, are:

relational: `EQ.INF`, `NEQ.INF`

For example, to determine the runtime type of *INF* value *R20*:

```
R21 := TYPE(R20)
```

Two values of type *INF* are equal if they are of the same TPL type, they have the same tag and they have equal values compared using the corresponding equality procedure.

7.8.12 Operations on *MAP*

Values of *MAP* type are used to hold maps from *CHARS* to (typed) values.

Constructors

To bind a value of type *MAP* to a new identifier *id*, both the *PROOT()* or *MAP()* constructors may be used. The *PROOT()* constructor binds to the identifier *id* the location of the distinguished root of persistence, which is of type *MAP*:

```
<id> := PROOT()
```

and *MAP()* creates a new object to hold a map:

```
<id> := MAP()
```

Primitive operations

The operation to insert a new entry into a map of type *MAP*, given the name (of type *CHARS*), the location of the map and the value of a compatible type, has the syntax:

```
INSERT.<mc-type>(<id>,<loc>,<value>)
```

and there is one instruction for each TPL type:

insertion: `INSERT.INT`, `INSERT.DOUBLE`, `INSERT.BOOL`, `INSERT.BITS`, `INSERT.PIXEL`, `INSERT.CHARS`, `INSERT.RECORD`, `INSERT.VECTOR`, `INSERT.PROC`, `INSERT.MAP`, `INSERT.INF`

The operation to lookup a value in the map and bind it to the identifier *id* of the corresponding type, have the syntax:

```
<id> := LOOKUP:INT(<id>,<loc>)
```

and there is one instruction for each TPL type:

lookup: `LOOKUP:INT`, `LOOKUP:DOUBLE`, `LOOKUP:BOOL`, `LOOKUP:BITS`, `LOOKUP:PIXEL`, `LOOKUP:CHARS`, `LOOKUP:RECORD`, `LOOKUP:VECTOR`, `LOOKUP:PROC`, `LOOKUP:MAP`, `LOOKUP:INF`

If the value does not exist in the map or it is not of the required type, a runtime error will be generated. It should be noted that for each new TPL type added, the *INSERT* and *LOOKUP* instructions must be provided to ensure full civil rights.

The operation *REMOVE()* removes the entry with the name specified as the first argument from the map specified as the second argument:

```
<REMOVE(<id>,<loc>)
```

The operations on values of type *MAP* that bind a value of type *BOOL*, are:

relational: `EQ.MAP`, `NEQ.MAP`

exists: `<id> := EXISTS(<id>,<loc>)`; tests the existence of an entry in the map

is-type: `<id> := ISTYPE(<id>,<loc>,<mc-type>)`; tests the type of an entry in the map

This type may help in structuring the object store into different zones and serve to give hints for clustering during object allocation. Operations to scan a map, like *FIRST*, *LAST* and *NEXT*, and operations to perform *in situ* updates of values in a map may prove to be necessary in the future and will be added.

7.8.13 Miscellaneous Operations and Statements

Included in this group are TPL operations that do not bind any value, the *statements*, store and runtime management, control-flow, assignment, type conversion and other miscellaneous operations.

Store and runtime management

The following operations are provided to deal with the store and the runtime management:

> **initialisation:** `INIT()`, runs the runtime initialisation routine
>
> **shutdown:** `CLOSE()`, runs the routine that performs the shutdown of the runtime environment
>
> **stabilise:** `STABLE()`, runs the runtime routine that performs a checkpoint in the object store
>
> **rollback:** `RESTART()`, runs the runtime routine that performs a rollback to the last checkpoint in the object store
>
> **garbage-collect:** `GC()`, runs the runtime routine that performs a garbage collection of the object store[13]

Control-flow

The *BRA()* statement is the only TPL instruction available to specify program control-flow. It can be used to specify controlled program jumps depending on the value of a condition:

```
BRA(<value>,<tpl-block>,JUMP(<id>),<tpl-block>,JUMP(<id>))
```

The first argument is a value of type *BOOL*. If its runtime value is equal to *TRUE* then the TPL block of code specified by the second argument is entered followed by an unconditional jump to the identifier specified by the third argument. If not, the TPL block of code specified by the fourth argument is entered followed by an unconditional jump to the identifier specified by the fifth argument.

Assignment

Assignment to modify the value of a TPL identifier with a new value of the same type is accomplished by:

```
UPDATE(<loc>,<value>)
```

No-operations

The following TPL instructions are provided to declare the structure of the program but they do not correspond to effective target-machine code:

> **no-operation:** `NOP()`, used by front-ends to code forward jumps
>
> **start-block:** `START`, marks the beginning of a new TPL block of scope
>
> **block-end:** `END`, marks the end of the current TPL block of scope
>
> **procedure-end:** `PROCEND`, marks the end of the current procedure body

[13]The runtime may have other threads doing incremental garbage collection, when appropriate.

From	To	Operation	Meaning
INT	DOUBLE	`<id> := DOUBLE.INT(<value>)`	the equivalent real number
INT	BOOL	`<id> := BOOL.INT(<value>)`	*FALSE* if *0*, *TRUE* if *1*, or an error
INT	BITS	`<id> := BITS.INT(<value>)`	the equivalent in base 2
INT	PIXEL	`<id> := PIXEL.INT(<value>)`	the equivalent *PIXEL*, or an error
INT	CHARS	`<id> := CHARS.INT(<value>)`	the equivalent in base 16
DOUBLE	INT	`<id> := INT.DOUBLE(<value>)`	the equivalent integer, or an error
BOOL	INT	`<id> := INT.BOOL(<value>)`	*0* if *FALSE*, *1* if *TRUE*
BITS	INT	`<id> := INT.BITS(<value>)`	the equivalent in base 10
BITS	PIXEL	`<id> := PIXEL.BITS(<value>)`	the equivalent *BITS*, or an error
BITS	CHARS	`<id> := CHARS.BITS(<value>)`	the equivalent in base 16, or an error
PIXEL	INT	`<id> := INT.PIXEL(<value>)`	the equivalent in base 10, or an error
PIXEL	BITS	`<id> := BITS.PIXEL(<value>)`	the equivalent *BITS* value
CHARS	INT	`<id> := INT.CHARS(<value>)`	the equivalent in base 10, or an error
CHARS	BITS	`<id> := BITS.CHARS(<value>)`	the equivalent in base 2

Table 7.2: Coercion Operations in TPL

Other instructions

Finally, there are the following operations:

move: `<id> := MOVE.<mc-type>(<value>)`, constructs a copy of the value passed as argument and binds it to *id*[14]

label: `LABEL(<id>)`, marks a TPL instruction to be used only by *JUMP()* in a *BRA()* instruction that must belong to the same TPL block; that is, from a *BRA()* the flow of control can only go to the closer outer scope.

7.8.14 Type Conversions

As there is no implicit coercion in TPL, coercion between values of different TPL types must be done explicitly by calling a TPL instruction of the form:

```
<id> := <mc-type>.<mc-type>(<value>)
```

as, for example, in:

```
R22 := DOUBLE.INT(123)
```

which should be read as: construct a value of type *DOUBLE* given a value of type *INT* and bind it to the identifier *R22*. The set of operations available are described in Table 7.2.

In the operations that bind a value of type *BOOL*, that value is the literal *FALSE* if the argument is *0* otherwise it will be *TRUE*. If the value to be coerced to a value of the new type does not fit, a runtime error is generated. For example, when a value of more than 24 bits is coerced to *PIXEL*, or if a value of type *BITS*, which is not multiple of 8, is coerced to type *CHARS*.

It is the case of coercing values with more than 24 bits to *PIXEL*, or values of type *BITS* not multiples of 8 to type *CHARS*.

[14]For base types it is equivalent to the use of the corresponding constructor for that type; for aggregate values it copy its pointer (shallow copy).

7.8.15 Standard Library

In order to interact with the environment and achieve operating system and architecture neutrality, a standard library is provided. It includes operations to deal with I/O, for example. Those procedures can be collected into a value of type *MAP*(see Section 8.5). The following program declares procedures to write an integer value and to write a string value.

```
R3 := PROC("INT->VOID",[A0],           % writeInt
  START
  VOID := CALLCC("printf",[" %d ",A0],"")
  END
  PROCEND)
...
R4 := PROC("CHARS->VOID",[A1],         % writeString
  START
  VOID := CALLCC("printf",[" %s ",A1],"")
  END
  PROCEND)
```

7.9 Conclusions

This chapter presented TPL as a language with a concrete syntax, a set of types, a set of predefined instructions and scope rules. The design decisions will be best understood after the illustration of TPL usage that will be presented in Chapter 8. It must be noted though, that TPL does not occur is this form and that the representation presented is only for human readability. In fact, TPL exists only as data structures shared by the high-level language front-end (2TPL) and the other components of the language framework. As 2TPL needs to fill the implementation internal data structures and the implementation compiler enforces the correct types, there is no need to have an explicit parsing and type-checking phase of TPL (CHECK component) in the prototype. The TPL programs could be statically type-checked, except for identifiers bound by *PROJ()* clauses, where a dynamic type-check is needed.

It should be noted that the set of operations presented here is by no means complete and also that TPL should be easily extensible in order to fulfil the requirements of longer-term persistence. The minimum set should include, for example, types for other base values such as *BYTE*, *SHORT*, *LONG* and *SINGLE* and the corresponding operations, monitors to enable concurrent programming, a statement to throw an exception that could serve in the event of a runtime error, a statement to set a breakpoint for debugging and support for different threads of execution. All of these are considered outside the scope of this thesis.

The following chapters will show how TPL can be used as a target for front-ends and present some transformations which can be performed in TPL in order to achieve better characteristics and to prepare TPL for target machine code generation.

Chapter 8

Compiling to TPL

This chapter continues the description of the language compilation framework. It shows a typical usage of TPL as a target language by front-ends to the source languages anticipated. For each relevant language construct, a small example is presented in the COREL language together with the corresponding TPL program. The translation rules involved in the process are enunciated. Finally, the front-end implemented in the prototype is briefly described.

8.1 Introduction

In order to show that TPL can be used to support the language constructs identified in Sections 3.5 to 3.8, several small programs are passed through the prototype language framework. Following sections will illustrate general language features, language control structures, recursion, aggregate types, first-class procedures, incremental binding, orthogonal persistence, polymorphism, union types and infinite union types. Except when explicitly stated, examples are taken directly from programs produced by the language framework. For each example, a source language COREL program is presented[1] and explained together with the corresponding TPL program[2] generated by 2TPL. The pretty-printer for TPL programs (component PP of the language framework prototype) follows the internal graph representation and prints each node internal number, inside square brackets, by the order of the visit. In the same line, the corresponding TPL instruction is printed in accordance with the syntax given in Chapter 7. For each high-level language construct, a translation rule to TPL is enunciated and its use shown in the corresponding translation example.

All TPL programs generated by this front-end have the same preamble and the same epilogue. The preamble initialises the runtime system (first instruction compulsory in any TPL program), fetches the root of persistence and the predefined procedure *matchType* (which matches two type representations as described in Section 8.5):

[1]COREL syntax and type rules are presented in Appendix B.

[2]See the abstract syntax in Appendix A.

```
[1]    INIT()                                # initialise runtime
[2]    START
[3]    R1 := PROOT()                         # root of persistence
[4]    R2 := LOOKUP:RECORD("matchType",R1)   # get matchType routine
[5]    R3 := LOOKUP:RECORD("matchConst",R1)  # get matchConst routine
```

In this dissertation, TPL programs are annotated manually with comments preceded by # when they are needed for better understanding[3]. TPL instruction 3 binds to *R1* the map which corresponds to the distinguished root of persistence[4]. The map can then be searched to get the required bindings, as is the case of the predefined procedures from the standard library. The epilogue closes down the runtime system by calling the appropriate TPL instruction:

```
[12]   STABLE()                              # checkpoint the store
[13]   END
[14]   CLOSE()                               # close down runtime
```

and the *START* and *END* limit the outer TPL block. The preamble and epilogue are omitted from the TPL examples which will be presented next.

8.2 General Language Features

The prototype language framework is able to accept a minimum set of general language features that are needed to test the PHOL's distinguishing features: orthogonal persistence, first-class procedures, polymorphism and reflection. This section illustrates some of those needed features, namely, declarations, assignment, control structures, arithmetic and boolean expressions, recursion and aggregate types.

8.2.1 Declaration, Assignment and Arithmetic Expressions

To illustrate the translation of declaration, assignment and arithmetic expressions, consider the following COREL program:

```
[1]    let a:= 3
[2]    let b= a-1
[3]    let c:= "Hi there"
[4]    a:= a+b*10
```

Constancy in COREL is signalled by the use of =, which establishes a R-value binding (as in *b*); otherwise, if := is used the name is variable, that is, a binding to an L-value is established (as with *a* or *c*). Each *let* clause adds a binding to an environment and so, by line 4 of the program, the lexical environment has the following bindings:

```
{{a,3,int,variable}, {b,2,int,constant}, {c,"Hi there",string,variable}}
```

and the front-end is able to decide if an assignment to the variable *a* is allowed.

The translation to TPL is trivially done using the following translation rules:

[3]The numbers on the left between square brackets serve only to help in the description of the programs, when referring to the lines.

[4]See Section 8.6.

TR 1 (declaration) *declarations are translated to a call to the constructor for the corresponding TPL types or, if there is an expression involved, a call to an operation*

TR 2 (expression) *expressions are translated to calls to appropriate TPL operations in an order which respects the precedence of operators involved*

TR 3 (assignment) *assignments are translated into an UPDATE() instruction with the location to be updated and the new value as parameters*

The compilation using 2TPL produces an internal representation in TPL. The relevant part can be pretty-printed as:

```
[6]     R4 := INT(3)                      % a     # R4 <- 3 (bind INT)
[7]     R5 := MINUS.INT(R4,1)             % b     # R5 <- R4-1 (bind INT)
[8]     R6 := CHARS("Hi there")           % c     # R6 <- "Hi there"
[9]     R7 := MULT.INT(R5,10)
[10]    R8 := PLUS.INT(R4,R7)
[11]    UPDATE(R4,R8)                             # R4 <- R8 (assign)
```

For the *INT* and *STRING* values involved in this example, there are matching TPL base types; when that is not the case, the front-end has to find a way of representing the desired set of values and operations using the TPL repertoire provided. It should be noted that the source program identifiers are known in TPL in order to enable proper error messages. They are printed in comments preceded by %, at the end of the corresponding TPL binding instructions, as in instructions 6, 7 and 8.

8.2.2 Control Structures and Boolean Expressions

This section presents examples which illustrate the translation of high-level control structures and boolean expressions into TPL.

To represent the value of a boolean expression, two techniques may be used [Aho *et al.*, 1986]:

- to encode the *TRUE* and *FALSE* truth values numerically and evaluate the expressions analogously to arithmetic expressions; or
- to represent the result of the expression by flow-of-control in the position it reaches in the code sequence.

2TPL uses the first technique which gives the high-level language COREL a strict semantics, that is, a boolean expression is always completely evaluated even when its final value is discovered[5]. That's not the case of the languages C and Napier88, for example. For this class of languages, the second technique can be used by the front-end to produce short-circuit code using the *BRA()* conditional instruction.

To translate the following program, involving an *if-then-else* clause with a boolean expression:

[5]It should be noted that optimisations may simplify the expression and avoid complete evaluation.

```
[1]     let a:= 20
[2]     let b= true
[3]     a:= if (a>5) or (a=2) and b then 10 else 2      !* block expression
```

The following rule for this control structure may be used:

TR 4 (if-then-else) *if-then-else clauses are translated to a BRA() instruction with TPL blocks corresponding to the then and else clauses both continuing with the following clause of the program*

Together with the rule for expressions; the program translates to:

```
[6]     R4 := INT(20)                   % a
[7]     R5 := BOOL(TRUE)                % b
[8]     R6 := GT.INT(R4,5)
[9]     R7 := EQ.INT(R4,2)
[10]    R8 := AND.BOOL(R7,R5)
[11]    R9 := OR.BOOL(R6,R8)
[12]    R10 := INT(0)                           # result of block expression
[13]    BRA(R9,
[14]      START
[15]      UPDATE(R10,10)
[16]      END,
          JUMP(L17),
[22]        START
[23]        UPDATE(R10,2)
[24]        END,
            JUMP(L17))
        LABEL(17)
[17]    NOP
[18]    UPDATE(R4,R10)
```

In order to represent the block-expression in the source program at line 2, a new variable *R10* is declared with the required type and a dummy value (at TPL instruction 12) which is afterwards updated accordingly to the value of the boolean expression by one of the branches of the *BRA()*. Its value is used at instruction 18 to update the value of *R4* (the location for *a*). The use of a *NOP* instruction to represent the instruction after a *BRA()* should be noted. It is a mark used by 2TPL to implement forward jumps and can be removed by the optimisation phase of the language compilation framework.

In order to illustrate the translation of *while* clause, consider the following program:

```
[1]     let a:= 5
[2]     while a>=2 do a:= a-1
```

and the translation rule:

TR 5 (while) *the control structures represented by a while clauses translates to a BRA() instruction with a label before the instruction that computes the value of the variable which controls the loop and a TPL block to be executed while the condition remains true*

The program may be translated to TPL and become:

```
[6]     R4 := INT(5)                         % a
        LABEL(7)
[7]     R5 := GTE.INT(R4,2)
[8]     BRA(R5,
[9]       START
[10]      R6 := MINUS.INT(R4,1)
[11]      UPDATE(R4,R6)
[12]      END,
          JUMP(L7),
[13]        START
[14]        END,
            JUMP(L15))
        LABEL(15)
[15]    NOP
```

8.2.3 Recursion

Recursive control structures supported by recursive or mutually recursive procedures can be used in COREL. As an example, consider the following factorial program:

```
[1]     let fact:= proc( i: int -> int ); 0
[2]     fact:= proc( n: int -> int ); if n=1 then 1 else n* fact( n-1 )
```

and the rule to translate procedures:

TR 6 (procedure) *procedures are translated to PROC() with the TPL type for the arguments and result, the list of parameters, the TPL code which corresponds to the body of the procedure and PROCEND*

The corresponding TPL program, is:

```
[6]     R4 := PROC("INT->INT",[A1],          % fact  # dummy value
[7]       START
[8]       R0 := INT(0)
[9]       END
[10]      PROCEND)
[11]    R10 := PROC("INT->INT",[A2],                 # new literal for fact
[12]      START
[13]      R5 := EQ.INT(A2,1)
[14]      R6 := INT(0)
[15]      BRA(R5,
[16]        START
[17]        UPDATE(R6,1)
[18]        END,
            JUMP(L19),
[23]          START
[24]          R7 := MINUS.INT(A2,1)
[25]          R8 := CALL(R4,[R7])
[26]          R9 := MULT.INT(A2,R8)
[27]          UPDATE(R6,R9)
[28]          END,
              JUMP(L19))
          LABEL(19)
[19]      NOP
[20]      R0 := INT(R6)
[21]       END
[22]       PROCEND)
[29]     UPDATE(R4,R10)                              # update fact
```

Recursive and mutually recursive procedures can be achieved in TPL by updating the value of a variable of *PROC* type. To be in scope, an identifier is declared with a dummy value so it can then be used in the body of a new *PROC* value to be assigned to the same identifier. Instructions 6 to 10 declare the *PROC* value *R4* initialised with a dummy value. Instructions 11 to 28 declare a new *PROC* value *R10* with the same type of *R4* and referring to *R4* in the body. If *R5* has the value *TRUE*, the basis rule is selected and the recursion ends with the result *1* in *R6* (look at instructions 17 and 20); otherwise, the induction rule decrements the argument at instruction 24 and calls *R4*, saving the result in *R6* (look at instruction 27). Finally, instruction 29 assigns the new *PROC* value to the location for *fact*.

The result of a procedure is always bound to the identifier *R0* in the last instruction of the external block before *PROCEND*, as in instruction 20. *R0* is the exception to the unique-binding rule, as it is used more than once, but in different scopes. This way the result is identified and can be moved to the correct location after a *CALL()* to the procedure.

8.2.4 Aggregate Types

In COREL, data values may be grouped into larger aggregate data values. If all the constituents are of the same type, a vector has to be used; otherwise a labelled cross-product or structure may be used. The source language may include image types to represent collections of pixels and collections of bindings are also aggregates which will be described in Section 8.4.

All aggregate data values have **pointer semantics**, that is, for each constructed data value a pointer to the location that holds the value is also created. The data value is held on the heap as a boxed value and is always referred to by the pointer which is also used for equality tests. On every assignment using an aggregate data value, the implicit pointer is copied to the new location. Pointer semantics fit well with TPL equality by identity.

Labelled Cross-products

Consider the following COREL program:

```
[1]     let a:= struct( f1= struct( ff1:= 5);
[2]                     f2= proc( b: bool -> int); if b then 5 else 10;
[3]                     f3= false )
[4]     a( f1 )( ff1 ):= 8
[5]     let b= a( f2 )( a( f3 ) )
```

which constructs a structure with three fields: the first (*f1*) is also a structure with one field (*ff1*) initialised to *5*, the second (*f2*) is a procedure value and the third (*f3*) a boolean value. The values are used by indexing with the field name; in line 4 a new value is assigned to the field *ff1* and in line 5, *b* is initialised with that value.

The rule which can be used in the translation of structures, is:

TR 7 (structure) *structures are translated to RECORD() with a list of the field values by the order given* and the program may be coded in TPL as:

```
[6]     R4 := RECORD([5])                                # f1 of a
[7]     R6 := PROC("BOOL->INT",[A1],                     # f2 of a
[8]       START
[9]       R5 := INT(0)                                   # result of block expression
[10]      BRA(A1,
[11]        START
[12]        UPDATE(R5,5)
[13]        END,
            JUMP(L14),
[18]          START
[19]          UPDATE(R5,10)
[20]          END,
              JUMP(L14))
          LABEL(14)
[14]      NOP
[15]      R0 := INT(R5)                                  # result
[16]      END
[17]      PROCEND)
[21]    R7 := RECORD([R4,R6,FALSE])            % a       # construct structure
[22]    R8 := MOVE.RECORD(R7!0)                          # address of field 1 of a
[23]    UPDATE(R8!0,8)                                   # a( f1 )( ff1 ) <- 8
[24]    R9 := CALL(R7!1,[R7!2])                % b       # call a( f2 ) with a( f3 )
```

R4 corresponds to *f1*, *R6* to the *f2* procedure literal and *R7* to the structure *a*. Before the *RECORD()* construct can be applied, the values must be ready to initialise the fields of the new data value. Therefore, all values that are not TPL literals are first bound to identifiers (e.g. *R4* and *R6* in instructions 6 and 7 respectively). The use of *MOVE.RECORD()* to access field *ff1* through two indirections at instructions 22 and 23 should be noted. *RECORD* fields are not named in TPL and their implicit order is used instead with the first field at index 0.

There is no notion of constancy for *RECORD* fields in COREL. If constancy is required, the front-end must take steps to check the constancy of fields. If constancy is statically determinable, the entire check can be done by front-ends. If it depends on the value as in Napier88, then the front-end may pack in a *RECORD* the value plus a constancy bitmap represented in values of type *BITS* and plant code to check for field constancy.

Vectors

To construct a vector in COREL, the lower bound, upper bound and initialising values must be provided, as in:

```
[1]     let s= struct( a= 1; b= true )
[2]     let v1= vector 0 to 9 of s(a)
[3]     let v2= vector 0 to 1 of s
[4]     v2(1)(a):= v1(5)
```

Using the translating rule:

TR 8 (vector) *vectors translate to VECTOR() with the lower bound, upper bound and initialisation value specified*

it is straightforward to translate the program to:

```
[6]     R4 := RECORD([1,TRUE])               % s
[7]     R5 := VECTOR(0,9,R4!0)               % v1
[8]     R6 := VECTOR(0,1,R4)                 % v2
[9]     R7 := MOVE.RECORD(R6@1)                           # address of v2( 1 )
[10]    UPDATE(R7!0,R5@5)                                 # v2( 1 )( a ) <- v1( 5 )
```

Indexing vector elements is performed in the same way as with structures by using the instruction *MOVE.RECORD()* (as in instruction 9, used to index *v2(1)(a)*). Checking for constancy must also be arranged by front-ends, as discussed above.

Images

Images were not included in the COREL language accepted by the implemented prototype. Nevertheless, the building block for images are present: types *PIXEL* for literals and *BITS* to support the bitmaps for each plane of the image. The *RECORD()* constructor may be used to aggregate plane bitmaps with other information, such as colour maps. The operations on *BITS* could be used to implement the raster image operations. An example of image support is in the PAM ([Brown *et al.*, 1994]) designed to support Napier88 values of type image and corresponding operations. As always the decision is:

1. provide a special data type with operations; or
2. leave the task of dealing with images using the existing data types to the front-end.

At the moment, TPL does not provide a type for images. The data type for images has to conform to hardware constraints regarding bit layout or to some image standard that is supported. The second option may not give architecture neutrality as the front-end may choose one particular representation for images with no warranty of future support.

8.3 First-class Higher-order Procedures

Both the high-level language COREL and the TPL internal representation are block-structured languages where procedures can be nested, passed as parameters and, more generally, where procedures are first-class citizens. Therefore, the translation of procedures to TPL using the rule TR 6 is straightforward. This section illustrates the compilation of nested procedures with free variables and the compilation of first-class procedures.

8.3.1 Nested Procedures

To illustrate the compilation of nested procedures with free-variables, consider the following program:

```
[1]     let x = proc()                   !* lexical level 0
[2]             begin
[3]               let a:= 1
[4]               let p= proc( -> int ); a     !* lexical level 1
[5]               let b= 2
[6]               let q= proc( -> int )        !* lexical level 1
```

```
[7]                begin
[8]                  let a:=p()          !* caller at level 1/callee at 1
[9]                  a+b
[10]               end
[11]        let c= q()
[12]        let r= proc( -> int )        !* lexical level 1
[13]               begin
[14]                 let s= proc( -> int )!* lexical level 2
[15]                        begin
[16]                          p()        !* caller at level 2/callee at 1
[17]                        end
[18]                 s()                 !* caller at level 1/callee at 2
[19]               end
[20]        let d= r()
[21]      end
```

Procedure *x* is declared at lexical level 0, procedures *p*, *q* and *r* are declared at lexical level 1 and procedure *s* is declared at level 2. This program contains all possible combinations of caller/callee levels:

same level the first call to *p* in the body of procedure *q* (at line 8); caller and callee have the same environment

callee deeper the call to *s* in the body of procedure *r* (at line 18); the environment of the callee is the same as the environment of the caller plus the current activation record

caller deeper the second call to *p* in the body of procedure *s* (at line 16); the environment of the callee is only a part of the environment of the caller plus the current activation record

The block-structure of TPL is used directly by 2TPL to represent programs and this way postponing for a later stage all memory allocation decisions. Access to non-locals by following links, using a display, or other strategy will be decided at the time memory allocation is decided (see Section 10.3). For each source program variable, instead of a level plus displacement to represent the access, it is enough to have the corresponding TPL variable in the 2TPL symbol table entry, as all TPL identifiers are unique over the scope of the entire program.

```
[6]     R16 := PROC("->VOID",[],            % x
[7]       START
[8]       R4 := INT(1)                      % a
[9]       R5 := PROC("->INT",[],            % p
[10]        START
[11]        R0 := INT(R4)
[12]        END
[13]        PROCEND)
[14]      R6 := INT(2)                      % b
[15]      R9 := PROC("->INT",[],            % q
[16]        START
[17]        R7 := CALL(R5,[])               % a     # call p
[18]        R8 := PLUS.INT(R7,R6)
[19]        R0 := INT(R8)
[20]        END
[21]        PROCEND)
[22]      R10 := CALL(R9,[])                % c     # call q
R14 := PROC("->INT",[],             % r
  START
  R12 := PROC("->INT",[],           % s
    START
    R11 := CALL(R5,[])                      # call p
    R0 := INT(R11)
    END
    PROCEND)
  R13 := CALL(R12,[])                       # call s
  R0 := INT(R13)
  END
  PROCEND)
R15 := CALL(R14,[])                 % d     # call r
END
PROCEND)
```

It should be noted that the source program free-variables are free variables in the TPL program, as well (e.g. variable *a* for procedure *p*).

8.3.2 First-class Procedures

As discussed in Section 3.5, first-class procedures together with true block scope pose interesting problems in the implementation. The following COREL program is similar to the program of Figure 3.2:

```
let p= proc( i: int -> proc( int -> int ) )
       begin
         let a= 10
         proc( f: int -> int ); f*i*a
       end
let p2= p( 2 )
let c= p2( 5 )
```

P2 is a procedure that accesses the free-variable *a* and is applied to *5* to obtain the integer value which will be bound to *c*. *a* is used to calculate the result of *p2(5)*, but it is not any more in scope.

Because procedures are first-class in TPL and may have free-variables, by using the rule TR 6 the translation is straightforward:

```
R8 := PROC("INT->PROC(INT->INT)",[A1],  % p     # proc literal
  START
  R4 := INT(10)                         % a
  R7 := PROC("INT->INT",[A2],                   # unnamed proc literal
    START
    R5 := MULT.INT(A2,A1)
    R6 := MULT.INT(R5,R4)
    R0 := INT(R6)                               # result of unnamed
    END
    PROCEND)
  R0 := MOVE.PROC(R7)                           # result of p
  END
  PROCEND)
R9 := CALL(R8,[2])                      % p2    # call p
R10 := CALL(R9,[5])                     % c     # call p2
```

A procedure literal is constructed in instructions 9 to 15 and it is the result of *R8* (the location for *p*). The block structure of the original program was preserved in the TPL program and the block retention mechanism mentioned in Section 3.5 still needs to be achieved.

8.4 Collections of Bindings

Environments in COREL provide support for control of names and allow incremental system construction, following the ideas of the language Napier88. While COREL structures are labelled cross-products determined statically, environments (values of type *env*) are infinite unions of labelled cross-products where bindings can be added or removed dynamically. Also, in contrast with structures, two values of type *env* are equivalent irrespective of the particular set of bindings they contain. The semantics of composition of environments is equivalent to the familiar block structure semantics of programming languages[6].

To illustrate the compilation of *env* values, consider the following program:

```
[1]    use PS() with
[2]      writeInt: proc( int );
[3]      environment: proc( -> env ) in
[4]    begin
[5]      let env1= environment()
[6]      in env1 let i:= 10
[7]      in env1 let p= proc( a: int -> int ); 2*a
[8]      use env1 with i: int; p: proc( int -> int ) in
[9]      begin
[10]       i:= 1000
[11]       writeInt( p(i) )          !* writes out 2000
[12]     end
[13]     in env1 let j:= 100
[14]     use env1 with constant j: int in
[15]       writeInt( j )             !* writes out 100
[16]   end
```

In COREL, *PS()* returns a value of type *env* which contains all predefined procedures, such as: error reporting procedures, procedures to interface the operating system and others (c.f. Napier88 Standard Library [Kirby *et al.*, 1994]). The predefined procedure *environment*, when applied, returns a new empty environment. The *use-in* clause (e.g. lines 1, 8 or 14 of the program) projects bindings from environments into scope. Only a partial match is needed as the environment may contain other bindings as well. Its syntax is:

```
use <clause> with <signature> in <clause>
<signature> ::= [constant] <id-list>:  <type-id> [; <signature>]
```

The effect of the *use* clause of line 3 is that the identifier *environment* will be in scope with the signature specified (only the type in this case). As a binding has associated a constancy property, in COREL it is possible request for a particular constancy value. In the use clause of line 14, both the type and constancy are specified and must be verified at runtime. It should be noted that *j* is used as a constant and was declared as a variable; the opposite is not possible though. The program is still strongly typed but not entirely statically. At runtime

[6]For a more complete description of Napier88 environments see [Dearle, 1989].

the corresponding value must exist in the environment with the declared type and constancy, and, if not, a runtime error will be raised. This mechanism allows for type-checking to be delayed until execution time at chosen program points. In order to permit this dynamic type-checking, the front-end type representation must be included in the stored value and the type equivalence routine used by the front-end, must be accessible during execution.

The *in-let* clause (e.g. lines 6 and 7 of the program) is used to introduce a new binding into an environment. Its syntax is:

```
in <clause> let <object-init>
<object-init> ::= <id> <init-op> <clause>
```

After line 7 the environment *env1* will contain the set of bindings:

```
{{i,10,int,variable},{p,proc ...,proc(int->int),constant}}
```

It should be noted that the lexical environment is not affected by the clauses of lines 6 and 7. As with assignment, bindings to environments can be by R-value, if the symbol = is used as in line 13, or to L-value, if the symbol := is used as in line 6.

The translation of this program may be done using the following rules:

TR 9 (in-let) *in-let clauses are translated to INSERT.RECORD() with the name, a RECORD(), containing the value, the front-end type representation and the constancy, and the map which corresponds to the environment*

TR 10 (use-in) *use-in clauses are translated to LOOKUP:RECORD() with the name and the environment (in order to get the boxed attributes of the name provided), a type-checking operation and sometimes a constancy test*

One representation in TPL may be the following:

```
[3]     R1 := PROOT()                                 # root of persistence
[4]     R2 := LOOKUP:RECORD("matchType",R1)
[5]     R3 := LOOKUP:RECORD("matchConst",R1)
[6]     R4 := LOOKUP:RECORD("writeInt",R1)            # from MAP
[7]     VOID := CALL(R2,[R4!1,"procedure(int->VOID)"]) # check its type
[8]     R5 := LOOKUP:RECORD("environment",R1)         # from MAP
[9]     VOID := CALL(R2,[R5!1,"procedure(->env)"])    # check its type
[10]    R6 := CALL(R5!0,[])                 % env1    # new empty MAP
[11]    R7 := INT(10)                                 # for i
[12]    R8 := RECORD([R7,"int",FALSE])                # value, type, constancy
[13]    INSERT.RECORD("i",R8,R6)                      # insert i into MAP
[14]    R10 := PROC("INT->INT",[A1],                  # for p
[15]      START
[16]      R9 := MULT.INT(2,A1)
[17]      R0 := INT(R9)
[18]      END
[19]      PROCEND)
[20]    R11 := RECORD([R10,"procedure(int->int)",TRUE]) # value, type, constancy
[21]    INSERT.RECORD("p",R11,R6)                     # insert p into MAP
[22]    R12 := LOOKUP:RECORD("i",R6)                  # i from MAP
[23]    VOID := CALL(R2,[R12!1,"int"])                # type-check value
[24]    R13 := LOOKUP:RECORD("p",R6)                  # p from MAP
[25]    VOID := CALL(R2,[R13!1,"procedure(int->int)"]) # type-check value
[26]    UPDATE(R12!0,1000)                            # update i
```

```
[27]    R14 := CALL(R13!0,[R12!0])              # call p(i)
[28]    VOID := CALL(R4!0,[R14])                # write result
[29]    R15 := INT(100)                         # for j
[30]    R16 := RECORD([R15,"int",FALSE])        # value, type, constancy
[31]    INSERT.RECORD("j",R16,R6)               # insert j into MAP
[32]    R17 := LOOKUP:RECORD("j",R6)            # from MAP
[33]    VOID := CALL(R2,[R17!1,"int"])          # type-check value
[34]    VOID := CALL(R3,[R17!2,TRUE])           # check constancy
[35]    VOID := CALL(R4!0,[R17!0])              # write result
```

2TPL chose to represent source program types as strings by using the *CHARS* constructor and therefore the equality between two types resumes to equality between strings. Should a more elaborate type representation be needed (see [Connor, 1991, Cutts, 1993]), the front-end can store the routines to deal with it in the standard library; namely, the type equality routine and the routine to construct a type representation. When constancy is specified in the *use* clause, then a test is planted together with the test for the type (at instructions 33 and 34). 2TPL packs all values with their type representation and constancy in a *RECORD* (as in instruction 12) before insertion into the map. This way, the map contains only boxed values. These values are accessed through one indirection, which preserves the semantics of assignment, in case they are in scope more then once (i.e. if they are aliased to more then one local identifier).

In the source language COREL there are two other clauses to manipulate environments: the *contains* clause, to test if a binding exists in an environment and the *drop-from* clause, to remove bindings. Their syntax is:

```
<clause> contains [constant] <id> [:<type-id>]
drop <id> from <clause>
```

To illustrate the translation of these two clauses, consider:

```
[1]     use PS() with
[2]       writeInt: proc( int );
[3]       environment: proc( -> env ) in
[4]     begin
[5]       let env2= environment()
[6]       in env2 let i= 100
[7]       use env2 with i: int in
[8]       begin
[9]         writeInt( i )              !* writes out 100
[10]        if env2 contains constant i: int do
[11]          drop i from env2
[12]        writeInt( i )              !* writes out 100
[13]      end
[14]    end
```

The program uses the *contains* clause in line 10 to check if integer *i* exists in environment *env2* and in line 11 removes that binding from the environment. It should be noted that identifier *i* will be still in scope after being dropped from the environment *env2* and can be used, as in line 12. The translation rules for these two clauses are:

TR 11 (contains) *contains clauses are translated to EXISTS() with the name and the map which corresponds to the environment given; if the result is TRUE then a test for constancy* *and type is performed in case they are specified; the result is the logical AND of all those three tests*

TR 12 (drop) *drop clauses are translated to REMOVE() with the name and the map*

One representation in TPL may be the following:

```
[3]    R1 := PROOT()                                  # preamble
[4]    R2 := LOOKUP:RECORD("matchType",R1)
[5]    R3 := LOOKUP:RECORD("matchConst",R1)
[6]    R4 := LOOKUP:RECORD("writeInt",R1)
[7]    VOID := CALL(R2,[R4!1,"procedure(int->VOID)"])
[8]    R5 := LOOKUP:RECORD("environment",R1)
[9]    VOID := CALL(R2,[R5!1,"procedure(->env)"])
[10]   R6 := CALL(R5!0,[])                   % env2   # new empty MAP
[11]   R7 := INT(100)                                 # for i
[12]   R8 := RECORD([R7,"int",TRUE])                  # box i
[13]   INSERT.RECORD("i",R8,R6)                       # insert into MAP
[14]   R9 := LOOKUP:RECORD("i",R6)                    # from MAP
[15]   VOID := CALL(R2,[R9!1,"int"])                  # type-check value
[16]   VOID := CALL(R4!0,[R9!0])                      # write integer
[17]   R10 := EXISTS("i",R6)                          # test existence
[18]   BRA(R10,
[19]     START
[20]     R11 := LOOKUP:RECORD("i",R6)                 # get attributes
[21]     R12 := EQ.BOOL(TRUE,R11!2)                   # test constancy equality
[22]     R13 := AND.BOOL(R10,R12)                     # update result
[23]     UPDATE(R10,R13)
[24]     R14 := EQ.CHARS("int",R11!1)                 # test type equality
[25]     R15 := AND.BOOL(R10,R14)                     # update result
[26]     UPDATE(R10,R15)
[27]     END,
         JUMP(L28),
[40]       START
[41]       END,
           JUMP(L28))
       LABEL(28)
[28]   NOP
[29]   BRA(R10,
[30]     START
[31]     REMOVE("i",R6)                               # remove i from MAP
[32]     END,
         JUMP(L33),
[38]       START
[39]       END,
           JUMP(L33))
       LABEL(33)
[33]   NOP
[34]   VOID := CALL(R4!0,[R9!0])                      # write integer
```

Environments are represented using values of type *MAP* and operations described in Section 7.8.12. Instruction 8 binds to *R5* a *RECORD* containing the predefined procedure *environment* from the value of type *MAP* which exists at the root of persistence, or raises an error at execution-time if it does not exist in the map. Instruction 10 creates a new environment (*env2*) and binds it to *R6*. Alternatively, the front-end could implement a map from string to values of type *any* using the corresponding TPL types *CHARS* and *INF* with the needed procedures hidden from the programmer in the same way *environment* was achieved in the prototype.

The result of the *contains* clause will be collected in *R10*. If the result is *FALSE*, the *BRA()* conditional continues immediately with instruction 28, followed by instruction 33. Otherwise, and if the *contains* clause has the constancy or type specifications, then code to perform the appropriate tests will be planted after the *BRA()*. To have access to the constancy and type attributes, a *LOOKUP()* instruction is first planted as shown in instruction 20. In this example a test for constancy is required, at instruction 21, and of type, at instruction 24.

8.5 Standard Library

The front-end uses the root of persistence at the map returned by *PROOT()* (*R1* in 2TPL) to store the standard library. This library contains predefined procedures which encapsulate the interaction with the operating system and the architecture (e.g. *writeInt* or *writeString* presented in Section 7.8.15) and other routines the front-end needs to compile COREL (e.g. *environment* and *matchType*).

The procedure *environment* returns an empty value of type *MAP* and is inserted in the *PROOT()* map by the following TPL program:

```
[3]     R1 := PROOT()
[4]     R2 := PROC("->MAP",[];                  % environment
[5]       START
[6]       R0 := MAP()                              # empty MAP
[7]       END
[8]       PROCEND)
[9]     R101 := RECORD([R2,"proc(->env)",FALSE])   # box with type and constancy
[10]    INSERT.RECORD("environment",R101,R1)       # into MAP
```

and the *matchType* routine used by 2TPL to perform dynamic type-checks, is:

```
[25]    R5 := PROC("CHARS,CHARS->VOID",[A2,A3],  % matchType
[26]      START
[27]      R6 := NEQ.CHARS(A2,A3)                   # arguments different ?
[28]      BRA(R6,
[29]        START
[30]        VOID := CALL(R4,["Type mismatch"])     # write string
[31]        RESTART()                              # rollback
[32]        CLOSE()                                # close down runtime
[33]        END,
            JUMP(L34),
[37]          START
[38]          END,
              JUMP(L34))
          LABEL(34)
[34]      NOP
[35]      END
[36]      PROCEND)
[39]    R104 := RECORD([R5,"proc(string,string)",FALSE])# value, type, constancy
[40]    INSERT.RECORD("matchType",R104,R1)         # into MAP
```

In the case the two types supplied do not match, the error is signalled, the persistent object store is brought to the last stable point by instruction 31 and the program terminates.

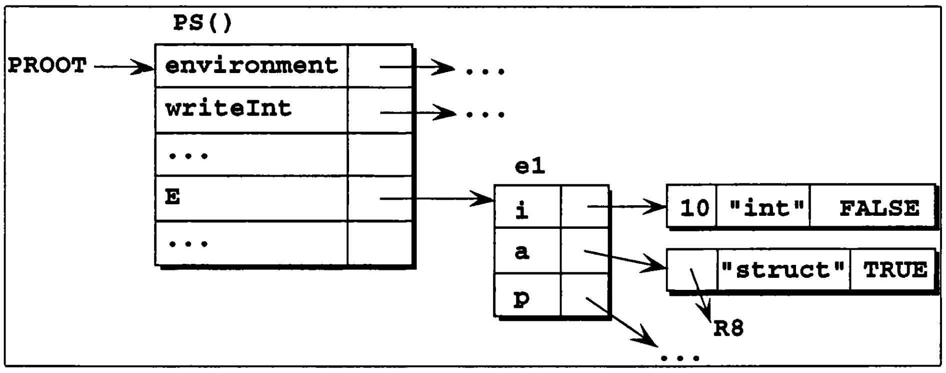


Figure.8.1: Persistent Store Graph

8.6 Orthogonal Persistence and Incremental System Construction

This section demonstrates one of the crucial features of the high-level languages anticipated, as referred to in Section 1.3.3: orthogonal persistence.

Environments are first-class in COREL and therefore can be used to model a naming convention in the persistent store. Consider the following program:

```
[1]     let ps= PS()
[2]     use ps with environment: proc( -> env ) in
[3]     begin
[4]       let e1= environment()
[5]       in e1 let i:= 10
[6]       in e1 let a= struct( a= 10; b= false )
[7]       in e1 let p= proc( i: int -> int ); i*10
[8]       in ps let E= e1
[9]     end
```

PS() returns a value of type *env* which is the root of persistence. Any value in the transitive closure of this root will be persistent. After executing this program, the store will look like the graph of Figure 8.1. Environment *e1* contains the bindings established in lines 5, 6 and 7 and will be reachable from the *PS()* environment because it is bound with the name *E* in line 8.

As TPL implements persistence by reachability from the map returned by *PROOT()*, it is sufficient to insert values into maps reachable from it to achieve persistence:

TR 13 (persistence) *the use of the environment at the root of persistence PS translates to the use of the value of type MAP which results from calling PROOT()*

The compilation into TPL produces the following program:

```
[3]     R1 := PROOT()
[4]     R2 := LOOKUP:RECORD("matchType",R1)
[5]     R3 := MOVE.MAP(R1)                       % ps     # PROOT MAP
[6]     R4 := LOOKUP:RECORD("environment",R3)
[7]     VOID := CALL(R2,[R4!1,"procedure(->env)"])        # type-check value
[8]     R5 := CALL(R4!0,[])                      % e1     # new empty MAP
[9]     R6 := INT(10)                                     # for i
```

```
[10]    R7 := RECORD([R6,"int",FALSE])                       # value, type, constancy
[11]    INSERT.RECORD("i",R7,R5)                             # into MAP e1
[12]    R8 := RECORD([10,FALSE])                             # for a
[13]    R9 := RECORD([R8,"structure(a,b)",TRUE])             # value, type, constancy
[14]    INSERT.RECORD("a",R9,R5)                             # into MAP e1
[15]    R11 := PROC("INT->INT",[A1],                         # for p
[16]      START
[17]      R10 := MULT.INT(A1,10)
[18]      R0 := INT(R10)
[19]      END
[20]      PROCEND)
[21]    R12 := RECORD([R11,"procedure(int->int)",TRUE])      # value, type, constancy
[22]    INSERT.RECORD("p",R12,R5)                            # into MAP e1
[23]    R13 := MOVE.MAP(R5)                                  # for E
[24]    R14 := RECORD([R13,"env",TRUE])                      # value, type, constancy
[25]    INSERT.RECORD("E",R14,R3)                            # into MAP PROOT
```

To demonstrate orthogonal persistence, consider a program that uses environment E (created by the previous program) and accesses the values of the bindings introduced into E for i, a and p and changes the persistent value for i:

```
[1]     use PS() with
[2]       writeInt: proc( int );
[3]       E: env in
[4]     use E with
[5]       i: int;
[6]       p: proc( int -> int );
[7]       a: struct( a:int; b: bool) in
[8]     begin
[9]       i:= 100
[10]      writeInt( p( a(a)*i ) ) !* writes out 1000
[11]    end
```

Line 6 of the program puts procedure p in scope. Because it is an L-value binding to p (a binding to the location that contains p and not the value of p) any change to the value (e.g. change in the body) by any other program will be visible when this program executes (that is, this program always uses the most recent value of p). It should be noted that this program could be compiled (but not executed!) before the program which creates environment E. If environments are in the transitive closure of the root of persistence, they will persist between program activations together with the bindings they contain. Persistence and procedures as first-class values enable **incremental construction of programs**, as a procedure body can be changed and used by other programs bound by L-value, without the need to recompile those programs. In the current example, the first program produces procedure p and the second applies that procedure. See reference [Connor, 1991] for a detailed example of incremental system construction using environments.

The program compiles to TPL as:

```
[3]     R1 := PROOT()
[4]     R2 := LOOKUP:RECORD("matchType",R1)
[5]     R3 := LOOKUP:RECORD("matchConst",R1)
[6]     R4 := LOOKUP:RECORD("writeInt",R1)
[7]     VOID := CALL(R2,[R4!1,"procedure(int->VOID)"])
[8]     R5 := LOOKUP:RECORD("E",R1)                          # from MAP
[9]     VOID := CALL(R2,[R5!1,"env"])                        # type-check value
```

```
[10]    R6 := LOOKUP:RECORD("i",R5!0)
[11]    VOID := CALL(R2,[R6!1,"int"])
[12]    R7 := LOOKUP:RECORD("p",R5!0)
[13]    VOID := CALL(R2,[R7!1,"procedure(int->int)"])
[14]    R8 := LOOKUP:RECORD("a",R5!0)
[15]    VOID := CALL(R2,[R8!1,"structure(a,b)"])
[16]    UPDATE(R6!0,100)                          # i <- 100
[17]    R9 := MOVE.RECORD(R8!0)                   # address of a
[18]    R10 := MULT.INT(R9!0,R6!0)                # R10 <- a(a) * i
[19]    R11 := CALL(R7!0,[R10])                   # call p
[20]    VOID := CALL(R4!0,[R11])                  # write integer
```

It should be noted that the protection mechanism supported by the type system of the high-level language works for the whole life of the value. This example also shows the linearisation which takes place when COREL programs translate to TPL programs; the argument for *writeInt* is first calculated in instructions 17 to 19 to be ready for the call.

8.7 Polymorphism

Polymorphism can be used in computations that do not depend on the types of the operands by abstracting over details and thus promoting reuse. A definition of polymorphism was presented in Section 3.3.2 on page 33, and the different ways of implementing polymorphism were surveyed in Section 3.6 on page 39.

Polymorphism is not supported in COREL to avoid complex type-checking in the front-end and therefore simplifying the prototype language framework. It can be seen that TPL contains the constructs identified in Section 3.6 needed to support, for example, universal parametric polymorphism. Using values of type *INF*, the implementation described in reference [Morrison *et al.*, 1991] which uses the retention mechanism needed to support first-class procedures can be implemented as follows.

For example, the program

```
let p3= proc[T]( x: T -> T ); x
let p3Int= p3[int]
```

This method uses the type parameter supplied when the procedure is specialised to tag the common representation used by the uniform polymorphic code. The previous program can be compiled as if it was ([Connor *et al.*, 1989]):

```
let p3= proc( t: int -> proc( ? -> ? ) )
            begin
              proc( x: ? -> ? ); x
            end
let p3Int= p3(int)
```

The translation rule for this situation is:

TR 14 (polymorphism) *parameter polymorphic procedures translate to two nested procedures with the polymorphic value represented in an INF value and the type parameter in the closure of the deeper procedure*

This rule can be used to produce the following TPL program for the previous COREL program:

```
[6]    R5 := PROC("INT->PROC(INF->INF",[A1],     % p3   # A1 is the type parameter
[7]      START
[8]      R4 := PROC("INF->INF",[A2],                   # uniform code for procedure
[9]        START
[10]       R3 := INF(A2,A1,0)                          # value, m/c type, tag
[11]       R0 := PROJ(R3,A1,0)                         # project as type given by
[12]       END                                         #  type parameter A1
[13]       PROCEND)
[14]     R0 := MOVE.PROC(R4)                           # result of p3
[15]     END
[16]     PROCEND)
[17]   R6 := CALL(R5,[INT])                     % p3Int # specialisation
```

The call to *R5* (*p3*) at instruction 17, supplies the type (*INT*) to specialise the generic code for the required type. That parameter is used in instruction 10 to inject the value in the *INF* and in instruction 11 to project the value from the *INF* to a value of type *INT* and the argument *A2* is the *x* argument of *p3*. It should be noted that the *INF* tag value is not needed to support parametric polymorphism.

Using this method, the requirements to support universal parametric polymorphism are:

1. mapping an infinite set of COREL types to the finite set of TPL types;
2. the injection and projection to a common representation in TPL using the *INF* type; and
3. support for higher-order procedures in order to generate, at specialisation time, the procedure with the type parameter as a free-variable.

and all of them can be achieved by the proposed architecture.

8.8 Union Types

Union types or variants represent labelled disjoint sums from the value space. For example, using the Napier88 syntax:

```
type a_union is variant(s:string; i:int)
```

declares a type *a_union* whose values may be strings *s* or integers *i*. To form a value of that type, *variant(s:string; i:int)*, a pair of identifier (tag) and corresponding value must be injected, as in:

```
let v:= a_union(i:99)
```

which binds to the identifier *v* a value of the *a_union* variant type. A value of type *variant* can be tested to have a particular tag by using the *is* or *isnt* clause, as in:

```
v is string
```

which in this case will yield the boolean value *false*. The value of a variant may be projected by using ' and a tag value, as in:

```
v'i
```

which will signal an error at runtime if *v* does not contain a value with tag *s*. In order to achieve static type-checking, there is also a *project* clause that may be used, as in the following program:

```
type a_union is variant(s: string; i: int)

use PS() with writeString: proc( string ); writeInt: proc(int ) in
begin
  let v= a_union(i: 99)
  project v as V onto
        s: writeString( V )
        i: writeInt( V )
  default: writeString("Unexpected value found.")
end
```

For each clause, the identifier *V* has the type of the matched tag and can therefore be used without fear of a runtime error. Further to injection and projection operations, two variant values can be tested for equality. Two variants are equal if they have the same tags and the same values of the same type.

COREL does not provide union types but it can be seen that the TPL *INF* type is enough to support such values. Firstly, the front-end must make a correspondence between the source level tags and integer values and use them consistently. For example, if *0* corresponds to tag *s* and *1* to tag *i* then the previous COREL program can be translated to TPL as:

```
R4 := LOOKUP:RECORD("writeInt",R1)
VOID := CALL(R2,[R4!1,"procedure(int->VOID)"])
R5 := LOOKUP:RECORD("writeString",R1)
VOID := CALL(R2,[R4!1,"procedure(string->VOID)"])
R6 := INF(99,INT,1)                    % v      # value, m/c type, tag
R7 := TAG(R6)                                   # get the tag
R8 := EQ.INT(R7,1)                              # test its value
BRA(R8,
  START
  R9 := PROJ(R6,INT,1)                          # project as an INT
  VOID := CALL(R4,[R9])                         # write integer
  END,
  JUMP(22),
    START
    R10 := PROJ(R6,CHARS,0)                     # project as CHARS
    VOID := CALL(R5,[R10])                      # write string
    END,
    JUMP(22))
LABEL(22)
NOP
```

The translation rule can be enunciated as:

TR 15 (union) *union types translate to INF() with the value, its machine type representation and a tag to represent the branch; the value is projected using the PROJ() instruction*

The translation of *is* or *isnt* clauses is achieved in the same way. It should be noted that the front-end flags an error if tags other then *i* or *s* are used in the source program, therefore there is no need to generate code for the *default* branch in the *project* clause in this example. The operation *EQ.INF()* may be used to test two variant representations for equality.

8.9 Infinite Union Types

Infinite union types are the union of all values in one language. They are universal and extensible union types [Atkinson and Morrison, 1990]. For example, the type *any* in Napier88 fills that rôle. To inject the value *99* into an *any* and bind it to the identifier *a1* the following Napier88 clause may be used:

```
let a1:= any(99)
```

which wraps the value *99* with a representation of its type (*INT*). Two values of type *any* are always type compatible so the decision on what is the type is traded for a restriction on the available operations. In an *any* only equality and assignment operations are allowed, further to the injection and projection operations. It is necessary to project the value to gain access to the other operations. The projection is performed by using the *project* clause, as in the following program:

```
[1]    use PS() with writeString: proc( string ); writeInt: proc(int ) in
[2]    begin
[3]      let a= any( false )
[4]      project a as A onto
[5]        int  : writeInt( A )
[6]        bool : writeString( "Value of type bool" )
[7]      default: writeString( "Unexpected type found." )
[8]    end
```

The representation for the high-level type is decided by the front-end. References [Connor, 1991] and [Cutts, 1993] contain a comprehensive discussion on the subject. The front-end also determines how type equality is decided.

COREL does not provide support for values of type *any* in order to avoid the complex type-checking related problems, but it can be seen that values of this type can be easily represented using *INF*.

The corresponding transition rule is:

TR 16 (any) *infinite union types translate to a value of type INF packed with a high-level language type representation inside a RECORD*

Considering that the front-end uses strings of characters to represent types, a procedure which returns the boolean value *TRUE* if called with two arguments of equivalent types and *FALSE* otherwise, can be easily coded in TPL as:

```
(23)    R5 := PROC("CHARS,CHARS->BOOL",[A2,A3],  % eqType
(24)      START
(25)      R0 := EQ.CHARS(A2,A3)
(26)      END
(27)      PROCEND)
```

Using a representation of high-level types as strings, the previous program may be translated to:

```
[6]     R4 := LOOKUP:RECORD("writeInt",R1)
[7]     VOID := CALL(R2,[R4!1,"procedure(int->VOID)"])
[8]     R5 := LOOKUP:RECORD("writeString",R1)
```

```
[9]    VOID := CALL(R2,[R4!1,"procedure(string->VOID)"])
[10]   R6 := CHARS("int")                              # type representation
[11]   R7 := CHARS("bool")                             # type representation
[12]   R8 := INF(FALSE,BOOL,0)                         # value, m/c type, tag
[13]   R9 := RECORD(R8,R7)                      % a    # box value and type-rep
[14]   R10 := EQ.CHARS([R9!1,R6])                      # type int ?
[15]   BRA(R10,
[16]     START
[17]     R11 := PROJ(R9!0,INT,0)                       # project as int
[18]     VOID := CALL(R4,[R11])                        # write integer
[19]     END,
         JUMP(20),
[21]       START
[22]       R12 := EQ.CHARS(R9!1,R7)                    # type bool ?
[23]       BRA(R12,
[24]         START
[25]         R13 := PROJ(R9!0,BOOL,0)                  # project as bool
[26]         VOID := CALL(R5,["Value of type bool"])   # write string
[27]         END,
             JUMP(31),
[28]           START
[29]           VOID := CALL(R5,["Unexpected type found."]) # default clause
[30]           END
               JUMP(31)
           LABEL(31)
[31]       NOP
[32]       END
           JUMP(20))
       LABEL(20)
[20]   NOP
```

The front-end makes a type representation for the types involved (instructions 10 and 11), constructs the value as an *INF* at instruction 12 and boxes it with the type representation in a *RECORD* value at instruction 13. The type is checked against *int* in instruction 14 and *bool* in instruction 22 and the *INF* value is then projected accordingly (instructions 17 and 25). The *INF* tag is not used in this case and may have any value (e.g. 0), which must be consistently used by the front-end.

Assignment may be translated to the use of a *INF()* constructor followed by the construction of a *RECORD* and an *UPDATE()* instruction. A test for equality of two *any* values involves a call to the compiler constant procedure *eqType* followed by the instruction *EQ.INF* when *eqType* returns *true*, if equality is considered as in Napier88. In this case, two values of type *any* are equal if and only if they can be projected on to equivalent types and the projected values are equal.

8.10 The Compiler Front-end

The 2TPL front-end parses programs written in COREL and generates TPL programs, as shown in previous sections.

8.10.1 Parsing

2TPL is a recursive descent parser that generates TPL in one pass without backpatching[7]. As usual in recursive-descent compilers [Davie and Morrison, 1981], there is roughly one procedure for each production of the source language. Each of these procedures parses one production rule and returns a type which is used by its caller procedure to perform the type-checking.

In order to have unique TPL identifiers to facilitate program analysis, 2TPL uses a different name irrespective of the scope level. The procedure parameter names are also unique for each TPL program. Otherwise, a first phase where bound variables are renamed, can be performed; this phase exists in several compilers and is called "alpha conversion" in the ORBIT compiler ([Kranz *et al.*, 1987]. Because of this property, optimisations can ignore the problem of name conflicts that arise when two or more variables have the same identifier in different scopes.

8.10.2 Internal Data Structures

As COREL is a block structured language, a symbol table is needed to model the scope. The compile-time environment is constructed as the COREL program is parsed. 2TPL collects information about program identifiers in the symbol table when they are declared, and subsequently uses that information when an identifier is used to check if it is a legal usage. Each entry in the symbol table contains the following information:

- **name** – the name of the identifier;
- **type** – the type of the identifier;
- **constancy** – true if the identifier is a constant;
- **ll** – lexical level where the identifier was found; and
- **dd** – location where the value can be found (an abstract machine register *Rn*).

The symbol table is implemented by a linked list of binary trees. For each scope there is a tree prepended to the list when the scope is entered and dropped from the list when it is exited.

In order to perform type-checking of programs according to the rules presented in Appendix B.2, types are represented by 2TPL in a Napier88 variant with the following tags:

- **scalart** — for atomic types; they are represented as strings: *int*, *real*, ...
- **proct** – for procedure types; they are represented as a structure with a list of types for arguments and a type for its result
- **struct** – for structure types; they are represented as a list of structures with a string (the label) and the corresponding type
- **vectort** – for vector types; they are represented by the type of its elements.

The front-end fills the nodes of the directed graph which represents the program in TPL, as parsing proceeds.

[7]Sometimes it introduces a *NOP* instruction when a forward label is needed at a position that is not yet known.

8.10.3 Collecting Blackboard Information

After the COREL program reaches its end, 2TPL traverses the TPL internal representation and collects administrative information together with blackboard information by calling the component LABEL of the language compilation framework. The directed graph TPL internal representation is traversed by a depth-first search and the nodes are numbered sequentially when visited. The collected blackboard information associated with the TPL identifier, includes the lexical level, its TPL type and the source program name when applicable (only for source language program declarations):

ID	NAME	LL	USED	COST	CALLS	FLAGS	TYPE
R1		1	3				MAP
R2		1	1	-	0		PROC(CHARS,CHARS->BOOL)
R3		1	0	-	0		PROC(BOOL,BOOL->BOOL)
R4	debug	1	1				BOOL
R5	i	1	3			U	INT
R6		1	3				RECORD(CLOSURE(),CHARS,BOOL)
R7		2	1				INT

The field "FLAGS" registers the information of the different transformations performed. The information whether an identifier is used in an *UPDATE* instruction that changes its value (e.g., *i* in the situation shown) and the number of uses of the identifier is also identified as important for optimisations. The number of uses is equal to the number of times an identifier is referred to (excluding its definition). The blackboard information is stored in a map with the identifier as a key and is available for all the other components of the language framework prototype.

8.11 Bootstrapping the Compilation Framework

The TPL code for predefined procedures is generated in the prototype by the 2TPL procedures used by the front-end to plant code and the front-end is implemented in Napier88 (as described in Section 6.6). This way, the prototype uses two object stores: the first (Napier Store) for the routines which compose the language compilation framework and the second (TPL Store) to save the persistent values manipulated by the compiled programs. At some point in the future, the front-end must be integrated into the TPL Store in order to achieve runtime linguistic reflection. This can be accomplished by having sufficient constructs in the COREL language to implement the front-end and the other components of the language compilation framework[8]. Following the bootstrapping technology, as described for example in [Aho *et al.*, 1986], the front-end written in COREL can compile itself into the TPL Store. Versions of its routines can then be optimised using the language compilation framework itself.

8.12 Conclusions

This chapter demonstrated that features which were identified as crucial for this work, and therefore needed to be supported, can be accommodated by instructions of the TPL language

[8]Of course, this can be done by achieving a front-end for the Napier88 subset used in the implementation.

described in Chapter 7. Further to persistence and first-class higher-order procedures, the prototype supports declarations, assignment, control structures, recursion, aggregate types and incremental binding. Support for polymorphism, union types and infinite union types can also be incorporated into the prototype, as was described. Runtime linguistic reflection can be achieved by having the compiler callable from the persistent object store, because the prototype already supports orthogonal persistence. The architecture needs to be bootstrapped in order to achieve this.

Other features, like ADT and inclusion polymorphism are not supported as yet. The design issues related to the support of ADT are considered orthogonal to the design issues of TPL, and can be achieved considering that first-class higher-order persistent procedures are enough to support ADT, as was proved in [Atkinson and Morrison, 1985]. Furthermore, supporting ADT poses the type-checking problems described in [Cutts, 1993] which would complicate the implementation. Inclusion polymorphism over records, may be supported by field addressing on values of type *RECORD*, as described in Section 3.6.2.

The following chapters will show how the TPL representation can be transformed in order to achieve better time and space characteristics and in order to achieve a program representation directly executable by a von Neumann machine. The runtime system and the persistent object store will be shown to enable longevity, code generation and program execution with dynamic binding.

Chapter 9

High-level Machine Independent Optimisations

The architecture proposed to meet the requirements of efficiency and longevity includes a high-level intermediate language and transformations intended to optimise the programs represented in that language. This chapter illustrates the support for some machine independent optimisations on TPL internal representations of programs. The transformations described in this chapter include partial evaluation techniques, such as constant folding and constant propagation; redundancy elimination techniques, such as unreachable-code elimination, useless-code elimination, common-subexpression elimination; and procedure call transformations, such as inlining, procedures called only once, dropping unused arguments and tail recursion. The components of the language framework which implement the transformations are described. This chapter finishes by describing the use of continuations as a vehicle for optimisation, the implementation of this transformation in TPL to produce a representation in CPS and the properties of TPL changed by this transformation.

9.1 Introduction

Optimisations are intended to improve the qualities of an internal program representation with respect to space usage and execution time. Because optimisations at this level are independent of the underlying architecture, they can be done once and for all. In order to cope with longevity demands, it may be necessary to generate different versions of target machine code for different *era*, but the transformations at this level will not be performed again.

Because CPU performance has doubled every two to three years while DRAM speeds have doubled only once per decade [Patterson and Hennessy, 1990], special attention must be paid to memory accesses. At this level, the number of loads and stores must be minimised to improve memory performance. Another crucial improvement is the efficient usage of machine registers, which is left to optimisations at a different level. Several optimisations at this level alleviate the pressure on machine registers, such as unreachable-code elimination or copy propagation transformations. Because TPL representations are intended to be stored together with the final target machine code, the optimisations at this level must pay special attention to the final size of the representation.

In the proposed architecture, optimisations can be performed at another level, closer to the hardware machine architecture. Important optimisations are left for this level that will "tune" the program representation to the target architecture; examples are loop manipulations, induction-variable elimination, efficient register usage and instruction scheduling. Data-flow based transformations, which involve complex analysis in order to build data structures to support them on top of the TPL representation, are also not studied in this work. Those transformations are commonly performed on top of three-address representations and therefore can also be applied with TPL.

In applying transformations to a program representation, care must be taken to ensure that the transformation does not change the meaning of the program or changes it only in a restricted way which is acceptable to the user. A transformation is **legal** "if the original and the transformed program produce exactly the same output for all identical executions" [Bacon *et al.*, 1994]. Two executions are considered **identical executions** if when they are supplied with the same input data and if every corresponding pair of non-deterministic operations produce the same result in the two executions[1]. Program representation *P1* is equivalent to program representation *P2* if and only if *P2* is the result of applying any legal transformation to *P1*. Thus, if *P1* and *P2* are equivalent, they produce the same output for identical executions. In order to maintain correctness when transforming a program representation, attention must be paid to several situations that may violate the definition. For example, overflow may occur if the order of operations is changed. As floating-point representations are approximations of real numbers, different results may occur if the order in which approximations (rounding) are applied is changed and so special care must be taken not to produce different results.

Optimisations can be applied at different levels of granularity, from the level of statement to the whole program:

1. statement transformations (e.g. arithmetic expression simplification);
2. basic block transformations, which are well studied and commonly used in optimising compilers (e.g. constant folding);
3. loop transformations, to target high-performance architectures (e.g. code motion transformations and induction-variable elimination);
4. procedure-level transformations, performed after intra-procedural analysis (e.g. memory access transformations); and
5. inter-procedural transformations, where several procedures can be considered at the same time (e.g. inlining procedure calls).

[1]Examples of non-deterministic operations are UNIX system calls such as *time()* or *read()*.

The cost of analysis increases with the scope of the transformations. In this chapter, transformations at the level of the statement (and thus directly supported by TPL representation) are investigated together with inter-procedural transformations supported by simple whole program analysis. Other optimisations, such as common-subexpression elimination, can be performed by using the appropriate data-flow analysis which traces flow of data through program's variables [Aho *et al.*, 1986]. These analysis may involve the construction of more elaborate data structures to represent data and control dependencies. The only analysis performed in TPL programs is a simple distinction of variables which are updated or not together with a counting of the number of procedure calls for each procedure definition, which enables important transformations such as inlining. Other important class of optimisations based on data-flow analysis are loop transformations, especially the inner loops where the program tends to spend more time. Further to languages such as C or PASCAL, this transformation has been applied with success to higher-order languages such as ML [Tarditi *et al.*, 1996].

Program execution speed can be improved by several loop optimisations: *code motion* which moves code outside a loop, *induction-variable elimination* which eliminates variables from inner loops, *loop unrolling* which replicates the body of the loop a number of times and changes the loop step accordingly and many others (see [Bacon *et al.*, 1994] for a comprehensive summary of loop transformations). As the target architectures usually include an on-chip cache, transformations such as inlining or code motion out of loops may have considerable merit as they can improve cache hit rates. From all of these, only inline is demonstrated in this chapter.

Because a compiler may find several optimisations that are worthwhile to apply in a program representation, it must decide on the best sequence and also when to stop applying those transformations. The problem of finding the optimal representation for a given program is, in general, NP-complete ([Appel, 1992]), but suboptimal solutions may be obtained with the use of heuristics as shown in this chapter. The following sections will present the sequence of transformations as implemented in TPL and describe transformations at the level of statement, at the level of the basic block, whole program transformations and the CPS transformation.

9.2 Optimising TPL Program Representations (OPT)

In the language framework prototype implemented, high-level machine independent optimisations are divided into several components, each one implementing a class of transformations. This separation is driven by the need to separate transformations that interfere with each other and also to simplify the usage of the TPL internal data structures. The transformations were grouped into the following components consisting only of non-interfering optimisations:

1. constant propagation, constant folding, copy propagation and algebraic simplifications (FOLD);

2. unreachable-code elimination or comparison folding (COMPAR);

```
FOLD                          % constant folding
repeat                        % start cycle
    total = COMPAR            % unreachable-code elimination
    total += UNUSED           % dead-variable elimination
    total += INLINE           % inline procedure calls
    total += FOLD             % constant folding
until (total < MIN)           % performed less than minimum
NOPS                          % remove NOP
```

Figure 9.1: Sequence of Transformations Implemented in OPT

3. useless-code and dead-variable elimination (UNUSED);
4. inline procedure calls (INLINE); and
5. remove *NOP* instructions (NOPS).

Each of these components may be run independently to transform a TPL representation into another and the resultant TPL may be inspected. By using this "plug and play" technology, heuristics may be collected on the better sequence to be applied by the TPL to TPL optimiser (OPT component).

Figure 9.1 presents the sequence of transformation applied by OPT as an example of a promising sequence. After partial evaluation performed by FOLD, each round of optimisations involves one pass of each of the first four groups of optimisations. As performing one transformation may enable others to be applied afterwards, several rounds may be performed in a run of the OPT language framework component. The variable *total* collects the number of transformations performed during each cycle as the sum of the transformations performed by each component. The value of *total* is used to determine when the cycle should stop. A minimum number of transformations (*MIN*) is requested to happen in order to keep the time spent into OPT bounded by reasonable limits. As inline creates new opportunities for constant propagation and useless-code elimination, the FOLD component is always called after INLINE. OPT ends by calling NOPS to remove all *NOP* instructions introduced by the front-end when the TPL program was generated or by the previous transformations performed by OPT.

In the following sections each component of OPT will be described. The transformations immediately supported by the TPL representation are discussed and illustrated with examples collected from runs of the language framework prototype.

9.3 Partial Evaluation

The technique of performing part of the computation at compile-time is called partial evaluation. Included in this class are transformations such as constant propagation, constant folding, copy propagation and strength reduction that improve both the size and speed of the program representation. These transformations will be described in the following subsections and a complete example will be presented in the end of this section.

9.3.1 Constant Propagation

Typically programs contain many constants introduced directly by the programmer or by previous program transformations. In a study of name usage in several Napier88 programs made by different programmers reported in [Sjøberg, 1993], of over 51328 lines of code and 21037 identifiers, 29.9% are declared as constants. It should be expected in the TPL internal representation that an even larger number of names may be recognised as constants, as a name may not be declared constant by the programmer but it is identified as constant because its value is not updated in the program. By propagating those constants through the program, new opportunities for optimisation are revealed. Some variables may become dead after this transformation and may then be removed; also branching choices may become known at compile-time, allowing for unreachable-code removal.

For example, for *INT* constants in TPL, the following program:

```
R1 := INT(10)
R2 := INT(R1)
UPDATE(R2,5)
R3 := PLUS.INT(R1,R2)
```

may be transformed to:

```
NOP
R2 := INT(10)
UPDATE(R2,5)
R3 := PLUS.INT(10,R2)
```

As can be seen from this example,·there is no dependency analysis done to know that the value of *R2* at the last instruction is also a constant value *5* that would enable further transformations. The algorithm used simply looks at variables that are initialised by a constructor and never updated, so their value is definitely known to be the initial constant (which is not the case of variable *R2* in the example). To be implemented, this algorithm needs only the knowledge of whether the identifier is used in an *UPDATE* instruction. This information is collected by LABEL after 2TPL finishes generating a program. Taking advantage of every name being unique, a map from the old identifier to the immediate value (constant) is interrogated every time the optimiser finds a possible propagation. The instruction that constructed the propagated constant value is substituted by a *NOP* as the identifier will not be used afterwards. Alternatively, if left, it would be removed by the useless-code elimination transformation as the identifier will be dead.

9.3.2 Constant Folding

Constant folding transformation consists of replacing an operation by its result if the operands are known to be constants. For example, the following relational instruction:

```
R1 := GT.INT(2,10)
```

will be transformed to the equivalent use of a constructor with the computed initial value:

```
R1 := BOOL(FALSE)
```

Care must be taken with situations which would produce illegal programs if the operations performed at compile-time produce different results from the operations at runtime. For example, if a *PLUS* operation involves two numbers equal to the maximum integer which can be represented by the optimiser, the overflow cannot be generated by the optimiser, but instead, it should be left to runtime in order to maintain identical executions[2].

The algorithm used in performing constant folding looks at the arguments of all primitive operations of all TPL types and assesses whether the result can be computed immediately. In those situations, that instruction is transformed to a call to a constructor of the same type with the corresponding literal.

9.3.3 Copy Propagation

It can be observed that the front-end may copy a value to a different variable (e.g. the result of procedures in the TPL program presented in page 106) or that certain transformations may do the same (e.g. inlining described in Section 9.5.1). In order to reduce register pressure by eliminating redundant register-to-register move instructions, this transformation substitutes each copy by its original value. For example, the following program:

```
R1 := INT(1)
R2 := INT(R1)
R3 := VECTOR(1,10,R2)
R4 := MOVE.VECTOR(R3)
R5 := PLUS.INT(R4@1,R2)
```

is transformed by copy propagation to:

```
R1 := INT(1)
NOP
R3 := VECTOR(1,10,R1)
NOP
R5 := PLUS.INT(R3@1,R1)
```

For all constructors (*INT(), BOOL(), RECORD(), ..., MOVE.<mc-type>()*) when used to construct a new value, the identifier introduced can be folded if its value is not updated afterwards and the value used as initialisation is an identifier which corresponds to a variable that is also not updated in the program. A small amount of data-flow analysis is required to identify the program constants. The instruction that was constructing the copy is substituted by a *NOP* instruction and the initialisation identifier is used in the place of the copy, as shown in the example.

9.3.4 Algebraic Manipulations

For TPL base types, the optimiser can simplify operations by the application of known algebraic rules. For example, the instruction that multiplies the value of *R1* by *1*:

```
R10 := MULT.INT(R1,1)
```

can be transformed to:

[2]It would be better to have the front-end to discover these situations and inform the programmer immediately of the error.

Operation	Transformed to
`PLUS.INT(x,0) = PLUS.INT(0,x)`	`INT(x)`
`MINUS.INT(x,0)`	`INT(x)`
`MINUS.INT(0,x)`	`NEG.INT(x)`
`MULT.INT(1,x) = MULT.INT(x,1)`	`INT(x)`
`MULT.INT(0,x) = MULT.INT(x,0)`	`INT(0)`
`DIV.INT(0,x)`	`INT(0)`
`DIV.INT(x,1)`	`INT(x)`
`REM.INT(0,x)`	`INT(0)`
`REM.INT(x,1)`	`INT(0)`
`OR.BOOL(TRUE,x) = OR.BOOL(x,TRUE)`	`BOOL(TRUE)`
`OR.BOOL(FALSE,x) = OR.BOOL(x,FALSE)`	`BOOL(x)`
`AND.BOOL(TRUE,x) = AND.BOOL(x,TRUE)`	`BOOL(x)`
`AND.BOOL(FALSE,x) = AND.BOOL(x,FALSE)`	`BOOL(FALSE)`
`NOT.BOOL(FALSE)`	`BOOL(TRUE)`
`NOT.BOOL(TRUE)`	`BOOL(FALSE)`
`EQ.<mc-type>(a,b)`	`BOOL(a=b)`
`CAT.CHARS(a,b)`	`CHARS(a++b)`
`SUB.CHARS(a,b,c)`	`CHARS(a(b|c))`

Table 9.1: Algebraic Rules Used in Optimisations

```
R10 := INT(R1)
```

revealing another possible opportunity for optimisation by applying a copy propagation transformation. Table 9.1 presents the algebraic transformations which can be performed in operations of the *INT*, *BOOL* and *CHARS* types. In those identities, *x* means any access mode for that operand and *a* or *b* an immediate or constant value of the required type. The optimiser performs operations on constant values, such as: equality =, concatenation ++ and sub-string |. Floating-point arithmetic may be more problematic to simplify due to the existence of special values; e.g. if *x* holds a constant value of type *DOUBLE* which is *NAN* (Not a number) then

```
R1 := MULT.DOUBLE(x,0)
```

is equal to

```
R1 := DOUBLE(x)
```

The algorithm consists of looking at the arguments of primitive operations and discovering whether any algebraic rule can be applied in which case a new TPL instruction is created which replaces the existing instruction.

9.3.5 Strength Reduction

The strength reduction transformation consists of changing expensive operators for equivalent less expensive operators. For example, the following instruction:

```
R1 := MULT.INT(x,2)
```

can be replaced by

```
R1 := PLUS.INT(x,x)
```

This transformation is particularly effective when the change lies inside an inner loop that is executed several times.

9.3.6 Putting It All Together — FOLD

The transformations discussed in this section are performed in one pass. The TPL internal representation is traversed once and each node is analysed according to the instruction it represents. The algorithms described for each transformation are combined in the implementation by testing the applicability of the transformations, for each instruction. A map from the old identifier to its substitute is maintained with a new entry added every time a new opportunity to perform a transformation is verified. For each access to a value in each instruction, the map is interrogated and the addressing mode changed if the corresponding identifier has already been changed. The result of FOLD is the number of transformations performed.

As an example, consider the following TPL program representation:

```
[6]     R4 := INT(10)                              % a1    # let a1= 10
[7]     R6 := PROC("INT->INT",[A1],                % p1    # let p1= proc(i: int -> int)
[8]       START
[9]       R5 := MULT.INT(A1,2)                             # i*2
[10]      R0 := INT(R5)
[11]      END
[12]      PROCEND)
[13]    R7 := CALL(R6,[2])                         % a2    # let a2= p1(2)
[14]    R8 := INT(R7)                              % a3    # let a3= a2
[15]    R9 := DIV.INT(R8,1)                        % a4    # let a4= a3 div 1
[16]    R10 := LOOKUP:RECORD("writeInt",R1)
[17]    VOID := CALL(R2,[R10!1,"procedure(int->VOID)"])
[18]    R11 := MULT.INT(R4,2)
[19]    R12 := GT.INT(R11,10)
[20]    BRA(R12,                                           # if a1*2>10
[21]      START                                            # then
[22]      VOID := CALL(R10!0,[R8])                         # writeInt(a3)
[23]      END,
          JUMP(L24),
[28]        START                                          # else
[29]        VOID := CALL(R10!0,[R9])                       # writeInt(a4)
[30]        END,
            JUMP(L24))
        LABEL(24)
[24]    NOP
```

that will be transformed by FOLD into:

```
[6]     NOP                                                # constant propagation of R4
[7]     R6 := PROC("INT->INT",[A1],                % p1
[8]       START
[9]       R5 := PLUS.INT(A1,A1)                            # strength reduction
[10]      R0 := INT(R5)
[11]      END
[12]      PROCEND)
[13]    R7 := CALL(R6,[2])                         % a2
[14]    NOP                                                # copy propagation of R7
[15]    NOP                                                # algebraic simplific of R9
```

```
[16]   R10 := LOOKUP:RECORD("writeInt",R1)
[17]   VOID := CALL(R2,[R10!1,"procedure(int->VOID)"])
[18]   NOP                                              # constant propagation of R4
[19]   NOP                                              # constant propagation of R4
[20]   BRA(TRUE,                                        # opportunity for COMPAR
[21]     START
[22]     VOID := CALL(R10!0,[R7])
[23]     END,
         JUMP(L24),
[28]       START
[29]       VOID := CALL(R10!0,[R7])
[30]       END,
           JUMP(L24))
       LABEL(24)
[24]   NOP
```

Variables *R4*, *R11*, *R12* were eliminated by propagating the constant value of *R4*. *R8* and *R9* were eliminated by copy propagation and algebraic simplification respectively. Finally, instruction 9 was transformed by strength reduction.

It should be noted that new opportunities for optimisation are now possible. The conditional branch of instruction 20 has a known constant value for the condition and can be further transformed by applying unreachable-code elimination. When a constant value is propagated, the corresponding variable becomes dead and the constructor that binds its value will be substituted by a *NOP* in the FOLD component, as in instruction 6. It could as well be left to be removed by the useless-code elimination transformation to be discussed in Section 9.4.2. When FOLD is run, the *NOP* instruction that substituted folded constructors are left to highlight the places where transformations occurred. *NOP* instructions will be collected by the NOPS component in the end of the optimiser OPT.

9.4 Redundancy Elimination

There is a class of transformations that improve performance by eliminating redundant computations. Included in this class are transformations that remove unreachable or useless computations. This section discusses these transformations together with common-subexpression elimination transformations which can also be included in this class.

9.4.1 Unreachable-code Elimination

A computation is **unreachable** if it is never executed. Removing such a computation from the program has no effect on the execution. Unreachable-code may be created by programmers (e.g. conditional debugging code) but most frequently is a result of other transformations (e.g. constant propagation). Applying this transformation can, in turn, create new opportunities for constant folding.

If the program representation has conditional predicates with a value known to be false or true, then the conditional and one of the branches may be safely removed. Another possibility for unreachable-code elimination is the identification of loops that are never executed. If the representation has only the unstructured *goto* to transfer control then the applicability of this transformation is not immediately obvious and the control flow-graph must be traversed. In TPL there is only the *BRA()* general branch construct for both conditional predicates and loops. Therefore, the algorithm for this transformation consists of looking at the truth value of the *BRA()* condition and when known to be a constant, drops the unreachable branch. This algorithm is implemented in the COMPAR component. COMPAR simply traverses the TPL internal representation and analyses each *BRA()* instruction. The fact that each TPL block must be enclosed by *START* and *END* helped in the implementation.

Consider for example the following COREL source program with conditional debugging code:

```
[1]     let debug = false
[2]     let i:= 10
[3]     use PS() with writeInt: proc( int ) in
[4]       if debug then { i:= i+1; writeInt( i ) } else writeInt( 2 )
```

and the corresponding TPL representation:

```
[6]     R4 := BOOL(FALSE)                     % debug # constant
[7]     R5 := INT(10)                         % i     # constant
[8]     R6 := LOOKUP:RECORD("writeInt",R1)
[9]     VOID := CALL(R2,[R6!1,"procedure(int->VOID)"])
[10]    BRA(R4,
[11]      START
[12]      R7 := PLUS.INT(R5,1)
[13]      UPDATE(R5,R7)                             # i <- i+1
[14]      VOID := CALL(R6!0,[R5])                   # write integer
[15]      END,
          JUMP(L16),
[20]        START
[21]        VOID := CALL(R6!0,[2])                  # write integer
[22]        END,
            JUMP(L16))
        LABEL(16)
[16]    NOP
```

In order to reveal unreachable-code the constant *R4* is firstly propagated, leading to:

```
[6]     R5 := INT(10)                         % i
[7]     R6 := LOOKUP:RECORD("writeInt",R1)
[8]     VOID := CALL(R2,[R6!1,"procedure(int->VOID)"])
[9]     BRA(FALSE,                                  # opportunity for COMPAR
[10]      START
[11]      R7 := PLUS.INT(R5,1)
[12]      UPDATE(R5,R7)
[13]      VOID := CALL(R6!0,[R5])
[14]      END,
          JUMP(L15),
[18]        START
[19]        VOID := CALL(R6!0,[2])
[20]        END,
            JUMP(L15))
        LABEL(15)
[15]    STABLE()
```

The transformation in the internal representation performed by COMPAR for the unreachable-code in the *BRA()* instruction is represented in Figure 9.2. This program is transformed to:

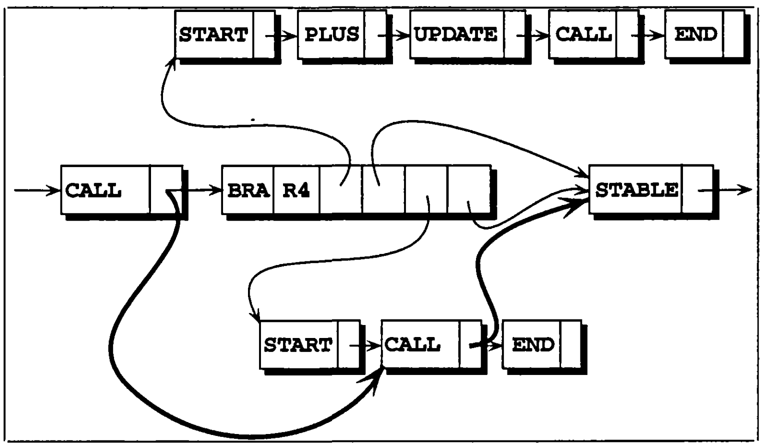


Figure 9.2: Unreachable-code Elimination

```
[3]     R1 := PROOT()
[4]     R2 := LOOKUP:RECORD("matchType",R1)
[5]     R3 := LOOKUP:RECORD("matchConst",R1)              # opportunity for UNUSED
[6]     R5 := INT(10)                          % i        # opportunity for UNUSED
[7]     R6 := LOOKUP:RECORD("writeInt",R1)
[8]     VOID := CALL(R2,[R6!1,"procedure(int->VOID)"])
[9]     VOID := CALL(R6!0,[2])
```

It should be noted that the transformed representation is substantially shorter than the original TPL program and that further transformations can now be performed. *R5* is not updated anymore and thus becomes dead and may be eliminated.

9.4.2 Useless-code Elimination

A computation is **useless** if none of the outputs of the program are dependent on it. For example, if a variable is declared but not used, the declaration can be removed as it is useless. Useless code is often created by other transformations such as constant propagation or unreachable-code elimination.

In TPL if a variable is used in an *UPDATE* to assign a new value and it is not used thereafter, then the *UPDATE* instruction could be removed. In order to perform this kind of transformation live-variable analysis[3] must be carried out in the TPL representation. The removal of unnecessary assigns is not performed at the moment.

Directly supported by TPL is **dead-variable elimination**. This transformation consists on removing declarations of variables that are never used in the program. For Napier88 programs, a study described in [Sjøberg *et al.*, 1994] reports that 8% of all value identifiers are declared but not used in the program. As this figure does not include those made redundant by other transformations, it is expected that this transformation can be applied to even more identifiers.

The algorithm involves a traversal of the TPL representation and the substitution by a *NOP* instruction of all constructors for identifiers on which the computed number of uses

[3]A well studied data-flow problem [Aho *et al.*, 1986].

is equal to zero. This algorithm is implemented in the UNUSED component which uses the *number of uses* information stored by LABEL in the field *used* of the blackboard information (see Section 8.10.3).

For example, the program representation that resulted from the unreachable-code elimination of page 132 can be further transformed by dead-variable elimination to:

```
[3]     R1 := PROOT()
[4]     R2 := LOOKUP:RECORD("matchType",R1)
[5]     NOP                                          # useless-code elim on R3
[6]     NOP                                          # useless-code elim on R5
[7]     R6 := LOOKUP:RECORD("writeInt",R1)
[8]     VOID := CALL(R2,[R6!1,"procedure(int->VOID)"])
[9]     VOID := CALL(R6!0,[2])
```

R5 was not used after its declaration and therefore could be removed. It should be noted that this transformation works also for procedures declared by the front-end and not used in the program as is the case of procedure *matchConst* in this example.

9.4.3 Common-subexpression Elimination

A set of computations may contain identical subexpressions introduced by programmer code or by other transformations. Common-subexpression elimination consists of using the value computed in the first place in all the other identical computations.

This transformation is not implemented as yet, but it can be shown how it can be performed in a TPL program. Consider, for example, the following program:

```
R1 := INT(10)
R2 := MULT.INT(R1,R1)
R3 := MULT.INT(2,R2)
R4 := MULT.INT(R1,R1)          # R4=R2
R5 := MULT.INT(2,R4)
R6 := PLUS.INT(R3,R5)
```

is transformed by eliminating the common-subexpression *MULT.INT(R1,R1)* to:

```
R1 := INT(10)
R2 := MULT.INT(R1,R1)
R3 := MULT.INT(2,R2)
NOP
R5 := MULT.INT(2,R2)           # R5=R3
R6 := PLUS.INT(R3,R5)
```

and further by eliminating *MULT.INT(2,R2)* to:

```
R1 := INT(10)
R2 := MULT.INT(R1,R1)
R3 := MULT.INT(2,R2)
NOP
NOP
R6 := PLUS.INT(R3,R3)
```

It should be noted that, as a result of this transformation, there is a decrease in size of the program representation. There is also a decrease on register pressure, as before the transformation all intermediate values were named.

In order to support this transformation in TPL programs, data-flow analysis similar to the analysis on 3-address code ([Aho *et al.*, 1986]), must be performed.

9.5 Procedure Call Transformations

Procedure calls involve overhead on entry and exit to store and restore machine registers, allocate and deallocate an activation record and store the actual parameters and the results of the procedure. There is a class of transformations targeted to eliminate or reduce the procedure call overhead. This section discusses inlining procedure calls, elimination of calls to procedures called only once, drop of unused arguments and tail-recursion.

9.5.1 Inlining

Procedure inlining eliminates all the overheads of procedure calls by replacing the procedure call with the body of the called procedure. Actual parameters replace the procedure's formal parameters and local variables may need to be renamed if the procedure is called more than once from the same scope, or when their names conflict with names in the caller scope as is the case of *R0*. Because inlining eliminates the overhead of procedure calls, there is an improvement in execution speed. The disadvantage of copying the body is that program representations may become larger and more registers may be needed. This change in size is particularly important for longevity, as the TPL representation is made persistent together with the object code. Copying the body introduces the possibility of extending the analysis from intra-procedural to inter-procedural with no extra cost. Constant propagation, constant folding, unreachable or useless-code transformations, etc. can be extended to parameters and the body of the procedure after inlining with no extra analysis cost. By eliminating redundant operations after inlining, the size of the program is reduced, further to the register pressure effects.

The impact of inlining in the hierarchy of memory is more complex and is dependent on the size of the cache. Instruction cache behaviour of the program is affected favourably because locality may be improved by eliminating the transfer of control. On the other hand, if the loop has several calls to a procedure, inlining these calls will cause several copies of the procedure body to be loaded into the cache. In some situations, an inlined program may become slower because of cache misses or page faults. The overhead of a context switch also depends on the target architecture. In modern architectures, a context switch is more efficient (e.g. register windows in the SPARC). Because of its dependency on the size and organisation of the cache and also in the context switch overhead of the target architecture, the applicability of this transformation may be better decided with the use of machine-dependent information [Benitez and Davidson, 1994]. Inlining must then be considered a machine independent optimisation which can be performed by a machine independent algorithm that needs machine dependent information to decide when the transformation may be applied.

Because of its dependency on the architecture and because it may lead to larger program representations, inlining of procedure calls in TPL must be carefully assessed, as together with a search for efficiency, there is a need to keep machine independent program representations as small as possible, for the sake of longevity. The space/time trade off puts more emphasis on space as there is an opportunity in the proposed architecture for optimisations (that may even include inlining) at a lower level.

To implement this transformation in TPL, another piece of simple analysis is needed in order to separate the uses of the procedure identifier in a *CALL()* from its other uses (such as the use in *UPDATE()* instruction), as the available blackboard information is only the number of uses. A procedure call will be inlined if it is the only call, because that will not make the program larger, or if the change in size is smaller then a maximum positive value *MAX*. The formula:

$$(\mathtt{noc} - 1) * \mathtt{cost} \leq \mathtt{MAX}$$

were *noc* is the number of calls and *cost* is the procedure cost proportional to the size of the procedure body, may be used to decide inlining. If one call to a particular procedure may be inlined all calls will be inlined and the procedure declaration becomes dead and may be eliminated by UNUSED. Because of that, *(noc-1)*cost* gives the effective change in size by the inlining transformation. It should be noted that, once inlined, there may be significant collapse of code via other transformations. The procedure cost is computed after constant folding and constant propagation transformations are applied to TPL by OPT, in order to be more accurate, when inlining has to be decided. In computing *cost*, the TPL instructions are considered with the size 1 and the cost of the *PROC()* constructor is equal to 1 plus the cost computed for the body (as procedures can be nested). It should be noted that it is usual to compute the cost in terms of target machine code instructions because more than the size, execution time is paramount. In that situation, the cost of TPL instructions would be different; e.g. the cost of *RECORD([R1,. . . ,Rn])* would be one plus the cost of moving the n arguments on initialisation.

The INLINE component performs inlining in TPL representations by using an algorithm that traverses the representation three times:

1. computes the cost for each *PROC()* instruction, the number of calls for each procedure identifier and stores this information plus a reference to the procedure body and parameters in a map indexed by the procedure identifier;

2. for each *CALL()* instruction, decide if inlining is worthwhile based on the computed cost;

3. if inlining was selected, declare actual parameters, copy procedure body renaming all locals (to maintain the TPL property of different names even if in different scopes); move the result to the identifier bound by *CALL()*; and store in the blackboard the information that the procedure was inlined.

Consider, for example, the following COREL program:

```
[1]     let p1= proc( i: int -> int ); i
[2]     let p2:= proc( i: int -> int ); 2
[3]     let v= vector 1 to 1 of p1
[4]     let a1= p1( 12 ) + p2( 22 )
```

```
[5]     let a2= v( 1 )( 32 )
[6]     p2:= p1
```

when translated to a TPL program, becomes:

```
[6]     R4 := PROC("INT->INT",[A1],           % p1
[7]       START
[8]       R0 := INT(A1)
[9]       END
[10]      PROCEND)
[11]    R5 := PROC("INT->INT",[A2],           % p2
[12]      START
[13]      R0 := INT(2)
[14]      END
[15]      PROCEND)
[16]    R6 := VECTOR(1,1,R4)                  % v
[17]    R7 := CALL(R4,[12])                           # call p1
[18]    R8 := CALL(R5,[22])                           # call p2
[19]    R9 := PLUS.INT(R7,R8)                 % a1
[20]    R10 := CALL(R6@1,[32])                % a2    # call v(1)
[21]    UPDATE(R5,R4)                                 # p2 <- p1
```

This program is transformed by INLINE into:

```
[6]     R4 := PROC("INT->INT",[A1],           % p1
[7]       START
[8]       R0 := INT(A1)
[9]       END
[10]      PROCEND)
[11]    R5 := PROC("INT->INT",[A2],           % p2
[12]      START
[13]      R0 := INT(2)
[14]      END
[15]      PROCEND)
[16]    R6 := VECTOR(1,1,R4)                  % v
[17]    R11 := INT(12)                                # parameter A1 of p1
[18]    R12 := INT(R11)                               # R0 renamed to R12
[19]    R7 := MOVE.INT(R12)                           # MOVE result of p1(12)
[20]    R8 := CALL(R5,[22])
[21]    R9 := PLUS.INT(R7,R8)                 % a1
[22]    R10 := CALL(R6@1,[32])                % a2
[23]    UPDATE(R5,R4)
```

Inlining the call to procedure *p1* (*R4*) in instruction 17 substitutes the *CALL()* instruction by instructions 17 to 19. It should be noted that although *p1* and *p2* (*R5*) have the same cost, only *p1* is inlined because the value of *p2* is updated in the program representation. A simple piece of analysis would reveal that the update is done after the call and therefore could be inlined as well. The call to *p1* through *v(1)(10)* is also not inlined. Procedure calls made by indexing vectors with constants or by using constant offsets from records, could be assessed by first performing appropriate alias analysis. As recursive procedures are implemented in TPL by updating an initial value with a new *PROC* value that makes references to the procedure identifier (see Section 8.2.3), they will not be inlined, even in the cases where that would be convenient. Finally, persistent values are "boxed" in a value of type *RECORD* (see Section 8.6) and so they will not be inlined as well. Persistent procedure calls could not be inlined also because those procedures may have state and references to other store objects. In summary, only the procedures declared in the program being analysed may be inlined.

To show the effectiveness of this transformation, for the current example, the TPL representation is further optimised by FOLD followed by UNUSED, leading to:

```
[3]     R1 := PROOT()                                 # opportunity for UNUSED
[4]     NOP                                           # useless-code elim on R2
[5]     NOP                                           # useless-code elim on R3
[6]     R4 := PROC("INT->INT",[A1],       % p1
[7]       START
[8]       R0 := INT(A1)
[9]       END
[10]      PROCEND)
[11]    R5 := PROC("INT->INT",[A2],       % p2
[12]      START
[13]      R0 := INT(2)
[14]      END
[15]      PROCEND)
[16]    R6 := VECTOR(1,1,R4)              % v         # opportunity for UNUSED
[17]    NOP                                           # constant propagation of R11
[18]    NOP                                           # constant propagation of R11
[19]    R7 := MOVE.INT(12)                            # opportunity for UNUSED
[20]    R8 := CALL(R5,[22])                           # opportunity for UNUSED
[21]    NOP                                           # useless-code elim on R9
[22]    NOP                                           # useless-code elim on R10
[23]    UPDATE(R5,R4)
```

By propagating constant *R11* introduced by INLINE, instructions 17 and 18 may be removed and the identifiers *R2*, *R3*, *R9* and *R10* become dead and the corresponding constructors can be removed by UNUSED. Another pass of UNUSED will find more dead identifiers: *R1*, *R6*, *R7* and *R8*. The final TPL program is thus:

```
[3]     NOP                                           # useless-code elim on R1
[4]     R4 := PROC("INT->INT",[A1],       % p1
[5]       START
[6]       R0 := INT(A1)
[7]       END
[8]       PROCEND)
[9]     R5 := PROC("INT->INT",[A2],       % p2
[10]      START
[11]      R0 := INT(2)
[12]      END
[13]      PROCEND)
[14]    NOP                                           # useless-code elim on R6
[15]    NOP                                           # useless-code elim on R7
[16]    NOP                                           # useless-code elim on R8
[17]    UPDATE(R5,R4)
```

Further analysis on updated values would reveal that *R4* and *R5* are also useless and the corresponding instructions could therefore be eliminated.

9.5.2 Procedures Called Only Once

For procedures defined and called just once, a different transformation referred to in [Appel, 1992] can be performed instead of the more general inlining transformation, that of course also works in this particular case. If the procedure identifier is not updated and the corresponding procedure is called only once, the *CALL()* instruction is replaced by constructors for all parameters, a copy of the procedure body and a constructor to move the result. The procedure constructor is immediately replaced by a *NOP*, and in this situation, there is no need to rename procedure arguments and locals in order to preserve the unique-binding rule as only one constructor for each variable will remain after the transformation.

9.5.3 Drop Unused Arguments

Procedure calls can be simplified by removing unused parameters in order to alleviate register pressure. If a parameter *Ax* is declared in the list of parameters and not used in the body of a procedure *pp*, then *Ax* can be removed from the procedure body similarly to a dead-variable transformation, along with the corresponding actuals in all calls to the procedure *pp*.

To implement this transformation in TPL, an algorithm could be used that would discover and eliminate the unused parameters from the procedure body, as in dead-variable elimination (see Section 9.4.2), and record a list of parameters to be removed from calls. For each *CALL()* to a procedure the list of parameters to be removed would be interrogated and the list of actual parameters changed accordingly.

9.5.4 Tail Recursion

A procedure is tail-recursive when its last act is a call to itself and return the value of the result of the recursive call without any further processing. For tail-recursive procedures, recursion can be replaced by iteration as the current invocation will not use its frame any longer, so the call can be substituted by a jump to the first instruction of the body.

This transformation is not directly supported in TPL due to the way recursion is implemented as described in Section 8.2.3. The introduction of a recursive procedure constructor (*FIX* in the work described in [Appel, 1992]) would enable this transformation.

9.6 Using Continuations

A **continuation** is a function that expresses the computation to do next. Continuations can be used to model the flow-of-control of programs as they represent the control point to which control will be transfered after the current computation. Continuation-passing style (CPS) was introduced in Section 4.3 as an internal representation used in the simplification of compiling processes of many higher-order languages. RABBIT for scheme [Steele Jr., 1978], ORBIT for scheme [Kranz *et al.*, 1987] and SML/NJ for Standard ML [Appel and MacQueen, 1987], are the better known examples of the use of CPS for optimisations and code generation. CPS consists of adding a continuation to every procedure to represent the remaining execution of the program. The continuation will be called by the procedure with its result, instead of returning.

The use of CPS in the context of the compilation of PHOLs may simplify the runtime system because a runtime stack is not needed. The use of this representation will be described in the following sections.

9.6.1 CPS Transformation

The transformation implemented by the CPSt component transforms a TPL program into a TPLk program, that is, a TPL program expressed in continuation-passing style (see Section 9.7). As for each TPL instruction the continuation is already known and represented in the internal data structure by a pointer to the next instruction, or by two continuation instructions in the case of a conditional branch *BRA()*, only the return of a procedure needs to be transformed to a call to a continuation.

In the process of performing this transformation, another formal parameter (the continuation) is added to the parameter list of all procedures and to each procedure call. At the end of the procedure, a call is made to the continuation parameter. For each procedure call, a new procedure is built to serve as the continuation for that call. The algorithm described is illustrated by the following example, and its implementation will be described in Section 9.6.3. Consider the following fragment of a COREL program:

```
let a:= 10
let p1= proc(x: int -> int); x*a
let p2= proc(x: int); a:= a+x
let p3= proc(x: int -> int)
begin
  a:= 2*a
  let b= p1(15)
  a:= 3*b
  p2(2*a)
  a+b
end
let c= p3(a)
```

The same COREL program, when expressed in CPS, would be:

```
let a:= 10
let p1k= proc(x: int; k: proc(int) )
    begin
    k(x*a)                      !# call the continuation k with the result
    end
let p2k= proc(x: int; k: proc())
    begin
    a:= a+x
    k()                         !# call the continuation k
    end
let p3k= proc(x: int; k: proc(int) )
    begin
    a:= 2*a
    let k1= proc(r: int)        !# to call after p1 return
        begin
        let b:= r
        a:= 3*b
        let k2= proc()          !# to call after p2 return
            begin
            k(a+b)              !# call the continuation k with the result
            end
        p2k(2*a,k2)             !# call p2
        end
    p1k(15,k1)                  !# call p1
    end
```

```
let k3= proc(r: int)          !# to call after p3 return
        begin
        let c:= r             !# no cont to main program
        end
p3k(a,k3)                     !# call p3
```

k is the new argument for each procedure to represent the continuation to be called in the return position. If the procedure has a result, then it will be the actual parameter for its continuation, and otherwise, a continuation for a *void* procedure does not have a parameter (as with *p2*). It should be noted that the first instruction of *k1* (the continuation for *p1*) moves the result of the call (passed as a parameter to the continuation) to the appropriate variable (*b* in this case).

9.6.2 Consequences of CPS Transformation

Some consequences of the CPS transformation outlined in the previous example can be enumerated:

1. procedure call and return are unified;
2. the number of procedure constructors in the program is increased[4]; and
3. procedures contain a larger number of free-variables.

The unification of procedure call and return may be seen as an advantage of CPS, as because procedures never return, a stack of activation records is not needed to simulate lexical scopes, which may lead to a simpler runtime system. As procedures are sliced to create the needed continuations, locals that are declared before a procedure call and used after, become free-variables in the continuation after the call. For example *b*, which is a local in *p3*, is a free-variable of *k2*. Therefore, not only is the number of procedures greater than in TPL, but they also contain more free-variables in TPLk. Moreover, even temporary variables in expressions may become free variables, as there is no distinction between programmer names and compiler generated names. This can be seen as an advantage as it is then known that only locals to the current procedure are alive. That is one of the advantages of CPS, as reported in [Appel, 1992], page 135:

> "only the local variables of the current procedure [...] are 'roots' of garbage collection. Variables that, in conventional compilation, would be in activation records of 'suspended' procedures are, in the CPS version, free-variables of continuation closures. But the continuation closures are just ordinary records reachable from the local variables of the current procedure."

This property may make garbage collection easier as only local variables of the current procedure need to be inspected by the garbage collector. Space exhaustion need only be tested before every procedure call (safe points in the program) if the size of the activation record is known in advance. As control-flow is explicit and there will be no free-variables after closure conversion (see Section 10.3), inlining procedures as much as possible can reduce the impact on closure creation ([Appel, 1992]).

[4]And therefore larger number of closures is needed.

9.6.3 Implementation — CPSt

In the language framework prototype, the front-end generates TPL that can afterwards be transformed to CPS by the use of the CPSt component. CPSt traverses the TPL internal representation once to implement the CPS algorithm described in Section 9.6.1:

- the continuation argument is added to the list of arguments in each *PROC()* constructor;
- for each *CALL()* a new continuation is started to represent "what to do" after the call; in that continuation, the result of the original call, which will be passed to the continuation just started, is moved to the appropriate variable; in the current continuation, the original call is substituted by a *CALLK()* instruction with the continuation just finished to be constructed as its last parameter;
- for each *PROCEND* a *CALLK()* is also planted instead to call the continuation added as a parameter to the *PROC()* constructor corresponding to the *PROCEND*.

Existing TPL *PROC()* constructs and these newly created continuations all are named *CONT()* to express the fact that they were CPS-transformed. A program expressed in TPLk will have *CONT()* and *CALLK()* in the place of *PROC()* and *CALL()* (as referred to in Section 9.7).

To illustrate this transformation in TPL, consider again the COREL program of page 140, which will be translated into the following TPL program:

```
[6]     R4 := INT(10)                     % a
[7]     R6 := PROC("INT->INT",[A1],       % p1
[8]       START
[9]       R5 := MULT.INT(A1,R4)
[10]      R0 := INT(R5)
[11]      END
[12]      PROCEND)
[13]    R8 := PROC("INT->VOID",[A2],      % p2
[14]      START
[15]      R7 := PLUS.INT(R4,A2)
[16]      UPDATE(R4,R7)
[17]      END
[18]      PROCEND)
[19]    R14 := PROC("INT->INT",[A3],      % p3
[20]      START
[21]      R9 := MULT.INT(2,R4)
[22]      UPDATE(R4,R9)
[23]      R10 := CALL(R6,[15])            % b     # call p1
[24]      R11 := MULT.INT(3,R10)
[25]      UPDATE(R4,R11)
[26]      R12 := MULT.INT(2,R4)
[27]      VOID := CALL(R8,[R12])                  # call p2
[28]      R13 := PLUS.INT(R4,R10)
[29]      R0 := INT(R13)
[30]      END
[31]      PROCEND)
[32]    R15 := CALL(R14,[R4])             % c     # call p3
```

This program is transformed by CPSt into:

```
[6]     R4 := INT(10)                           % a
[7]     R6 := CONT("INT,PROC(INT->VOID)->VOID",[A1,A4],  % p1 # added cont A4
[8]        START
[9]        R5 := MULT.INT(A1,R4)
[10]       R0 := INT(R5)
[11]       END
[12]       CALLK(A4,[R0]))                            # call cont with result
[13]     R8 := CONT("INT,PROC(VOID->VOID)->VOID",[A2,A5],  % p2 # added cont A5
[14]       START
[15]       R7 := PLUS.INT(R4,A2)
[16]       UPDATE(R4,R7)
[17]       END
[18]       CALLK(A5,[]))                              # call cont
[19]     R14 := CONT("INT,PROC(INT->VOID)->VOID",[A3,A6],  % p3
[20]       START
[21]       R9 := MULT.INT(2,R4)
[22]       UPDATE(R4,R9)
[23]       R16 := CONT("INT->VOID",[A7],              # K1, cont for p1()
[24]         START
[25]         R10 := INT(A7)                 % b      # result of p1()
[26]         R11 := MULT.INT(3,R10)
[27]         UPDATE(R4,R11)
[28]         R12 := MULT.INT(2,R4)
[29]         R17 := CONT("->VOID",[],                # K2, cont for p2()
[30]           START
[31]           R13 := PLUS.INT(R4,R10)
[32]           R0 := INT(R13)
[33]           END
[34]           CALLK(A6,[R0]))                       # call cont for p3()
[35]         END
[36]         CALLK(R8,[R12,R17]))                    # call p2
[37]       END
[38]       CALLK(R6,[15,R16]))                       # call p1
[39]     R18 := CONT("INT->VOID",[A8],                # K3, cont for p3()
[40]       START
[41]       R15 := INT(A8)                   % c
[42]       STABLE()
[43]       END
[44]       CLOSE())
[45]     END
[46]     CALLK(R14,[R4,R18])                          # call p3
```

To the existing procedures *R6*, *R8* and *R14*, three new procedures, *R16*, *R17* and *R18*, which correspond to continuations, were added. The TPLk program can be transformed by OPT in order to try to get better characteristics. Instruction 25, declaring identifier *R10*, may be eliminated by copy propagation and the *MULT.INT()* operation on instructions 21 and 28 may also be modified to a *PLUS.INT()* operation by the application of a strength reduction transformation.

Inlining a TPL program expressed in CPS (that is, a TPLk program) can be performed using the same algorithm as described in Section 9.5.1 with *PROC* substituted by *CONT* and *CALL* by *CALLK*. The first inline transformation can only consider user-defined procedures as potential candidates, but if applied again, then it could inline calls for continuations as well (see [Appel, 1992] for examples of inlining CPS code).

9.7 TPLk and the Changes to TPL

CPSt transforms a program representation expressed in TPL into another equivalent representation following the "Continuation-Passing Style" expressed in TPLk. The differences between these two languages are the substitution of *PROC()* and *CALL()* by *CONT()* and *CALLK()* respectively. The instruction *PROCEND* is also substituted by a *CALLK()*. *CONT()* constructs a value of type *PROC* and has the same syntax as the *PROC()* constructor:

```
<id> := CONT("<type-rep-list>-><type-rep>",<id-list>,<tpl-block>)
```

CALLK() is used to apply a value of type *PROC*, but unlike *CALL()*, it is never followed by another TPLk instruction and it never binds a value to *R0* (the result):

```
CALLK(<loc>,<value-list>)
```

9.8 Conclusions

As TPL programs are saved in order to achieve longer-term persistence, the main goal of the high-level machine independent optimisation stage in the proposed architecture, is to minimise the size of each procedure in terms of TPL instructions. There is opportunities in the architecture for optimisations at another stage when code for a particular architecture is generated. The program representation will be analysed again, this time to perform machine dependent optimisations putting emphasis on the execution time as well as the size of the program. At this later stage, it is usual to apply "peephole optimisations" which include the same transformations already applied in the high-level representation, such as algebraic manipulation and useless-code elimination.

The optimisations discussed in this chapter make the program representation smaller by eliminating instructions, with the exception of inlining transformations which may enlarge the program. Despite that, and its dependency on target machine code information, inlining is considered because it introduces opportunities for other machine independent transformations. Inlining enables inter-procedural analysis which may reduce the size of the programs. The change in size after inlining, is taken into account to decide whether the transformation is worthwhile. A transformation of TPL into continuation-passing style was investigated, since it may simplify the runtime system. The salient consequences of this transformation are a larger number of procedure closures and a larger number of free-variables; the price paid for a simplification of the runtime system.

The transformations presented in this chapter were intended to assess the adequacy of the TPL design decisions. Transformation at the level of the instruction, basic block and procedure were investigated. All the transformations implemented involve only a very limited amount of analysis. More elaborate transformations, at the other levels of granularity (loops and whole program), could be performed as long as data-flow is performed to identify the loops, for example. A restricted version of code motion for constant values out of loops could be easily implemented in TPL based on the update information. TPL programs are suitable for loop transformations, as all intermediates are explicit and constants inside cycles are exposed. The transformations were performed only for variables which are not updated. The fact that updates are clearly identified by a single *UPDATE()* instruction makes it easier to discover program constants and so constant folding and constant propagation opportunities. Because in TPL all intermediates are named, the expressions to be analysed were very simple and the algorithms easy to implement. Having restricted the control-flow after the *then* and *else* of the conditional instruction *BRA()*, comparison folding transformations were easier. A simplification of the addressing modes in TPL to have only immediate and register addressing, together with the introduction of an explicit *SELECT()* operation to access fields of structures, would help in performing alias analysis, which could extend the scope of applicability of some transformations (e.g. inlining).

The internal graph representation is traversed (sometimes several times) by following the control-flow links in order to perform the discussed program transformations. There is a node in the graph for each TPL instruction and as many different node structures as TPL instructions. Because of that, implementation programs are verbose but easy to write. An improvement may be to have links to the instructions in the internal representation where a variable is defined, as this data-flow information would help in tracking all uses of a variable. To cope with longevity requirements, it should be possible to supply the procedures needed to perform all transformations when a new type is introduced in the framework. In the prototype, the addition of a new type and its instructions involve a change in all programs related to the optimiser.

Because a persistent programming language was used in the implementation, the internal representation could be used efficiently by different programs. This made the construction of a "plug and play" framework possible where each component performs a single analysis or transformation in the internal representation. The flexibility achieved helped during experimentation which supported the decision as to which transformations should be performed and how they should be performed. An algorithm for the optimisation of TPL programs was proposed which combines all the components and ensures termination. More heuristics need to be collected on larger programs and measurements need to be made in order to "tune" the proposed transformations.

The following chapter will describe the transformation which must be performed in TPL in order to have a representation suitable to be executed by a target machine.

Chapter 10

Abstract Machine and Object Store

For the three-stage architecture proposed in this thesis to support orthogonally persistent, reflective, higher-order, polymorphic languages, the usage of an internal representation independent of the source languages and of target architectures was illustrated and high-level and machine independent transformations in that representation were described. As procedures have nested scope and free-variables, and the program representation must have a predefined order of evaluation in order to be executable by a von Neumann machine, changes in the internal representation are still needed.

This chapter presents the design of a low-level abstract machine and discusses how it can be used to support TPL. The transformations which must be performed in TPL in order to achieve a representation suitable for execution are described and illustrated by fragments of programs. The runtime system, which supports object creation and access and the interaction with the underlying layers, is presented. Finally, the use of a garbage-collected object store to achieve persistence, target machine code generation and program execution with dynamic binding, are discussed.

10.1 Introduction

In Section 6.5 the set of possible compilation strategies was discussed. A strategy uses an abstract machine which consists of a set of registers plus a model of memory and corresponding addressing modes. Each strategy requires the following choices to be made:

1. how procedure closures are stored;

2. how procedure activation records are represented;

3. how parameters are passed from caller to callee;

4. how procedure call and return is performed; and
5. how the C stack can be used to enable low-level optimisations.

After analysing the set of possible compilation strategies in Section 6.5, it was decided that in this experiment:

1. the TPL program would be translated to a cTPL program where nested scope is eliminated and environment analysis has been performed in order to arrange for access to free-variables;
2. an infinite supply of registers would be used to store local and intermediate variables;
3. parameters would be packed into heap objects constructed on a per call basis;
4. C calls without parameters would support calls and return, or alternately, the CPS technique could be used; and
5. the infinite supply of registers would be supported by C locals.

The following sections will describe the abstract machine which supports cTPL, the environment analysis and closure conversion transformation performed in TPL so procedures will have everything they need to execute, the runtime system which supports object creation and accessing and the interaction with a garbage-collected object store to ensure persistence and stability. Finally, code-generation, the construction of executables and the dynamic linking of code into executables is described.

10.2 Low-level Abstract Machine

The cTPL abstract machine memory consists of a sufficient supply of registers, and a garbage-collectible heap, as represented in Figure 10.1. The **set of registers** contains three special registers with the following meaning:

R0 — result, on exit from a procedure;

A — argument pointer, referring to an object holding the parameters on entry to a procedure; and

C — closure pointer, referring to the closure record for the current procedure.

All the other registers are used to store locals and intermediate variables. The **heap** contains procedure closures and all non-atomic values, including code vectors which contain machine code and parameter packs. The abstract machine does not have a runtime stack, therefore data items are stored in registers and heap objects. Registers and heap objects may contain scalars, which represent atomic values (such as *R3* and *R4*), or pointers to objects, which represent aggregate values (such as *R1*, *R2* and *R5*). In order to track pointers for garbage-collection, accesses to atomic values are separated from accesses to pointers to objects. Access to values in cTPL can be interpreted as described in Table 10.1,. These addressing modes are the same as in Table 7.1 with register offset separated into scalar offset and pointer offset,

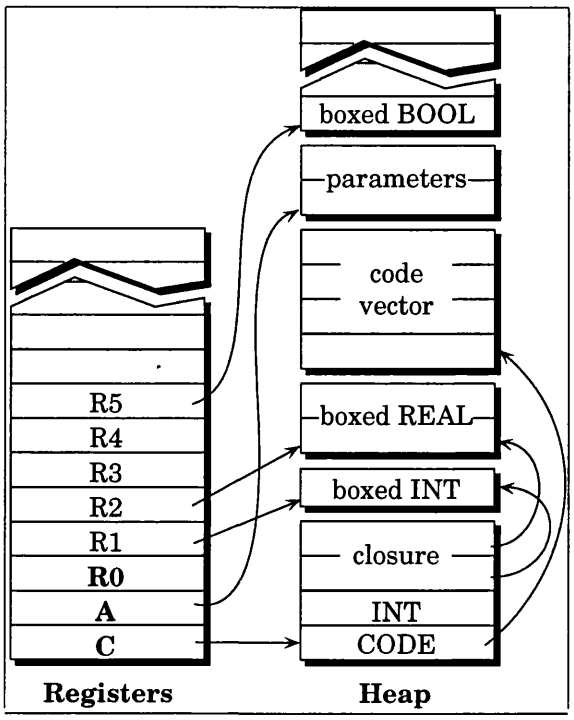


Figure 10.1: cTPL Abstract Machine

and register index into scalar index and offset index. Both the offsets and the indexes are in machine words. The instructions and data types in cTPL are basically the same as in TPL or in TPLk, with the differences enumerated in Section 10.5.

10.3 Environment Analysis and Closure Conversion

As explained in Section 3.5, when procedures are first-class values in a block structured language, then some sort of block retention mechanism is needed because variables may be used after leaving their scope. In TPL, procedures are first-class values and may access variables from outer scopes. For example, in the following program, there are four nested procedures, *R14*, *R12*, *R11*, *R10*:

```
[6]     R14 := PROC("INT,INT,INT->VOID",[A1,A2,A3],  % p1
[7]       START
[8]       R4 := INT(A1)                          % v1    # R4 <- A1
```

Addressing mode	Example	Meaning
immediate	x	immediate value x, e.g. *100*, *TRUE*, *String25*
register	Rn	value is in location *Rn*
register scalar offset	Rn!sx	value is in the x-th scalar from *RECORD* location *Rn*
register pointer offset	Rn!px	value is in the x-th pointer from *RECORD* location *Rn*
register scalar index	Rn@sx	value is in the x-th scalar from *VECTOR* location *Rn*
register pointer index	Rn@px	value is in the x-th pointer from *VECTOR* location *Rn*

Table 10.1: Addressing Modes in cTPL

```
[9]     R12 := PROC("->VOID",[],                % p2    # R12 nested inside R14
[10]      START
[11]      R5 := MULT.INT(R4,2)                   % v2    # R4 is free
[12]      R11 := PROC("->VOID",[],               % p3    # R11 nested inside R12
[13]        START
[14]        R6 := MULT.INT(R5,3)                 % v3    # R5 is free
[15]        R10 := PROC("->INT",[],              % p4    # R10 nested inside R11
[16]          START
[17]          R7 := PLUS.INT(A1,A2)                      # A1 and A2 free
[18]          R8 := PLUS.INT(R7,A3)                      # A3 free
[19]          R9 := PLUS.INT(R8,R6)                      # R6 free
[20]          R0 := INT(R9)
[21]          END
[22]          PROCEND)                                   # end R10
[23]        END
[24]        PROCEND)                                     # end R11
[25]      END
[26]      PROCEND)                                       # end R12
[27]    R13 := PLUS.INT(R4,1)
[28]    UPDATE(R4,R13)                                   # R4 <- R4 +1
[29]    END
[30]    PROCEND)
```

A procedure may access its local variables in addition to its formal parameters. Procedure *R14* accesses its local variable *R4* and its formal parameter *A1*. A procedure may also access a variable declared in a lexically enclosing scope, as does procedure *R12* in instruction 11 to use the value of *R4* in order to bind *R5*. Variable *R4* is a **bound** or **local** variable with respect to procedure *R14* and a **global** or **free** variable with respect to procedure *R12*.

The closure of a procedure, which includes all information needed for its execution, can be represented by any combination of memory and registers. In the experimental prototype, closures are represented by records in the heap following the technique described in [Davie, 1979] and similarly to the work of Cardelli in the implementation of the FAM [Cardelli, 1983]. For each *PROC()* instruction, a closure record is constructed which provides runtime access to all its free-variables. If a free-variable is not updated in the program (and is then, in fact, a constant) its value is copied to the closure; if not, then in order to maintain the program semantics the variable must be boxed inside a record and it is the pointer to the record that is copied to the closure. If a free-variable may be used later in the computation there must be some reference to it from some accessible closure, and therefore it will be retained; otherwise the space it uses may be garbage-collected. The closure record also contains a pointer to the code vector containing the executable code which knows how to access its free-variables by offsets from the closure, as represented in Figure 10.2 for procedure *R12* in the previous example.

Each TPL program must pass through an **environment analysis** pass which, for each procedure abstraction determines the set of its free-variables and arranges for access to the free-variables to be available in a closure record. This pass also determines if a variable needs to be boxed inside a *RECORD* object by evaluating the predicate:

$$box(x) : -free(x) \wedge updated(x)$$

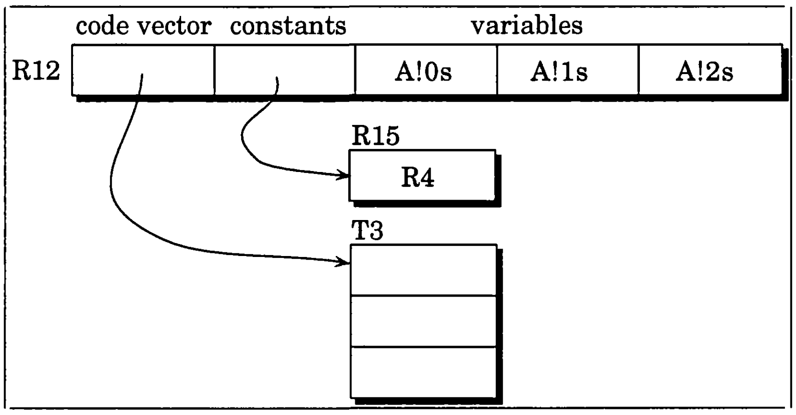


Figure 10.2: Closures in cTPL

If the variable is not updated, its identifier is copied to the closure; otherwise, the identifier to the enclosing *RECORD* is copied to the closure. In order to "close" a procedure definition, for each free-variable, an entry is made in the corresponding closure and also in all the enclosing closures until the closure which corresponds to the scope where the name is declared. This flat representation of closures involves a lot of variable copying between closures.

Other possible representations for the closures of program presented in page 148 are illustrated in Figure 10.3. In the **flat closures** representation, in (a), variables *A1*, *A2* and *A3* must be copied from the closure for *R12* into the closure for *R11* and then into the closure for *R10*, as they are used in *R10* as free-variables. In the **linked closures** representation, in (b), closures for procedures *R10* and *R11* are reusing the closure for *R12* and thus retaining the free-variable *R4* which is not free in *R11* or *R10*[1]. Another overhead involved in this later representation is the need to traverse the links in order to access some free-variables, e.g. variables *A1*, *A2* and *A3* from procedure *R10*. In the **shared closures** representation, in (c), variable copying is avoided by grouping variables with the same *lifetime* into a shareable record [Shao and Appel, 1994]. Variables *A1*, *A2* and *A3* can be shared by all three procedures. There is always only one link to follow, and therefore, access to free-variables is more efficient than in the linked closures representation. As access to free-variables is represented in cTPL by using a flat closure representation, faster access for each call is traded for more expensive creation of closures, which is performed only once. If a procedure is called several times, there will be gains in following this approach and because procedures are persistent it is expected that this situation is more likely to happen.

Following environment analysis there is a **closure conversion** pass which changes references to free-variables to explicit offsets from the closure [Appel and Jim, 1989]. After closure conversion, procedures know everything they need to execute. To pass a procedure as a parameter or store it in a data structure, the closure is used. Because access to free-variables is arranged through closures, procedure declarations may be lifted to the outer scope and thus the nested scope of TPL programs eliminated. This way, the cTPL program that results from environment analysis and closure conversion is closer to being executable by a von Neumann

[1]This space leak problem is even worse if activation records are not separated from the closure because then the all activation record will be retained.

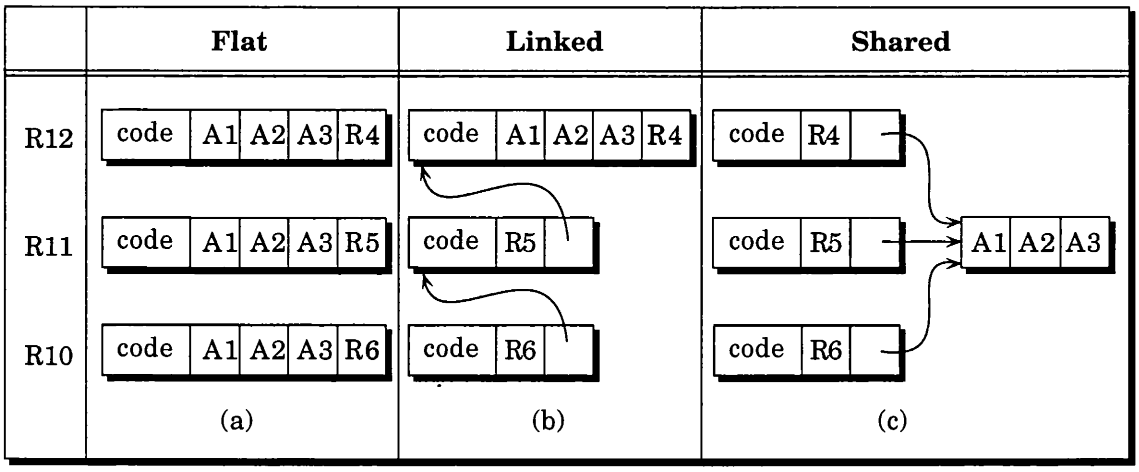


Figure 10.3: Possible Closure Representations

machine.

In the TPL program presented in page 148, the environment analysis around *R12* produces for the *PROC()* instruction of line 9:

```
[24]     R4 := INT(A!s0)                          % v1    # local of R14
[25]     R15 := RECORD([R4])                              # box R4 free in R12
[26]     R12 := CLOSURE([T3,R15,A!s0,A!s1,A!s2])  % p2 # code, R4, A1, A2, A3
```

Instruction 23 constructs *R4* a local of *R14*, which is free in *R12*. *R15* is an identifier to the *RECORD* which "boxes" *R4* as *R4* is updated in the program. The *CLOSURE()* instruction contains in the first field a value of type *CODE*, which knows how to access free-variable *R4* by using this closure, and a copy of *R15*. The formal parameters of the *R14* are also copied to the closure as they are free because they are used in the enclosed procedure *R10*. The code value referred to in the closure is:

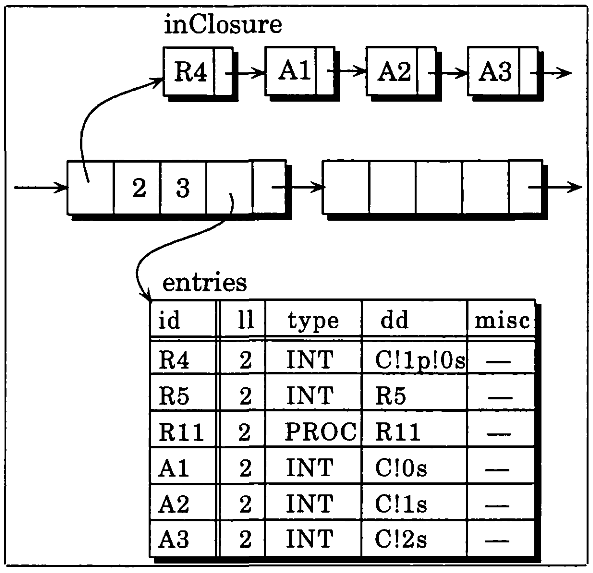
```
[15]    T3 := CODE(                               % p2
[16]      START
[17]      R16 := RECORD([C!p1])                           # R4 address
[18]      R5 := MULT.INT(R16!p0,2)                % v2    # R5 <- R4 * 2
[19]      R11 := CLOSURE([T2,R5,C!s0,C!s1,C!s2])  % p3    # code, R5, A1, A2, A3
[20]      END
[21]      PROCEND)
```

and the closure for *R12* is represented in Figure 10.2.

10.4 Putting It All Together — CLOSE

Environment analysis and closure conversion are implemented in the component CLOSE of the experimental prototype. This component is required in the compilation framework to commit TPL programs to a particular abstract machine architecture by establishing an access path for each variable and a contract for parameter passing between callers and callee. TPLk programs are transformed in the same way.

To perform environment analysis, TPL scoping is modelled by a symbol table. In Figure 10.4, the symbol table is represented for the situation where procedure *R12* is being



id	ll	type	dd	misc
R4	2	INT	C!1p!0s	—
R5	2	INT	R5	—
R11	2	PROC	R11	—
A1	2	INT	C!0s	—
A2	2	INT	C!1s	—
A3	2	INT	C!2s	—

Figure 10.4: CLOSE Symbol Table

processed. It should be noted that *(C!1p)!0s* is shorthand notation for *Rx!0s* after obtaining the address with *Rx:= RECORD([C!1p])*. The symbol table is implemented as a linked list of records, one for each TPL scope. The information in the record for each scope includes *inClosure*, a list of access paths to be included in the closure (one for each free-variable or free-constant), the last pointer slot and the last scalar slot used so far in the closure (2 and 3 in the figure) and *entries*, a map with entries for each identifier in the current scope. Each entry in this map has the following information associated with the identifier:

- **id** — the identifier;
- **ll** — the lexical level;
- **type** — the TPL type of the identifier;
- **dd** — the location where the value can be found; the location can be local or through the closure record for free-variables; and
- **miscellaneous** — scalar/pointer information needed for fields of values of type *RECORD* or indexes for values of type *VECTOR*.

In order to perform environment analysis and closure conversion and establish an abstract machine model of memory and execution, CLOSE traverses the internal representation of the TPL program four times. It should be noted that as environment analysis constitutes a whole program analysis, it prevents the program representation in TPL from being persistent, so TPL cannot constitute the basis for longevity and the result of this transformation, cTPL code, may be used instead.

first pass — uses the symbol table to determine access paths and discriminate pointer/scalar access to store objects. For each *PROC()* body, free-variables are identified, accesses are changed accordingly and stored in the *inClosure* structure of the symbol table so they can be used to build a closure when *PROCEND* is reached; updated locals and parameters used as free-variables are marked in the blackboard as "to be boxed later"; each *PROC()* is substituted by a *CODE()* constructor, which contains the changed code, and a *CLOSURE()* constructor for the record closure; *CODE()* instructions are lifted to the outer level; and finally, the sequence of instructions *START INIT* is inverted to *INIT START* and the same happens with the *END CLOSE* sequence.

second pass — performs the boxing of locals and parameters using the blackboard entries marked by the previous pass; accesses to those variables are changed accordingly.

This two passes through each TPL program are needed, as only at the end of each procedure (*PROCEND*) will it be known which variables need to be boxed as they may be free in enclosed closures. After this analysis, the procedure code will "know" how to access all values through its closure or, for values in the local scope, through registers.

The other two passes are needed to decide the parameter-passing strategy. Parameters are treated as locals declared before the procedure body. Later, a strategy of passing parameters in a heap object is implemented and the access paths changed to use an indirection through the *A* register, which points to the parameter pack object.

third pass — store in the blackboard access paths using the *A* register for parameters discriminating pointer/scalar access to the parameter pack store object.

fourth pass — change all access paths for parameters using the information in the blackboard.

Figure 10.5 summarises the different possible access paths for local variables, the corresponding access paths using the closure when the variable becomes free and a picture of the memory for each situation. Atomic values are exemplified by *INT* values and aggregate values contain only one slot with an *INT* value. If a local variable *Rx* holding a value of type *INT* is free and is not updated by the program, then it need not be boxed and is simply copied to one slot in the closure record (e.g. slot 0). After environment analysis, the access path will be *C!0s*, the first scalar in the closure (and similarly to the other atomic values of types *BOOL*, *DOUBLE*). If the same variable is updated, then it is boxed in *Ry* and the access path becomes *Ry!0s* or *(C!1p)!0s* for local access or access to a free-variable, respectively. Similarly, for non-atomic values (which are values that need a store object) the identifier if not updated can be used directly to access the object locally and *C!1p* to access the same object as a free-variable. To access the value of type *INT* which *Rz* contains, *Rz!0s* and *(C!1p)!0s* must be used instead. If the variable is updated, then it needs to be boxed in the record *Rw* and one more level of indirection must be used, as represented in the Figure 10.5.

To illustrate the action of CLOSE, consider for example the following TPL program:

```
[6]     R4 := VECTOR(1,3,2)                    % v     # let v= vector 1 to 3 of 2
[7]     R5 := INT(R4@2)                        % a     # let a= v(2)
[8]     R6 := INT(R5)                          % b     # let b:= a
[9]     R8 := PROC("INT->INT",[A1],            % p     # let p= proc(i: int -> int)
[10]      START
[11]      R7 := PLUS.INT(R6,R5)                        # b+a
[12]      R0 := INT(R7)
```

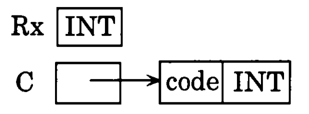
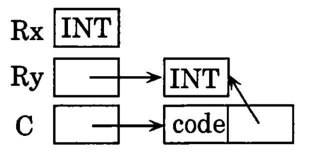
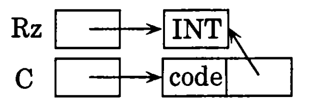
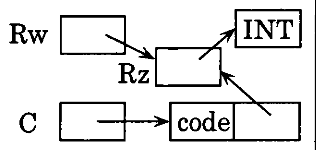
		Local	Free	Memory
atomic value	not updated	Rx	C!0s	
	updated	Ry:=RECORD([Rx]) Ry!0s	(C!1p)!0s	
aggregate value	not updated	Rz	C!1p	
		Rz!0s	(C!1p)!0s	
	updated	Rw:=RECORD([Rz]) Rw!0p	(C!1p)!0p	
		(Rw!0p)!0s	((C!1p)!0p)!0s	

Figure 10.5: cTPL Access Paths

```
  END
  PROCEND)
UPDATE(R6,10)                                  # b:= 10
R9 := MOVE.PROC(R8)                      % q   # let q:= p
R11 := PROC("INT->INT",[A2],                   # unnamed= proc(x: int -> int)
  START
  R10 := PLUS.INT(A2,10)                       # x+10
  R0 := INT(R10)
  END
  PROCEND)
UPDATE(R9,R11)                                 # q:= unnamed
R15 := PROC("->INT",[],                  % s   # let s= proc(-> int)
  START
  R12 := CALL(R8,[2])                          # call p, R8 free
  R13 := CALL(R9,[4])                          # call q, R9 free
  R14 := PLUS.INT(R12,R13)                     # p(2)+q(4)
  R0 := INT(R14)
  END
  PROCEND)
```

After closure conversion and environment analysis it will be represented in cTPL as:

```
T101 := CODE(                            % p   # code for p
  START
  R18 := RECORD([C!p1])                        # address of R6
  R7 := PLUS.INT(R18!p0,C!s0)                  # R6 (free) + R5 (free)
  R0 := INT(R7)
  END
  PROCEND)
T102 := CODE(                                  # code for unnamed
  START
  R10 := PLUS.INT(A!s0,10)                     # A2 + 10
[11]      R0 := INT(R10)
[12]      END
[13]      PROCEND)
[14]    T103 := CODE(                     % s     # code for s
[15]      START
[16]      R12 := CALLC(C!p1,[2])                  # call p (free)
[17]      R19 := RECORD([C!p2])                   # address of q
[18]      R13 := CALLC(R19!p0,[4])                # call q (free)
[19]      R14 := PLUS.INT(R12,R13)
[20]      R0 := INT(R14)
[21]      END
[22]      PROCEND)
[23]    T0 := CODE(                               # main
[24]      START
[25]      INIT()
[26]      R1 := PROOT()
[27]      R2 := LOOKUP:RECORD("matchType",R1)
[28]      R3 := LOOKUP:RECORD("matchConst",R1)
[29]      R4 := VECTOR(1,3,2)             % v
[30]      R5 := INT(R4@p2)                % a
[31]      R6 := INT(R5)                   % b
[32]      R16 := RECORD([R6])                     # box b
[33]      R8 := CLOSURE([T101,R16,R5])    % p     # code, boxed b, a
[34]      UPDATE(R16!s0,10)                       # b <- 10
[35]      R9 := MOVE.RECORD(R8)           % q
[36]      R17 := RECORD([R9])                     # box q
[37]      R11 := CLOSURE([T102])                  # closure for unnamed
[38]      UPDATE(R17!p0,R11)                      # q <- unnamed
[39]      R15 := CLOSURE([T103,R8,R17])   % s     # code, p, boxed q
[40]      STABLE()
[41]      CLOSE()
[42]      END
[43]      PROCEND)
```

It should be noted that the identifier *T0* always represents the outer scope of a cTPL program and the way *C!p2!p0* is calculated in instructions 17 and 18 to access *q*. The use that this program makes of the cTPL abstract machine is represented in Figure 10.6. The vector *R4* constructed at instruction 29 will be allocated in the heap and initialised with 2, as represented. *R6* is boxed inside the record *R16* because it is a free-variable of procedure *R8*. The closure for *R8* is built in instruction 33 with a pointer to the code vector *T101*, *R16* and a copy of the free-variable *R4* (which is not updated). At instruction 35, the closure for *R8* is copied to identifier *R9*. *R9* is then boxed inside *R17* because it will be used as a free-variable and is updated at instruction 38 with the closure for the unnamed procedure. Instruction 39 constructs a closure to procedure *R15* with the procedures *R8* and *R9* (boxed inside *R17*) as free-variables.

10.5 cTPL and the Changes to TPL

After performing environment analysis and closure conversion in a TPL or TPLk program, it will be expressed in cTPL. The differences between the two languages are the absence of *PROC()*, *CONT()*, *CALL()* and *CALLK()* instructions in the program representation and the introduction of the new instructions *CODE()*, *CLOSURE()*, *CALLC()* and *CALLKC()*. Each

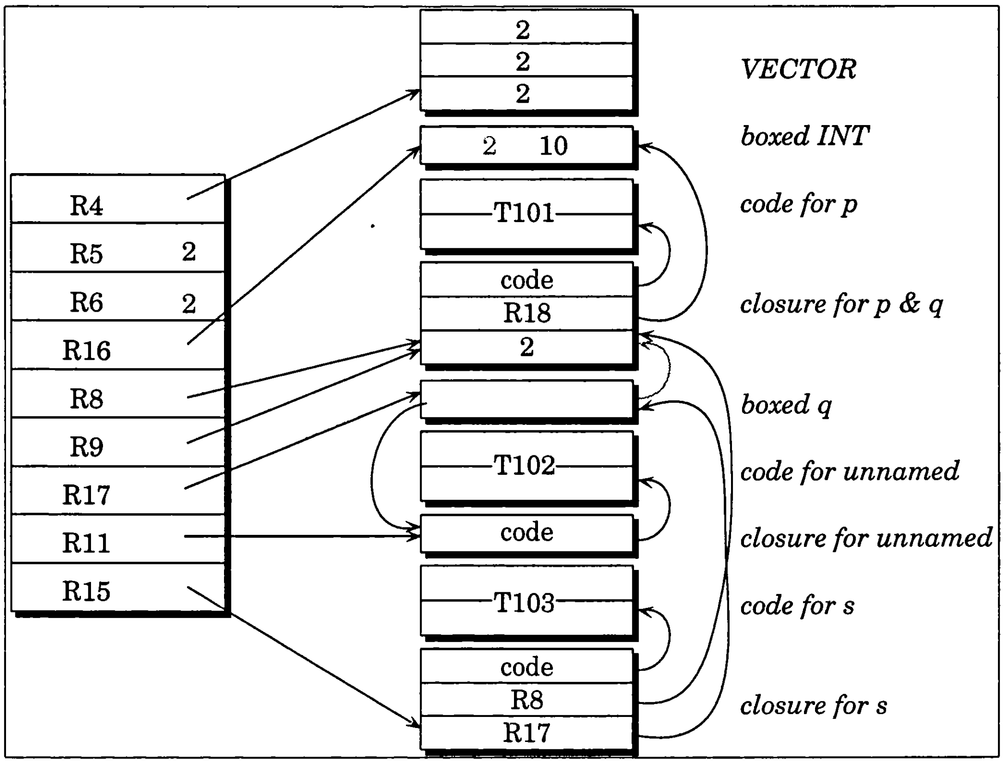


Figure 10.6: Example of Value Creation and Access in cTPL

occurrence of both *PROC()* and *CONT()* is substituted by the use of the new constructors *CODE()* and *CLOSURE()*. *CLOSURE()* constructs a value of type *RECORD* and has the same syntax as the *RECORD()* constructor:

```
<id> := CLOSURE(<value-list>)
```

Occurrences of *CALL()* are substituted by the new *CALLC()* instruction, which has the same syntax as *CALL()* and can be used to apply a closure:

```
<id> := CALLC(<loc>,<value-list>)
```

Occurrences of *CALLK()* are substituted by the new *CALLKC()* instruction, which has the same syntax as *CALLK()* and can be used to apply a closure for CPS code:

```
<id> := CALLKC(<loc>,<value-list>)
```

10.5.1 Operations on Code Vectors

cTPL has support for code vectors as value of type *CODE*.

Constructors To bind a value of type *CODE* to a new identifier *id*, the *CODE()* constructor may be used. The *CODE()* constructor has the following syntax:

```
<id> := CODE(<tpl-block>)
```

and can be used only in the outer scope. Examples of the construction of a values of type *CODE* were presented in the program on page 154.

TAG	Object Type
CHDR	Scalar Cell
DHDR	Pointer Cell
LHDR	Record
RHDR	Persistent Root
SHDR	Byte Array
THDR	Code Array
VHDR	Scalar Array
WHDR	Pointer Array

Table 10.2: Runtime Object Formats

10.6 The Runtime System

The runtime system is intended to support cTPL programs in terms of object creation and accessing (the abstract machine heap), its interaction with a persistent object store and its interaction with the operating system environment. Therefore the runtime system constitutes a layer that isolates the code-generator from the details of the object store and operating system.

In the prototype language framework, the runtime system is implemented in C through a library of functions that can be used by the C-- generated from cTPL by 2C. This library is linked with the C object files in order to make the executable program (see Section 10.9). Ideally, the interface should be described by a set of macros, and inlined code should be generated in the prototype in order to achieve faster target machine code. At the moment, inlining can only be achieved by having the function calls inlined by the compiler for the C code generated. The runtime set of macros would change for each different object store in order to adapt the runtime needs to the particular store interface. In this situation, inlined code for direct manipulation of objects could not, however, be generated.

10.6.1 Runtime Support for cTPL

For all non-atomic values in cTPL, the runtime system must provide ways to create a new value or get components of the aggregate object and update those values. Atomic values will reside in general purpose registers of the abstract machine and may be copied to fields of values of type *RECORD* when boxing is needed. They may also be aggregated in values of type *VECTOR* and *RECORD* or stored into an object of type *MAP*. Aggregate values go to the heap and are tagged with their type.

Object types include the formats presented in Table 10.2. Objects of type *Cell* are used to support *MAP* values (used in turn to support COREL *env* values as shown in Section 8.6) and its operations; the *Persistent Root* object supports the distinguished root of persistence; the object type *Byte Array* can be used to support values of types *CHARS*, *BITS* and *PIXEL*, (alternatively, a more specialised *Bit Array* object could be introduced to support *BITS* and *PIXEL* values); to support values of type *VECTOR* and their operations, the *Scalar Array* and *Pointer Array* object formats may be used.

The *Record* object format is the most general format; it aggregates any number of scalar and pointer components and can thus be used to support new cTPL types. This format directly supports values of type *RECORD* and its operations. In cTPL programs, values or type *RECORD* are mapped in the prototype to C-- using the C functions with the following signatures:

```
OID mkrecord(const int np,OID ptrs[],const int ns,const WORD scalars[]);
```

which returns the object identifier (OID) of a heap object of format *Record* holding the number of pointers given by the first parameter and the number of scalars given by the third. The object is initialised with the pointers given by the second argument and the scalars given by the fourth argument.

```
int poffword(OID r,const int n,const WORD w);
int poffwordp(OID r,const int n,const OID wp);
```

poffword and *poffwordp* update a field given by the second argument of the object whose OID is the first argument, respectively with the scalar or pointer given as the third argument.

```
WORD goffword(const OID r,const int n);
OID goffwordp(const OID r,const int n);
```

Similarly, *goffword* and *goffwordp* return fields fetched from an object.

Values of type *CODE* are supported by the runtime system in objects of type *Byte Array*. The C function

```
OID mkcode(const char *fname);
```

returns the OID of an object of format *Code Array* filled with the object code read from the operating system file with basename given by the argument and suffix *.o*. The basename of the file is stored in the object to be used as the name of the function. Different names are generated for each C function and therefore the names are unique.

```
OID gcode(const OID c);
```

returns the address in the memory map of the running process of the code stored in an object of format *Code Array* with OID given by the argument. That address can be used as entry point in a call to the corresponding procedure. The procedure name is fetched from the *Code Array* object and used to see if it is already linked in the running executable; if it is, then the memory address is returned. If the name is not known to the executable, the code needs to be copied to the memory map of the process and the name linked into the executable; the entry point is returned in case of success.

The runtime system also includes C functions to initialise and shutdown the heap, with the signatures:

```
void init();
void shutdown();
```

It must also include functions to support operations on values of the cTPL type *INF*, functions to support the cTPL instructions *STABLE*, *RESTART*, *GC*, functions to perform coercion between cTPL values of different types (as presented in Table 7.2), support for checkpointing and rollback, concurrency and threads, etc.

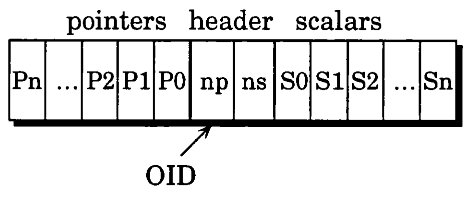


Figure 10.7: Store Object Format

10.7 The Persistent Object Store

In order to support the abstract machine heap of objects and ensure persistence and stability, a persistent object store is used. This store must support all TPL features introduced to match the needs of the PHOL language (identified in Section 10.6) namely, the provision of orthogonal persistence and efficient access to data across the spectrum of persistence, the support of polymorphic data structures, higher-order procedures and abstract data types, and finally, the provision of dynamic binding to support reflection and incremental program construction. The runtime system uses the functionality of the underlying object store to provide object creation and access.

As the main goal of this research is not about *persistent object stores,* a simple store is presented in the following sections just for illustrative purposes. This is a prototype of a single-user, memory-mapped, single-level store that was built to make the prototype complete for the compilation framework being demonstrated.

10.7.1 Store Object Formats

The objects in the store conform to a single general format, having pointers separated from scalars and with pointers always pointing to the header of an object. This condition enables the operation of garbage collectors, which need not know details of the content of the objects but must be able to find and follow all valid pointers. Other agents which may be scanning the object store can only find the pointers by starting from an object and then following all its pointers. The store object format is shown in Figure 10.7. An object is always referred to by its OID which is guaranteed to be unique in the store. The header, which is pointed to by an OID, contains the number of pointers and the number of scalars. Pointers and scalars grow in opposite directions from each side of the object header. Given this format, both the number of pointers *np* and the number of scalars *ns* must be known when an object, is created and moreover, it can never change. This prevents the object format from being dynamically changed, which was a need not felt in this experiment. Assuming that the *OID* has the size of a machine word, scalar and pointer field indexes start at word *0* and reside in the word given by:

$$scalar_i = OID + 2 + i$$

$$pointer_p = OID - p$$

The runtime data formats map directly into the store object formats shown in Figure 10.8,

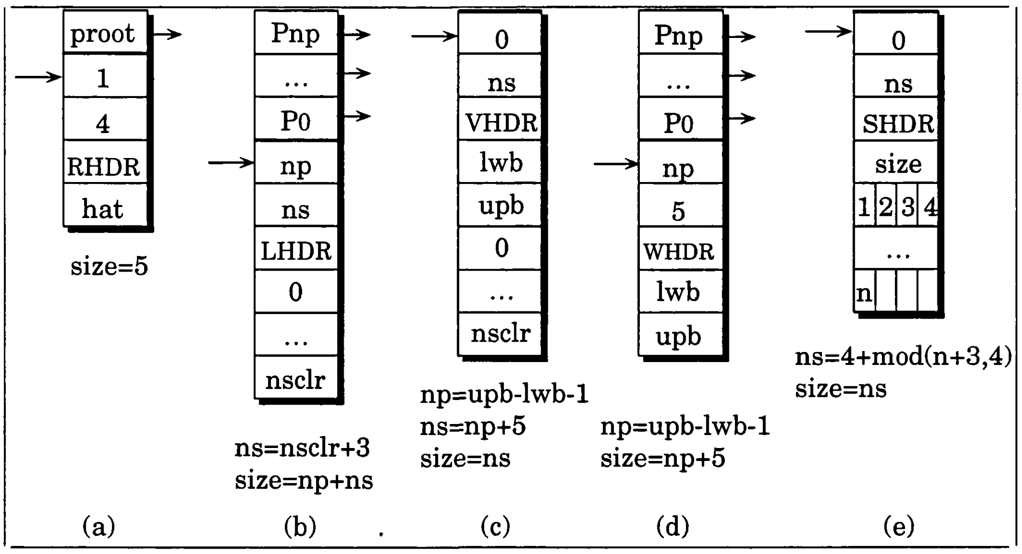


Figure 10.8: Store Objects: (a) Persistent Root; (b) Record (c) Scalar Array; (d) Pointer Array; and (e) Byte Array

10.9 and 10.10. The size of each object is given in machine words and the object tag is always at the first scalar (*S0*). The way procedure closures are represented which is illustrated in Figure 10.9 should be noted. Target machine object code is stored in an object with the *Code Array* format (THDR) which contains a pointer to an object of format *Byte Array* (SHDR) used to store the name of the corresponding function. The OID for the *Code Array* (THDR) object is stored into the object of type *Record* that holds the procedure closure. As was mentioned in Section 5.1 on page 60, the store may move to new *eras* in order to cope with longevity. Because of that, the *Code Array* object also includes two more fields:

era – which encodes the code creation era; and

ctpl – which points to a *Byte Array* object which contains the cTPL code linearised and expressed into a platform-independent representation (a stream of bytes).

An example of a platform-independent data representation as a linear stream of bytes is TXR [Mueller *et al.*, 1997], which may be used to represent the stored platform-independent internal program representation. This representation makes the target machine object code available with only one indirection, but the change to a new *era* involve the creation of a new *Code Array* object and copying the *ctpl* and *name* fields.

10.7.2 Persistent Values

In cTPL, persistence is implemented by reachability from a distinguished root of persistence. A naming convention can be imposed on the persistent store by using values of type *MAP*, as shown in Section 8.6. A value can itself be another map and thus maps can be nested.

Figure 10.10 shows the state of the persistent store for the example presented in Section 8.6 on page 113, Figure 8.1. The *Persistent Root* object (RHDR) points to a *Pointer Cell*

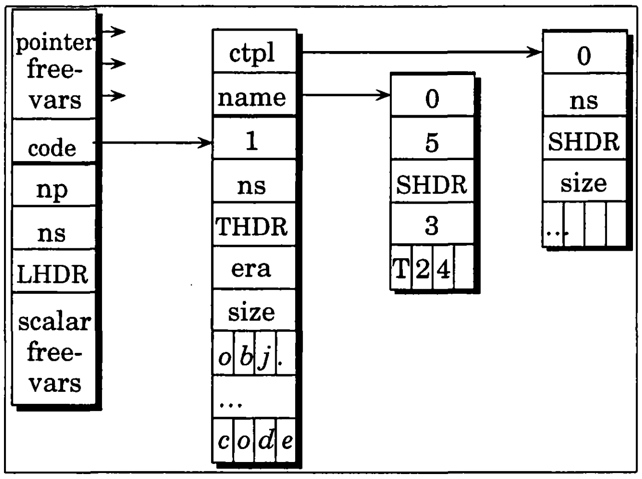


Figure 10.9: Closures and Code Array Objects

object holding the map corresponding to *PS()* which contains another map *E*. The last element of the map *E* is the first value inserted; a record with two fields: the value *10* of type *INT* and the *BOOL* value *FALSE* to represent the constancy. It is represented by a *Pointer Cell* object (DHDR) with type encoding *14*, name *i* and a value in a *Record* object (LHDR) containing value *10* and constancy *0* (FALSE). Similarly, the map that represents *PS()* is terminated by the closure for predefined procedure *environment()* boxed inside a *RECORD* with the constancy flag.

It should be noted that if a closure is made persistent, then the corresponding target machine code will also persist due to the fact that closures always contain a pointer to the code in their first pointer. Similarly for free-variables and the cTPL intermediate representation since they are accessible from the closure.

10.7.3 Garbage Collection

The garbage collector is not yet implemented in the language framework prototype. In this section, its outline design is presented to indicate its implementation is feasible.

Whenever there is no more space available to create a new object on the heap, an explicit garbage collection is requested by a program *GC()* instruction or a checkpoint is performed by *STABLE()*, and a garbage collection takes place. A garbage collector works by following all the pointers and retains all objects reachable from a set of roots. The garbage collector must therefore be informed of all possible roots of persistence, which include the root of persistence given by *PROOT()*, the abstract machine registers *A*, *C*, *R0* and all general purpose registers *R1* to *Rn* holding pointer values[2]. In cTPL, it is easy to identify which are the registers that contain pointers, as the instructions that bind a new value are typed. Therefore a mask, to inform the garbage collector of the pointers in general purpose registers, can be constructed at any time.

[2]Agents working in the store in order to improve object clustering are other examples where pointers need to be identified, further to the garbage collector and in stabilisations.

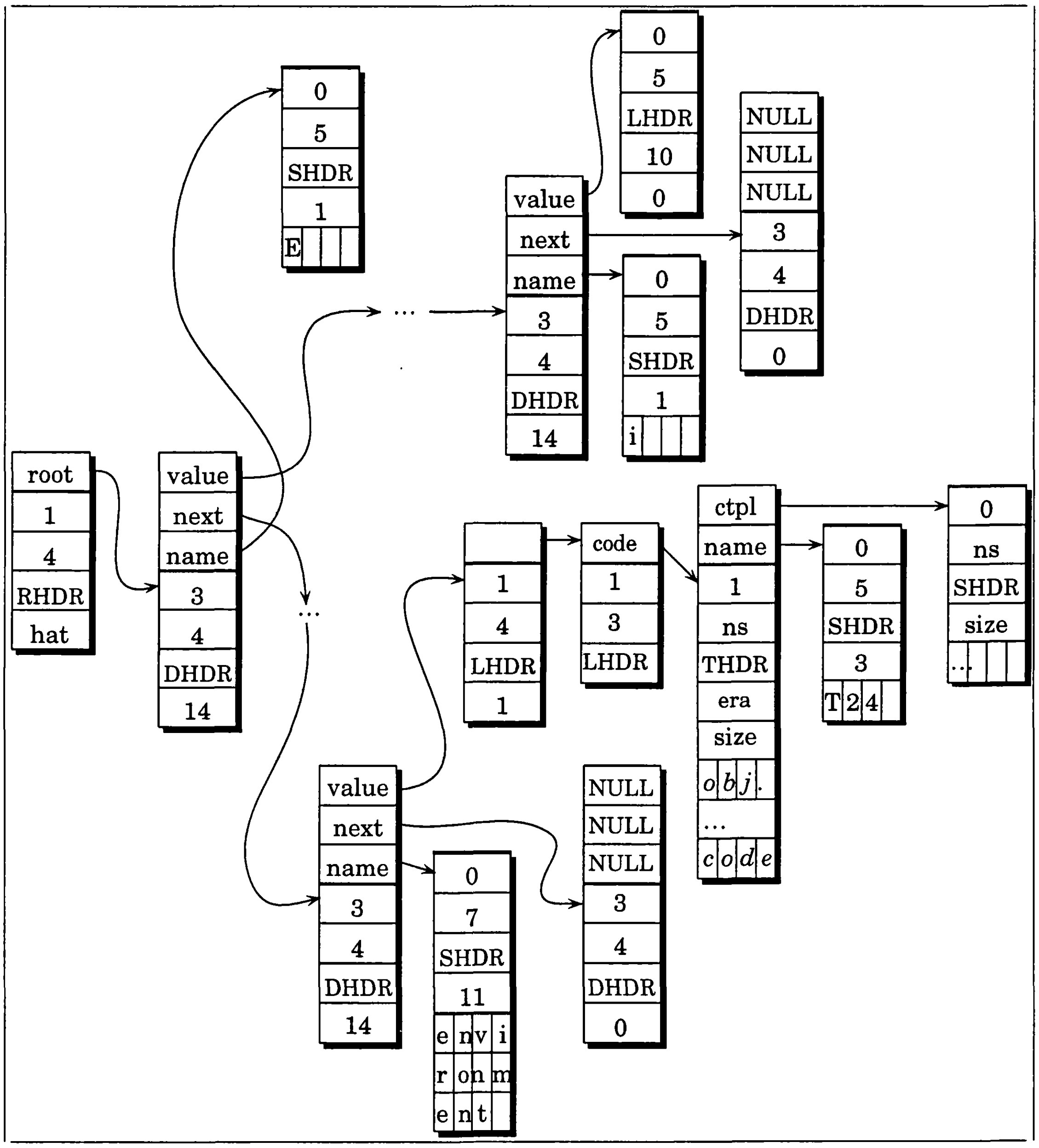


Figure 10.10: Support for Persistent Values

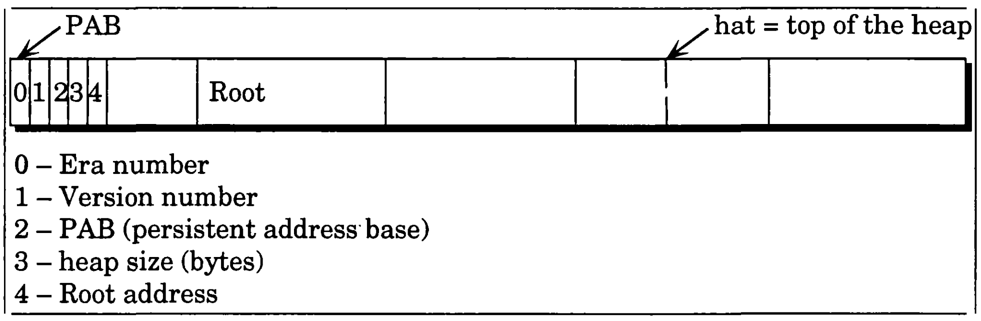


Figure 10.11: Memory Mapped Store

In the process of garbage collecting the heap, objects may move their positions and therefore variables pointing to them must change their values accordingly. Code vectors with target machine object code may also change position and so object code must be relocatable. The decision made to use C locals to support the general purpose registers complicates the task of identifying pointers as will be discussed in Section 11.6.4.

10.7.4 Implementation — STORE

The object store prototype (STORE) is implemented using SUNOS memory mapped I/O. The implementation was based on experiments made by Paul Philbrow on using the kernel call *mmap()*. Memory mapped I/O consists of mapping stable storage in a disk file into a buffer in memory so that when bytes are fetched from the buffer, the corresponding bytes of the file are read. Similarly, data stored in the buffer is automatically written to the file. A call to *mmap()* informs the kernel which file to map into what memory region.

The store implementation includes the C functions:

```
OID create();
OID awake();
void asleep();
```

create() opens an operating system file to hold the persistent store, sets the size of the store, maps the file in the process address space using *mmap()*, writes in the first page of the store the words that identify the store and writes the *Persistent Root* object (RHDR) as depicted in Figure 10.11. An empty map is also created and its OID stored into the *proot* field of the *Persistent Root* object. *awake()* opens the operating system file, confirms whether it represents the expected store by comparing the *era* and *version* words with their values in the store and maps the file into the process address space. *asleep()*, un-maps and closes the file.

10.7.5 Discussion

The store used in this experiment is minimal and should be seen as a component built with the sole purpose of enabling the demonstration that the proposed architecture is feasible. It lacks important constructs. The most critical for this research is the non-existence of a garbage collector. An emphasis on efficiency is needed in a more elaborate store. For example, a hash-table or a B-tree must be used instead of the list of *cell* objects to support persistent value creation and access. The store interface could also contain memory move functions to speed-up copies between objects and all allocation should be done inline.

Other stores may be used to support the runtime system. The Napier88 Munro Store [Munro, 1993] or the Texas Persistent Store [Singhal *et al.*, 1992] are natural candidates. It suffices to map the runtime functions to the particular interface to have cTPL programs using it. However, extra information can be supplied by compiled code to the underlying store. Alternatively, a lower level support may be used to build the needed functionality of the object store. RVM, a recoverable virtual memory system in UNIX [Mashburn and Satyanarayanan, 1994], could serve as such a support tool to build object abstractions.

10.8 Code Generation

After the conversion of a TPL program into cTPL, all procedure definitions are at the same level (there are no nested scopes) and the access to all values is explicit. The program is thus in a form suitable for execution by a von Neumann machine. The transformation of this intermediate representation into C--, a restricted form of C language programs, will be described in the following sections.

10.8.1 Using C--

The cTPL internal representation is traversed and each instruction is expanded into a fragment of C source code by the component 2C. cTPL variables in general purpose registers are mapped on to C local variables and the C operators are used to perform cTPL primitive operations whenever they have the required semantics; otherwise, a C function is added to the runtime system to implement the operation. The abstract machine registers are held in C global variables. Alternatively, a non-standard feature of GCC can be used to keep those C variables in machine registers. All heap objects are created by calls to C functions of the runtime system, as well as accesses to fields of heap objects, and *CODE* values are represented by C functions without arguments.

The body of each *CODE* value is translated into C and written to a UNIX file. At the end of the body, the GCC compiler is called on that file, and the resultant relocatable object code is stored into a code object by using *mkcode()*. Consider, for example, the following part of a cTPL program:

```
T40 := CODE(                    % p     # code vector for p
  START
  R5 := RECORD([10])            % s     # struct(a= 10)
  R6 := MULT.INT(A!s0,C!s0)             # A1 * R4 (free)
  R7 := PLUS.INT(R6,R5!s0)              # R6 * s(a)
  R0 := INT(R7)                         # result
  END
  PROCEND)
```

It will be translated to the following C code which was manually annotated for better understanding:

```
#include "../C/runtime.h"
void T40()                                          /* code for procedure R8 */
{
  WORD R5 = 0; WORD R6 = 0; WORD R7 = 0;                   /* declare locals */
  {
    int np = 0; WORD *ptrs[1];                             /* make record R5 */
    int ns = 0; WORD sclrs[1];
    sclrs[ns++] = (WORD) 10;                                   /* init field */
    R5 = (WORD) mkrecord(np,ptrs,ns,sclrs);
  }
  R6 = goffword((OID) A,0) * goffword((OID) C,0);     /* A1 * R4 (C!0s, free) */
  R7 = R6 + goffword((OID) R5,0);                               /* R6 * s(a) */
  R0 = (WORD) R7;                                                  /* result */
}
```

Before a *CALLC()* instruction, parameters are moved to a new heap object, current values of machine registers *C* and *A* are saved and changed to new values corresponding to the OIDs of the closure and of the parameter pack, respectively. The target machine object code is obtained from the *Code Array* object referenced from pointer slot 0 of the closure into the process memory and dynamically linked by using the *gcode()* runtime function. The entry-point to the C function is applied, and after its return the abstract machine registers are restored and, if there is a result, it will be moved from *R0* to the intended location. To illustrate the call sequence, consider another part of the previous cTPL program:

```
[15]      R4 := INT(0)                    % a
[16]      R8 := CLOSURE([T40,R4])         % p    # code, free R4
[17]      R9 := CALLC(R8,[10])            % b    # call p
```

and the corresponding annotated C code:

```
#include "../C/runtime.h"
int main()
{
  WORD R1 = 0; WORD R2 = 0; WORD R3 = 0;          /* declare global variables */
  /* ... */                                                     /* preamble */
  R4 = (WORD) 0;                                           /* initialise R4 */
  {
    int np = 0; WORD *ptrs[1];
    int ns = 0; WORD sclrs[1];
    ptrs[np++] = (OID) mkcode("T40");     /* produce relocatable object code */
    sclrs[ns++] = (WORD) R4;                                     /* free R4 */
    R8 = (WORD) mkrecord(np,ptrs,ns,sclrs);                /* closure record */
  }
  {
    int np = 0; WORD *ptrs[1];
    int ns = 0; WORD sclrs[1];
    sclrs[ns++] = (WORD) 10;
    pbuf = (WORD) mkrecord(np,ptrs,ns,sclrs);              /* parameter pack */
  }
  {
    int sc = C; int sp = A;                        /* save machine registers */
    C = (WORD) R8; A = pbuf;                               /* set new values */
    ((int (*) ()) gcode(goffwordp((word *) C,0)))();                /* apply */
    C = sc; A = sp;                                     /* restore registers */
  }
```

```
    R9 = R0;                                                /* move result */
    stabilise();                                               /* epilogue */
    shutdown();
    return 1;
  }
```

Section C.4.6 contains a complete example with the C code generated for a cTPL program.

As explained in Section 7.8.10, external C functions may be called by using the *CALLCC()* instruction and the result of the call, if any, can be bound to a TPL identifier, as in:

```
[6] R10 := CALLCC("malloc",[1024],"/usr/lib/libc.a")
```

The library *libc.a* with the path name provided is dynamically linked into the executable at the time of the call. The C code generated is:

```
dld_link_file("/usr/lib/libc.a");
R10 = (WORD) malloc(1024);
```

The dynamic link-editing is performed by the call to the *dld* function, as will be described in Section 10.9.1.

10.8.2 Code for TPLk Programs

When the program is expressed in TPLk because it was transformed by CPSt, the C-- program is different from that presented above; instead of *CALLC()* there will be *CALLKC()* instruction and the translation to C-- is also different.

In an *CALLK()* instruction there is no need to save and restore abstract machine registers as the current closure or parameter record will not be needed anymore. The C stack frame is also not needed and may be discarded. Therefore, instead of calling the continuation, which would make the C stack grow, the C function returns the address of the function to be called [Steele Jr., 1978, Tarditi *et al.*, 1990, Peyton-Jones, 1992]. An interpreter dispatches function calls and with that extra cost the C stack never grows more than 2 levels[3], and the dispatcher is simply:

```
int (*cont)() = (int (*)()) T0;                           /* first function */
while (cont)
    cont= (int (*)()) (*cont)();                           /* dispatch next */
```

and the last function returns *FALSE* to finish dispatching.

Consider, for example, the TPLk program presented in Section 9.6.3 page 142. Part of the translation of procedure *R14* (which calls its continuation in the end) together with the translation of procedure *R18* (which is the last function to be called) are presented here for illustrative purposes:

```
#include "../C/runtime.h"
int T64()                                                  /* code for R14 */
{
    /* ... */
  {
```

[3]In [Peyton-Jones *et al.*, 1993] it is mentioned how this overhead can be removed by using jumps with the inline assembly facilities of *GCC*. Because GCC has first-class labels, this non-standard feature could be used with gotos instead.

```
      int np = 0; WORD *ptrs[4];                                /* closure record */
      int ns = 0; WORD sclrs[1];
      ptrs[np++] = (OID) mkcode("T63");                  /* code for continuation */
      ptrs[np++] = (OID) goffwordp((OID) C,1);                /* free-variables */
      ptrs[np++] = (OID) goffwordp((OID) A,0);
      ptrs[np++] = (OID) goffwordp((OID) C,2);
      R16 = (WORD) mkrecord(np,ptrs,ns,sclrs);           /* cont closure for p1() */
    }
    {
      int np = 0; WORD *ptrs[1];
      int ns = 0; WORD sclrs[1];
      sclrs[ns++] = (WORD) 15;
      ptrs[np++] = (OID) R16;
      pbuf = (WORD) mkrecord(np,ptrs,ns,sclrs);               /* parameter pack */
    }
    C = goffwordp((OID) C,3);                        /* set to closure to be called */
    A = pbuf;                                           /* set to parameter pack */
    return (int) gcode(goffwordp((OID) C,0));                        /* call p1 */
  }

  #include "../C/runtime.h"
  int T65()                                                      /* code for R18 */
  {
    WORD R15 = 0;
    R15 = (WORD) goffword((OID) A,0);                        /* R15= INT(A!0s) */
    stabilise();                                                     /* epilogue */
    shutdown();
    return FALSE;                                        /* to finish dispatching */
  }
```

The C program which will be transformed into an executable includes the procedure *T0* and the C main function (which comprises only the dispatcher cycle).

10.8.3 Machine Dependent Optimisations

The C programs are transformed into target machine code by the GCC compiler. At the time target machine code generation is performed by GCC, several optimisations may take place [Stallman, 1992]. GCC performs inlining even for procedures that contain loops, tails-recursive calls and gotos if the flag *-O3* is specified. It simplifies jumps to the following instruction and jumps to jumps followed by unreachable-code elimination, common subexpression elimination and constant propagation, instruction scheduling and local register allocation. This way, low-level and machine dependent optimisations are applied to the cTPL program leading to better target machine code programs. The GCC optimisations are somehow limited by the fact that global inter-procedural optimisations are not possible because of the way C code is generated.

10.9 Constructing an Executable — JUICE

In order to construct a UNIX executable (with name *tpl.e*) the JUICE component links the object code for program's outer scope, which was stored into a file called *T0.o* by 2C, with the runtime library and the dynamic-linking library. The following command may be used:

```
gcc -static -L./C -L./C/dld -o tpl.e CPROGS/TO.o -lrun -ldld
```

The program may then be executed simply by issuing the command:

```
tpl.e
```

10.9.1 Dynamic Binding and Linking

During the execution of the program, dynamic linking takes place whenever a procedure closure is called and its code is not yet linked into the executable. This may involve a previous compilation of cTPL code into C-- followed by the code generation of target machine code if the *Code Vector* belongs to a previous *era*. In a persistent environment allowing incremental compilation, one program may have instructions to fetch code from the store and call it. A runtime error must occur if the corresponding executable is launched before the executable that creates the code. The *gcode* runtime function ensures that.

Static linking cannot be used to solve the requirements of performing a link-editing before the call, as it requires all global symbols to be well defined at link time and, as shown, procedure closures are fetched dynamically from the persistent object store after the executable is constructed. In the SUNOS 4.x operating system, load-time linking is available and used usually to link shared libraries. This solution is not enough, as the dynamic link-editor (ld.so) is called only once to search and load the missing routines before the control is passed to the main procedure of the program. A simple solution is available, when using an interpreter and interpretative code for an abstract machine, since the new code can be bound to the runtime support by the interpreter.

The dynamic link-editing is performing in the prototype by using the GNU **dld** library package [Ho and Olsson, 1991], which allows a process to add, remove, replace or relocate object modules within its address space during execution. This solution retains the efficiency of executing native machine code (instead of having an interpreter) and adds the flexibility of modifying a program during its execution. The cost is the onetime overhead in copying object modules from memory or reading library files from disc, as the link-editing takes only small time. A small change in *dld* was required to copy the object code from memory instead of a disc file. In the compilation framework, object code is stored in a *Code Array* object which resides in the process memory map by the time link-editing is required.

Alternatively, solutions with some form of indirection could be used. In [Bushell *et al.*, 1994] register indirection is used to a structure with global data to dynamically bind native code generated for Napier88 procedures with the runtime system. Linkage between generated code is based on indexing pointers to code objects.

10.10 Summary and Conclusions

This chapter described the changes to TPL programs that must be performed before generation of target machine code. Environment analysis and closure conversion and the use of a low-level abstract machine (or a model of memory) were shown to provide adequate support for target machine code generation.

A persistent object store was used by the runtime system to achieve persistence. Because the object store needs to be garbage-collected to recover space and objects may change their position as a consequence, demands for identifying and preserving pointer values were recognised. To provide the ability to recover from transaction, system or media failures, the store needs to be checkpointed. At those points, the dynamic state of the running program must also be preserved.

To support incremental construction of programs and reflection, it must be possible to bind generated programs to existing data. This problem is solved by dynamic link-editing procedures with the executable representation of the program. Target machine object code resides in an object in the store and is loaded and bound on demand. Whenever the store moves to a new *era*, native code in the store becomes invalid and on-the-fly code generation is needed. Therefore, the components CLOSE and JUICE of the proposed architecture have to be available in the store. This leads to the need to bootstrap these components.

For practical reasons, C-- was used as a UMC enabling machine dependent optimisations and portability. By compiling into C--, good portability is achieved, as a compiler to the language exists for almost all platforms. By using GCC very good low-level, platform specific optimisations are also achieved. In addition, using C-- also provides the possibility of linking programs written in C, and other programming languages that support an interface to C, into the executable. The use of GCC provides useful non-standard features which may be used to improve efficiency, namely first-class labels allowing label arithmetic, and the possibility of explicitly map global variables on to fixed machine registers.

This chapter finished the description of the implementation of the compilation framework prototype built to demonstrate the feasibility of the proposed three-level architecture. The next chapter will evaluate the results of the experiment conducted and propose design changes.

Chapter 11

Evaluation and Discussion

This chapter evaluates the experiment conducted in building a prototype for the three-stage architecture proposed to support the compilation of persistent higher-order reflective languages. The achievements and the limitations of both the prototype and the experimental work are presented. The design space is covered by describing the implications of the goals of supporting this class of languages while ensuring longer-term persistence of data and sufficient efficiency. For each of the goals, the design decisions are evaluated in the face of the results.

11.1 Introduction

To achieve the goals presented in Section 2.3, a three-stage architecture was proposed and an experiment conducted to prove it feasible. The research described in this dissertation is about an internal representation, TPL, intended to support the compilation of persistent higher-order reflective languages. TPL is independent both of the source language and of the target platform comprising an operating system and a particular hardware architecture. It enables high-level optimisation and target-machine code generation. A prototype of an instance of the compilation framework, as presented in Figure 6.2, was implemented in order to demonstrate that the proposed three-stage architecture, which includes TPL as a high-level intermediate representation, is feasible. Using a control graph as the persistent internal data structure, a modified three-address internal representation is generated and optimised and low-level C code is generated from it. Storage for aggregate values is provided by a persistent heap of objects and dynamic link-editing is used to bind persistent code with the executing process containing the runtime system and the accumulated application code. An experiment in the use of CPS in the context of PHOL compilation was conducted by transforming the three-address representation into a CPS representation in order to evaluate the simplifications in the runtime system. After some experiments with the use of a stack, an abstract machine with only registers and a heap was used and the infinite supply of registers was mapped to C local variables. Pointers in the C stack need to be identified to permit garbage collection and stabilisation.

The intermediate representation designed to support the compilation of persistent higher-order reflective languages, TPL, provides first-class procedures in order to facilitate the work of front-ends for this class of languages. Because uniform polymorphism was chosen to implement parametric polymorphism, TPL is constrained to have formats which accommodate any of the values of the language. TPL must have a stabilising operation together with dynamic-binding to ensure the persistence of data values. Dynamic-binding and dynamic-typechecking are needed to support reflective programs and are present in the proposed architecture. All these constraints make TPL very different from machine level languages and this imposes constraints on efficiency. Therefore, optimisations are applied to TPL and target machine code is generated in order to achieve program representations with better space and time behaviour.

This chapter looks at each of the requirements for the compilation language framework with the architecture proposed, and discusses what was achieved and what are the limitations of the prototype implemented. Design decisions are evaluated and changes are proposed where appropriate.

11.2 Language Features

The language features intended to be supported by TPL were identified in Section 3.4 and the constructs needed to support those features were identified in Sections 3.5 to 3.8. The following sections will discuss each in turn.

11.2.1 Orthogonal Persistence

As described in Section 3.8, orthogonal persistence is achieved by software through the use of a persistent object store. The alternatives are to embed the runtime system in a persistent address space [Vaughan, 1994, Dearle *et al.*, 1994] provided by the operating system or implemented by hardware [Rosenberg and Keedy, 1987, Russell *et al.*, 1994]. Because persistence by reachability facilitates the work of the programmer, this model was followed and objects reachable from distinguished roots are persistent. To deliver persistence, the object store must provide stability, recovery and an identity mechanism. Concurrency is often a complementary requirement. To perform garbage collections and checkpoints, store management must be able to identify all the pointers and know the size of each object. To maintain identity, when objects move position as a result of a garbage-collection, all references to them must be revised accordingly (see Section 10.7.3). Finally, in order to support the binding of new and persistent values, dynamic binding and type-checking are needed.

In the proposed architecture, there are two main approaches to use an object store: to pick an "off-the-shelf" object store and have the runtime system use its interface or to access the internals of the store and manage the space directly through inlined code. The trade-off here is between more generality, in the first case, or more efficiency, in the second.

In the prototype implemented by the experiment described in this dissertation, the first approach was followed (as described in Section 10.7.4) by constructing a minimal object store and a runtime system which uses its interface. The choice was mainly concerned with implementation considerations; the code for the first approach is easy to construct and maintain because it partitions the implementation. Alternatively, the runtime system could have a mapping to a store protocol, such as TSP described in [Mueller *et al.*, 1997], which would allow the usage of all object stores conforming to that protocol.

11.2.2 First-class Procedures

A form of block retention is needed to support first-class procedures, as described in Section 3.5. A way of representing procedure closures with access to all free-variables must be devised in order to have access to values of variables which have left their scope. The trade-off here is between the space used to represent the closure and the time efficiency of calls and accesses to free-variables.

In the proposed architecture, the high-level intermediate representation TPL has support for first-class procedures (see Section 7.8.10 on page 91). At a second stage, this higher-order representation is mapped on to a flat representation suitable to run on a von Neumann machine. The mapping can be performed by following the closure conversion algorithm which changes accesses to free-variables to access to slots with computed offsets in the procedure closure (see Section 10.3).

In the prototype implemented, closure conversion is performed and a flat closure representation is used. A similar approach is also used in the PAMCASE abstract machine designed to support the same class of languages [Cutts *et al.*, 1997]. This representation uses more space at closure-creation time than a shared representation, as shown in Figure 10.3, but avoids the extra indirection which is otherwise needed. This avoidance of an indirection is important in persistent systems since each indirection may incur the overhead of an object fault. Measurement is still required to quantify both approaches in order to decide which will perform better for a particular class of languages and work loads.

11.2.3 Polymorphism

A means to accommodate all values of the language must be provided in order to support a uniform representation of universal parametric, polymorphism as described in Section 3.6. Inclusion polymorphism in statically-typed languages may be supported by virtual method tables and dynamically-typed languages need more elaborate representation where dispatch tables must be used.

In the proposed architecture, the TPL intermediate representation has the building blocks needed to support both forms of polymorphism. The type *INF* together with its operations and support for higher-order procedures can be used to support universal parametric polymorphism (as described in Section 8.7) and values of type *RECORD* may be used to implement subtyping over records.

At the moment, the prototype language framework does not support any form of polymorphism as the front-end is not able to parse polymorphic declarations. It was shown in Section 8.7 how parametric polymorphism can be accommodated in the language framework and in Section 3.6.2 that the front-end may use TPL records to support inclusion polymorphism.

11.2.4 Reflection

As described in Section 3.7, type safe runtime linguistic reflection may be achieved by calling the compiler during the execution of a program and performing dynamic link-editing and type-checking to link the result of the compilation with the executing program.

In the proposed architecture, dynamic binding can be performed by projection from values of type *INF* which generates a runtime error if the value is not of the required TPL type. To ensure that the value is of the required high-level type, the front-end can plant code to call the procedure *matchType* from the standard library (see Section 8.5) with the stored type representation and the intended type representation as parameters, as shown for example in Section 8.4 on page 109.

The prototype does not support reflection as yet. As persistence is already supported by the prototype (by having closures in the store), it is only necessary to implement a compiler as a procedure in a language that can be compiled to TPL and then to install that procedure in the persistent store.

11.3 Longevity

As described in Section 2.3.4, to achieve longer-term persistence it is necessary to preserve the semantics and bindings of data items in the persistent store over changes in the supporting layers of software and hardware. It is therefore necessary to introduce an intermediate representation of data (including programs) independent of the supporting layers in an architecture to guarantee, at any time, the transformation from that intermediate representation into a form which can run in a new supporting set of layers. It must be possible for the independent representation to evolve easily in a way that keeps old program representations with unchanged semantics.

For each change in the underlying architecture, a new *era* is initiated: all values in the persistent store need to be translated to the new physical formats (e.g. new byte-orderings) and the persistent target machine code must be invalidated. Further to the translation of values to the new architecture, new target machine code needs to be generated, bound and loaded on demand and cached for future use.

This thesis is concerned with achieving a description of data values which is independent of the supporting layers such that the precise semantics of all values is well defined and that code for new supporting layers can be generated whenever needed. A change in the order of bytes within a word from "big endian" to "little endian" [Patterson and Hennessy, 1990] leads to different integer quantities being represented by the same bit-pattern. Therefore, the problem of moving the binary contents of the store to a new architecture remains to be solved by lower-level technology. ANDF achieved relevance in supporting program portability but still makes no provision to achieve data portability, as support for hiding the byte-ordering of the platform is not provided [Macrakis, 1993].

In summary, the two requirements for longevity are: to achieve an architecture which includes an independent representation of data values and to plan for easy extension of that representation.

11.3.1 Architecture-independence

In the proposed architecture, the intermediate representation TPL, in its different forms, can constitute a suitable architecture-independent representation. TPL is typed so that for each data value, the operations which may be applied have known precise and unchangeable semantics enabling longer-term persistence. In order to get a new target machine code representation for programs to run on a new platform it is necessary to support dynamic binding and loading, as was the case with reflection. The internal representation must be constructed from a linear stream of bytes which all machines and operating systems are able to read or write.

In the prototype implemented, cTPL is the intermediate representation which will constitute the basis for longevity; C--, and target machine code may be generated from cTPL whenever required. TPL or TPLk cannot serve for that purpose as whole program analysis is needed on those representations in order to perform environment analysis and closure conversion (as described in Section 10.3) before generating cTPL. When a procedure is applied and the existing native code cannot run in the supporting platform because it belongs to a different *era*, new target machine code needs to be generated "on-the-fly". Therefore, procedures must know everything they need to execute and cTPL is the only intermediate representation filling those requisites. It should be noted that C--, because it is a subset of C, cannot be used, as the semantics of its operations is not the same on all platforms.

In order to demonstrate support for longer-term persistence, the components of the prototype 2C and JUICE must be callable on program execution and therefore, they must be in the store. Again, there is a need to bootstrap the architecture as was the case with support for reflection.

The technology used to ensure longevity, that can transfer arbitrarily complex graphs of objects from one *era* to another with a different supporting platform, may also be used to achieve distribution across machines with different architectures. Because the data values are typed and thus have a precise semantics, interoperability among different source languages can also be achieved.

11.3.2 Extensibility

In order to achieve longer-term persistence, it is essential that the intermediate language TPL can evolve easily by the introduction of new types with operations of the required semantics (see *INT32* example on Section 2.3.4 on page 20).

In the proposed architecture, the introduction of a new TPL type implies the provision, at back-end compile-time of:

- a constructor for values of that type;

- instructions corresponding to the first-class citizenship rights: *EQ.type*, *NEQ.type*, *INSERT.type* and *LOOKUP:type*;
- instructions to operate with values of the type;
- instructions to perform explicit coercion between values of different types;
- the cost of inlining for all instructions of the type;
- a fold predicate for all instructions of the type; and
- a code emitter function for all instructions of the type.

The introduction of new TPL types in the present prototype involves a change of several programs which implement all optimisations, environment-analysis and closure conversion, translation to C-- and sometimes even the introduction of a new store object type. This is not satisfactory, and a redesign of the internal structure of the prototype is needed to achieve easy extensibility with minimum impact on the existing components. Ideally, the components should work with information in tables to implement the required operations on TPL values allowing the compiler framework to remain unchanged.

11.3.3 Generality and Portability

As mentioned in Section 2.3.1, the architecture proposed is intended to provide generality by accepting front-ends for different PHOLs and back-ends using different platforms. It is also intended to be easily supported in new platforms.

By introducing a three-stage architecture with two intermediate representations for programs, TPL and cTPL, generality and portability are achieved. TPL is intended to simplify the task of front-ends and to support machine independent optimisations. cTPL supports longevity and can be used to generate new target machine code. As cTPL code generates to a subset of C, portability is achieved by using a C compiler for the new platform whenever native code is needed. For now, we assume a consistent C compiler for C-- to be ubiquitous and permanently available

Further to the use of a native compiler, other approaches leading to slower solutions are also in use to run binary code for an old architecture on a new architecture. Those approaches, ordered from the slower to the the fastest, are the use of a software interpreter, a microcoded emulator or binary translation. Binary translation consists in translating an old architecture binary program to a sequence of new-architecture instructions which reproduce its behaviour. Industrial work described in [Sites *et al.*, 1993] succeeded in translating VAX native code programs to the Alpha architecture. Even with two related architectures, the work was qualified as hard; translation between dissimilar architectures would be very difficult. Therefore, to keep up with the generality and portability demands and to achieve fast execution, a native compiler must be used.

11.3.4 Recent Work

While this dissertation was being written, the Java [Gosling *et al.*, 1996, Arnold and Gosling, 1996] programming language has emerged and enjoyed a rise in popularity as a means of developing distributed and mobile platform independent applications (called applets), as well as in the construction of applications.

The Java source code is compiled to bytecodes for a virtual machine and shipped to a client machine using the WWW, where it will be interpreted and checked for safety during execution. In order to achieve better performance, native machine code may be generated using a "just-in-time" code generator.

The Java virtual machine [Lindholm and Yellin, 1997] constitutes a means of achieving neutrality and type-safety, as defined on Section 2.3, and it was designed to support inclusion polymorphism (object-orientation). Java lacks orthogonal persistence but several approaches have been proposed recently [Atkinson and Jordan, 1996] to add persistence to Java.

Relevant to the work described in this dissertation is PJava [Atkinson *et al.*, 1996] which is an orthogonally persistent version of Java following the principles described in Section 1.3.3 on page 6. PJava is supported by a garbage-collected object cache and persistence by reachability is achieved through the use of a persistent object store. Code can also persist together with the data items it manipulates. Further to neutrality, which is achieved by compiling to bytecodes for Java's virtual machine, persistence is achieved by the provision of a class (*PJavaStore*) which provides access to persistent facilities. PJava has a simple model of concurrency and recovery, supports threads and exceptions. It lacks parametric polymorphism and higher-order functions as first-class values, from the set of features identified as needed for the class of languages anticipated (see Section 3.2 on page 31). However, languages with these properties have already been compiled to Java bytecodes [Odersky and Wadler, 1997].

11.4 Efficiency

This work is intended to achieve efficient implementations on stock hardware. The required language features, particularly first-class procedures and polymorphism, make it harder to achieve efficiency. These were tackled by devising a three-stage architecture which includes an intermediate representation, TPL (suitable to accommodate a set of high-level machine independent optimisations) and by code generation, during which machine-dependent optimisations may be performed.

11.4.1 High-level Optimisations

High-level machine independent optimisations supported directly on top of TPL, or which require only limited amount of analysis, were demonstrated in Chapter 9 together with an experiment on the use of CPS in order to simplify the runtime system. This sample set of optimisations demonstrates that the compilation architecture supports the analysis and transformation needed for machine-independent optimisations. It is reasonable to propose that were a full set of such optimisations implemented, they would yield the same improvements that they deliver in other compilation architectures [Aho *et al.*, 1986, Appel, 1992, Tarditi *et al.*, 1996]

The optimisations supported by the prototype have validated the TPL design choices, such as: the named intermediates, the unique binding rule, the explicit assign and the restricted control-flow. Other optimisations not supported yet (e.g. whole program transformations and loop transformations) can be applied as long as appropriate analysis is performed. The CPS experiment proved that by introducing more procedures and free-variables, the runtime system can be simplified as a runtime stack is not needed. It is only needed to analyse the current activation record to find roots for garbage-collection (as described in Section 9.6.2); other variables containing pointers (in older activation records) will be free-variables of continuation closures and therefore they will be reachable, as those closures are reachable from the current closure. In summary, CPS simplifies the runtime environment by having only tail-calls but increases heap allocation. A great deal of effort has been spent in closure analysis in order to use a stack to allocate activation records when they follow a LIFO policy and allocate activation records in the heap for higher-order function closures which have an indefinite extent (i.e. that outlive its lexically enclosing activation record). In heap allocation, an activation record may point to another and it will be alive for as long as there are external references to it. Several compilers that use CPS—idxuseCPS as intermediate representation "undo" CPS before they generate code, in order to achieve better performance and better space usage [Flanagan *et al.*, 1993]. For example, ORBIT and RABBIT change the allocation strategy when the closure is a continuation.

If CPS is used, then it seems preferable to also perform register allocation [Norris and Pollock, 1994] and instruction scheduling together with direct target machine code generation, as in the SML/NJ compiler [Appel and MacQueen, 1987]. This approach does not fit with the longevity requirements because it requires highly-skilled labour intensive work to build a new back-end for every new platform. The use of code-generator generators [George *et al.*, 1994] may constitute an interesting approach in this situation. Generating low-level C could also facilitate the construction of the back-end. For the SML/NJ compiler, this approach leaded to an overhead in runtime of about 70% to 100%, when compared to native machine code, as reported in [Tarditi *et al.*, 1992].

11.4.2 Code Generation

In the proposed architecture, for practical reasons cTPL is translated to C--, a restricted form of C. Good portability is achieved as a C compiler is available for almost all platforms. These C compilers usually perform a comprehensive set of optimisations during the translation to target machine code, and therefore efficient program representations are achieved.

In the prototype, target machine code resides in code objects in the persistent store and this code is dynamically-linked to the execution process the first time it is required in a corresponding procedure call. Alternatively, the C-- code could be compiled only when native code is required, following the dynamic translation approach of the Deutsch-Schiffman Smalltalk-80 system [Deutsch and Schiffman, 1984]. A similar approach (called dynamic compilation) is used in a compiler for the SELF programming language to achieve better performance than interpretative code. It is reported to reduce compile-time and code-space costs when compared with a static compiler [Chambers and Ungar, 1991, Hölzle and Ungar, 1994]. A parser translates SELF source code into a simple bytecode intermediate representation. A compiler compiles and optimises the program when the program is invoked, and the resulting object code is cached for future use. Dynamic compilation gives particularly good results for dynamically-typed languages, as in this class of languages, types are known only at runtime. For the class of languages treated in this thesis, it is better to generate code at compilation time, as most of the types are known and use dynamic compilation only to deal with polymorphism, dynamic binding and longevity issues, as described in Section 11.3.1.

Recently, dynamic compilation (also called Just-In-Time compilation) has been used to replace an interpretative layer by better code (as it uses information available at runtime) for several programming languages [Lee and Leone, 1996, Auslander *et al.*, 1996, Engler, 1996]. There is also recent work targeted at achieving language independence which uses this technique [Adl-Tabatabai *et al.*, 1996]. Portability can also be achieved with little cost, as dynamic code generation is reported to require little more time than the input of equivalent native code from disk storage medium [Franz, 1995]. Also with the goal of achieving portability, work has recently been reported on **mobile programs**. Mobile programs may be shipped unchanged to a heterogeneous collection of processors and executed with identical semantics on each processor. The combination of electronic documents with the use of network protocols on the Internet requires safe execution of document contents on different platforms. Java [Arnold and Gosling, 1996] has recently gained wide use for such tasks. Java's internal representation was designed for fast interpretation by a stack-machine [Gosling, 1995, Lindholm and Yellin, 1997] but native code can also be generated just-in-time from Java bytecodes.

This extensive interest in dynamic compilation indicates that interpretation is often combined with dynamic code generation to achieve efficiency.

11.5 TPL Design Choices

In this section, the properties of the high-level intermediate language TPL are summarised and put in the context of the support of PHOLs and the fulfilment of the goals already enunciated. A discussion of the set of types and instructions that should be supported in TPL is also presented.

11.5.1 TPL Properties

TPL was designed to achieve several goals. As a result of the choices made, TPL is an intermediate language with the following characteristics:

it is higher-order — in order to simplify the work of the different front-ends which parse the PHOLs that it is intended to support;

it is linear — i.e. calls are ordered and arguments for procedure calls are atomic[1], in order to make code generation easier and in order to make simpler the analysis needed to

[1]That is, they can not be expressions or functions calls, for example.

perform high-level machine independent optimisations;

intermediates are named — in order to simplify the algorithms for optimisations, enable loop transformations and simplify the generation of native machine code;

identifiers are unique — in order to simplify the analysis during high-level optimisations;

assignment is explicit — i.e. there is only one way of changing the value of a variable, the *UPDATE* instruction, in order to allow constants to be easily discovered;

it is safe — i.e. a low-level type system is used for protection, in order to enable longevity, to help in the optimisation phase and in order to enable garbage-collection by identifying the pointers;

locations are mutable — i.e. there is no constancy check in TPL because the front-end can make the necessary arrangements if needed;

equality by identity — in order to suit some of the PHOLs to be supported (the other different policies may be ensured by the front-end);

persistence by reachability — in order to provide a model of persistence to the front-end programmer;

only one branch construct — i.e. the structured *BRA* instruction, in order to simplify the unreachable-code removal transformation by preserving the structure of the source program.

The investigations so far have not revealed deficiencies in this design (see Chapters 8 and 9).

11.5.2 TPL Set of Types

The set of types to be supported by TPL is intended to cover the minimum functionality required by the PHOL languages anticipated: Napier88, TL and Fibonacci. In the prototype implemented, to support COREL, only some of the types provided in TPL are used. The other types are only used in TPL test programs written by directly generating the internal representation.

Further to the types enumerated in Chapter 7, there are other types which should be included in TPL. To satisfy compiler requirements, the set of base types should probably include the types for different storage sizes; for example, the types *BYTE*, *SHORT*, *LONG* and *SINGLE*, and the corresponding operations implemented efficiently. Types dedicated to the support of multimedia values may also be added in the future if standards are widely accepted. Each TPL type which may be added requires its own collection of operations along with a high-performance implementation involving appropriate data structures and access methods. On the other hand, some types now present in TPL may be dropped from the language in future prototypes, if they prove not to be as useful as anticipated. For example, instead of providing direct support in TPL for tables and vectors, a different approach can be followed.

Instead of providing the *MAP* type, a set of constant procedures can be inserted by the front-end in the Standard Library. For example, to support collections of bindings, a map from *STRING* to *INF* could be implemented in TPL. To insert a new value in the map, the procedure:

```
INSERT(<id>,<loc>,<value>)
```

may be used. This procedure has type *PROC(CHARS,RECORD,INF->VOID)* and takes the name, the identifier holding the map location and the values of type *INF* to be inserted into the map. To get the value back from the map, the procedure:

```
<id> := LOOKUP(<id>,<loc>)
```

with type *PROC(CHARS,RECORD->INF)*, may be used. The projection from the *INF* value would achieve dynamic-binding. To also perform a dynamic type-check, the high-level type should also be provided by the front-end together with the value, both packed in a value of type *RECORD*. In this case, the procedure to insert the value would have the type:

```
PROC(CHARS,RECORD,RECORD->VOID)
```

Support for values of type *VECTOR* were introduced in TPL in order to provide more specialised, and therefore more efficient, operations and addressing modes on structured values containing elements of the same type. Alternatively, vectors can be accommodated in values of type *RECORD*. For example, to support Napier88 vectors, a constancy bit is needed and therefore a value of type *RECORD* will be needed to pack the constancy with the value. In the prototype, to bind a constant vector with two integers to *R12*, the following instruction may be used:

```
R12 := VECTOR(1,2,10)
R13 := RECORD([TRUE,R12])
```

Instead, it could be used:

```
R14 := RECORD([TRUE,10,10])
```

This second approach requires more work to be done by the front-end to initialise large vectors but would avoid the extra level of indirection present in the first approach.

If there is no direct support for vectors in TPL, then indexing may be eliminated from the supported addressing modes and thus the runtime system will be simpler. The addressing modes presented in Table 10.1 on page 148 can be further simplified by eliminating offsetting as well and introducing an explicit operation to select a field from a record (*SELECT* in [Appel, 1992]). For example, the following TPL program:

```
R15 := RECORD([1,TRUE)
R16 := INT(R15!0)
```

would be coded as:

```
R17 := RECORD([1,TRUE)
R18 := SELECT(R17,0)
R19 := INT(R18)
```

This simplification in addressing modes would make more information explicit and therefore could extend the scope of applicability of optimisations such as, for example, inlining procedure calls.

11.5.3 TPL Instruction Set

Further to the instructions to operate with the base types introduced in the last section, there are also other instructions which must be considered.

The introduction of the structured branch instruction *BRA* helps in performing transformations such as unreachable-code elimination as it keeps the structure of the program. The alternative would be to use unrestricted *goto*, but in this case, the program control-flow graph needs to be traversed in order to perform those transformations. Recursion is achieved with the use of first-class procedures by assigning a new value of the same type which refers to the procedure itself. Instead, a recursive constructor could be provided which would simplify the work of the front-end and would enable the transformation of tail-recursive calls into iterations, as described in [Appel, 1992] for the SML/NJ compiler. Outside the scope of this thesis, there are instructions to deal with concurrency (*UP* and *DOWN* on values of type *MUTEX*), exceptions (*RAISE* and *HANDLE*) and debugging (*CHECKPOINT*[2]) which also need to be included in TPL.

During the translation of a program, the compiler may have more information about the use the program makes of the objects it manipulates than the information which is expressed in the intermediate representation. Such knowledge may be transmitted to the runtime system. New instructions may be introduced in TPL to convey information to the runtime system. For example:

- to request that an object remains in the heap as it will be used frequently in one region of the program, *PIN* and *UNPIN* instructions could be used to mark the region;
- to relax some constraints that need to be checked by the runtime when a vector is indexed, instructions to check the bounds (*CHECKUPB* and *CHECKLWB*) could be introduced together with a fast assign instruction, which would not check bounds (*FASTUPDATE*)[3]; and finally,
- the compiler could also plant a *SCHEDULE* instruction to call the thread scheduler.

11.5.4 Conclusions

TPL properties were carefully chosen in order to fulfil the requirements derived from the goals of achieving generality, efficiency and portability when supporting higher-order reflective languages. The TPL low-level type system enables longevity, helps in optimisations and guarantees the discovery of *all* the pointers for garbage-collection purposes. The TPL instruction-set implemented in the prototype is insufficient to support all the planned features of the intermediate representation and more instructions need to be introduced in TPL, for example, those in Section 11.5.3.

[2]This is not the same as *STABLE* used to preserve the store's state.

[3]It should be noted that in order to relax those constraints, the TPL layers would need to trust the compiler front-end.

11.6 The Abstract Machine and Runtime System

Chapter 10 described a low-level abstract machine and discussed how it can be used to support TPL. Other possible strategies, as presented in Section 6.5 on page 71, are also possible. The allocation policy for activation records will be revisited, another strategy to pass parameters in function calls will be put forward, a discussion concerning the level of the store interface and a discussion of the implications of the interactions between the use of the the C stack and garbage-collection are presented in this section.

11.6.1 The Allocation of Activation Records

After environment analysis and closure conversion have been performed and the corresponding closures have been constructed, the runtime system must allocate an activation record for each activation of the procedure, containing its local and intermediate values. The three strategies which can be used to allocate activation records were described in Section 6.5: heap allocation, stack allocation or allocation in registers.

The strategy used in the experiment in order to build the prototype consisted in laying down activation records into an infinite supply of registers and of using the C stack to allocate space for each procedure's local and intermediate values. The decision to use the C stack in the experiment was concerned with a search for efficient implementations by exposing the C-- code to the C compiler in order to enable low-level optimisations. Because using the C stack for local and intermediate values enables optimisations and because heap allocation and stack allocation would imply an extra level of indirection in all access to values, these two other strategies were not considered in the experiment.

By relaxing the constraint of having explicit C variables for each local and intermediate value, alternative strategies may be followed. Figure 11.1 presents a putative abstract machine for cTPL where activation records are allocated in the heap, instead of registers. The former *C* register is substituted by an *SP* register which points to the activation record for the current executing procedure. Local and intermediate values of scalars and pointers for the procedure activation, can be accessed through this pointer. The activation record includes a pointer to the closure which corresponds to the procedure (in the field *closure*) and in the field *previous*, a pointer to the activation record for the calling procedure. Its value will be used to update the register *SP* on procedure return. Each procedure closure contains access paths for each free-variable and constant, and a code vector pointer in the field *CODE*. The code knows how to access local and intermediate values by using *SP* and free-variables by using *SP!closure*[4]. The number of pointers in the field *no-of-pointers* and the number of scalars in *no-of-scalars*, enable the runtime to ask for an object for the activation record with the required format. This strategy to allocate activation records does not contain any space leaks, as only closures are retained (similarly to the register allocation strategy). The benefit is that pointers are always easily located but there may be some cost in inefficiency compared with the strategy used in the prototype, as the C optimisations are reduced.

Further changes to this abstract machine are possible. Parameters can also be allocated in

[4]To avoid another level of indirection, *SP!closure* can be cached in a C register in the C-- code.

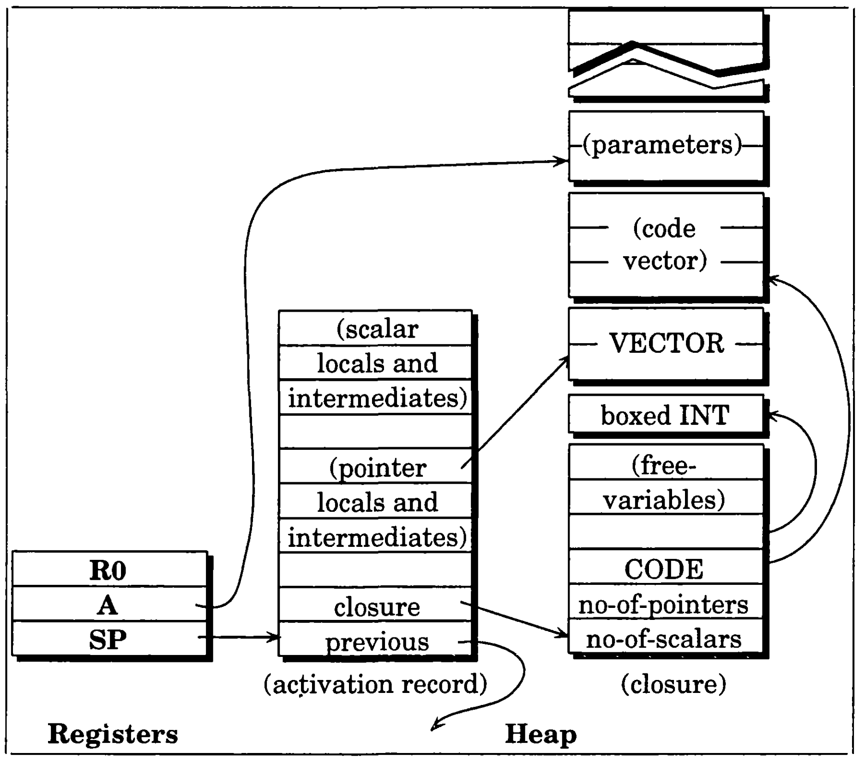


Figure 11.1: A New cTPL Abstract Machine

the activation record at the lower positions of the pointer and scalar part, and the *A* register eliminated. The *R0* register can also be eliminated by using adequate space for results in the activation record object, for example, before the parameters. These proposed changes in the cTPL abstract machine are localised to the programs which implement the CLOSE and 2C components.

11.6.2 Parameter-Passing Strategies

In the prototype, parameters are packed in a heap object and accessed through an indirection via the *A* abstract machine register in C-- (as described in Section 10.4 on page 151). The experiment was conducted this way for the sake of simplicity and some inefficiency is expected to take place.

The parameters in a TPL program, represented by identifiers *A0*, *A1*, ..., *An*, can alternatively be represented in C-- by C function parameters and their actual values can be passed to the C function, at the time of the call. If activation records are allocated in the heap or stacks, the parameters can use that space as well. The modifications proposed are localised to the programs which implement the 2C component.

11.6.3 The Level of the Store Interface

The interface to the store can be at a higher-level, where the runtime system requests object creation and manipulation, or at a lower-level, where the runtime system is responsible for space allocation and must know the object layout and plant tests for space exhaustion.

For the sake of efficiency, the runtime system should take more responsibility for space management and so a lower-level store interface should be used. The runtime could then use information provided by the front-end to improve performance; examples of such information are pinning objects in memory, explicit residency checks and explicit bound checks. These would then allow the use of faster TPL instructions for vector indexing, for example. In the prototype, the store contains specialised objects, e.g. vectors or tables, in order to provide more efficient operations. Following the trend of giving more responsibility to the runtime system, the TPL operations for table management (*INSERT*, *LOOKUP*, *EXISTS* and *REMOVE*) could be mapped into simpler store objects instead of having direct support provided by the object store.

11.6.4 Use of C as a Target Language

A subset of C code was used in the experiment conducted to build a prototype of the three-stage architecture proposed. The use of the C stack for local and intermediate procedure variables, is a good solution to achieve the generality, portability and efficiency goals, but introduces a difficulty in discovering where the pointers in the C stack are. Pointers need to be distinguished from scalars during garbage-collection, object clustering and pointer swizzling operations. As a common optimisation performed by C compilers is to store frequently accessed values in registers, an additional difficulty arises when pointers need to be followed or modified during garbage-collections; is the pointer value in the stack or in a register? The use of the *volatile* qualifier in pointer declarations may be used to guarantee that values do not go to registers but then, part of the potential efficiency gains related to the use of the C stack are lost.

Garbage-collections, for example, may move objects to different positions and then all pointers to them must be updated. Pointers may reside in abstract machine registers, activation records, local and intermediate variables and in heap objects. Object code in code objects is relocatable and can therefore move around and only the entry point is needed the first time the object code is linked into the executable. The pointer to code object will be updated in the relevant closure record. Therefore, the sole problem to be solved is finding all the pointers, currently notionally on the C stack.

An experiment for the language Napier88 (one of the PHOL languages intended to be supported by TPL) on native code generation using C differently from the approach taken in the prototype is reported in [Bushell *et al.*, 1994]. The paper is about mapping all code and data on to persistent objects taking into account garbage collection and the need to save the dynamic state of a running system. The compiler generates a C program with one C function (which does not contain any non-local references) per Napier88 procedure; the C program is compiled and when executed, copies the executable code into a temporary file. The compiler then copies the instruction sequence from the temporary file to a persistent object. Dynamic binding is achieved by indirecting through a global register to access other data, such as the stack frame pointer register. In the event of a garbage collection or checkpoint, the C stack approach reverts to the usual approach taken by the Napier88 abstract machine, with activation records on the heap. When such an event occurs, all locals must be copied to the corresponding abstract machine frame. This involves a lot of copying. C functions can restart after these events. As closure conversion is not performed, activation records in the heap are also used for nested procedures and this limits the number of frames allocated on the C stack. By using such a scheme, there is no problem with garbage-collections or checkpoints, as has been demonstrated before by the PAM abstract machine, but this scheme incurs a substantial overhead each time a garbage collection or a checkpoint occurs.

Pointer Tracking

Pointers must be found during several heap operations and may change their value as a result of performing those operations. In the literature, several approaches have been proposed to deal with tracking pointers. One solution would be to use a tagged hardware architecture where pointers are discriminated from scalars [Bawden *et al.*, 1977]. Other solutions consist of emitting masks which describe where pointers are, to discover the pointers in the stack or to use conservative garbage-collection on stack values.

The SML/NJ compiler emits a mask describing precisely which registers contain pointers at safe points in the program, at which a garbage-collection might take place [Appel, 1992]. This way, only data reachable from "live" registers is retained by the SML/NJ collector.

The solution described in reference [Diwan *et al.*, 1992] for a Modula-3 compiler, effectively "decodes" all of the C stack. Tables identifying pointers are constructed at possible garbage-collection points and the return address stored in stack activation records is used to consult the tables for previous activation records. Even after applying compaction techniques, this solution leads to a space overhead of 16% to 45%.

Conservative garbage-collectors have been used in situations where the C stack is used. The Boehm-Demers-Weiser collector developed at Xerox Parc does not assume any cooperation from the compiler and is, thus, fully conservative [Boehm and Wieser, 1988]. Because it cannot be certain that any value is a pointer, it can not move objects as, otherwise, it could risk changing a scalar data value. Another collector designed by Bartlett to support the Scheme language using C as an intermediate language, works differently [Bartlett, 1988]. In this case a more liberal approach could be followed, as all pointers in the heap-allocated data could be found accurately. The collector is hybrid, with the objects which might be referred to from the stack, from registers and from the static area, treated conservatively and not moved (as in the previously referred collector) and objects only accessible from the other heap-allocated objects, being copied. This leads to a faster and more accurate collector.

In the proposed architecture, pointers were carefully discriminated from scalars, and because of strong typing at the TPL-level, cannot be forged. Therefore, in a TPL intermediate representation of a program, it is easy to know which identifiers are pointers and their names. This provides enough information to emit masks. Objects reachable through pointers from other objects in the heap can always be found accurately, enabling the use of a copying collector, as in the work of Bartlett. The use of the C stack poses the same problems as in the Bartlett Scheme-to-C compiler and could be solved in the same way.

11.6.5 Measurements and Conclusions

The language framework prototype did not reach a stage where a complete and well-engineered implementation could be achieved. The lower layers of the prototype, the runtime system and object store, are very primitive because their main goal was to prove the proposed architecture feasible and not to investigate better implementations. Therefore it is not expected that the language framework will perform well when compared with other stable environments, built by teams of engineers working for years. It is also difficult to find a convincing interpretation in terms of the design decisions taken from the results of the measurements, as *a priori*, it is known that the object store will not perform well. Nevertheless, measurements were taken for the execution of an implementation in COREL of a procedure which computes the Fibonacci numbers. This is an intensively recursive algorithm as can be seen in Appendix C.3.

Measurements

Table 11.1 presents results obtained using the UNIX *time* command on the execution of implementations of the procedure in the same machine. The columns presented have the following meaning:

n the argument given to the procedure which computes Fibonacci numbers;

calls number of procedure calls performed in the execution;

CPU the percentage of the CPU time used during the execution;

user the time in seconds spent in user mode;

system the time in seconds spent in kernel mode during system calls;

elapsed the elapsed time in seconds;

memory *text* memory space plus *data* and *stack* memory space used, expressed in kilobytes;

IO input plus output operations performed; and

faults page faults plus process swaps.

Table a presents the results from the execution of the Napier88 interpreter (NPR) and **table b** presents the results taken from the execution of the UNIX executable which was the outcome of the language framework prototype (TPL). For a small number of procedure calls, the native code executable (TPL) performs better than the interpreted code (NPR), as expected. Further to the interpretative overhead, the interpreted code incurs also in the constant cost of starting the interpreter. While in NPR the figures for system time, memory space used, I/O operations and page faults kept roughly constant and only user time increased more than proportionately, with TPL, all figures kept rising more than proportionately with the number of procedure calls. The poor results can be attributed to the low-performance of the one-level store used and to the parameter-passing strategy used. Because parameters were allocated in a heap object supported by a memory mapped store which was not garbage-collected, when

n	calls	CPU	user	system	elapsed	memory	I/O	faults
10	177	45.7%	0.49	0.91	0:03.06	0+787k	0+25	0
20	21 892	50.4%	1.23	0.87	0:04.16	0+926k	0+27	1
25	242 785	49.7%	9.17	0.97	0:20.37	0+1188k	0+27	7
27	635 621	45.6%	22.52	0.97	0:51.48	0+1231k	0+27	0

a) NPR

n	calls	CPU	user	system	elapsed	memory	I/O	faults
10	177	35.0%	0.02	0.05	0:00.20	0+117k	0+2	0
20	21 892	31.2%	2.27	1.21	0:11.13	0+211k	2+44	45
25	242 785	31.6%	19.38	11.32	1:37.06	0+1070k	2+171	479
27	635 621	35.2%	56.30	30.11	3:17.91	0+2611k	2+135	1244

b) TPL

Table 11.1: Measurements

large number of objects are allocated, the store needed to be extended leading to page faults and I/O operations on disk. Better designs for parameter passing and better implementations are therefore needed as discussed in Section 11.6.1 and 11.6.2.

Conclusions

On the one hand, the use of the C stack implies a lot of work in keeping track of pointers for garbage-collection, and on the other hand, the optimisations that can be performed are limited, as registers cannot be used to cache locals and the C expressions used are very simple because all intermediates are named. In retrospect, the use of the C stack does not seem to be much more attractive that using heap or stack allocation, as described in Section 11.6.1.

11.7 Enabling Technology and Internal Representation

The front-end and most of the components of the language framework prototype are implemented in the PPL Napier88; the persistent object store (STORE) is the only component implemented in ANSI C; the final executables (exec) are constructed by linking the C-- generated programs with the runtime system (also written in ANSI C) by the use of the GCC optimising compiler.

Further to the simplifications described in Section 1.3.3 on page 6 and because of its characteristics of being strongly-typed, polymorphic and higher-order, as described in Section 6.6 on page 75, the use of the PPL Napier88 proved to be an excellent choice for the experimentation described in this dissertation. Building the prototype for the compilation framework was greatly facilitated by using incremental construction of the programs. The possibility of sharing the internal data structures between different phases of the compilation framework being implemented, enabled the construction of a plug-and-play framework where components could be composed in different permutations to test particular characteristics (e.g. for the optimisation phase where components were built for each independent transformation). This greater flexibility simplified the experimentation.

The use of ANSI C for the runtime and object store, enabled the efficient use of the operating system facilities and gave easier access to the internals of the underlying machine architecture. The GCC compiler generates good quality code and gave access to non-standard features, such as the ability to map the abstract machine registers (C global variables) on to hardware machine registers, in order to improve efficiency.

The internal data structures used to support the TPL abstract syntax tree, described in Section 6.7 on page 76, lead to easily written, but verbose programs. These programs always need to scan all nodes of the TPL DAG for each navigation, even when only one node type is of interest for the particular operation being implemented. Further to that, the compilation framework prototype is difficult to extend with new TPL types and operations. A complete redesign of the representation and implementation programs needs to take place in the next phase of this research.

The choice of light data structures associated with a minimal approach to the amount of information present in the intermediate representation, must be re-evaluated in order to permit good debugging messages. For all places where a runtime error may occur, the source language name may be fetched from the blackboard (see Section 8.10.3 on page 121) and used to give sensible error information. The relationship between the point where the error is detected at runtime and the original high-level expression may be difficult to reconstruct, because high-level language expressions are broken down in small pieces in order to generate TPL programs. To improve debugging, each intermediate value could refer to the high-level source which should then be kept in the blackboard. A balance must be sought in the future between the need to have good error messages and the space and time overhead of keeping the source code together with the intermediate representation.

11.8 Limitations of the Experimental Work

The experimental work described in this dissertation was intended to develop and validate a three-stage architecture which can support the compilation of persistent higher-order and reflective languages. The work was mainly centred on the design and validation of a persistent target intermediate representation for programs. In order to validate the proposed architecture and the intermediate representation, it is necessary to have a front-end parsing the high-level language and a runtime system. Included in the runtime are other lower layers of the architecture, such as, the persistent object store. Several simplifications were made in order to have a project compatible with the time constraints of this research. These have been explained elsewhere in this dissertation. This section presents a summary of the limitations of the experimental work which remain as challenges for the future.

In order to simplify the front-end and the parsing process, some high-level language features were not supported directly in the prototype, though the necessary TPL constructs are identified and already present in the target language. Among these language features are parametric and inclusion polymorphism, bulk data types, graphic data types, modules and ADTs. Implementation of these missing features is necessary to confirm they can be supported, to bootstrap the whole framework and to achieve reflection. Parametric polymorphism, for example, is used in the programs which implement the framework. From the class of languages anticipated by this work as source languages, one may be chosen to be fully supported enabling *real* programs to be used by the language framework.

The stable store used in the experiment does not satisfy the minimum requirements of functionality and efficiency. Firstly, it lacks a garbage-collector which is fundamental to support the heap of objects giving the illusion of an unbound persistent space and, secondly, the approach chosen to use a subset of the C language as a target language interacts badly with the garbage-collection needs, as explained above. The absence of the garbage-collector greatly simplified the implementation work, though. Future work to achieve a suitable foundation for TPL will be directed towards one of the following technologies: use of an off-the-shelf persistent object store, such as the Munro Store [Munro, 1993] or the Texas Persistent Store [Singhal *et al.*, 1992], or else, to build object abstractions on top of a lower-level support technology such as the RVM UNIX package [Mashburn and Satyanarayanan, 1994], together with the use of a garbage-collector working in the C environment, such as the Bartlett mostly copying collector [Bartlett, 1988].

As mentioned before in this chapter, the TPL set of types and corresponding instruction set is by no means complete and types to deal with concurrency and threads, exceptions, breakpoints and other base types to support different sizes of integers and floating-point values, need to be introduced into the language framework. The internal data structures need to be refined in order to better facilitate extension of TPL, as in the prototype there is no easy way of introducing new types into the compilation framework.

The work done in transforming the intermediate representation in order to get programs with better characteristics, must be taken forward in order to include elaborate transformations such as whole program transformations and loop transformations, which need a more complex program analysis. Specialisations of polymorphic code can also be performed in order to achieve better performance.

The requirements for longevity were identified and taken into consideration in the design of the internal representation but complimentary work needs to be done in order to demonstrate that it is always possible to move a representation, using one set of data formats, to another set of data formats, whenever the store of objects moves to a new architecture.

Finally, after the front-end re-implementation to include some of the missing language features and specially the re-implementation of the persistent object store and runtime system, precise measurements must take place in order to substantiate the choices made. Examples are measurements of execution speed, of compilation speed, of code size and of the impact on cache usage. Measurements on the effectiveness of the transformations in order to collect more heuristics to improve the optimisation algorithm, must also be performed.

The experimental work needs to be taken further in order to test the suitability of other possible intermediate representations for TPL, which may be found more appropriate than the modified three-address representation described in this dissertation. The GSA intermediate representation referred to in Section 4.7.1 on page 53 and the BURG code-generator generator referred to in Section 5.3 on page 62, are candidate technologies.

11.9 Conclusions

After analysing the limitations of the experimental work and of the prototype implemented, future work was proposed in this chapter. This future work is intended to complete and make the language framework more efficient and to carry forward the experimental work in intermediate representations for persistent, higher-order, reflective languages and longer-term persistence. The experiment conducted and described in this dissertation suggests that the proposed three-stage architecture is feasible and worthwhile. The prototype constructed may constitute a suitable framework for cooperative work between several researchers, as it promotes re-use of language back-ends and object stores on the one hand and of front-ends, on the other hand.

Chapter 12

Conclusions and Future Work

This work set out to identify the technical challenges that arise when attempting to support persistent higher-order and reflective languages for building persistent application systems with sufficient longevity, adequate performance and in amortising costs by providing general services. These challenges were tackled by the proposal of a new architecture with three stages, which accommodates high-level and machine independent optimisations and evolution of the supporting technology for multiple languages. A more detailed summary and discussion appears in Chapter 11.

It is believed that some of the issues referred to in this thesis, such as portability and neutrality of safe code will become more and more important with the deployment of digital telecommunication networks. Beyond mobile computing, representations of programs independent of the platform are also needed to maintain long-living and evolving persistent application systems.

12.1 Summary

The viability of the proposed architecture was demonstrated and the following issues were investigated:

- a new architecture for compilation of persistent higher-order reflective languages, with a context for high-level machine independent optimisations and code generation, in order to achieve performance;
- the constructs needed to support persistent higher-order and reflective programming languages;
- an intermediate language which supports the class of languages anticipated and which can serve as a mean to ensure longevity of the data and programs involved in PAS coding and maintenance;

- the use of the continuation-passing style program transformation as a means to achieve performance and to simplify the runtime system;
- a set of machine independent optimisations to be applied to the intermediate representation; and
- the use of the C programming language as a portable representation of programs expressed in the intermediate representation.

An experiment to build a prototype of the proposed three-stage architecture was designed and constructed and it is evaluated in Chapter 11. Throughout this dissertation, small example programs were used to illustrate the relevant aspects; the technology described also supports the construction and correct execution of large programs.

The limitations and achievements of both the prototype and of the experimental work were presented. The design space was covered by describing the implications of the goals of supporting the class of languages anticipated while ensuring longer-term persistence of data and sufficient efficiency. For each of the goals, the design decisions were evaluated in the face of the results.

12.2 Future Work

Several avenues are now open for future work. The first concerns implementation of the prototype. The front-end must be able to deal with a sufficiently complete programming language so that the bootstrap of the language framework implemented can be achieved in order to provide reflection, longevity and just-in-time compilation. There are several language features that should be added to the language framework in order to refine the design presented, but these are not crucial to the arguments made. They include: parametric polymorphism, inclusion polymorphism, bulk data types, graphic data types, ADTs and modules. The back-end can also be improved by interfacing it with an off-the-self persistent object store or a recoverable virtual memory package. A garbage-collector, such as the Bartlett mostly copying collector, must be introduced in order to achieve a complete implementation of persistence by reachability and achieve stability. In the end, a complete compilation framework able to support a persistent higher-order reflective language, ought to be built.

The second avenue of development is related with the intermediate representation. TPL needs to be developed to include instructions to deal with concurrency, recovery and exceptions, as described in Chapter 11. Rather than using the modified three-address instructions as the basis for TPL, the modification of the GSA internal representation (or similar approaches) should be investigated. The use of the BURG technology to achieve code-generators for each new platform when needed, rather than using C as a target language, may also be worth some consideration.

The third avenue for further research concerns opportunities for optimisations introduced by the three-stage architecture proposed. High-level optimisations, such as whole program transformations, which prove to be very effective in other language environments, should be investigated. To perform these transformations, complex analysis and the construction of complex data structures will be needed. Similarly, low-level optimisations which can be performed by the back-end, such as the substitution of polymorphic code by specialised code when the types are known, would be beneficial. A possible new target for the low-level system is provided by the Java Virtual Machine bytecodes.

Finally a great deal of work remains to provide support for longevity, as in this thesis longevity issues were only planned. Issues related to data portability and the low-level translations needed to ensure long-term persistence of data, also remain.

12.3 Conclusion

The architecture presented in this dissertation, with a high-level intermediate representation, proved to be appropriate in the construction of supporting technology for persistence. It enables high-level optimisations and code generation and it can effectively support persistent reflective higher-order polymorphic languages, ensuring longevity, safety and persistence. An initial design of the architecture was presented and the intermediate language crucial features were identified and validated.

Appendix A

TPL

The syntax of TPL is described as a set of productions using the Extended Backus-Naur Form (EBNF) notation, where:

- productions take the form *<prod> ::= <exp>*, meaning that the nonterminal *<prod>* is defined to be equal to the syntactic expression *<exp>*;
- uppercase strings, "*(*", "*)*", "*[*", "*]*", "*:=*", "*;*", "*.*", "*,*", "*+*", "*-*" are terminals;
- *<text>* is nonterminal;
- *<exp-a> <exp-b>* is the sequence of *<exp-a>* followed by *<exp-b>*;
- *<exp-a>* | *<exp-b>* is the alternative;
- {*<exp>*} is an optional syntactic expression; and
- *, is zero or more repetitions of the preceding syntactic expression.

In order to emphasise the regularity of the TPL abstract machine operations are grouped by operation instead of type of arguments. In this description of the TPL language the types *BYTE*, *SHORT*, *LONG*, *SINGLE* and *UNICODE* were included. They are not referred to in the text that reports on the experiment conducted. For each operation its semantics is informally stated as an English text comment introduced by the symbol %..

A.1 TPL Abstract Syntax

TPL abstract syntax is presented, starting with the machine types:

```
!******* TPL types  *******************

<integer>    ::= BYTE | SHORT | INT | LONG
<real>       ::= SINGLE | DOUBLE
<arith>      ::= <integer> | <real>
<string>     ::= PIXEL | BITS | CHARS | UNICODE
<ordered>    ::= <arith> | <string>
<base-type>  ::= <ordered> | BOOL
<mc-type>    ::= <base-type> | RECORD | VECTOR | PROC | INF | MAP
<type>       ::= <mc-type> | VOID

!******* TPL context free syntax  *******************

<pgm>        ::= INIT() <tpl-block> CLOSE()            % TPL program

<tpl-block>  ::= START {<tpl>}* END                    % TPL block of instructions
```

```
<tpl>        ::= <binding> | <statement>                    % TPL instructions

<binding>    ::= <id> := <bind-value>                        % bind a value to id

<bind-value>::= PROOT()                                     % get the persistent root
              | <construct>                                 % construct a new value
              | <prim-op>                                   % primitive operations
              | <coercion>                                  % coercion between values
              | <map-op>                                    % MAP binding operations
              | <call>                                      % procedure call
              | <callcc>                                    % call a C function

<construct> ::= <base-type>(<value>)                        % construct a new value
              | RECORD(<value-list>)
              | VECTOR(<value>,<value>,<value>)
              | PROC("<type-rep-list>-><type-rep>",<id-list>,<tpl-block>PROCEND)
              | INF(<value>,<mc-type>,<value>)
              | MAP()

<prim-op>    ::= <universal> | <special>

<universal> ::= EQ.<mc-type>(<value>,<value>)               % equality
              | NEQ.<mc-type>(<value>,<value>)              % non-equality
              | MOVE.<mc-type>(<value>)                     % copy the value

<special>    ::= <arithmetic> | <relational> | <logical>
              | <bitwise> | <string-op> | <other>

<arithmetic>::= PLUS.<arith>(<value>,<value>)               % addition
              | MINUS.<arith>(<value>,<value>)              % subtraction
              | MULT.<arith>(<value>,<value>)               % multiplication
              | DIV.<arith>(<value>,<value>)                % division
              | NEG.<arith>(<value>)                        % negation
              | REM.<int>(<value>,<value>)                  % remainder
              | ABS.<real>(<value>)                         % absolute value

<relational>::= GT.<ordered>(<value>,<value>)               % greater then
              | GTE.<ordered>(<value>,<value>)              % greater then or equal
              | LT.<ordered>(<value>,<value>)               % less then
              | LTE.<ordered>(<value>,<value>)              % less then or equal

<logical>    ::= AND.BOOL(<value>,<value>)                  % logical and
              | OR.BOOL(<value>,<value>)                    % logical or
              | NOT.BOOL(<value>)                           % logical negation

<bitwise>    ::= BAND.<int>(<value>,<value>)                % bitwise and
              | BOR.<int>(<value>,<value>)                  % bitwise or
              | BXOR.<int>(<value>,<value>)                 % bitwise exclusive or
              | BNOT.<int>(<value>)                         % bitwise negation
              | BSHIFTR.<int>(<value>,<value>)              % bitwise shift right
              | BSHIFTL.<int>(<value>,<value>)              % bitwise shift left

<string-op> ::= CAT.<string>(<value>,<value>)               % string concatenation
              | SUB.<string>(<value>,<value>,<value>)       % sub-string
              | LEN.<string>(<value>)                       % string length

<other-op>   ::= <vector-op> | <inf-op>

<vector-op> ::= LWB.VECTOR(<value>)                         % inspect lower-bound
              | UPB.VECTOR(<value>)                         % inspect upper-bound

<inf-op>     ::= TAG(<value>)                               % inspect INF tag
              | TYPE(<value>)                               % inspect INF type-rep
              | PROJ(<value>,<mc-type>,<value>)             % dynamic projection

<coercion>   ::= <mc-type>.<mc-type>(<value>)               % coerce value to new type
```

```
<map-op>     ::= EXISTS(<id>,<loc>)                     % test entry in MAP
               | LOOKUP:<mc-type>(<id>,<loc>)           % get entry from MAP
               | ISTYPE(<id>,<loc>,<mc-type>)           % test MAP entry type-rep

<call>       ::= CALL(<loc>,<value-list>)               % procedure call
<callcc>     ::= CALLCC(<string>,<value-list>,<string>) % C function call

<statement> ::= STABLE()                                % checkpoint the store
               | RESTART()                              % rollback to the last checkpoint
               | GC()                                   % garbage collects the heap
               | LABEL(<id>)                            % set a mark for JUMP()
               | NOP()                                  % no-operation
               | <branch>                               % the only branch instruction
               | <update>                               % assign a new value to an id
               | <map-stmt>                             % MAP statements
               | <bits-stmt>                            % in-situ BITS statements

<branch>     ::= BRA(<value>,<tpl-block>,JUMP(<id>),<tpl-block>,JUMP(<id>))

<update>     ::= UPDATE(<loc>,<value>)                  % loc <- v

<map-stmt>   ::= REMOVE(<id>,<loc>)                     % remove from MAP
               | INSERT.<mc-type>(<id>,<loc>,<value>)   % insert in MAP

<bits-stmt> ::= NOT.BITS(<value>,<value>,<value>)       % in situ negate range
               | SET.BITS(<value>,<value>,<value>)      % in situ set range
               | CLEAR.BITS(<value>,<value>,<value>)    % in situ clear range

<type-rep-list> ::= <type-rep>{,<type-rep-list>}        % type representation
<type-rep>    ::= <base-type>
               | RECORD(<type-rep-list>)
               | VECTOR(<type-rep>)
               | PROC(<type-rep-list> -> <type-rep>)
               | MAP
               | INF

<id-list>    ::= [<id>{,<id-list>}]                     % identifier
<id>         ::= <letter> <digit>*

<value-list>::= [<value>{,<value-list>}]                % addressing modes
<value>      ::= <literal> | <loc>
<loc>        ::= <id> | <offset> | <index>
<offset>     ::= <id>!<slot>
<index>      ::= <id>@<slot>
<slot>       ::= <id> | int

<literal>    ::= <int> | <double> | <bool> | <bits> | <pixel> | <chars> | <unicode>
```

A.2 TPL Micro-syntax

TPL micro-syntax for literals:

```
<int>         ::= {<add-op>}<digit><digit>*
<double>      ::= <int>.<digit>*{E<int>}
<bool>        ::= TRUE | FALSE
<bits>        ::= #<bit>* | <hex-number>
<pixel>       ::= #<bit><bit>*
<chars>       ::= <hexadecimal> | <string>

<hex-number> ::= X<byte><byte>*
<byte>        ::= <hexadecimal><hexadecimal>
<hexadecimal>::= <digit> |A|B|C|D|E|F
```

```
<string>     ::= "<print-char>*"
<print-char> ::= <digit> | <letter> | <special> | <other-iso> | \" | \\
<bit>        ::= 0|1
<digit>      ::= 0|1|2|3|4|5|6|7|8|9
<letter>     ::= a-z | A-Z
<special>    ::= ‘|’|~|^|:|;|,|.|?|!|@|$|#|%|&|_|(|)|{|}|[|]|<|>|=|+|-|*|/|...
<other-iso>  ::= 8-bits_iso_chars

<add-op>     ::= +|-
```

A.3 Changes for TPLk

The following TPL instructions were dropped in TPLk:

```
<id> := PROC("<type-rep-list>-><type-rep>",<id-list>,<tpl-block>PROCEND)
<id> := CALL(<loc>,<value-list>)
```

and the following new instructions were added:

```
<id> := CONT("<type-rep-list>-><type-rep>",<id-list>,<tpl-block>)  % continuation
CALLK(<loc>,<value-list>)                                           % call continuation
```

A.4 Changes for cTPL

The following TPL instructions were dropped in cTPL:

```
<id> := PROC("<type-rep-list>-><type-rep>",<id-list>,<tpl-block>PROCEND)
<id> := CONT("<type-rep-list>-><type-rep>",<id-list>,<tpl-block>)
<id> := CALL(<loc>,<value-list>)
CALLK(<loc>,<value-list>)
```

and the following new instructions were added:

```
<id> := CODE(<tpl-block>)              % bind procedure code
<id> := CLOSURE(<value-list>)          % bind procedure closure
<id> := CALLC(<loc>,<value-list>)      % call closure
CALLKC(<loc>,<value-list>)             % call closure for CPS code
```

Appendix B

COREL

COREL abstract syntax is described using the same notation as in Appendix A with the exception of *[* and *]* that are not terminal symbols, and { and } which which are not terminals. This allow *[<exp>]* to have the usual meaning of optional syntactic expression *<exp>*.

B.1 COREL Abstract Syntax

COREL abstract syntax:

```
!******* context free syntax  *******************

<pgm>          ::= <sequence> [?]

<sequence>     ::= <decl> ; <sequence>
                 | <clause> [; <sequence>]

<decl>         ::= let <object-init>
<object-init>  ::= <id> <init-op> <clause>

<clause>       ::= <name> := <clause>
                 | if <clause> then <clause> else <clause>
                 | if <clause> do <clause>
                 | while <clause> do <clause>
                 | use <clause> with <signature> in <clause>
                 | drop <id> from <clause>
                 | in <clause> let <object-init>
                 | <expr>

<signature>    ::= <named-param-list>

<expr>         ::= <exp1> [or <exp1>]*
<exp1>         ::= <exp2> [and <exp2>]*
<exp2>         ::= [~] <exp3> [<comp-op> <exp3>]
<exp3>         ::= <exp4> [<add-op> <exp4>]*
<exp4>         ::= <exp5> [<mult-op> <exp5>]*
<exp5>         ::= [<add-op>] <exp6> [^ <exp5>]*
<exp6>         ::= <literal>
                 | ( <clause> )
                 | { <sequence> }
                 | begin <sequence> end
                 | <expr> ( [<application>] )
                 | <expr> ( <dereference> )
                 | <value-constr>
                 | <clause> contains [constant] <id> [:<type-id>]
                 | PS()
                 | <id>
```

```
<name>         ::= <id> [( <id> )]*
                 | <expr> ( <clause-list> ) [<clause-list>]*

<application> ::= <clause-list>
<clause-list> ::= <clause> [, clause-list>]

<dereference> ::= <clause>

<literal>      ::= <int> | <bool> | <string> | <proc-literal>

<proc-literal>::= proc ( [<named-param-list>] [-> <type-id>] ); <clause>
<named-param-list>::= [constant] <id-list>: <type-id> [; <named-param-list>]

<value-constr>::= <struct-constr> | <vector-constr>
<struct-constr>::= struct ( [<struct-init-list>] )
<struct-init-list>::= <id> <init-op> <clause> [; <struct-init-list>]

<vector-constr>::= vector <clause> to <clause> of <clause>

<id-list>      ::= <id> [, <id-list>]

<type-id>      ::= int
                 | bool
                 | string
                 | env
                 | proc ( [<type-list>] [-> <type-id>] )
                 | structure ( [<named-param-list>] )
<type-list>    ::= <type-id> [, <type-list>]

<rel-op>       ::= <eq-op> | <comp-op>
<eq-op>        ::= = | ~=
<comp-op>      ::= < | <= | > | >=
<add-op>       ::= + | -
<mult-op>      ::= * | div | rem | ++
<init-op>      ::= = | :=

<int>          ::= <digit> <digit>*
<bool>         ::= true | false
<string>       ::= "[<letter> | <digit>]*"
<id>           ::= <letter> ( <letter> | <digit> )*
<digit>        ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter>       ::= a-z | A-Z

!* SPACE, TAB, NL, COMMENT ignored
```

B.2 COREL Type Rules

Type rules used to typecheck COREL programs:

```
!******  TYPE RULES  *******************

<pgm>:
        <sequence>: VOID [?] => VOID

        <decl>: VOID ; <sequence>:T => T
        <clause>: VOID ; <sequence>:T => T
        <clause>: T => T

        <decl> => VOID
        <id>: T := <clause>:T => VOID
        if <clause>: BOOL do <clause>: T => VOID
        if <clause>: BOOL then <clause>: T else <clause>: T => T
        while <clause>: BOOL do <clause>: VOID => VOID
        use <clause>: ENV with <signature> in <clause>: T => T
        drop <id> from <clause>: ENV => VOID
        in <clause>: ENV let <id> <init-op> <clause: NONVOID> => VOID
        <clause>: ENV [constant]contains <id> => BOOL
        <id>: T => T

        <expr>: BOOL or <expr>: BOOL => BOOL
        <expr>: BOOL and <expr>: BOOL => BOOL
        ~ <expr>: BOOL => BOOL
        <expr>: T <eq-op> <expr>: T => BOOL
        <expr>: T <add-op> <expr>: T => T
        <expr>: T <mult-op> <expr>: T => T
        <expr>: T ^ <expr>: T => T
        <add-op> <expr>: T => T

        <literal>: T => T
        ( <clause>: T ) => T
        begin <sequence>:T end => T
        <expr>: *T ( <clause>: INT ) => T

        T: nonvoid, <value-constr>: T => T

        struct( <struct-init-list> ) => STRUCTURE
        where <struct-init-list>::= <id> <init-op> <clause>: NONVOID
                                          [, <struct-init-list>]

        vector <clause>: INT to <clause>: INT of <clause>: T => *T

        true | false => BOOL
        [<add-op>] <digit> [<digit>]* => int
        proc( [<named-param-list>] [-> <type-id>]: T ); <clause>: T

!* type rules valid for all T
```

Appendix C

Using the Language Framework Prototype

In order to make concrete illustrations of previous descriptions, an example of the use of the language framework, constructed during the experimental work described in Chapter 6, will be presented in this appendix.

C.1 Procedures as Parameters

In the following program, procedure *f* using free-variable *z*, is passed in the procedure call *p(f)*.

```
let p= proc( x: proc( int -> int ) -> int )
       begin
         let dd= x( 11 )
         dd
       end
let r= proc( -> int )
       begin
         let z= 10
         let f= proc( i: int -> int )
                begin
                  let a:= i * z
                  a + 2
                end;
         let y= f( 1 )
         let a= p( f )
         a + y
       end
use PS() with writeInt: proc( int ) in
  writeInt( r() )  !* 124
?
```

It is represented in TPL as:

```
(1)     INIT()
(2)     START
(3)     R1 := PROOT()
(4)     R2 := LOOKUP:RECORD("PS",R1)
(5)     R4 := PROC("PROC(INT->INT)->INT",[A1],  % p
(6)       START
(7)       R3 := CALL(A1,[11])                 % dd
(8)       R0 := INT(R3)
(9)       END
(10)      PROCEND)
(11)    R12 := PROC("->INT",[],                % r
(12)      START
(13)      R5 := INT(10)                        % z
(14)      R8 := PROC("INT->INT",[A2],          % f
(15)        START
(16)        R6 := MULT.INT(A2,R5)              % a
(17)        R7 := PLUS.INT(R6,2)
(18)        R0 := INT(R7)
(19)        END
(20)        PROCEND)
(21)      R9 := CALL(R8,[1])                   % y
(22)      R10 := CALL(R4,[R8])                 % a
(23)      R11 := PLUS.INT(R10,R9)
(24)      R0 := INT(R11)
(25)      END
(26)      PROCEND)
(27)    R13 := CALL(R2,[])
(28)    R14 := LOOKUP:RECORD("writeInt",R13)
(29)    R15 := CALL(R12,[])
(30)    VOID := CALL(R14!0,[R15])
(31)    END
```

The identifier that corresponds to the procedure *f* (R8) is passed as parameter in the call to the procedure that corresponds to *p* (R6) in instruction 22.

C.2 Mutual Recursion

The following program illustrates the compilation to TPL of two mutually recursive procedures.

```
let fact2:= proc(i,j: int -> int); 0
let fact:= proc(i: int -> int); fact2(1,i)
fact2:= proc(n,m: int -> int)
        if n=m then n
        else fact2(n, (m+n) div 2) * fact2((m+n) div 2+1, m)
let s:= 0
let i:= 5; while i >0 do
  begin
     s := s + fact(i); i := i-1
  end
use PS() with writeInt: proc( int ) in writeInt( s )
```

It is represented in TPL as:

```
(1)     INIT()
(2)     START
(3)     R1 := PROOT()
(4)     R2 := LOOKUP:RECORD("PS",R1)
(5)     R3 := PROC("INT,INT->INT",[A1,A2],     % fact2
(6)       START
(7)       R0 := INT(0)
(8)       END
(9)       PROCEND)
(10)    R5 := PROC("INT->INT",[A3],            % fact
(11)      START
(12)      R4 := CALL(R3,[1,A3])
(13)      R0 := INT(R4)
(14)      END
(15)      PROCEND)
(16)    R16 := PROC("INT,INT->INT",[A4,A5],
(17)      START
(18)      R6 := EQ.INT(A4,A5)
(19)       R7 := INT(0)
(20)       BRA(R6,
(21)         START
(22)         UPDATE(R7,A4)
(23)         END,
             JUMP(L24),
(28)           START
(29)           R8 := PLUS.INT(A5,A4)
(30)           R9 := DIV.INT(R8,2)
(31)           R10 := CALL(R3,[A4,R9])
(32)           R11 := PLUS.INT(A5,A4)
(33)           R12 := DIV.INT(R11,2)
(34)           R13 := PLUS.INT(R12,1)
(35)           R14 := CALL(R3,[R13,A5])
(36)           R15 := MULT.INT(R10,R14)
(37)           UPDATE(R7,R15)
(38)           END,
               JUMP(L24))
           LABEL(24)
(24)       NOP
(25)       R0 := INT(R7)
(26)       END
(27)       PROCEND)
(39)     UPDATE(R3,R16)
(40)     R17 := INT(0)                     % s
(41)     R18 := INT(5)                     % i
         LABEL(42)
(42)     R19 := GT.INT(R18,0)
(43)     BRA(R19,
(44)       START
(45)       R20 := CALL(R5,[R18])
(46)       R21 := PLUS.INT(R17,R20)
(47)       UPDATE(R17,R21)
(48)       R22 := MINUS.INT(R18,1)
(49)       UPDATE(R18,R22)
(50)       END,
           JUMP(L42),
(51)         START
(52)         END,
             JUMP(L53))
         LABEL(53)
(53)     NOP
(54)     R23 := CALL(R2,[])
(55)     R24 := LOOKUP:RECORD("writeInt",R23)
(56)     VOID := CALL(R24!0,[R17])
(57)     END
```

C.3 Fibonacci Numbers

The following program illustrates the compilation to TPL of a COREL program which computes Fibonacci numbers.

```
let nfc:= 0
let nfib:= proc( n: int -> int ); 0
nfib:= proc( n: int -> int )
       begin
         nfc:= nfc +1
         if n < 2 then 1 else 1 + nfib( n-1 ) + nfib( n-2 )
       end
let discard= nfib( 28 )
use PS() with writeInt : proc( int ) in writeInt( nfc )
```

It is represented in TPL as:

```
(1)     INIT()
(2)     START
(3)     R1 := PROOT()
(4)     R2 := LOOKUP:RECORD("PS",R1)
(5)     R3 := INT(0)                            % nfc
(6)     R4 := PROC("INT->INT",[A1],             % nfib
(7)       START
(8)       R0 := INT(0)
(9)       END
(10)      PROCEND)
(11)    R14 := PROC("INT->INT",[A2],
(12)      START
(13)      R5 := PLUS.INT(R3,1)
(14)      UPDATE(R3,R5)
(15)      R6 := LT.INT(A2,2)
(16)      R7 := INT(0)
(17)      BRA(R6,
(18)        START
(19)        UPDATE(R7,1)
(20)        END,
            JUMP(L21),
(25)          START
(26)          R8 := MINUS.INT(A2,1)
(27)          R9 := CALL(R4,[R8])
(28)          R10 := PLUS.INT(1,R9)
(29)          R11 := MINUS.INT(A2,2)
(30)          R12 := CALL(R4,[R11])
(31)          R13 := PLUS.INT(R10,R12)
(32)          UPDATE(R7,R13)
(33)          END,
              JUMP(L21))
          LABEL(21)
(21)      NOP
(22)      R0 := INT(R7)
(23)      END
(24)      PROCEND)
(34)    UPDATE(R4,R14)
(35)    R15 := CALL(R4,[28])                    % discard
(36)    R16 := CALL(R2,[])
(37)    R17 := LOOKUP:RECORD("writeInt",R16)
(38)    VOID := CALL(R17!0,[R3])
(39)    END
```

C.4 A complete example

This section presents a complete example of the use of all components of the TPL language framework described in this dissertation.

C.4.1 The COREL Program

The COREL program to be compiled is as follows:

```
let a1= 10
let a2= a1+20
let p1= proc(i:int -> proc(int->int))
begin
  let a1=5
  proc(x:int->int); x*i*a1
end
let v1= vector 0 to 1 of p1(2)
let a3= v1(1)(5)
use PS() with writeInt: proc(int) in
        writeInt(a3)
?
```

C.4.2 The TPL Representation

The compilation of the COREL program by using 2TPL produces an internal representation in TPL that can externally be pretty-printed, with the node numbers inside parentheses, as represented:

```
jlopes@hawaii:/users/rapids/jlopes/tpl:89> 2tpl pgm.N
** Compilation into TPL done
** Annotating TPL... done.
jlopes@hawaii:/users/rapids/jlopes/tpl:90> pp
(1)     INIT()
(2)     START
(3)     R1 := PROOT()
(4)     R2 := LOOKUP:RECORD("PS",R1)
(5)     R3 := INT(10)                          % a1
(6)     R4 := PLUS.INT(R3,20)                  % a2
(7)     R9 := PROC("INT->PROC(INT->INT)",[A1], % p1
(8)       START
(9)       R5 := INT(5)                         % a1
(10)      R8 := PROC("INT->INT",[A2],
(11)        START
(12)        R6 := MULT.INT(A2,A1)
(13)        R7 := MULT.INT(R6,R5)
(14)        R0 := INT(R7)
(15)        END
(16)        PROCEND)
(17)      R0 := MOVE.PROC(R8)
(18)      END
(19)      PROCEND)
(20)    R10 := CALL(R9,[2])
(21)    R11 := VECTOR(0,1,R10)                 % v1
(22)    R12 := CALL(R11@1,[5])                 % a3
(23)    R13 := CALL(R2,[])
(24)    R14 := LOOKUP:RECORD("writeInt",R13)
(25)    VOID := CALL(R14!0,[R12])
(26)    END
(27)    CLOSE()Thread 168 is finished
jlopes@hawaii:/users/rapids/jlopes/tpl:91>
```

C.4.3 Optimisations in TPL

The TPL internal representation can be improved by performing the constant folding and constant propagation transformations. The transformation applied by FOLD substitutes the bindings for integer variables *R3*, *R4* and *R5* by *NOP* instructions and the usage of *R5* by its value: *5*. *NOP* instructions can then be removed by NOPS.

The optimised TPL pretty-prints as:

```
jlopes@hawaii:/users/rapids/jlopes/tpl:91> fold
** Constant folding and constant propagation... 3 done.
** Annotating TPL... done.
jlopes@hawaii:/users/rapids/jlopes/tpl:92> nops
** Removing NOPs... 3 done.
** Renumbering TPL... done.
jlopes@hawaii:/users/rapids/jlopes/tpl:93> pp
(1)     INIT()
(2)     START
(3)     R1 := PROOT()
(4)     R2 := LOOKUP:RECORD("PS",R1)
(5)     R9 := PROC("INT->PROC(INT->INT)",[A1],  % p1
(6)       START
(7)       R8 := PROC("INT->INT",[A2],
(8)         START
(9)         R6 := MULT.INT(A2,A1)
(10)        R7 := MULT.INT(R6,5)
(11)        R0 := INT(R7)
(12)        END
(13)        PROCEND)
(14)      R0 := MOVE.PROC(R8)
(15)      END
(16)      PROCEND)
(17)    R10 := CALL(R9,[2])
(18)    R11 := VECTOR(0,1,R10)                 % v1
(19)    R12 := CALL(R11@1,[5])                 % a3
(20)    R13 := CALL(R2,[])
(21)    R14 := LOOKUP:RECORD("writeInt",R13)
(22)    VOID := CALL(R14!0,[R12])
(23)    END
(24)    CLOSE()
jlopes@hawaii:/users/rapids/jlopes/tpl:94>
```

C.4.4 Closed TPL (cTPL)

To the TPL program is then applied the closure conversion transformation performed by CLOSE. It produces the cTPL code with closures *8* and *R9*.

```
jlopes@hawaii:/users/rapids/jlopes/tpl:94> clos
** Closure conversion...  I II III IV 2 done.
** Renumbering TPL... done.
jlopes@hawaii:/users/rapids/jlopes/tpl:95> pp
(1)     T23 := CODE(
(2)       START
(3)       R6 := MULT.INT(A!s0,C!s0)
(4)       R7 := MULT.INT(R6,5)
(5)       R0 := INT(R7)
(6)       END
(7)       PROCEND)
(8)     T24 := CODE(                           % p1
(9)       START
(10)      R8 := CLOSURE([T23,A!s0])
(11)      R0 := MOVE.PROC(R8)
(12)      END
(13)      PROCEND)
(14)    T0 := CODE(
(15)      START
(16)      INIT()
(17)      R1 := PROOT()
(18)      R2 := LOOKUP:RECORD("PS",R1)
(19)      R9 := CLOSURE([T24])                 % p1
(20)      R10 := CALL(R9,[2])
(21)      R11 := VECTOR(0,1,R10)               % v1
(22)      R12 := CALL(R11@p1,[5])              % a3
(23)      R13 := CALL(R2,[])
(24)       R14 := LOOKUP:RECORD("writeInt",R13)
(25)       VOID := CALL(R14!p0,[R12])
(26)       CLOSE()
(27)       END
(28)       PROCEND)
jlopes@hawaii:/users/rapids/jlopes/tpl:96>
```

C.4.5 Blackboard Information

The information kept internally for each TPL variable (lexical level, number of usages and type) is the following:

```
jlopes@hawaii:/users/rapids/jlopes/tpl:101> list
** Blackboard Information:
ID     NAME        LL     USED   COST   CALLS  FLAGS      TYPE
A1                 2      1                               INT
A2                 4      1                               INT
R0                 3      0      -      0                 PROC(INT->INT)
R1                 1      1                               MAP
R10                1      1      -      0                 PROC(INT->INT)
R11    v1          1      1                               VECTOR(PROC(INT->INT))
R12    a3          1      1                               INT
R13                1      1                               MAP
R14                1      1                               RECORD(CLOSURE(),BOOL)
R2                 1      1                               CLOSURE()
R3     a1          1      1                    F          INT
R4     a2          1      0                    F          INT
R5     a1          3      1                    F          INT
R6                 5      1                               INT
R7                 5      1                               INT
R8                 3      1      -      0                 PROC(INT->INT)
R9     p1          1      1      -      0                 PROC(INT->PROC(INT->INT))
jlopes@hawaii:/users/rapids/jlopes/tpl:102>
```

C.4.6 The C-- Code Generated

For each code object of cTPL, 2C generates object-code which is then stored inside a store object. The code objects *T23* and *T24* and the main code *T0* are represented in C-- as follows:

```
jlopes@hawaii:/users/rapids/jlopes/tpl:96> tpl2c
** Generating C code...
Compilation of CPROGS/T23.c OK
Compilation of CPROGS/T24.c OK
Compilation of CPROGS/T0.c OK
** Renumbering TPL... done.
Thread 173 is finished
jlopes@hawaii:/users/rapids/jlopes/tpl:97> cat CPROGS/T23.c
#include "../C/runtime.h"

void T23()
{
  WORD R6 = 0;
  WORD R7 = 0;
  R6 = goffword((OID) A,0) * goffword((OID) C,0);
  R7 = R6 * 5;
  R0 = (WORD) R7;
}
jlopes@hawaii:/users/rapids/jlopes/tpl:98> cat CPROGS/T24.c
```

```
#include "../C/runtime.h"

void T24()
{
  WORD R8 = 0;
  {
    int np = 0; word *ptrs[1];
    int ns = 0; word sclrs[1];
    ptrs[np++] = (OID) mkcode("T23");
    sclrs[ns++] = (word) goffword((OID) A,0);
    R8 = (WORD) mkrecord(np,ptrs,ns,sclrs);
  }
  R0 = (WORD) R8;
}
jlopes@hawaii:/users/rapids/jlopes/tpl:99> cat CPROGS/T0.c
#include "../C/runtime.h"

int main()
{
  WORD R1 = 0;
  WORD R2 = 0;
  WORD R9 = 0;
  WORD R10 = 0;
  WORD R11 = 0;
  WORD R12 = 0;
  WORD R13 = 0;
  WORD R14 = 0;
  init();
  R1 = (WORD) gcons((OID) 0, "", 0);
  R2 = (WORD) gcons((OID) R1, "PS", 16);
  {
    int np = 0; word *ptrs[1];
    int ns = 0; word sclrs[1];
    ptrs[np++] = (OID) mkcode("T24");
    R9 = (WORD) mkrecord(np,ptrs,ns,sclrs);
  }
  {
    int np = 0; word *ptrs[1];
    int ns = 0; word sclrs[1];
    sclrs[ns++] = (word) 2;
    pbuf = (WORD) mkrecord(np,ptrs,ns,sclrs);
  }
  call((WORD) R9,pbuf);
  R10 = R0;
  R11 = (WORD) mkvwordp((int) 0,(int) 1,(OID) R10);
  {
    int np = 0; word *ptrs[1];
    int ns = 0; word sclrs[1];
    sclrs[ns++] = (word) 5;
    pbuf = (WORD) mkrecord(np,ptrs,ns,sclrs);
  }
  call((WORD) gidxwordp((OID) R11,1),pbuf);
  R12 = R0;
  call((WORD) R2,pbuf);
  R13 = R0;
  R14 = (WORD) gcons((OID) R13, "writeInt", 15);
  {
    int np = 0; word *ptrs[1];
    int ns = 0; word sclrs[1];
    sclrs[ns++] = (word) R12;
    pbuf = (WORD) mkrecord(np,ptrs,ns,sclrs);
  }
  call((WORD) goffwordp((OID) R14,0),pbuf);
  shutdown();
  return 1;
}
jlopes@hawaii:/users/rapids/jlopes/tpl:100>
```

C.4.7 Executing a C-- Program

Link-editing the main program with the runtime environment is done by JUICE and the resulting executable *tpl.e* when executed with the debugging flag at ON, produces the following listing. The listing includes the program result to be printed as *50*.

```
jlopes@hawaii:/users/rapids/jlopes/tpl:100> juice
Compiling C...
gcc -static -L./C -L./C/dld -o tpl.e CPROGS/T0.o -lrun -ldld
Executing...
        awake: brek=193952, pab=233472
        awake: root=237572
        init()  hops: gcons(0,"",0) (PROOT) 237600
        hops: gcons(237600,"PS",16) 237916
        hops: mkcode (from T24):  heap: HAT=240312 cstring(3)  hop(0,5) heap: HAT=240332 ccode(476)
 hop(1,124) 240336
        hops: mkrecord:  heap: HAT=240832 crecord(1,0)  hop(1,3) 240836
        hops: mkrecord:  heap: HAT=240848 crecord(0,1)  hop(0,4) 240848
        CALL: A=240848 C=240836
        hops: goffwordp(240836,0) 240336
        hops: gstring(240312) 1412576256
        hops: gcode(240336) T24
        hops: mkcode (from T23):  heap: HAT=240864 cstring(3)  hop(0,5) heap: HAT=240884 ccode(424)
 hop(1,111) 240888
        hops: goffword(240848,0) 2
        hops: mkrecord:  heap: HAT=241332 crecord(1,1)  hop(1,4) 241336
        hops: mkvwordp:  heap: HAT=241352 cvwordp(0,1)  hop(2,5) 241360
        hops: mkrecord:  heap: HAT=241380 crecord(0,1)  hop(0,4) 241380
        hops: gidxwordp(241360,1) 241336
        CALL: A=241380 C=241336
        hops: goffwordp(241336,0) 240888
        hops: gstring(240864) 1412576000
        hops: gcode(240888) T23
        hops: goffword(241380,0) 5
        hops: goffword(241336,0) 2
        CALL: A=241380 C=237916
        hops: goffwordp(237916,0) 237640
        hops: gstring(237616) 1412759552
        hops: gcode(237640) T5
        hops: gcons(0,"",0) (PROOT) 237600
        hops: gcons(237600,"writeInt",15) 239072
        hops: mkrecord:  heap: HAT=241396 crecord(0,1)  hop(0,4) 241396
        hops: goffwordp(239072,0) 238556
        CALL: A=241396 C=238556
        hops: goffwordp(238556,0) 238228
        hops: gstring(238204) 1412890624
        hops: gcode(238228) T7
        hops: goffword(241396,0) 50
 50     shutdown()
jlopes@hawaii:/users/rapids/jlopes/tpl:101>
```

C.4.8 CPS Transformation to Produce TPLk

It should be noted that, in the prototype, the COREL program to be compiled must use versions of the constant procedures transformed themselves by CPSt. The program is as follows:

```
let a1= 10
let a2= a1+20
let p1= proc(i:int -> proc(int->int))
begin
  let a1=5
  proc(x:int->int); x*i*a1
end
let v1= vector 0 to 1 of p1(2)
let a3= v1(1)(5)
use PS() with writeIntK: proc(int) in
        writeIntK(a3)
?
```

The TPL representation of the program, before applying closure conversion, is then transformed into TPLk by the CPSt component. Four continuations are built, as represented:

```
jlopes@hawaii:/users/rapids/jlopes/tpl:104> cps
** CPS transformation... 4 done.
** Renumbering TPL... done.
Thread 164 is finished
jlopes@hawaii:/users/rapids/jlopes/tpl:105> pp
(4)     R2 := LOOKUP:RECORD("PS",R1)
(5)     R3 := INT(10)                           % a1
(6)     R4 := PLUS.INT(R3,20)                   % a2
(7)     R9 := CONT("INT,PROC(PROC(INT->INT)->VOID)->VOID",[A1,A3],  % p1
(8)       START
(9)       R5 := INT(5)                          % a1
(10)      R8 := CONT("INT,PROC(INT->VOID)->VOID",[A2,A4],
(11)        START
(12)        R6 := MULT.INT(A2,A1)
(13)        R7 := MULT.INT(R6,R5)
(14)        R0 := INT(R7)
(15)        END
(16)        CALLK(A4,[R0]))
(17)      R0 := MOVE.PROC(R8)
(18)      END
(19)      CALLK(A3,[R0]))
(20)    R15 := CONT("PROC(INT->INT)->VOID",[A5],
(21)      START
(22)      R10 := MOVE.PROC(A5)
(23)      R11 := VECTOR(0,1,R10)                % v1
(24)      R16 := CONT("INT->VOID",[A6],
(25)        START
(26)        R12 := INT(A6)
(27)        R17 := CONT("MAP->VOID",[A7],
(28)          START
(29)          R13 := MOVE.MAP(A7)
(30)          R14 := LOOKUP:RECORD("writeInt",R13)
(31)          R18 := CONT("->VOID",[],
(32)            START
(33)            END
(34)            CLOSE())
(35)          END
(36)          CALLK(R14!0,[R12,R18]))
(37)        END
(38)        CALLK(R2,[R17]))
(39)      END
(40)      CALLK(R11@1,[5,R16]))
(41)    END
(42)    CALLK(R9,[2,R15])
jlopes@hawaii:/users/rapids/jlopes/tpl:106>
```

Bibliography

[Abiteboul and Hull, 1987] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987. Cited on page 3.

[Adl-Tabatabai *et al.*, 1996] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In PLDI [1996], pages 127–136. Published as *SIGPLAN Notices 31*(5), May 1996. Cited on page 178.

[Aho *et al.*, 1986] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986. Cited on pages 26, 45–47, 49, 62, 76–77, 100, 121, 124, 133–134, and 176.

[Albano *et al.*, 1985] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985. Cited on pages 8 and 34.

[Albano *et al.*, 1994] A. Albano, G. Ghelli, and R. Orsini. Fibonacci reference manual: A preliminary version. Technical Report FIDE/94/102, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994. Cited on page 8.

[Albano *et al.*, 1995] A. Albano, G. Ghelli, and R. Orsini. An introduction to Fibonacci: A programming language for object databases. Technical Report FIDE/95/120, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1995. Cited on page 26.

[Andrews *et al.*, 1989] T. Andrews, C. Harris, K. Sinkel, and J. Duhl. The ONTOS object database. Technical report, Ontologic Inc., Burlinghton, MA, 1989. Cited on page 3.

[Appel and Jim, 1989] A.W. Appel and T. Jim. Continuation-passing, closure-passing style. In POPL [1989], pages 293–302. Cited on pages 38, 47, and 150.

[Appel and MacQueen, 1987] A.W. Appel and D.B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 301–324. Springer-Verlag, NY, 1987. Cited on pages 22, 38, 47, 139, and 177.

[Appel and MacQueen, 1991] A.W. Appel and D.B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third International Symposium on Programming Languages Implementation and Logic Programming (Passau, Germany, 26–28 August 1991)*, pages 1–13, Berlin, Germany, 1991. Springer-Verlag. Cited on page 38.

[Appel and Shao, 1994] A.W. Appel and Z. Shao. An empirical and analytical study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton University, Department of Computer Science, Princeton, NJ, March 1994. Cited on page 72.

[Appel, 1987] A.W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987. Cited on page 72.

[Appel, 1990] A.W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, November 1990. Cited on page 26.

[Appel, 1992] A.W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992. Cited on pages 26, 47–48, 50, 73, 76, 125, 138–139, 141, 143, 176, 180–181, and 185.

[Arnold and Gosling, 1996] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, Reading, MA, 1996. Cited on pages 175 and 178.

[Atkinson and Buneman, 1987] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987. Cited on page 6.

[Atkinson and Jordan, 1996] M.P. Atkinson and M.J. Jordan, editors. *Proceedings of the First International Workshop on Persistence and Java (September 1996, Drymen, Scotland)*. Sunlabs Technical Report, September 1996. Cited on page 176.

[Atkinson and Morrison, 1985] M.P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 4(7):539–559, October 1985. Cited on pages 4, 30, 33, 36, and 122.

[Atkinson and Morrison, 1987] M.P. Atkinson and R. Morrison. Polymorphic names, types, constancy and magic in a type secure persistent object store. In Carrick and Cooper [1987], pages 1–12. Proceedings of the Second International Workshop on Persistent Object Systems (Appin, Scotland, 25th–28th August 1987). Cited on page 30.

[Atkinson and Morrison, 1988] M.P. Atkinson and R. Morrison. Types, bindings and parameters in a persistent environment. In Atkinson et al. [1988], chapter 1, pages 3–20. Edited Papers from the Proceedings of the First Workshop on Persistent Object Systems (Appin, Scotland, August 1985). Cited on pages 30 and 41.

[Atkinson and Morrison, 1990] M.P. Atkinson and R. Morrison. Polymorphic names and iterations. In Bancilhon and Buneman [1990], chapter 14, pages 241–256. Edited Proceedings of the Workshop on Database Programming Languages (Roscoff, Brittany, France, September 1987). Cited on pages 30 and 117.

[Atkinson and Morrison, 1995] M.P. Atkinson and R. Morrison. Orthogonal persistent object systems. *VLDB Journal*, 4(3), 1995. Cited on pages 2, 4, and 7.

[Atkinson *et al.*, 1982] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-algol: An algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982. Cited on pages 3, 6 and 8.

[Atkinson *et al.*, 1983a] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983. Cited on page 4.

[Atkinson *et al.*, 1983b] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. CMS—a chunk management system. *Software Practice and Experience*, 13(3):273–285, March 1983. Cited on page 30.

[Atkinson *et al.*, 1983c] M.P. Atkinson, K.J. Chisholm, W.P. Cockshott, and R.M. Marshall. Algorithms for a persistent heap. *Software Practice and Experience*, 13(3):259–272, March 1983. Cited on page 30.

[Atkinson *et al.*, 1987] M.P. Atkinson, J.R. Lucking, R. Morrison, and G.D. Pratten. Persistent Information Space Architecture — PISA club rules. Technical Report PPRR-47-87, Universities of Glasgow and St Andrews, 1987. Cited on page 55.

[Atkinson *et al.*, 1988] M.P. Atkinson, O.P. Buneman, and R. Morrison, editors. *Data Types and Persistence*. Topics in Information Systems, series editors M.L. Brodie, J. Mylopoulos and Schmidt, J.W. Springer-Verlag, 1988. Edited Papers from the Proceedings of the First Workshop on Persistent Object Systems (Appin, Scotland, August 1985). Cited on pages 212, 219, and 221.

[Atkinson *et al.*, 1989] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases. Proceedings of the First International Conference on Deductive and Object-Oriented Databases (Kyoto, Japan, 4th–6th December 1989)*. Elsevier Science Publisher B.V., 1989. Cited on page 4.

[Atkinson *et al.*, 1994] M.P. Atkinson, V. Benzaken, and D. Maier, editors. *Persistent Object Systems*. Workshops in Computing. Springer-Verlag in collaboration with the British Computer Society, 1994. Proceedings of the Sixth International Workshop on Persistent Object Systems (Tarascon, Provence, France, 5th–9th September 1994). Cited on pages 215, 217, and 225.

[Atkinson *et al.*, 1996] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, December 1996. Cited on page 176.

[Atkinson, 1978] M.P. Atkinson. Programming languages and databases. In S.B. Yao, editor, *The Fourth International Conference on Very Large Data Bases (Berlin, West Germany, September 1978)*, pages 408–419, September 1978. Cited on pages 4 and 6.

[Atkinson, 1991] M.P. Atkinson. A vision of persistent systems. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases (Munich, December 1991)*, pages 453–459, 1991. Invited paper. Cited on page 16.

[Atkinson, 1992a] M.P. Atkinson. Persistent foundations for scalable multi-paradigmal systems. Invited paper. In Özsu et al. [1992]. Cited on pages 7, 9, and 16.

[Atkinson, 1992b] M.P. Atkinson. SPF scalable persistent foundations: Well engineered support for very high performance persistent systems. SERC case for support, January 1992. Cited on page 20.

[Atkinson, 1997] M.P. Atkinson, editor. *Fully Integrated Data Environments*. Springer-Verlag, 1997. To be published. Cited on pages 217 and 224.

[Auslander *et al.*, 1996] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In PLDI [1996], pages 149–159. Published as *SIGPLAN Notices 31*(5), May 1996. Cited on page 178.

[Bacon *et al.*, 1994] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994. Cited on page 124.

[Bailey *et al.*, 1980] P.J. Bailey, P. Maritz, and R. Morrison. The s-algol abstract machine. Technical Report CS-80-2, Department of Computational Science, University of St Andrews, 1980. Cited on page 54.

[Bancilhon and Buneman, 1990] F. Bancilhon and O.P. Buneman, editors. *Advances in Database Programming Languages*. ACM Press, Frontier Series. Addison-Wesley Publishing Company and ACM Press, 1990. Edited Proceedings of the Workshop on Database Programming Languages (Roscoff, Brittany, France, September 1987). Cited on pages 212 and 221.

[Bancilhon *et al.*, 1988] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, and F. Velez. The design and implementation of o_2, an object-oriented database system. In *Advances in Object-Oriented Database Systems, Proceedings of the Second International Workshop on Object-Oriented Database Systems*, number 334 in Lecture Notes in Computer Science, pages 1–22. Springer-Verlag, 1988. Cited on pages 3 and 8.

[Bancilhon *et al.*, 1992] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, 1992. Cited on page 42.

[Bartlett, 1988] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report WRL, Research Report 88/2, DEC Western Research Laboratory, Palo Alto, California, February 1988. Cited on pages 185 and 189.

[Bartlett, 1989] Joel F. Bartlett. SCHEME→C: a portable Scheme-to-C compiler. Technical Report WRL, Research Report 89/1, DEC Western Research Laboratory, Palo Alto, California, January 1989. Cited on pages 27 and 64.

[Bawden *et al.*, 1977] A. Bawden, R. Greenblatt, J. Holloway, T. Knight, D. Moon, and D. Weinreb. LISP machine progress report. A.I. Lab Memo 444, MIT, August 1977. Cited on page 185.

[Beeri *et al.*, 1987] C. Beeri, S. Naqvi, R. Ramakrishan, O. Schmueli, and S. Tsur. Sets and negation in a logic and database language (LDL1). In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*. ACM Press, 1987. Cited on page 3.

[Benitez and Davidson, 1994] M.E. Benitez and J.W. Davidson. The advantages of machine-dependent global optimization. In J. Gutknecht, editor, *Programming Languages and System Architectures, International Conference (Zurich, Switzerland, March 1994) Proceedings*, volume 782 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1994. Cited on page 135.

[Benzaken and Delobel, 1990] V. Benzaken and C. Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. In A. Dearle, G.M. Shaw, and S.B. Zdonik, editors, *Implementing Persistent Object Bases, Principles and Practice*, pages 403–412. San Mateo, CA: Morgan Kaufmann Publishers, 1990. Proceedings of the Fourth International Workshop on Persistent Object Systems, Their Design, Implementation and Use (Martha's Vineyard, USA, September 1990). Cited on page 4.

[Berners-Lee *et al.*, 1992] T. Berners-Lee, R. Cailliau, and B. Pollermann. World-wide web: The information universe. *Electronic Networking: Research, Applications and Policy*, 1(2):52–58, 1992. Cited on page 1.

[Bobrow and Wegbreit, 1973] D.G. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16(10):591–603, October 1973. Cited on page 72.

[Boehm and Wieser, 1988] A. Boehm and M. Wieser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988. Cited on page 185.

[Boehm, 1991] A. Boehm. Simple GC-safe compilation. In *OOPSLA '91, Workshop on Garbage Collection in Object Oriented Systems*, 1991. Position paper. Cited on page 65.

[Bolognesi and Brinksma, 1989] B. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In *The formal description Technique LOTOS*, pages 303–326. North-Holland Publishing Company, Amsterdam, 1989. Cited on page 4.

[Booch, 1991] Grady Booch. *Object Oriented Design with applications*. Benjamin/Cummings, Redwood City, CA, 1991. Cited on page 4.

[Bor, 1990] Borland International. *PARADOX Relational Database, User's Guide, version 3.5*, 1990. Cited on page 3.

[Brandis and Mössenböck, 1994] M.M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994. Cited on page 53.

[Brandis, 1995] M.M. Brandis. *Optimizing Compilers for Structured Programming Languages*. PhD thesis, Swiss Federal Institute of Technology Zurich, ETH Zürich, 1995. Cited on pages 45 and 53.

[Brodie, 1984] M.L Brodie. On the development of data models. In M.L. Brodie, J. Mylopoulos, and Schmidt J.W., editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1984. Cited on page 3.

[Brown *et al.*, 1988] A.L. Brown, R. Carrick, R.C.H. Connor, A. Dearle, and R. Morrison. The Persistent Abstract Machine. Technical Report PPRR-59-88, Universities of Glasgow and St Andrews, 1988. Cited on pages 22, 26, 42, and 54.

[Brown *et al.*, 1992] A.L. Brown, G. Mainetto, F. Matthes, R. Mueller, and D.J. McNally. An open system architecture for a persistent object store. In Morrison and Atkinson (minitrack coordinators) [1992], pages 766–776. Cited on page 42.

[Brown *et al.*, 1994] A.L. Brown, R. Carrick, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby, R. Morrison, and D.S. Munro. The Persistent Abstract Machine Version 10 / Napier88 (Release 2.0). Universities of St Andrews and Adelaide, 1994. Cited on page 105.

[Brown, 1989] A.L. Brown. *Persistent Object Stores*. PhD thesis, University of St Andrews, 1989. Cited on page 42.

[Bushell *et al.*, 1994] S.J. Bushell, A.L. Brown, A. Dearle, and F.A Vaugham. Native code generation in persistent systems. In Atkinson et al. [1994]. Proceedings of the Sixth International Workshop on Persistent Object Systems (Tarascon, Provence, France, 5th–9th September 1994). Cited on pages 64, 168, and 184.

[Cardelli and Wegner, 1985] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985. Cited on pages 33–34 and 75.

[Cardelli, 1983] L. Cardelli. The functional abstract machine. *Polymorphism*, 1(1), 1983. Cited on pages 26, 38, 54, and 149.

[Cardelli, 1986] L. Cardelli. Amber. In G. Cousineau, P. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*, number 242 in Lecture Notes in Computer Science, pages 21–47. Springer-Verlag, 1986. Cited on pages 8 and 32.

[Cardelli, 1989] L. Cardelli. Typeful programming. Digital Systems Research Center Report 45, Digital Equipment Corporation, Systems Research Centre, 130 Lytton Avenue, Palo Alto, Calif., USA, May 1989. Cited on page 42.

[Carey and DeWitt, 1996] M.J. Carey and D.J. DeWitt. Of objects and databases: A decade of turmoil. In *Proceedings of the 22th International Conference on Very Large Data Bases (Mumbay (Bombay), India, September 3-6th 1996)*, pages 1–12, 1996. Cited on page 7.

[Carey *et al.*, 1988] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for EXODUS. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, Chicago, USA, May 1988. Cited on page 3.

[Carey *et al.*, 1994] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, and M.J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (Minneapolis, Minnesota, May 24-27, 1994)*, pages 383–394, 1994. Cited on page 8.

[Carrick and Cooper, 1987] R. Carrick and R.L. Cooper, editors. *Persistent Object Systems: Their Design, Implementation and Use.* Universities of Glasgow and St Andrews Technical Report PPRR-44-87, 1987. Proceedings of the Second International Workshop on Persistent Object Systems (Appin, Scotland, 25th–28th August 1987). Cited on pages 212, 221, and 225.

[Carson, 1989] C. Carson. *CASE*DESIGNER User's Guide and Tutorial.* ORACLE Corporation UK Limited, Chertsey, Surrey, November 1989. Cited on page 3.

[Cattell, 1991a] R.G.G. Cattell. Next generation database systems. *Communications of the ACM*, 34(10), October 1991. Editor. Cited on page 3.

[Cattell, 1991b] R.G.G. Cattell. *Object Data Management.* Addison-Wesley Publishing Company, Reading, MA, 1991. Cited on page 3.

[Cattell, 1994] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994. Cited on page 3.

[Chambers and Ungar, 1991] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA '91 (Phoenix, Arizona, October 1991)*, pages 1–15, 1991. Published as *SIGPLAN Notices 26*(11), November 1993. Cited on pages 22 and 177.

[Chase, 1990] David Chase. Private communication (posted to the USENET newsgroup comp.compilers), August 1990. Cited on pages 27 and 64.

[Chen, 1976] P.P. Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976. Cited on page 3.

[Coad and Yourdon, 1990] P. Coad and E. Yourdon. *Object-oriented Analysis.* Yourdon Press and Prentice-Hall, Englewood Cliffs, New Jersey, 1990. Cited on page 4.

[Cockshott *et al.*, 1984] W.P. Cockshott, M.P. Atkinson, K.J. Chisholm, P.J. Bailey, and R. Morrison. POMS: a persistent object management system. *Software Practice and Experience*, 14(1):49–71, January 1984. Cited on page 42.

[Codasyl Committee on Data System Languages, 1971] Codasyl Committee on Data System Languages. Codasyl data base task group report. Technical report, Association for Computing Machinery, 1971. Cited on page 2.

[Codd, 1970] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970. Cited on page 3.

[Codd, 1972] E.F. Codd. Further normalisation of the data base relational model. In R. Rustin, editor, *Data Base Systems, Courant Computer Science Symposia Series*, volume 6, pages 33–64. Prentice-Hall, Englewood Cliffs, NJ, 1972. Cited on page 3.

[Codd, 1979] E.F. Codd. Extending the relational model of data to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979. Cited on page 3.

[Connor *et al.*, 1989] R.C.H. Connor, A.L. Brown, R. Carrick, A. Dearle, and R. Morrison. The Persistent Abstract Machine. In Rosenberg and Koch [1989], pages 353–366. Proceedings of the Third International Workshop on Persistent Object Systems (10th–13th January 1989, Newcastle, New South Wales, Australia). Cited on pages 22, 26, 37, 54, and 115.

[Connor, 1991] R.C.H. Connor. *Types and Polymorphism in Persistent Programming Systems*. PhD thesis, University of St Andrews, 1991. Cited on pages 7, 39, 41, 75, 110, 114, and 118.

[Cooper *et al.*, 1987] R.L. Cooper, M.P. Atkinson, A. Dearle, and D. Abderrahmane. Constructing database systems in a persistent environment. In P.M. Stocker and W. Kent, editors, *Proceedings of the Thirteenth International Conference on Very Large Data Bases (Brighton, England, 1987)*, pages 117–125. Los Altos, CA: Morgan Kaufmann Publishers, September 1987. Cited on pages 30 and 35.

[Cox, 1984] B.J. Cox. Message/object programming: An evolutionary change in programming technology. *IEEE Software*, 1(1):12–18, August 1984. Cited on page 40.

[Cutts *et al.*, 1997] Q.I. Cutts, R.C.H. Connor, and R. Morrison. The PamCase machine. In Atkinson [1997], chapter 2.1.3. To be published. Cited on pages 38 and 172.

[Cutts, 1993] Q.I. Cutts. *Delivering the Benefits of Persistence to System Construction and Execution*. PhD thesis, University of St Andrews, 1993. Cited on pages 7, 110, 118, and 122.

[Cytron *et al.*, 1989] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K Zadeck. An efficient method of computing static single assignment form. In POPL [1989], pages 25–35. Cited on page 52.

[Davelaar and van Kooten, 1996] S. Davelaar and S.T. van Kooten. *Custom Development Systems Design and Generation using Designer/2000*. Oracle Corporation, 1996. Cited on page 4.

[Davie and McNally, 1990a] Antony J.T. Davie and David J. McNally. CASE — a lazy version of an SECD machine with a flat environment. Technical Report CS/90/19, University of St Andrews, 1990. Cited on page 26.

[Davie and McNally, 1990b] Antony J.T. Davie and David J. McNally. The Staple language reference manual. Technical Report CS/90/16, University of St Andrews, 1990. Cited on page 8.

[Davie and McNally, 1992] Antony J.T. Davie and David J. McNally. PCASE — a persistent lazy version of an SECD machine. Technical Report CS/92/7, University of St Andrews, 1992. Cited on pages 26 and 42.

[Davie and Morrison, 1981] Antony J.T. Davie and R. Morrison. *Recursive Descent Compiling*. Ellis Horwood Publishers, Chichester, UK, 1981. Cited on pages 36, 72, and 119.

[Davie, 1979] Antony J.T. Davie. Variable access in languages in which procedures are first class citizens. Technical Report CS/79/2, Department of Computational Science, University of St Andrews, 1979. Cited on pages 38 and 149.

[Dearle *et al.*, 1994] A. Dearle, R. di Bona, J. Farrow, F. Henskens, D . Hulse, A. Lindström, S. Norris, J. Rosenberg, and F. Vaughan. Protection in Grasshopper: A Persistent Operating System. In Atkinson et al. [1994]. Proceedings of the Sixth International Workshop on Persistent Object Systems (Tarascon, Provence, France, 5th–9th September 1994). Cited on pages 17 and 171.

[Dearle, 1988] A. Dearle. *On the Construction of Persistent Programming Environments*. PhD thesis, University of St Andrews, 1988. Cited on pages 7, 55–56, and 75.

[Dearle, 1989] A. Dearle. Environments: A flexible binding mechanism to support system evolution. In B.H. Shriver, editor, *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, Volume II Software Track (January 1989)*, pages 46–55, 1989. Cited on page 108.

[Defence Research Agency, 1991] Defence Research Agency. TDF specification part I. Technical report, United Kingdom's Defence Research Agency, RSRE, Malvern, 1991. Contact N.E. Peeling. Cited on pages 27 and 61.

[Defence Research Agency, 1992] Defence Research Agency. TDF facts & figures. Technical report, United Kingdom's Defence Research Agency, RSRE, Malvern, 1992. Cited on page 61.

[Defence Research Agency, 1994] Defence Research Agency. A guide to TDF specification. Technical report, United Kingdom's Defence Research Agency, RSRE, Malvern, June 1994. Issue 3.0. Cited on page 61.

[Delobel *et al.*, 1995] C. Delobel, C. Lécluse, and P. Richard. *Databases: From Relational to Object-Oriented Systems*. International Thomson Publishing, London, UK, 1995. Cited on page 4.

[DeMarco, 1978] Tom DeMarco. *Structured Design and System Specification*. Yourdon Press and Prentice-Hall, Englewood Cliffs, NJ, 1978. Cited on page 4.

[Deutsch and Schiffman, 1984] P.L. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages—POPL (Salt Lake City, Utah, January 1984)*, pages 297–30. Association for Computing Machinery, 1984. Cited on pages 22 and 177.

[Dilles, 1990] A. Dilles. *Z: An Introduction to Formal Methods*. Wiley, 1990. Cited on page 4.

[Diwan *et al.*, 1992] A. Diwan, J.E.B. Moss, and R. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation (San Francisco, CA, June 1992)*, pages 273–282. Association for Computing Machinery, 1992. Published as *SIGPLAN Notices 27*(7), July 1992. Cited on pages 65 and 185.

[Diwan *et al.*, 1995] A. Diwan, D. Tarditi, and J.E.B. Moss. Memory subsystem performance of programs with intensive heap allocation. An earlier version is available as a CMU technical report: CMU-CS-93-227, 1995. Cited on page 72.

[Ellis and Stroustrup, 1990] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990. Cited on page 41.

[Engler, 1996] D.R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In PLDI [1996], pages 160–170. Published as *SIGPLAN Notices 31*(5), May 1996. Cited on page 178.

[Evered, 1985] M. Evered. *Leibniz — a Language to Support Software Engineering*. PhD thesis, Technical University of Darmstad, 1985. Cited on page 8.

[Feldman *et al.*, 1990] S.I. Feldman, D.M. Gay, M.W. Maimone, and N.L. Schryer. A Fortran-to-C converter. Technical Report Computer Science, No. 149, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1990. Cited on page 27.

[Ferrante *et al.*, 1987] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987. Cited on page 52.

[Fischer and Leblanc, Jr., 1988] C.N. Fischer and R.J. Leblanc, Jr. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA, 1988. Cited on page 54.

[Fishman *et al.*, 1987] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan. Iris, an object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987. Cited on page 3.

[Flanagan *et al.*, 1993] C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, June 1993. Cited on pages 48 and 177.

[Franz, 1995] M.S.O. Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, Swiss Federal Institute of Technology Zurich, ETH Zürich, 1995. Cited on pages 22 and 178.

[Fraser *et al.*, 1992] C.H. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992. Cited on page 62.

[Gane and Sarson, 1982] T. Gane and C. Sarson. *Structured Systems Design*. McDonell Douglas, 1982. Cited on page 4.

[Gawecki and Matthes, 1994] A. Gawecki and F. Matthes. The Tycoon Machine Language TML an optimizable persistent program representation. Technical Report FIDE/94/100, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994. Cited on pages 26, 47, 49–50, and 76.

[George *et al.*, 1994] L. George, F. Guillame, and J.H. Reppy. A portable and optimizing back end for the SML/NJ compiler. In P.A. Fritzson, editor, *Compiler Construction, 5th International Conference, CC'94, (Edinburgh, U.K., April 1994) Proceedings*, volume 786 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1994. Cited on pages 63 and 177.

[Gillespie, 1989] D. Gillespie. The p2c translator. Available from csvax.cs.caltech.edu by anonymous ftp under the GNUcopyleft, 1989. Cited on page 27.

[Goldberg and Robson, 1983] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Company, Reading, MA, 1983. Cited on page 40.

[Goos *et al.*, 1983] G. Goos, W.A. Wulf, A. Evans Jr., and K.J. Butler, editors. *DIANA An Intermediate Language for ADA*. Number 161 in Lecture Notes in Computer Science. Springer-Verlag, 1983. Cited on page 55.

[Gosling *et al.*, 1996] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley Publishing Company, Reading, MA, 1996. Cited on page 175.

[Gosling, 1995] J. Gosling. Java intermediate bytecodes. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations (San Francisco, CA, January 22, 1995)*, pages 111–118. Association for Computing Machinery, 1995. Published as *SIGPLAN Notices 30*(3), March 1995. Cited on pages 1 and 178.

[Gray *et al.*, 1988] P.M.D. Gray, D.S. Moffat, and J.B.H. du Boulay. Persistent prolog: A searching storage manager for prolog. In Atkinson et al. [1988], pages 353–368. Edited Papers from the Proceedings of the First Workshop on Persistent Object Systems (Appin, Scotland, August 1985). Cited on page 8.

[Gruber and Valduriez, 1994] O. Gruber and P. Valduriez. An object-oriented foundation for desktop computing. Technical Report FIDE/94/80, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994. Cited on page 16.

[Gruber *et al.*, 1992] O. Gruber, L. Amsaleg, L. Daynès, and P. Valduriez. Eos, an environment for object-based systems. In J. Rosenberg (minitrack coordinator), editor, *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, Volume I, Emerging Technologies, Architectural and Operating System Support for Persistent Object Systems*, pages 757–768, 1992. Cited on page 42.

[Hammer and McLeod, 1981] M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981. Cited on page 3.

[Hieb *et al.*, 1990] R. Hieb, R.K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation (White Plains, NY, June 1990)*, pages 66–77. Association for Computing Machinery, 1990. Published as *SIGPLAN Notices 25*(6), June 1990. Cited on page 72.

[Ho and Olsson, 1991] W.W. Ho and R.A. Olsson. An approach to genuine dynamic linking. *Software Practice and Experience*, 21(4):375–390, April 1991. Cited on page 168.

[Hölzle and Ungar, 1994] U Hölzle and D. Ungar. Optimizing dynamically-dispatched call with run-time type feedback. In PLDI [1994], pages 326–336. Published as *SIGPLAN Notices 29*(6), June 1994. Cited on page 177.

[Hölzle *et al.*, 1991] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamic-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-oriented Programming (1991)*, Lecture Notes in Computer Science 512. Springer-Verlag, 1991. Cited on page 39.

[Horwitz *et al.*, 1990] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. Cited on pages 53 and 79.

[Hosking *et al.*, 1990] A. Hosking, J.E.B Moss, and C. Bliss. Design of an object faulting persistent smalltalk. Technical Report 90-45, University of Massachusetts, Amherst, Massachusetts, May 1990. Cited on page 8.

[Hull *et al.*, 1989] R. Hull, R. Morrison, and D. Stemple, editors. *Database Programming Languages*. San Mateo, CA: Morgan Kaufmann Publishers, 1989. Proceedings of the Second International Workshop on Database Programming Languages (Salishan Lodge, Gleneden Beach, Oregon, June 1989). Cited on page 222.

[Hurst and Sajeev, 1989] A.J. Hurst and A.S.M. Sajeev. A capability based language for persistent programming: Implementation issues. In Rosenberg and Koch [1989], pages 109–125. Proceedings of the Third International Workshop on Persistent Object Systems (10th–13th January 1989, Newcastle, New South Wales, Australia). Cited on page 8.

[IBM, 1978] IBM. Ibm internal report on the contents of a sample of programs surveyed. Technical report, IBM Research Centre San Jose, California, 1978. Cited on page 5.

[Jackson, 1983] M.A. Jackson. *System Development.* Prentice-Hall, 1983. Cited on page 4.

[Jensen and Wirth, 1975] K. Jensen and N. Wirth. *PASCAL User Manual and Report.* Springer-Verlag, Berlin, Germany, second edition, 1975. Cited on page 33.

[Johnston, 1971] J.B. Johnston. The contour model of block structure processes. *ACM SIGPLAN Notices,* 6(2):56–82, 1971. Cited on page 37.

[Jones, 1990] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, NJ, second edition, 1990. ISBN 0-13-880733-7. Cited on page 4.

[Kato and Ohori, 1992] K. Kato and A. Ohori. An approach to multilanguage persistent type system. In Morrison and Atkinson (minitrack coordinators) [1992], pages 810–819. Cited on page 20.

[Kernighan and Ritchie, 1988] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, second edition, 1988. Cited on page 37.

[Kim *et al.*, 1988] W. Kim, N. Ballou, J. Banerjee, H. Chou, J. Garza, and D. Woelk. Integrating an object-oriented programming system with a database system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (San Diego, CA, 25th–30th September, 1988),* 1988. Cited on page 8.

[Kirby *et al.*, 1994] G.N.C. Kirby, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, V.S. Moore, R. Morrison, and D.S. Munro. The Napier88 standard library reference manual (version 2.2). Technical Report FIDE/94/105, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994. Cited on page 108.

[Kirby, 1993] G.N.C. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems.* PhD thesis, University of St Andrews, 1993. Cited on pages 34–35 and 41.

[Krablin, 1987] G.L. Krablin. Using abstract data type techniques in a concurrent persistent programming system. In Carrick and Cooper [1987]. Proceedings of the Second International Workshop on Persistent Object Systems (Appin, Scotland, 25th–28th August 1987). Cited on page 30.

[Krablin, 1988] G.L. Krablin. Building flexible multilevel transactions in a distributed persistent environment. In Atkinson et al. [1988], chapter 14, pages 213–234. Edited Papers from the Proceedings of the First Workshop on Persistent Object Systems (Appin, Scotland, August 1985). Cited on page 30.

[Kranz *et al.*, 1987] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction (Palo Alto, CA, June 1986),* pages 219–233. Association for Computing Machinery, 1987. Published as *SIGPLAN Notices 21*(7), July 1986. Cited on pages 47, 120, and 139.

[Lamb *et al.*, 1991] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM,* 34(10):50–63, October 1991. Cited on pages 3 and 8.

[Lécluse *et al.*, 1990] C. Lécluse, P. Richard, and F. Velez. O$_2$, an object-oriented data model. In Bancilhon and Buneman [1990], chapter 15, pages 257–276. Edited Proceedings of the Workshop on Database Programming Languages (Roscoff, Brittany, France, September 1987). Cited on page 4.

[Lee and Leone, 1996] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI [1996], pages 137–148. Published as *SIGPLAN Notices 31*(5), May 1996. Cited on page 178.

[Leroy, 1990] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Rapport Techniques 117, INRIA, Rocquencourt, February 1990. Cited on page 23.

[Lindholm and Yellin, 1997] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley Publishing Company, Reading, MA, 1997. Cited on pages 176 and 178.

[Liskov *et al.*, 1994] B. Liskov, D. Curtis, M. Day, S. Ghemawat, P. Gruber, R. Johnson, and A.C. Myers. Theta reference manual. Technical Report Memo 88, Programming Methodology Group, MIT Laboratory for Computer Science, February 1994. Also available at http://www.pmg.lcs.mit.edu/papers/thetaref/. Cited on page 8.

[Lopes, 1992] J.C. Lopes. High performance target language for persistent systems. Department of Computing Science, University of Glasgow, First Year Report, June 1992. Cited on page 21.

[Lopes, 1993] J.C. Lopes. ShTh–Show Thesaurus user interface. Technical Report FIDE/93/76, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1993. Cited on page 21.

[Macrakis, 1993] S. Macrakis. Delivering applications to multiple platforms using ANDF. *AIXpert*, August 1993. Cited on pages 62 and 173.

[Maes, 1987] Patricia Maes. *Computational Reflection*. PhD thesis, Universiteit Brussel, 1987. Cited on pages 34 and 41.

[Maier and Stein, 1987] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B.S. Shriver and P. Wegner, editors, *Research Directions in Object Oriented Programming*, Computer Systems, pages 355–392. MIT Press, Cambridge, MA, 1987. Cited on pages 3 and 8.

[Makins, 1991] Marian Makins, editor. *Collins English Dictionary*. HarperCollins Publishers, Glasgow, UK, third edition, 1991. Cited on page 18.

[Mashburn and Satyanarayanan, 1994] H.H. Mashburn and Satyanarayanan. RVM recoverable virtual memory. RVM Release 1.3, January 1994. Cited on pages 164 and 189.

[Matthes and Schmidt, 1989] F. Matthes and J.W. Schmidt. The type system of DBPL. In Hull et al. [1989], pages 219–225. Proceedings of the Second International Workshop on Database Programming Languages (Salishan Lodge, Gleneden Beach, Oregon, June 1989). Cited on page 8.

[Matthes *et al.*, 1992] F. Matthes, R. Mueller, and J.W. Schmidt. Object stores as servers in persistent programming environments—the P-Quest experience. Technical Report FIDE/92/48, ESPRIT Basic Research Action, Project Number 3070—FIDE, 1992. Cited on pages 22, 26, and 42.

[Matthes *et al.*, 1994] F. Matthes, S. Müßig, and J.W. Schmidt. Persistent polymorphic programming in Tycoon: An introduction. Technical Report FIDE/94/106, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994. Cited on pages 8 and 49.

[Matthes, 1991] F. Matthes. P-Quest: Installation and user manual. DBIS Tycoon Report 101-91, Universität Hamburg, Germany, October 1991. Cited on page 8.

[Matthews, 1985] D.C.J. Matthews. Poly manual. *ACM SIGPLAN Notices*, 20(9):52–76, September 1985. Cited on page 8.

[McCarthy and others, 1962] J. McCarthy et al. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, 1962. Cited on page 33.

[McNally, 1993] D.J. McNally. *Models of Persistence in Lazy Functional Programming Systems*. PhD thesis, Department of Computational Science, University of St Andrews, 1993. Cited on pages 26 and 42.

[Meyer, 1988] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall International, Hemel Hempstead, UK, 1988. Cited on page 40.

[Mic, 1992] Microsoft Corporation. *Microsoft Access Language Reference*, 1992. Cited on page 3.

[Milner, 1983] R. Milner. A proposal for standard ML. *Polymorphism*, 1(3), December 1983. Cited on page 33.

[Morrison and Atkinson (minitrack coordinators), 1992] R. Morrison and M.P. Atkinson (minitrack coordinators), editors. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, Volume II, Software Technology, Persistent Object Systems*, 1992. Cited on pages 215 and 221.

[Morrison and Atkinson, 1990] R. Morrison and M.P. Atkinson. Persistent languages and architectures. In Rosenberg and Keedy [1990], pages 9–28. Invited paper. Cited on page 7.

[Morrison *et al.*, 1986] R. Morrison, A. Dearle, A.L. Brown, and M.P. Atkinson. An integrated graphics programming environment. *Computer Graphics Forum*, 5(2):147–157, June 1986. Also available as PPRR-14-86. Cited on page 30.

[Morrison *et al.*, 1989] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. The Napier88 reference manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989. Cited on pages 3, 8, 30, and 33.

[Morrison *et al.*, 1990] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby, J. Rosenberg, and D. Stemple. Protection in persistent object systems. In Rosenberg and Keedy [1990], pages 48–66. Cited on page 20.

[Morrison *et al.*, 1991] R. Morrison, A. Dearle, R.C.H. Connor, and A.L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13(3):342–371, July 1991. Cited on pages 37, 39, and 115.

[Morrison *et al.*, 1994] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby, and D.S. Munro. The Napier88 reference manual (release 2.0). Technical Report FIDE/94/104, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994. Cited on page 30.

[Morrison, 1979] R. Morrison. *On the development of algol*. PhD thesis, University of St Andrews, 1979. Cited on page 6.

[Morrison, 1982] R. Morrison. Low cost computer graphics for micro computers. *Software Practice and Experience*, 12(8):767–776, 1982. Cited on page 30.

[Moss, 1987] J.E.B. Moss. Managing stack frames in Smalltalk. In *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretative Techniques (St. Paul, Minnesota, June 1987)*, pages 229–240. Association for Computing Machinery, 1987. Published as *SIGPLAN Notices 22*(7), July 1987. Cited on page 72.

[Moss, 1989] J.E.B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. In Hull et al. [1989], pages 358–374. Proceedings of the Second International Workshop on Database Programming Languages (Salishan Lodge, Gleneden Beach, Oregon, June 1989). Cited on page 42.

[Moss, 1990] J.E.B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990. Cited on page 42.

[Moss, 1993] J.E.B. Moss. Private communication, March 1993. Cited on page 64.

[Mueller *et al.*, 1997] R. Mueller, F. Matthes, and J.W. Schmidt. Towards a unified model of untyped object stores: Experience with the Tycoon store protocol. In Atkinson [1997], chapter 2.2.4. To be published. Cited on pages 160 and 171.

[Munro, 1993] D.S. Munro. *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, University of St Andrews, 1993. Cited on pages 164 and 189.

[Mylopoulos *et al.*, 1980] J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong. A language facility for designing database intensive applications. *ACM Transactions on Database Systems*, 5(2):185–207, June 1980. Cited on pages 3 and 8.

[Nori *et al.*, 1981] K.V. Nori, U. Ammann, K. Jensen, H.H. Nageli, and C. Jacobi. Pascal-P implementation notes. In D.W. Barron, editor, *Pascal – The Language and its Implementation*, pages 125–170. Wiley, 1981. Cited on page 54.

[Norris and Pollock, 1994] C. Norris and L. Pollock. Register allocation over the program dependence graph. In PLDI [1994], pages 266–277. Published as *SIGPLAN Notices 29*(6), June 1994. Cited on pages 53 and 177.

[Odersky and Wadler, 1997] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997. Cited on page 176.

[OMG, 1991] The common object request broker: Architecture and specification. Published jointly by Object Management Group and X/Open, 1991. Cited on page 10.

[Ontologic Inc., 1991] Ontologic Inc. *ONTOS Reference Manual*. Ontologic Inc., Billerica, Massachusetts, USA, 1991. Cited on page 8.

[Özsu *et al.*, 1992] M.T. Özsu, U. Dayal, and P. Valduriez, editors. *Proceedings of the International Workshop on Distributed Object Management (Edmonton, Canada, 18th–21st August 1992)*. Morgan Kaufmann Publishers, 1992. Cited on pages 213 and 226.

[Patterson and Hennessy, 1990] D.A. Patterson and J. Hennessy. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990. Cited on pages 23, 123, and 173.

[Persistent Programming Research Group, 1985] Persistent Programming Research Group. PS-algol abstract machine manual. Technical Report PPRR-11-85, Universities of Glasgow and St Andrews, 1985. Cited on page 54.

[Persistent Programming Research Group, 1987] Persistent Programming Research Group. PS-algol reference manual — fourth edition. Technical Report PPRR-12-87, Universities of Glasgow and St Andrews, 1987. Cited on pages 30 and 41.

[Peyton-Jones *et al.*, 1993] S.L. Peyton-Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference (Keele, 1993)*, 1993. Cited on page 166.

[Peyton-Jones, 1987] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice-Hall International, Hemel Hempstead, UK, 1987. Cited on page 50.

[Peyton-Jones, 1992] S.L. Peyton-Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992. Cited on pages 26–27, 38, 50, 64, and 166.

[Peyton-Jones, 1994] S.L. Peyton-Jones. Private communication, February 1994. Cited on page 51.

[PLDI, 1994] *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (Orlando, Florida, June 20-24, 1994)*. Association for Computing Machinery, 1994. Published as *SIGPLAN Notices 29*(6), June 1994. Cited on pages 220 and 224.

[PLDI, 1996] *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (Philadelphia, PA, May 21-24, 1996)*. Association for Computing Machinery, 1996. Published as *SIGPLAN Notices 31*(5), May 1996. Cited on pages 211, 213, 218, 221, and 227.

[POPL, 1989] *Conference Record of the Sixteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL (Austin, Texas, January 1989)*. ACM Press, 1989. Cited on pages 211 and 217.

[Ramamoorthy *et al.*, 1984] C.V. Ramamoorthy, A. Prakash, W. Tsai, and Y. Usuda. Software engineering: Problems and perspectives. *IEEE Computer*, 17(10), October 1984. Cited on page 1.

[Rees and Clinger, 1986] Jonathan Rees and W. Clinger. The revised^3 report on the algorithmic language Scheme. AI Memo 848a, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1986. Cited on page 41.

[Richardson *et al.*, 1993] J.E. Richardson, M.J. Carey, and D.T. Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993. Cited on page 32.

[Richardson, 1989] J.E. Richardson. *E: A Persistent Systems Implementation Language*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, 1989. Cited on pages 8 and 32.

[Rosenberg and Keedy, 1987] J. Rosenberg and J.L. Keedy. Object management and addressing in the MONADS architecture. In Carrick and Cooper [1987], pages 114–133. Proceedings of the Second International Workshop on Persistent Object Systems (Appin, Scotland, 25th–28th August 1987). Cited on pages 18 and 171.

[Rosenberg and Keedy, 1990] J. Rosenberg and J.L. Keedy, editors. *Security and Persistence. Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information (Bremen, West Germany, 8–11 May 1990)*, Workshops in Computing. Springer-Verlag in collaboration with the British Computer Society, 1990. Cited on page 223.

[Rosenberg and Koch, 1989] J. Rosenberg and D. Koch, editors. *Persistent Object Stores*. Workshops in Computing. Springer-Verlag in collaboration with the British Computer Society, 1989. Proceedings of the Third International Workshop on Persistent Object Systems (10th–13th January 1989, Newcastle, New South Wales, Australia). Cited on pages 217, 220, and 228.

[Rumbaugh, 1991] J. Rumbaugh. *Object-oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991. Cited on page 4.

[Russell *et al.*, 1994] G. Russell, P. Shaw, and W.P. Cockshott. DAIS: An Object-Addressed Processor Cache. In Atkinson et al. [1994]. Proceedings of the Sixth International Workshop on Persistent Object Systems (Tarascon, Provence, France, 5th–9th September 1994). Cited on pages 18 and 171.

[Russell, 1995] Gordon Russell. *DOLPHIN: Persistent, Object-oriented and Networked*. PhD thesis, Department of Computer Science, University of Strathclyde, Glasgow, Scotland, 1995. Cited on page 18.

[Sabry and Felleisen, 1992] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming (San Francisco, CA, June 1992)*, pages 288–298. Association for Computing Machinery, 1992. Cited on page 51.

[Sajeev and Hurst, 1992] A.S.M. Sajeev and A.J. Hurst. Programming persistence in χ. *IEEE Computer*, pages 57–66, September 1992. Cited on page 55.

[Schaffert, 1992] C. Schaffert. CORBA: OMG's object request broker. In Özsu et al. [1992]. Cited on page 9.

[Schmidt, 1977] J.W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977. Cited on pages 3 and 8.

[Schwartz *et al.*, 1986] P. Schwartz, W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the STARBURST database system. In K.R. Dittrich and U. Dayal, editors, *Proceedings of the ACM/IEEE International Workshop on Object-Oriented Database Systems (23rd–26th September 1986, Pacific Grove, CA)*, pages 85–92. IEEE Computer Society Press, 1986. Cited on page 3.

[Serrano, 1994] M. Serrano. *Vers une Compilation Portable et Performant des Langages Fonctionnels*. PhD thesis, Université Pierre et Marie Curie (Paris VI), France, December 1994. Cited on page 27.

[Shao and Appel, 1994] Z. Shao and A.W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming (New York, June 1994)*, pages 150–161. Association for Computing Machinery, 1994. Cited on page 150.

[Shipman, 1981] D.W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981. Cited on pages 3 and 8.

[Shivers, 1988] O. Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (Atlanta, Georgia, June 22-24, 1988)*, pages 164–174. Association for Computing Machinery, 1988. Published as *SIGPLAN Notices 23*(7), July 1988. Cited on page 47.

[Shu *et al.*, 1977] N.C. Shu, B.C. Housel, R.W. Taylor, S.P. Ghosh, and V.Y. Lum. EXPRESS: A data EXtraction, Processing, and REStructuring System. *ACM Transactions on Database Systems*, 2(2):134–174, June 1977. Cited on page 21.

[Singhal *et al.*, 1992] V. Singhal, S.V. Kakkad, and P.R. Wilson. Texas: An efficient, portable persistent store. In A. Albano and R. Morrison, editors, *Persistent Object Systems: Implementation and Use*, Workshops in Computing, pages 11–33. Springer-Verlag in collaboration with the British Computer Society, 1992. Proceedings of the Fifth International Workshop on Persistent Object Systems (San Miniato, Italy, 1st–4th September 1992). Cited on pages 42, 164, and 189.

[Sites *et al.*, 1993] R. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993. Cited on page 175.

[Sjøberg *et al.*, 1993] D.I.K. Sjøberg, M.P. Atkinson, J.C. Lopes, and P.W. Trinder. Building an integrated persistent application. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Database Programming Languages (DBPL-4)*, Workshops in Computing, pages 359–375. Springer-Verlag in collaboration with the British Computer Society, 1993. Proceedings of the Fourth International Workshop on Database Programming Languages—Object Models and Languages (Manhattan, New York City, USA, 30th August–1st September 1993). Cited on page 21.

[Sjøberg *et al.*, 1994] D.I.K. Sjøberg, Q.I. Cutts, R.C. Welland, and M.P. Atkinson. Analysing persistent language applications. Technical Report FIDE/94/109, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994. Cited on page 133.

[Sjøberg, 1993] D.I.K. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems*. PhD thesis, University of Glasgow, July 1993. Cited on pages 7 and 126.

[Spivey, 1989] J.M. Spivey. *The Z Notation*. Prentice-Hall, 1989. Cited on page 4.

[Stallman, 1992] R. Stallman. Using and porting Gnu CC version 2.0. Documentation for the Gnu C compiler, Free Software Foundation, Cambridge, Mass, February 1992. Cited on pages 27, 63, and 167.

[Stallman, 1993] R. Stallman. Private communication (posted to the USENET newsgroup comp.compilers), January 1993. Cited on page 64.

[Steele Jr., 1978] G.L. Steele Jr. RABBIT: A compiler for scheme (a study in compiler optimization). Master's thesis, AI Laboratory, MIT, Cambridge, Mass., May 1978. Also as a Technical Report, AI-TR-474. Cited on pages 47, 139, and 166.

[Stemple *et al.*, 1992] D. Stemple, R.B. Stanton, T. Sheard, P.C. 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. Technical Report FIDE/92/49, ESPRIT Basic Research Action, Project Number 3070—FIDE, 1992. 29pp. Cited on pages 34 and 41.

[Stonebraker and Rowe, 1986] M. Stonebraker and L. Rowe. The design of Postgres. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, Washington D. C., May 1986. Cited on page 3.

[Strong *et al.*, 1958] J. Strong, A. Wegstein, J. Tritter, O. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines: a proposed solution. *Communications of the ACM*, 1(8):12–18, August 1958. Part 2: 1(9):9–15, September 1958. Report of the Share Ad-Hoc Committee on Universal Languages. Cited on page 18.

[Stroustrup, 1986] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1986. Cited on page 40.

[Tarditi *et al.*, 1990] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990. Cited on page 166.

[Tarditi *et al.*, 1992] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992. Cited on pages 27, 64, 73, and 177.

[Tarditi *et al.*, 1996] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In PLDI [1996], pages 181–192. Published as *SIGPLAN Notices 31*(5), May 1996. Cited on pages 39, 124, and 176.

[Taylor and Frank, 1976] R.W. Taylor and R.L. Frank. CODASYL data-base management systems. *ACM Computing Surveys*, 8(1):67–103, March 1976. Cited on page 2.

[Tennent, 1977] R.D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977. Cited on page 6.

[Teodosiu, 1991] D. Teodosiu. HARE: An optimizing portable compiler for scheme. *ACM SIGPLAN Notices*, 26(1):109–120, January 1991. Cited on page 47.

[Tsichritzis and Lochovsky, 1982] D.C. Tsichritzis and F.H. Lochovsky. *Data Models*. Prentice-Hall, 1982. Cited on page 3.

[Tsur and Zaniolo, 1986] S. Tsur and C. Zaniolo. LDL: a logic-based data language. In *Proceedings of the Twelfth International Conference on Very Large Data Bases (Kyoto, Japan, 24th–28th September 1986)*, 1986. Cited on page 8.

[Ullman, 1988] J.D. Ullman. *Principles of Data and Knowledge Bases*, volume 1. Computer Science Press, 1988. Cited on page 2.

[Ungar and Smith, 1991] D. Ungar and R. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):1–20, 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241. Cited on page 22.

[Vaughan, 1994] F. Vaughan. *Implementation of Distributed Orthogonal Persistence Using Virtual Memory*. PhD thesis, University of Adelaide, Faculty of Mathematical and Computer Sciences, December 1994. Cited on pages 31 and 171.

[Velez *et al.*, 1989] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager, an Overview. In P.M.G. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases (Amsterdam, Netherlands, 22nd–25th August 1989)*, 1989. Cited on page 42.

[Wai, 1989] F. Wai. Distributed PS-algol. In Rosenberg and Koch [1989], pages 126–140. Proceedings of the Third International Workshop on Persistent Object Systems (10th–13th January 1989, Newcastle, New South Wales, Australia). Cited on page 30.

[Wilson, 1992] P.R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management (St Malo, France, September 1992)*, number 637 in Lecture Notes in Computer Science. Springer-Verlag, 1992. Cited on page 42.

[Wirth, 1983] N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, Germany, second edition, 1983. Cited on page 33.

[Yourdon and Constantine, 1978] E. Yourdon and L. Constantine. *Structured Design*. Yourdon Press, Englewood Cliffs, New Jersey, 1978. Cited on page 4.

Abbreviations

ADT	Abstract Data Type
ANDF	Architecture-Neutral Distribution Format
API	Application Programming Interface
BURS	Bottom-Up Rewrite System
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CASE	Computer-Aided Software Engineering
CISC	Complex Instruction Set Computer
CORBA	Common Object Request Broker Architecture
CPS	Continuation-Passing Style
DBMS	Database Management System
DBPL	Database Programming Language
FAM	Functional Abstract Machine
GCC	GNU C compiler
GSA	Guarded Single-Assignment Forms
IDL	Interface Definition Language
LLPL	Low-Level Programming Language
ODBMS	Object-Oriented Database Management System
OMG	Object Management Group
ORB	Object Request Broker
OSF	Open Software Foundation
PAM	Persistent Abstract Machine
PAS	Persistent Application System
PCASE	Persistent CASE Machine
PDG	Program Dependence Graph
PHAM	Persistent Hierarchical Abstract Machine
PHOL	Persistent Higher-Order Reflective Language
POS	Persistent Object Store
PPL	Persistent Programming Language
PQM	P-Quest Machine
RISC	Reduced Instruction Set Computer
RTL	GNU Register Transfer Language
SPF	Scalable Persistent Foundation
SSA	Static Single Assignment Forms
STG	Spineless Tagless G-machine
TDF	TenDRA Distribution Format
TML	Tycoon Machine Language
UNCOL	UNiversal COmpiler-oriented Language

Index