Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming

by
Diomidis D. Spinellis

May 1993

A thesis submitted for the degree of
Doctor of Philosophy of the University of London
and for the Diploma of Membership of Imperial College

Department of Computing
Imperial College of Science, Technology and Medicine
University of London
London SW7 2BZ
United Kingdom

Abstract

The word paradigm, is used in computer science to talk about a family of notations, that share a common way for describing program implementations. Since each paradigm is well suited for solving only a range of problems, ideally a large system should be subdivided into components, each of which should be implemented in the most appropriate paradigm. Multiparadigm programming, allows the programmer to implement a system, in a number of different paradigms. The use of multiparadigm programming techniques, could lower implementation costs, and result in more reliable and efficient applications.

The difficulties that arise with multiparadigm programming can be separated into the areas of application development in multiple paradigms, design and implementation of multiparadigm environment, and generators for creating such environments.

We propose the use of object-oriented design techniques as a method for encapsulating programming paradigms within multiparadigm applications, and for abstracting common characteristics across paradigms. These techniques can be used to design practical multiparadigm environment generators, and implement multiparadigm programming environments supporting a wide variety of paradigms.

In order to demonstrate the validity of our approach, we describe the design and implementation of three prototypes: one in each problem area.

The *integrator* is a multiparadigm application dealing with the numeric and symbolic evaluation of integrals. The symbolic evaluation is based on the backtracking resolution mechanism offered by the logic programming paradigm, and the numeric evaluation, on the infinite streams implemented in the functional paradigm. Additionally, lexical analysis of the input expressions is described using regular expressions, and the expression grammar is described using a BNF syntax. Finally, expression simplification uses a term rewrite system, and graphing of functions is done by directly interacting with Unix tools.

Integrator is implemented in the *blueprint* multiparadigm programming environment, a prototype implementation of the six paradigms based on our object-oriented approach. Many different implementation techniques have been applied in order to demonstrate the wide applicability of our approach. Some of the paradigms are implemented as compilers (using existing implementations where possible), and others are implemented as interpreters.

Finally, the implementation of *blueprint* is based on the MPSS multiparadigm environment generator. This, allows the description of paradigms as object classes using paradigm description files. Additionally, it provides a multiparadigm link editor, and support for incorporating existing compilers into multiparadigm programming environments.

Acknowledgements

A great number of people have helped me in bringing this research to fruition. I would like to thank:

My parents, who have helped me in more ways than I could have thought were possible, Susan Eisenbach and Sophia Drossopoulou for supervising the project, hours of discussions, patient and careful reading of the thesis and insightful comments, Chris Hankin for his general guidance, Paul Kelly and Alexander Dimakis for interesting discussions, Mireille Ducassé for her valuable comments on drafts of the thesis, Edward Janies for his motivating thesis talk, the members of the Distributed Software Engineering Group of Imperial College and in particular Naranker Dulay, Anthony Finkelstein, Jeff Kramer, Jeff Magee, Bashar Nuseibeh, and Morris Sloman for their comments, my fellow PhD students Hiu Fai Chau, Kostis Dryllerakis, Konstantinos Moutsopoulos, Ian Mackie, and David Man for their support during the long hours of work and finally, various contributors of the project GNU, 4.4 BSD and Larry Wall whose tools I have used.

Last, but not least, the British Science and Engineering Research Council (SERC) which provided financial support during the whole period of my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we will describe our motivation for undertaking this research, by identifying our work context, outlining the previous work, and presenting our goals. We will finish with an outline of the whole thesis, in order to provide a road map for our reader.

1.1 Work Context

The word paradigm, is used in computer science to talk about a family of notations, that share a common way for describing program implementations. Thus, we talk about the *imperative*, *functional*, and *logic programming* paradigms, to denote programming notations based respectively, on explicit control, the theory of functions, and Horn-clause logic.

Since every paradigm is best suited for solving different problems, ideally, each system part should be implemented in the most appropriate paradigm. Multiparadigm programming, allows the programmer to write the implementation of a system, in a number of different paradigms. The use of multiparadigm programming techniques, could lower implementation costs, and result in more reliable and efficient applications.

The area of multiparadigm programming is relatively new. A number of multiparadigm languages have been documented in the literature, but most offer only a limited set of programming paradigms. In addition to the multiparadigm languages, there are some approaches based on system frameworks, for combining arbitrary paradigms.

1.2 Goals

Our approach is based on the idea of a multiparadigm programming system framework, rather than a specific multiparadigm language. We distinguish the problems of multiparadigm programming into the areas of application development in multiple paradigms, design and implementation of multiparadigm environments, and generators for creating such environments. This separation, allows the methodical study of the different issues and solutions that exist on each level. We will demonstrate how object-oriented techniques, can be used to organise multiparadigm programming environments in a simple, yet powerful way. We will advance the concept of a *call gate*, to

describe the inter-operation of arbitrary programming paradigms, with implementation overhead linear to the number of paradigms supported. Finally, in order to support our claims, for each of the subject areas, we will detail the design and implementation of a system, built according to our approach.

1.3 Thesis Outline

In the **second chapter** we examine existing multiparadigm languages, and programming frameworks. We present multiparadigm languages categorised according to the paradigms they support. After a description of multiparadigm frameworks, we analyse and evaluate all the available solutions. Related work concerning implementation issues has been separately summarised in sections 4.4.4, 4.4.5, 4.4.6, 4.4.7, 4.4.8, and 4.5.4.

Our approach to multiparadigm programming is detailed in the **third chapter**. There, we define the problem areas, identify multiparadigm system requirements, and present our solution based on object-oriented principles. We then, describe how our approach applies to, application development in multiple paradigms, design and implementation of multiparadigm environments, and generators for creating such environments.

Moving from an abstract description of our approach to a concrete realisation of it, in the **fourth chapter** we design a multiparadigm programming environment generator, MPSS, providing tools for implementing multiparadigm environments, a multiparadigm programming environment, *blueprint*, which supports programming in six different programming paradigms, and an application, the *integrator*, utilising the paradigms provided by *blueprint*.

The **fifth chapter** contains the implementation details of the systems designed in the previous chapter. Thus, we present the implementation, of the tools comprising MPSS, each paradigm of the *blueprint* environment, and the *integrator* application.

Having defined our approach, designed a prototype realisation of it, and described its implementation, in the **sixth chapter** we provide a critical evaluation of our work. We evaluate our system contributions to multiparadigm research in the problem areas defined in the previous chapters, our system as a programming language, MPSS as a process support environment, and finish with the evaluation of *blueprint* as a programming environment.

Multiparadigm programming is a relatively new computer science research area. Any research in it, is bound to raise many new questions; this thesis is no exception. In the final, **seventh chapter**, we will summarise various possible enhancements to our work, and outline possible new applications based on it.

At the end of the thesis we have provided a glossary (page 179) to guide the reader in the terminology we have introduced, and an index (page 205) to provide “random access” within the thesis structure. Detailed implementation notes on the realised systems can be found in appendix A.

Chapter 2

Related Work: Multiparadigm Programming

In this chapter we begin our exploration of the area of multiparadigm programming by examining the work that has been done up to now. Research in the area of multiparadigm programming can be divided in three different areas:

1. Programming paradigms: work in this area examines the notion of programming paradigms, their relationship to language design, and their effect on the software production environment.

2. Multiparadigm languages: during the literature research for this thesis we found more than 90 languages supporting more than one programming paradigm. Although not all languages were explicitly directed towards multiparadigm programming *per se* we believe that there were lessons to be learned from their collective study.

3. Multiparadigm programming frameworks: some researchers have come up with suitable abstractions and systems that support multiparadigm programming in general without targeting specific programming paradigms. Again in this case at least one of the systems covered was not the result of explicit multiparadigm research effort although it supports programming in multiple programming paradigms.

Our approach is geared towards producing a multiparadigm design methodology, a prototype system based on that methodology, and multiparadigm programming environment built upon that system. Therefore, the two last research areas are directly relevant to our research. We also examine the first area, because we believe that the notion of a programming paradigm is central to the theme of this thesis. Mixed language programming environments which only deal with languages based on a single paradigm (such as [Ein84]), and generic concurrent, distributed, heterogeneous systems, and module interface languages [Tic92] that could potentialy be used as multiparadigm frameworks (such as [Bea92]) are not examined. A thorough survey of distributed system languages can be found in [BST89], and of concurrent logic programming languages in [Sha89]; a set of articles on concurrent object-oriented programming can be found in [CAC93], and a survey of specific concurrent Smalltalk implementations in

[aYCK93]. Implementations of various paradigms on parallel computers are examined in [Tre90]. Furthermore, we will not examine systems that address the multiple programming paradigms of the program development cycle from specification to implementation, such as [MR89] supporting structured program graphs to provide different views of the program according to the software-process evolution stage, [FKG90] supporting multiple system viewpoints during the design phase of the system, and [Old92] defining the interface between the specification, implementation, and machine languages.

2.1 Programming Paradigms

The word paradigm (from the Greek word $\pi\alpha\rho\acute{\alpha}\delta\epsilon\iota\gamma\mu\alpha$ which means example) is commonly used to refer to a category of entities sharing a common characteristic. Wittgenstein [Wit60, p. 48] defines a paradigm by examining all the activities we call games. Among those activities there are some which possess some characteristic similarities equivalent to those exhibited by the members of a family. The *notion* "game" can only be defined by creating a list of all these typical cases that we call games, without being able to prescribe specific conditions for labelling an activity as a "game." In other words, we define games by listing some exemplar cases. In order to define an activity as a "game" it must share some common, but unspecified characteristics with those exhibited by the other members of the family; therefore the notion is only vaguely defined.

Kuhn used the notion of a *paradigm* in the scientific process by defining it as the scientist's view of the world and the structure of their assumptions and theories[1] [Gem87, pp. 301–303]. According to Kuhn [Kuh70, p. 10] a paradigm has a wider meaning than that of a scientific theory; it encompasses "law, theory, application and instrumentation together." Although Kuhn's examples are drawn from the history of physical science, his paradigm notion has been extended to a number of sciences [Gre80, EH80]. Paradigms are the basis of *normal science* which is related to all the activities of the established scientific tradition. Therefore, the formation of a paradigm is a sign of maturity for a given science.

In [Weg89] it is suggested that as programming languages mature, attention is turning from languages to paradigms. In trying to define the notion of *programming* paradigm[2] the most common definition found is that of "a style of programming expressing the programmer's intent" [Bob84] or the similar "model or approach in solving a problem" [Shr86], "approach to solving programming problems" [Fri91, p. 188], and "way of thinking about computer systems" [Zav89]. A more general definition is given in [Weg90, p. 21], where paradigms are rules for determining classes of languages according to some testable conditions. These conditions can be based on a number of abstraction criteria, such as the structure of a program or its state, or the development methodology. Furthermore, according to the same author, paradigms can be defined in four different ways:

1. extensional: by the languages that belong to that paradigm,

[1] Our apologies to the many people who are offended by Kuhn's misuse of the word paradigm.

[2] A distincton between (verbal) language-motivated and task motivated linguistic paradigms is made in [And90, p. 331].

2. intentional: by the properties that determine whether a language belongs to it or not,

3. historic: by the paradigm's evolution, and

4. exemplar: by specific example languages.

The distinction between the *extensional* and the *intentional* definition is also made by [Shr86]. [Fri91] divides the programming paradigms into process centered (imperative, dataflow, functional), and *data centered* (constraint, rule, object, database), listing them in a tree hierarchy. In sections 3.5.1, and 3.3.11 we will use this tree-based approach to structure our multiparadigm system. Finally, in a number of articles paradigms are defined by giving examples of differing ones.

In articles dealing with programming paradigms different authors have differing views on what constitutes a programming paradigm. Thus, paradigm examples given are: the block structure, procedural or imperative paradigm [Zav89, Weg90, Shr86, Bey86, WK89, Fri91], the functional, logic programming, and object-based or object-oriented paradigms [Zav89, Weg90, Bob84, Shr86, Bey86, WK89, Fri91], the database, and the concurrent or distributed paradigm [Weg90], the rule-based paradigm [Zav89, Fri91], the visual paradigm [Shr86], and the constraint, and spreadsheet paradigms[3] [Bey86, Fri91]. At this point we need to point out that often the denotation of a programming *paradigm* is confused with the denotation of a programming *methodology* (e.g. structured programming, data hiding, object-oriented programming). A programming paradigm can *support* a programming methodology, but it is a distinct entity. Most programming methodologies can be followed irrespective of the programming paradigms supported by the underlying language (e.g. structured programming in Fortran) [Str88, p. 11].

Paradigms are also distinguished and characterised according to their costs, the leverage they provide, and environment for using them [Bob84]. The costs associated with a paradigm are those of learning to write programs in it, debugging them, changing them, and running them; in short all the aspects of the software life-cycle. The leverage provided by a paradigm is associated with the notion of elision: the facts that need *not* be stated. Elision in the context of programming paradigms is better defined in [SBK86]:

> "Elision is the ability to state concisely and without redundancy what is intended. This is a hallmark of appropriate language support for a paradigm. By eliminating verbiage the programmer can focus on the essentials having both less opportunity for mistakes and more easily understood programs."

Finally, the environment associated with programming paradigms is characterised according to the support it provides for detecting errors and tracing the program, its analysis and performance tuning tools, and the tools available for generating user interfaces.

[3] Given paradigm status in [Cas92, BP93].

2.2 Multiparadigm Languages

In this section we present a number of languages supporting more than one programming paradigm. Other such surveys can be found in [Hai86b, Hai86a], and [Pla89, pp. 13–17]. We attempt to order the languages into different categories according to the paradigms they support. In each category we provide a list of the languages and the way they have been implemented, their main features, a brief description of each language and, finally, try to draw some conclusions from the languages presented.

We divide the languages into nine categories by choosing as categories those groups where more than three languages can be placed. Languages that can not be placed in any of the nine categories are examined separately at the end. Those languages use seldom-mentioned or specialised paradigms, based on sets, generators, BNF grammars, and regular expressions. We use the following broad paradigm definitions in order to divide the languages:

Imperative: Based on change of state, assignment, direct control of the execution strategy.

Functional: Based on simplification by function reduction. Many of the languages that fit in this paradigm provide higher order functions, lazy evaluation, and referential transparency. Functional languages often offer pattern matching, and polymorphic type systems.

Logic: Based on solving by SLD resolution. The concepts of relations, unification, backtracking and the logical variable are also closely related.

Object-Oriented: Based around the provision of some of the following: objects with encapsulated state and methods, inheritance mechanisms, classes, messages.

Distributed: Supporting the execution on more than one processing unit.

Constraint: Containing a solver to satisfy solutions of problems within a specific domain (usually equations).

We are certain most people will disagree with at least one of the above definitions. They may rest assured that they are not alone. The literature is full of specific paradigm definitions and no two of them are identical. However, the above definitions are not critical to our examination, but just rough guides used to group languages together into the various categories. We have attempted to characterise the languages using a set of objective criteria. Thus, all languages will be examined using a set of characteristics which provide much finer control over the language features. The characteristics are divided into two categories: general language features and execution strategies. They are listed in table 2.1 and will be used in all our language evaluations. Some languages are listed as supporting two conflicting or opposite characteristics; this is the case when a language has one subset supporting a given characteristic and another subset supporting another. The execution strategy is denoted as (*) in the cases where a language uses a strategy that is not included in table 2.1.

Some of the characteristics are important by their absence as much as by their presence (e.g. deterministic execution vs. non-determinism). In that case we list the

Name	Explanation	Summary
DT	Determinism	Same arguments produce same results
$\cup$	Unification	Logical variables
f	Functions	Functions can be composed
λ	Lambda expressions	Anonymous functions
R	Relations	Support for expressing relations
RT	Referential Transparency	Expression has single value
BR	Branches	Control of evaluation order
MD	Multiple Data	More than one data item is being operated upon
O	Objects	Entities responding to messages
$\parallel$	Parallelism	Support for parallel processes
IN	Inheritance	Implicit procedure use through a class tree hierarchy
MI	Multiple Inheritance	Implicit procedure use through object DAG hierarchy
Execution Strategies		
(X)	Explicit	Explicit
(FR)	Function Reduction	Solution by function reduction
(SLD)	SLD Resolution	Solution by resolution

Table 2.1: Multiparadigm Language Features

Paradigm	Characteristics
Functional	RT DT f λ (FR)
(Applicative)	RT DT f (FR)
Logic-programming	$\cup$ R RT $\overline{\text{DT}}$ (SLD)
Imperative	BR $\overline{\text{RT}}$ (X)
Declarative	$\overline{\text{BR}}$ (*)
Object-Oriented	IN O

Table 2.2: Common paradigms

Characteristic	Constituent Characteristics
Non-Determinism	RT $\overline{\text{DT}}$
Assignment	$\overline{\text{RT}}$
Superscalar	$\parallel$ $\overline{\text{BR}}$

Table 2.3: Common composite characteristics

characteristic with a horizontal bar over it. In order to give a rough idea of how these higher-granularity language characterisation entities relate to the paradigms that are commonly discussed we list in table 2.2 some common paradigms — following the most widely accepted paradigm definition — together with their composite characteristics. There are also some characteristics that are often used to characterise languages that are *composite* in our characterisation schema i.e. they are defined by more than one of our primary characteristics. Table 2.3 contains the definitions of some commonly used composite characteristics that are defined in this way.

2.2.1 Combinations of Functional and Logic Paradigms

The logic and functional programming paradigms are both based on mathematical foundations: logic programming is based on Horn-clause logic, and functional programming on equational logic (substitution of equals for equals) [GM86a]. Thus, there is the tempting possibility to perform the multiparadigm integration by integrating the underlying logics [GM87, p. 470], [Bol86]. The relationship between logic and functional languages is surveyed in [BL86, Red86, DFP86, BDL82, DL86, JLM86, GLDD90]. A comparison between Prolog and Lisp can be found in [WP77]. Integration of the relational paradigm [Cod70] (supported by logic programming) with other paradigms is examined in [Kor86b]. We found 24 languages that combine the functional and logic programming paradigms. Their implementations are summarised in table 2.4 and their characteristics in table 2.5. In the following paragraphs we list the most important features of each language.

ALF [Han90a, Han91] Defined in terms of Horn-clauses and equality. Any functional expression can be used in a goal literal and arbitrary predicates can occur in conditions of equations. The compilation process is based on a modified version of the Warren abstract-machine [War83].

ALICE [CST87] Based on the concept of mutual reflection [4]. The two systems are built as procedural introspective systems on top of Lisp-3 and then made mutually reflective.

Applog [Coh86] A functional programming interpreter implemented on top of Prolog. The interpreter provides facilities for defining and evaluating functions and an interface for calling Prolog and converting between lists and terms. It is possible to call Prolog from Applog using the `goal` function and Applog from Prolog using the `eval` predicate.

Bon87 [Bon87] Meta interpreter built on top of Scheme, modifying its semantics to evaluate both functions and predicates using a common environment.

EqL, E [JSG86, JS86] Support Horn-clause logic and first-order functional programming by means of equation solving rewrite rules. Their operation semantics are based on object refinement equation solving.

FGL+LV [Lin85] Functional graph reduction language, with unification. No backtracking is offered, so the effects of the unification can not be undone.

[4] Reflection is the ability to inspect and influence the language's interpreter from within the language.

FPL [BDL82] Lazy functional language with unification and a relational syntax for expressing streams.

Fresh [Smo86] A higher order functional language is extended by adding unification and non-determinism. Operational semantics are defined using a structural approach.

Funlog [SY86] Computational model supporting functional and logic programming. The reduction strategy is based on pattern-driven lazy reduction. The unification method used is semantic unification, or unification based on equality.

HASL [Abr86] A purely functional language, that supports one way unification. Of the two terms to be unified, only one can contain non-ground variables.

HCPRVR [Che80] This Horn-clause theorem prover is a Prolog interpreter implemented on top of Lisp. A way is provided to call Lisp functions from within Prolog.

HHT82 [HHT82] Contains all Prolog features and, in addition, first order deterministic functions and lazy evaluation. The logical evaluation system is natural deduction.

Han90 [Han90b] An explicitly typed polymorphic integration of functional and logic languages. Its operational semantics are based on resolution for predicates, and narrowing for functions. Higher order programming techniques can be used.

Id Nouveau [JP91] A first order functional language that contains logical arrays. These can be declared inside function blocks, and unification can be performed between them. *Id Nouveau* programs are compiled into a subset called *Cid*, for which full operational semantics are provided.

LML [BMPT90] A higher-order functional language where logic objects are represented as a special data-type called *theories*. These are manipulated using the functional constructs, but can be queried to produce multiple values.

LOGLISP [RS82] Extension to Lisp, using Lisp syntax, allowing the expression of non-deterministic goals and unification. A special form of resolution called *LUSH* is used. Lisp interface functions are provided.

Leaf [BBLM86, BBLM84] Provides a logic programming language with equational reasoning. The inference rule used, corresponds to call-by-name semantics. A deterministic, first-order functional subset of the logic language is provided, by substituting relations with tuple-valued functions, and unification with pattern matching. (In [BBLM86] the logic component is called ‘declarative’ and the functional, ‘procedural’).

Nar85 [Nar85] Presents a technique for the lazy evaluation of functions from within Prolog. This is achieved by defining these functions as rules of a `reduce` predicate, which performs the function reduction.

Qute [SS86a] Functional language supporting full unification and AND parallelism. The evaluation of Qute programs enjoys the Church-Rosser property. Parallel running processes communicate through shared variables.

SProlog [Smo84] Prolog with additional constructs to make the writing of deterministic functions possible.

SchemeLog [Bon91] The language is based on Scheme with the addition of logical variables and clause expressions. Its operational semantics are defined in terms of an abstract-machine interpreter borrowing instructions from the P-Machine, the SECD-Machine and the Warren Abstract Machine.

TABLOG [MMW84, MMW86] Extension to Prolog by allowing the definition of equations. A deductive-tableau proof system is used in place of Prolog's resolution proof system.

Term Desc. [Nak85] Term description provides a functional extension to Prolog by allowing terms followed by a description of their value. These descriptors are Prolog predicates, that are executed when such terms are unified. Lazy unification allows the expression of infinite data structures.

YS86 [YS86] Logic programs are represented as logic equations and are therefore solved by equational methods. A 'deletion upon unification' rule is used to handle unification. Lazy narrowing is the rewrite method used.

Many of the systems examined, integrated the two logics using the operational semantics of *narrowing* proposed in [Red85], instead of function reduction and SLD-resolution. The research on the integration of the two paradigms started relatively early [Che80]; we think that the design and implementation of such systems proved that multiparadigm programming was possible, and opened the road to broader multiparadigm programming languages.

2.2.2 Combinations of Imperative and Logic Paradigms

The classic equation [Kow79]

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

on which logic programming languages are based, implies that *control* is part of the program solution. Furthermore, in contrast to most functional programming languages, due to the search strategy used by the majority of implementations, the control of the execution order is an important aspect of many "logic" programs. For example, in the case of the often cited program `a :- a. a.` the goal `a` will not succeed, although it will if set for the logically equivalent one, `a. a :- a.`. It is thus clear, that control of the evaluation strategy can be an important aspect of practical logic programming languages. Most logic programming languages offer a way of manipulating the evaluation order by means of the `cut` predicate. A more drastic way to achieve this result is to couple a logic language with the traditional paradigm for expressing control: the imperative paradigm.

Name	References	Implementation
ALF	[Han90a, Han91]	WAM extension
ALICE	[CST87]	Meta-interpreter on top of 3-Lisp
Applog	[Coh86]	Interpreter written in Prolog
Bon87	[Bon87]	Meta-interpreter on Scheme
EqL, E	[JSG86, JS86]	Language
FGL+LV	[Lin85]	Extension to the graph reduction language FGL
FPL	[BDL82]	Extension to TEL functional language
Fresh	[Smo86]	Extensions to functional
Funlog	[SY86]	Interpreter implemented in Prolog
HASL	[Abr86]	Implemented in C-Prolog
HCPRVR	[Che80]	Implemented on top of Lisp
HHT82	[HHT82]	Extension to Prolog
Han90	[Han90b]	Theoretical framework
Id Nouveau	[JP91]	Operational Semantics
LML	[BMPT90]	Extension to functional
LOGLISP	[RS82]	Extension to Lisp
Leaf	[BBLM86, BBLM84]	Plan for hardware implementation
Nar85	[Nar85]	Technique
Qute	[SS86a]	Implemented in Prolog as a translator to Prolog
SProlog	[Smo84]	Implemented on top of Prolog
SchemeLog	[Bon91]	Meta-interpreter on Scheme
TABLOG	[MMW84, MMW86]	Language implemented in Lisp
Term Desc.	[Nak85]	Prolog extension
YS86	[YS86]	Semantic framework

Table 2.4: Implementations combining the functional and logic paradigms

Name	Characteristics										Control
	BR	DT	f	λ	$\|\|$	R	RT	$\cup$	$\overline{\text{DT}}$	$\overline{\text{RT}}$	
ALF	√	√				√	√	√	√		SLD, narrowing
ALICE		√	√	√		√	√	√	√		SLD, FR
Applog		√	√	√		√	√	√	√		SLD, FR
Bon87			√	√		√		√	√		FR, SLD
EqL, E						√	√	√	√		*
FGL+LV		√	√	√			√	√			FR
FPL		√				√		√			*
Fresh		√	√	√			√	√	√		FR, SLD
Funlog		√	√	√		√	√	√	√		FR
HASL		√	√	√			√	√			FR
HCPRVR	√	√	√	√		√	√	√	√		SLD
HHT82	√	√				√	√	√	√		*
Han90	√	√	√	√		√	√	√	√		SLD, narrowing
Id Nouveau	√	√	√				√	√		√	*
LML	√	√	√			√	√	√	√		FR
LOGLISP		√	√	√			√	√	√		FR
Leaf		√				√	√	√	√		*
Nar85	√					√	√	√	√		SLD, FR
Qute		√	√	√	√		√	√			FR
SProlog	√	√				√	√	√	√		SLD
SchemeLog			√	√		√		√	√		*
TABLOG	√					√	√	√	√		*
Term Desc.	√		√			√	√	√	√		SLD
YS86		√				√	√	√	√		*

Table 2.5: Characteristics of functional and logic paradigm combinations

It is not only the logic paradigm that benefits from the synergy with the imperative one. Traditional imperative languages can become more powerful, by tapping into the power of the logic variable, backtracking, and unification offered by the logic programming paradigm. In this section we will examine languages built upon such combinations.

The relationship between imperative and logic languages is examined in [DAT91, Dra87, HKW85, Fle90]. Integration of the relational paradigm [Cod70] with other paradigms is examined in [Kor86b]. An interesting approach, based on flow diagrams, towards the paradigm integration can be found in [DAT91]. We found ten languages that combine the imperative and logic programming paradigms. Their implementations are summarised in table 2.6 and their characteristics in table 2.7. In the following paragraphs we list the most important features of each language.

2.PAK [Mel75] Block structured language offering user-defined pattern matching and backtracking.

C with Rule Extensions [MS90] Based on the C programming language [KR78] with an extended syntax, a richer set of data types, a flexible input/output system and a forward chaining [Ric83, p. 56] execution strategy.

Leda [Bud91] Language with syntax similar to that of Pascal, with an additional code abstraction facility, the *relation*. The data-space for all entities contains the *undefined* value. Relations are coded as Prolog rules, and allow backtracking.

Logicon [LC86] A Prolog interpreter implemented as an Icon [GG83] procedure. The interpreter supports all Icon data types. It is possible to insert Prolog code into an Icon program as a generator and to use Icon code in a Prolog predicate. Both are realised using procedures implemented in Icon.

Modula-Prolog [Mul86] The facilities of a Prolog interpreter are provided to a Modula-2 programmer through a library. Predicates, that can be called from the Prolog interpreter, are written in Modula-2. The library includes term handling procedures.

PIC [BK90] The logic and imperative programming paradigms are combined, by translating Prolog predicates into C functions, and providing interfaces between the two languages. Each Prolog predicate is translated into a single equivalent C function. The arguments of the C function are the copy and trail stacks, the argument list, the nearest choice point, and a continuation stack. This implementation is similar to the Warren abstract-machine [War83]. A limited class of C functions can be called from Prolog. Prolog can be called from C, using a special tool to generate the interface code.

Paslog [Rad90] An extension to the Pascal [JW75] programming language providing logic programming capabilities. This is achieved by additional control structures, enhanced operators and statements, interactive features, and a suitable programming methodology. Non-determinism is provided by the special `split` statement.

Name	References	Implementation
2.PAK	[Mel75]	Language
C with Rule Extensions	[MS90]	Extension of C, preprocessor
Leda	[Bud91]	Language
Logicon	[LC86]	Prolog interpreter in Icon
Modula-Prolog	[Mul86]	Run-time library for Modula-2
PIC	[BK90]	Translator of Prolog to readable C
Paslog	[Rad90]	Language, extension of Pascal
Planlog	[Fro87]	Theoretical framework
Predicate Logic in APL	[EGM89]	Implemented on top of APL
Strand	[FT90]	Language

Table 2.6: Implementations combining the imperative and logic paradigms

Planlog [Fro87] A theoretical framework for cleanly incorporating procedural execution into a logical environment. Implemented by transforming Horn-clauses into special predicate logic rules. Deriving a goal using an initial situation and the rules via a linear logic proof [Gab90], generates a *plan* for that proof. In a pure logical world the plan's existence represents the state of the world after its execution. Adding built-in non-pure predicates i.e. predicates that have side effects or predicates that reflect external events, imposes the requirement of the execution of a plan in order to realize the effects of those predicates. Procedural aspects of an execution can be expressed in the form of pre-fabricated plans.

Predicate Logic in APL [EGM89] Predicate logic in APL is a set of functions that allow the representation of knowledge bases, and the execution of queries on them using the APL data structures, functions, and interface.

Strand [FT90] The language provides a "foreign language interface." This consists of a set of functions and a suitable methodology [SRA89, pp. 9.1–9.48] for linking C code into Strand programs. Facilities are provided for testing and building arguments on the C side, and for providing argument directionality information on the Strand side.

In the following sections we will attempt to analyse the approaches towards the integration of the two paradigms, by examining the the handling of unification, nondeterminism, and additional features found in those languages.

Handling of Unification

Unification is a central operation in logic programming because of the use of resolution for theorem proving [Kni89, p. 101]. We expected languages catering for the logic programming paradigm to provide some form of unification. Surprisingly not all languages provided it. As summarised in table 2.8 unification is offered in various degrees. We can distinguish the following levels of support for unification:

Name	Characteristics							Control
	BR	$\|\|$	R	RT	$\cup$	$\overline{\text{DT}}$	$\overline{\text{RT}}$	
2.PAK	√				√	√	√	*
C with Rule Extensions	√		√	√	√	√	√	*, X
Leda	√		√	√	√	√	√	SLD, X
Logicon	√		√	√	√	√	√	X, SLD
Modula-Prolog	√		√	√	√	√	√	SLD, X
PIC	√		√	√	√	√	√	SLD, X
Paslog	√		√	√	√	√	√	SLD, X
Planlog	√		√	√	√	√	√	SLD
Predicate Logic in APL	√		√	√	√	√	√	SLD, X
Strand	√	√	√	√	√	√	√	SLD

Table 2.7: Characteristics of imperative and logic paradigm combinations

Language	Unification	Backtracking	I/O extensions
Modula-Prolog	√	√	√
Planlog	√	√	
Predicates in APL	√	√	
Paslog	Explicit	√	√
C with Rules	?	√	√
PIC	√	√	
Leda	1 level	√	

Table 2.8: Language characteristics

Full support (*M-Prolog*, *Planlog*, *P-APL*, *PIC*). Unification of data structures of arbitrary depth is provided in the same way as in conventional logic programming language implementations.

Single level (*Leda*). This form of unification only unifies simple variables and does not work for recursive or nested data structures. When a variable with an undefined value is unified with a variable with a defined value both variables end up with the same, defined, value. More complicated patterns are not supported. For example `unify(X, john)` will result to `X = John`, but `unify(loves(john, mary), loves(john, X))` will fail.

Explicit unification (*Paslog*). Under this approach unification and its implementation strategy is under the control of the programmer. No support for unification is provided by the language, but the programmer can write his own function that will unify two arbitrary data structures. This is the most flexible, but least expressive method.

Handling of Non-determinism

Non-determinism, the existence of a branched computation tree for a given program [Hog84, p. 50], can be supported in two different ways:

Implicit generation of choice points (*M-Prolog*, *Planlog*, *P-APL*, *Paslog*, *C-Rule*, *PIC*, *Leda*). Choice points, i.e. points where more than one execution path can be followed, are implicitly generated by the use of non-deterministic constructs such as the existence of a number of clauses matching a given goal.

Explicit generation of choice points (*Leda*). Non-determinism can also be explicitly controlled by the programmer. At a given point of the program execution the programmer can direct the generation of a choice point. If at a latter point the program backtracks, execution will be resumed at that choice point.

Additional Features

In this section we outline various notable features of the languages examined.

Input/output support (*M-Prolog*, *Paslog*, *C-Rule*). A number of languages with an imperative base language enhanced their input/output system to parse and print composite and recursive data structures. This capability is common in most Prolog implementations and useful for rapid prototyping and debugging. In more solid applications a separate parser and pretty-printer are usually implemented.

Dynamic compilation (*M-Prolog*, *P-APL*, *Paslog*). The majority of Prolog implementations allow for the dynamic modification of the knowledge base by means of the `assert` and `retract` predicates and, sometimes, the `dynamic` keyword. Some of the languages examined also provide this feature.

2.2.3 Combinations of Functional and Imperative Paradigms

Some of the drawbacks of pure functional languages have to do with their inability to deal with *state* changes. State, represented by direct changes on a tangible medium, is an important part of a problem solution process: the solution of a problem needs at some point to be represented in a tangible way, in order be used. State modifications are used for two main reasons:

1. Input/output i.e. the control and sensing of the outside world via display screens, keyboards printers, machine actuators, sensors, etc.

2. Storage of data on various forms of memory.

Both are two facets of the same coin, as one can be performed in terms of the other. There are two things that can be done with state information: accessing it and modifying it. If a functional language dealt with state information, then either the *Church-Rosser* theorem [FH88, p. 121], on which many functional language properties are based, would be violated, or the way state information was dealt with would be useless. The reasons are the following:

Access: State information, by its nature, can change values between two accesses of it. Reducing an expression using such state information in two different ways would produce two different results, hence violating the *Church-Rosser* theorem.

Modification: By the *Church-Rosser* theorem, two different terminating sequences of reductions of the same expression to normal form will produce the same result (up to alphabetic equivalence). If an expression modified external state, the order of these modifications would be not be defined. Interfaces to the real world often impose a single correct sequential ordering (e.g. characters must be transmitted to a printer in the order in which they are to be printed) and therefore modification of state by a pure functional language can not be performed in a generally useful way.

Imperative languages are based on state changes; for this reason the integration of the functional and logic paradigms seems to be able to provide a way for producing useful systems utilising functional programming technology. This was advocated by Strachey in the discussion following [Lan66, p. 165]. The integration of imperative constructs within a functional framework can be realised in two different ways [WW88]:

- The convenient method, as implemented by ML [Mil85, MH88, MHMA89, AM87] and Scheme [REA$^+$86] is to introduce primitive operations with side effects.
- The pure method, as implemented in LispKit Lisp [HJJ83], and SASL [Tur79] exploits the capabilities of lazy evaluation to treat I/O and data-base handling primitives as specific instances of stream-valued stream functions. Applicative state transition systems are described in [Bac78a, pp. 635–638].

We found nine languages that combine the functional and imperative programming paradigms. Their implementations are summarised in table 2.9 and their characteristics in table 2.10. In the following paragraphs we list the most important features of each language.

FL [WW88] Side effects are represented by modifying a special object that all programs map, the *history*. Special primitives are provided to modify it.

Fluent [GL86] Class of functional languages that have sub-languages with side-effects. Sub-language invariants can be checked by static checking called *effect checking*. Four sub-languages are defined according to their ability to allocate, read, and write to memory locations.

Gedanken [Rey70] Functional language based exactly on the *completeness* (all values are first class citizens) and *reference* concepts, excluding all other features. Assignments are possible by using *references*: objects which point to values.

Lucid [FL86, AW76] Family of dataflow-based languages, each based on a formal algebra. The properties of each family member are independently determined by an algebra defined as a set together with operations over its elements.

Name	References	Implementation
FL	[WW88]	Language
Fluent	[GL86]	Language class
Gedanken	[Rey70]	Gedanken-experiment, Interpreter on Lisp
Lucid	[FL86, AW76]	Language
ML	[Mil85, MHMA89, Har86]	Language
Nial	[JGM86]	Interpreter
Scheme	[REA$^+$86]	Language
Spreadsheet	[Cas92]	Application
Viron	[Pra83]	Language

Table 2.9: Implementations combining the functional and imperative paradigms

ML [Mil85, MHMA89, Har86] Functional language with imperative features. These are: primitives used for input/output, exceptions allowing to escape from a control structure to the point where an exception is trapped, assignment by means of references (pointers to the heap), a sequencing operator, and an iteration construct.

Nial [JGM86] Array based language with assignment and explicit sequencing control. Extensions for logic programming and a methodology for object-oriented programming are provided for pedagogical purposes.

Scheme [REA$^+$86] Functional language providing static scoping and primitive operations with side effects. The call with current continuation primitive allows the creation of arbitrary control and environment graphs, and hence the explicit control of control flow. Coroutines, generators, classes, actors, and exception handling can be implemented in this way.

Spreadsheet [Cas92] Business spreadsheets contain a functional programming language coupled with an array structure and assignment operations. The programming language often features imperative control structures.

Viron [Pra83] Treats functions and memory cells as processes. Uses a small set of language elements (atoms, arrays, filters, sets, records, and cells) which are combined as processes. Program control flow is implicit.

The languages examined can be split into two categories:

- functional languages with imperative constructs, and
- imperative languages with functional constructs.

In general, the functional programming style (λ-expressions, currying, higher-order functions) was better supported in languages of the first category. The most frequent imperative constructs provided were assignment and input/output capabilities. Modification of control-flow was not a common feature, and when a language provided it, it would provide it in a structured way. In general, one may conclude that combinations of these two paradigms offer a pragmatic approach to functional programming.

Name	Characteristics							Control
	BR	DT	f	λ	$\|\|$	RT	$\overline{\text{RT}}$	
FL		√	√	√		√	√	FR
Fluent		√	√	√		√	√	FR
Gedanken		√	√	√		√	√	FR
Lucid		√				√		*
ML		√	√	√		√	√	FR
Nial	√	√	√	√		√	√	X, FR
Scheme	√	√	√	√		√	√	FR
Spreadsheet	√	√	√	√		√	√	FR
Viron		√	√	√	√	√		*

Table 2.10: Characteristics of functional and imperative paradigm combinations

2.2.4 Combinations of Functional and Object-Oriented Paradigms

Adding objects to a functional programming language can be used as a way to cleanly integrate some imperative features in a functional language. Object-oriented languages deal with objects containing state information — a concept functional languages have difficulty dealing with. As objects are typically encapsulated entities, their addition in the language space of a functional language would be less intrusive than the addition of generic imperative features. Some of the systems contain objects that have no identity that persists between changes of state. These 'functional objects' are mentioned in [Weg90, p. 29]. A discussion of Lisp object-oriented extensions can be found in [Mey88, p. 442].

An additional reason for the attractiveness of this approach, is the historic relationship of certain functional languages with the artificial intelligence community. Objects can be used to naturally represent many real world situations. Their addition to a functional language used for artificial intelligence research, would naturally appeal to the traditional language users. We found eight languages that combine the functional and object-oriented programming paradigms. Their implementations are summarised in table 2.11 and their characteristics in table 2.12. In the following paragraphs we list the most important features of each language.

Common Lisp Object System [BDG$^+$88] An object-oriented extension of Common Lisp. Messages, polymorphism, and multiple inheritance are supported. The implementation of classes, methods, and discriminators is based on a meta-object protocol.

Common Loops [BKK$^+$86, KHDS87] An object-oriented kernel implemented on Common Lisp. Messages, polymorphism, and multiple inheritance are efficiently supported. The implementation of classes, methods, and discriminators is based on meta-objects.

Common Objects [Sny86, KHDS87] An object oriented language implemented on Common Lisp and on CommonLoops. It has extensive support for encapsulation and allows the dynamic redefinition of a class in the running environment.

Name	References	Implementation
Common Lisp Object System	[BDG+88]	Common Lisp extension
Common Loops	[BKK+86, KHDS87]	Implemented on top of Common Lisp
Common Objects	[Sny86, KHDS87]	Implemented on top of Common Lisp
Flavors	[Moo86]	Language
Foops	[GM86b, GM87]	Language
Loops	[SBK86]	Implemented on top of Interlisp-D
T Object	[AR88]	Scheme extension
YAPS	[All83]	Implemented on top of Lisp

Table 2.11: Implementations combining the functional and object-oriented paradigms

Name	Characteristics							Control
	DT	f	IN	λ	MI	O	RT	
Common Lisp Object System	√	√	√	√	√	√	√	FR
Common Loops	√	√	√	√	√	√	√	FR
Common Objects	√	√	√	√	√	√	√	FR
Flavors	√	√	√	√	√	√	√	FR
Foops	√	√	√	√		√	√	*
Loops	√	√	√	√	√	√	√	FR
T Object	√	√		√	√	√	√	FR
YAPS	√	√		√		√	√	*

Table 2.12: Characteristics of functional and object-oriented paradigm combinations

Flavors [Moo86] Object-oriented programming integrated with Lisp. Instance variables, generic functions, and multiple inheritance are available. The implementation includes program development tools.

Foops [GM86b, GM87] Based on reflective equational logic.

Loops [SBK86] An object oriented language based on Interlisp-D. Messages, polymorphism, and multiple inheritance are supported. Supports access-oriented programming by associating annotations with data. These can introduce side effects on variable access, are transparent, and contain their own state.

T Object [AR88] A small number of extensions to Scheme that give it the capability to map functions to values using inheritance.

YAPS [All83] Built on top of Franz Lisp to provide an antecedent-driven production system with daemon objects.

From the languages we examined we conclude that object-oriented programming is typically combined with functional languages by adding objects to an existing functional programming language. This seems to be an established tradition in the Lisp community, and most of the languages examined fall in this category. We view the combination of these two paradigms as important, because by adding objects to a functional language it is possible to integrate various differing evaluation methods, thus paving the way for broader multiparadigm languages.

2.2.5 Combinations of Logic and Object-Oriented Paradigms

We believe that the main driving forces behind the integration of the logic and object-oriented paradigms are the suitability of the object-oriented approach for expressing artificial intelligence problems (a traditional domain of logic programming), and the flat structure of logic programs. Objects can provide the structure on which elaborate logic systems can be built. In systems where such features are not available, one can often see complicated logic programs containing custom-built *ad hoc* mechanisms for managing program complexity. The relationship between logic and object-oriented languages is surveyed in [McC92, pp. 18–30], [Uus92, pp. 99–101], and [VLM88, p. 194]. We found eleven languages that combine the logic and object-oriented programming paradigms. Their implementations are summarised in table 2.13 and their characteristics in table 2.14. In the following paragraphs we list the most important features of each language.

Intermission [Kah82] An object-oriented extension to Prolog. Objects that respond to messages can be defined. All programming is done by the single predicate `send(object, message, result).`

LAP [IK87] Object model relations supporting multiple inheritance can be defined in a Prolog-like environment. Demons can be attached to objects.

LOGIN [AKN86] Based on a ψ-term lattice structure to represent the object hierarchy, and a modified unification procedure.

L&O [McC92] Classes are described as *theories*: labelled sets of axioms. The axioms can be arbitrary predicates which are accessed by prefixing them with the name of their class. The logical variable can be used over theory names. Object state is implemented in terms of *mutable* theories.

LogiC++ [Wu91] A Prolog extension to C++. C++ methods can be written as Prolog Horn-clauses. Clauses are translated into C++ functions using the procedural interpretation of Prolog. The functions are executed at run-time, using activation frames and the usual two stack representation of the backtracking data.

MU [Uus92] Proposal for object-oriented and logic programming based on modal logic. Three logics are developed providing different inheritance models. For those logics a resolution calculus is provided.

PAL [Aka86] Organises Prolog predicates around an inheritance hierarchy. Prolog variables are represented as belonging to a given class, becoming “class bound

Name	References	Implementation
Intermission	[Kah82]	Implemented on top of Prolog
LAP	[IK87]	Prolog library
LOGIN	[AKN86]	Prolog extension
L&O	[McC92]	Prolog pre-processor
LogiC++	[Wu91]	Implemented as a translator to C++
MU	[Uus92]	Theoretical framework
PAL	[Aka86]	Language
PEACE	[Kos87]	Interpreter in Prolog
POL	[Gal86]	OO Extension of Prolog
Prolog/KR	[Nak84]	Prolog extension
Zan84	[Zan84]	On top of Prolog

Table 2.13: Implementations combining the logic and object-oriented paradigms

variables." The unification procedure is modified to unify only variables of matching class instances.

PEACE [Kos87] Objects contain instance variables, named slots and methods, described as predicates. They are organised as a flat network. Slots can be associated with demons that preform actions when some operation is executed on that slot. The demons provided are: `when_empty`, `referred`, `constrain`, `after_put`, `removed`, and `after_add`.

POL [Gal86] The POL system adds the notions of a class, object, method and inheritance to a Prolog-like language. The syntax and semantics of Prolog are retained and the relational framework is used to refine the object-oriented concepts. The system support deterministic methods and higher order functions.

Prolog/KR [Nak84] Allows the structuring of the Prolog database by introducing a world hierarchy in which the predicates are defined. The multiple inheritance provided can be controlled, so that from different viewpoints different ancestors are seen.

Zan84 [Zan84] Objects are defined with some attributes, and a set of Prolog clauses as methods. The primitive `isa` defines the relationship of the object with other objects in a multiple-inheritance hierarchy, and gives values to the attributes of the objects. The attributes of an object remain fixed over its lifetime.

Of the paradigm integrations examined, we found that the integration of logic with object-oriented programming was the one with the widest design and implementation choices and variety of approaches. These ranged from frameworks based on sound theories, to pragmatic approaches targeting real-life applications. Many of the systems offer genuine features that can be immediately utilised to the advantage of the application builder.

Name	Characteristics									Control
	BR	IN	MI	O	R	RT	$\cup$	$\overline{\text{DT}}$	$\overline{\text{RT}}$	
Intermission	√			√	√	√	√	√		SLD
LAP	√		√	√	√	√	√	√		SLD
LOGIN	√				√	√	√	√		SLD
L&O	√	√	√	√	√	√	√	√		SLD
LogiC++	√	√		√	√	√	√	√		SLD
MU		√		√	√	√		√		SLD
PAL	√	√	√		√	√	√	√		SLD
PEACE		√	√	√	√	√	√	√	√	SLD
POL		√	√	√	√	√	√	√		SLD
Prolog/KR	√	√	√		√	√	√	√		SLD
Zan84	√	√	√	√	√	√	√	√		SLD

Table 2.14: Characteristics of logic and object-oriented paradigm combinations

2.2.6 Combinations of Imperative and Object-Oriented Paradigms

The object-oriented paradigm differs from most paradigms in that it deals with data abstractions, rather than control abstractions. Although it is a useful mechanism for organising information, it needs to be coupled with another paradigm in order to make it possible to write useful programs. It can thus be said, that the object-oriented paradigm is *orthogonal* to all other programming paradigms. Naturally, the imperative paradigm being the one with the widest use, would be seen as the most likely candidate for such integration. The objects used in imperative languages are classified as 'imperative objects' in [Weg90, p. 29]. A discussion on object-oriented programming using imperative languages can be found in [Mey88, pp. 373–383]. In [LMT89] the authors argue that "pure" object-oriented systems are preferable to object-oriented extensions to the C programming language. We found seven languages that combine the imperative and object-oriented programming paradigms. Their implementations are summarised in table 2.15 and their characteristics in table 2.16. In the following paragraphs we list the most important features of each language.

C++ [Str86b, ES90] Extends the C programming language by adding objects, classes, polymorphism, inheritance and overloading.

Eiffel [Mey88, Mey92] Block based, procedural, object-oriented programming language with emphasis on reusability, software components, and reliability.

Met87 [Met87] A methodology for object-oriented programming in C. The objects are stored in C structures and unions.

Modula-3 [CDG+92] Block structured, imperative, systems programming language based on Modula-2. Supports objects (as references to data records with methods) and single inheritance. Supports a polymorphic type system.

Objective C [Cox86] Extends the C programming language by adding objects, classes, polymorphism, inheritance, and overloading. A Smalltalk-like syntax is employed. The objects created by Objective-C can be made persistent and be distributed.

Name	References	Implementation
C++	[Str86b, ES90]	Language / C preprocessor
Eiffel	[Mey88, Mey92]	Language / preprocessor to C
Met87	[Met87]	Methodology for object-oriented programming in C
Modula-3	[CDG+92]	Language
Objective C	[Cox86]	Language
Pool2	[Ame89]	Language
Sather	[Omo91]	Compiler, generates C

Table 2.15: Implementations combining the imperative and object-oriented paradigms

Name	Characteristics						Control
	BR	IN	MI	O	$\|\|$	$\overline{\text{RT}}$	
C++	√	√		√		√	X
Eiffel	√	√	√	√		√	X
Met87	√			√		√	X
Modula-3	√	√		√	√	√	X
Objective C	√	√		√		√	X
Pool2	√			√	√	√	X
Sather	√	√	√	√		√	X

Table 2.16: Characteristics of imperative and object-oriented paradigm combinations

Pool2 [Ame89] Object-oriented programming language with message passing across modules running in parallel. Algol-style syntax is provided using a built-in "syntactic-sugar" notation for expressing message passing.

Sather [Omo91] Imperative language offering classes, messages and multiple inheritance. Simplified, optimised variant of Eiffel [Mey88].

Given the maturity and pervasiveness of the imperative paradigm, it did not surprise us a lot to find that most languages combining the two paradigms were quite similar. We believe that in many cases, implementation efficiency concerns considerably limited the leeway available to the language designers. Nevertheless, the languages examined, offered a convincing proof of the orthogonality of the object-oriented paradigm to the imperative one.

2.2.7 Combinations of Functional, Imperative, Logic and Object-Oriented Paradigms

The functional, imperative, logic, and object-oriented paradigms appear to be the most visible programming paradigms. Therefore their combination would appear to be the starting point for multiparadigm language research. We found five languages that combine the functional, imperative, logic and object-oriented programming paradigms. Their implementations are summarised in table 2.17 and their characteristics in table 2.18. In the following paragraphs we list the most important features of each language.

Name	References	Implementation
G	[Pla89, Pla91b, Pla91a]	New language compiler
G-2	[Pla92]	New language compiler
Modcap	[Wel89]	Compiler
Multiparadigm Pseudocode	[WK89]	Pseudocode teaching aid
TAO	[TOO86]	Lisp-based meta-interpreter

Table 2.17: Implementations combining the functional, imperative, logic and object-oriented paradigms

G-2 [Pla92] A multiparadigm language with an attempt to provide semantic consistency in its functional part. The destructive assignment is encapsulated within a *block* construct which insulates it from the rest of the program.

G [Pla89, Pla91b, Pla91a] A language based on demand-driven stream evaluations. Functions are first class citizens, although there is no λ-abstraction mechanism. Logic programming is possible using generators. Logical variables are not provided.

Modcap [Wel89] Language based on functions, with assignment and sequencing providing the imperative features, encapsulation and genericity providing the object-oriented features, and the possibility of parallel evaluation catering for the logic non-determinism.

Multiparadigm Pseudocode [WK89] An expression-based pseudocode notation to be used as a teaching aid. The language features are derived from Modcap [Wel89].

TAO [TOO86] Lisp-based language supporting assignment, messages and multiple inheritance, unification and backtracking. Object-oriented features are implemented by defining classes, as collections of methods belonging to some superclasses. Logic programming is built into the language by offering two *resolvers*, similar to the *lambda* operator. One defines defines unification, and the other non-deterministic choice. Evaluating objects constructed using these resolvers leads to logic semantics. Both resolvers can be user-defined. Concurrency is provided by treating a process as an object. The language is efficiently implemented using micro-coded primitives on the ELIS Lisp-machine.

We found that languages offering the combination of the functional, imperative, logic, and object-oriented paradigms were attempts to provide a broad multiparadigm language, rather than paradigm reconciliation attempts — a common driving force behind languages combining two paradigms. All articles surveyed were explicitly basing the language design rationale on the virtues of multiparadigm programming, and the support for these paradigms — according to the authors — offers a sound base for a multiparadigm language.

Name	Characteristics												Control
	BR	DT	f	IN	λ	MI	O	R	RT	$\cup$	$\overline{\text{DT}}$	$\overline{\text{RT}}$	
G	√	√	√	√			√	√	√		√	√	FR
G-2	√	√	√	√			√	√	√		√	√	FR
Modcap	√	√	√				√	√	√		√	√	SLD, FR, X
Multiparadigm Pseudocode	√	√	√				√	√	√		√	√	SLD, FR, X
TAO	√	√	√	√	√	√	√	√	√	√	√	√	SLD, FR, X

Table 2.18: Characteristics of functional, imperative, logic and object-oriented paradigm combinations

2.2.8 Combinations of Distributed, Logic and Object-Oriented Paradigms

One reason for the combination of parallel, logic, and object-oriented paradigms is that the object-oriented paradigm seems to arrive "for-free", once a parallel implementation of a logic programming language is realised. Thus, slight modifications to the language [KTMB86], or even a suitable programming methodology [ST83] allow the creation of objects containing state. These are represented as the various branches of the logic computation. Given the number of languages combining the logic with the object-oriented paradigms, and the suitability of logic languages to run on parallel hardware, we expect to see more and more languages combining these three paradigms. We found four languages that combine the distributed, logic and object-oriented programming paradigms. Their implementations are summarised in table 2.19 and their characteristics in table 2.20. In the following paragraphs we list the most important features of each language.

Concurrent Prolog [ST83] The facilities available in concurrent Prolog make object-oriented programming natural and easy. Object state changes are represented by the reincarnation of an object with new parameters — a process similar to tail recursion. The objects instantiated are active, executing continually.

Orient84/K [IT86] Supports Smalltalk-like object-oriented programming. In addition objects execute concurrently in a monitor-like fashion and object methods can be written in a language similar to Prolog. Each object has its own, local knowledge base.

SCOOP [VLM88] Prolog extended with a block structured syntax and classes. Parallelism is provided by distinguishing betweem static entities (objects) and dynamic computing agents(processes). The dynamic Prolog clauses are used to represent object state variables, while the static clauses represent class methods.

Vulcan [KTMB86, KTMB87] A preprocessor for concurrent Prolog that allows the expression of object-oriented notions in a natural way. The facilities available in concurrent Prolog already allow object-oriented programming. Object state changes are represented by the reincarnation of an object with new parameters in a way similar to tail recursion. The objects instantiated are active, executing continually.

Name	References	Implementation
Concurrent Prolog	[ST83]	Language
Orient84/K	[IT86]	Language and Kernel
SCOOP	[VLM88]	Implemented on top of Prolog
Vulcan	[KTMB86, KTMB87]	Preprocessor for concurrent Prolog

Table 2.19: Implementations combining the distributed, logic and object-oriented paradigms

Name	Characteristics										Control
	BR	DT	IN	O	$\|\|$	R	RT	$\cup$	$\overline{\text{DT}}$	$\overline{\text{RT}}$	
Concurrent Prolog	√	√	√	√	√	√	√	√	√		SLD
Orient84/K	√	√	√	√	√	√	√	√	√		SLD
SCOOP	√		√	√	√	√	√	√	√	√	SLD
Vulcan	√	√	√	√	√	√	√	√	√		SLD

Table 2.20: Characteristics of distributed, logic and object-oriented paradigm combinations

Although the languages we examined support the creation of objects with mutable states, we did not find support for any inheritance mechanisms — a basic ingredient of object-oriented programming [Weg87, p. 169]. This can be attributed to the fact that sharing of variables among object instances across different processors is a difficult problem.

2.2.9 Combinations of Constraint, Functional and Logic Paradigms

Programming using constraints allows the succinct representation of many difficult problems [Col90, MD88, Ber88]. A problem is expressed as a set of constraints in a domain over some variables in a declarative way. A *solver* built into the constraint programming language will then attempt to provide a solution (in the form of variable values) which satisfies the constraints of the problem. Constraint logic programming languages are examined in [Coh90], while an analysis of the design issues behind integrating the three paradigms can be found in [DGP91a, DGP91b]. We found five languages that combine the constraint, functional and logic programming paradigms. Their implementations are summarised in table 2.21 and their characteristics in table 2.22. In the following paragraphs we list the most important features of each language.

Eqlog [GM86a] Pure first order language combining pure logic programming with equality, and first order functional programming. The operational semantics of the Horn-clause logic are provided by narrowing. It has powerful capabilities as a constraint language.

Falcon [DGP92] Integration of functional, typed-logic and constraint programming language. Relations are specified in functional syntax, but are evaluated using a logic solver and can use the capabilities of the logical variable.

Name	References	Implementation
Eqlog	[GM86a]	Language
Falcon	[DGP92]	Language
Flang	[Man91]	Language
Prolog-with-Equality	[Kor86a]	Extension to Prolog
Unicorn	[Ban86]	Language on top of Prolog

Table 2.21: Implementations combining the constraint, functional and logic paradigms

Name	Characteristics						Control
	DT	f	R	RT	$\cup$	$\overline{\text{DT}}$	
Eqlog	√		√	√	√	√	FR, *
Falcon	√	√	√	√	√	√	SLD, FR
Flang	√	√	√	√	√	√	*
Prolog-with-Equality		√	√		√	√	SLD, *
Unicorn	√			√	√	√	*

Table 2.22: Characteristics of constraint, functional and logic paradigm combinations

Flang [Man91] Functional language where relations are treated as functions. When a function returns a fail value, backtracking occurs. Function variables support unification. The definition of "algebraic" functions allows constraint programming.

Prolog-with-Equality [Kor86a] An extension to Prolog based on equality. Unification is enhanced by solving equality relations whenever two terms fail to unify syntactically. This provides the language with constraint solving capability. The language is expressed using a functional syntax.

Unicorn [Ban86] A first-order functional language based on constraining unification. This, together with semantic unification give it constraint programming capabilities.

All languages we examined, are based on theoretical examination of the underlying logics and solving methods. No *ad hoc* approaches similar to those found in other paradigm combinations were found among these languages. In fact, some of the approaches seem to provide constraint programming as a byproduct of the integration of the other two paradigms using a unified logic approach [Kor86a, GM86a].

2.2.10 Combinations of Various Paradigms

In this section we include languages that could not be fitted into one of the preceding sections. Table 2.23 contains the names of the languages and the paradigms supported. We found 15 languages that combine the various programming paradigms. Their implementations are summarised in table 2.24 and their characteristics in table 2.25. In the following paragraphs we list the most important features of each language.

DSM [Rum87] A C-based object-oriented language is extended by adding the concept of a relation to it. Relations are defined at the same abstraction level as classes.

Echidna [HSS+92] Expert system shell with an execution mechanism based on intelligent backtracking. An object structure is used to organise the knowledge representation schema.

Educe [Boc86] Extends and adds persistence to Prolog by coupling it with the INGRES DBMS.

Enhanced C [Kat83] An extension to the C programming language that allows operations to be performed on whole sets. The set programming paradigm, is thus supported.

Fooplog [GM87] First-order object-based functional language supporting logic programming. Based on reflective Horn-clause logic with equality.

Icon [OG87, Gri84, GG83] Expression-based language offering the concept of generators; non-deterministic expressions with local scope. An expression can fail which leads to backtracking. Provides extensive capabilities for handling of strings.

KE88 [KE88] A bridge between LOOPS and Prolog. LOOPS objects can be accessed from Prolog, LOOPS object methods can be defined in Prolog, and Prolog clauses can be treated as LOOPS objects.

Kaleidoscope [FBB92] Kaleidoscope '91 provides the integration of object-oriented features with a declarative constraint system. The object system is used to organise the contraints in hierarchies via constraint constructors.

Lex [Les75] Allows the definition of text processors and lexical analysers using regular expressions. Each regular-expression can have some code in C associated with it. There is an escape mechanism for explicitly manipulating the input character stream.

ML-Lex [AMT89] Allows the definition of lexical analysers using regular expressions. Each regular-expression has ML code associated with it.

ML-Yacc [TA90] Parsers can be defined in terms of BNF productions. Each rule can have a semantic action written in ML associated with it. The user can specify that the semantic rules are free from side-effects and thus, enhance the error recovery process by allowing the multiple execution of some rules.

SB86 [SB86] Supports the idea of meta-interpreter components to define special evaluation strategies for expert system development. The meta-interpreters are then partially evaluated and mixed with the object base, thus improving efficiency.

SPOOL [FiH86, Yok86] An object-oriented extension to Prolog. Instance and class variables are supported and store state information which is persistent on backtracking. Additional Prolog operators are defined for defining classes and sending messages.

Name	Paradigms
DSM	Imperative, Object-oriented and Relational
Echidna	Constraint, Logic and Object-oriented
Educe	Database and Logic
Enhanced C	Imperative and Set
Fooplog	Functional, Logic and Object-oriented
Icon	Generators and Imperative
KE88	Functional, Logic and Object-oriented
Kaleidoscope	Constraint, Imperative and Object-oriented
Lex	Imperative and Regular-Expression
ML-Lex	Functional and Regular-Expression
ML-Yacc	BNF and Functional
SB86	Any and Logic
SPOOL	Imperative, Logic and Object-oriented
Uniform	Constraint, Functional, Logic and Object-oriented
Yacc	(Object-oriented), BNF and Imperative

Table 2.23: Languages and paradigms they support

Uniform [Kah86] A partly-implemented language based on unification augmented with equality. The unification algorithm provides all the language features. The author comments, that an efficient implementation may be impossible.

Yacc [Joh75] Allows the definition of parsers using BNF rules. Each rule can have some code in C associated with it. Contains mechanisms for controlling expression operator precedence and error recovery. A C++ version [Joh88] allows the use of object-oriented programming techniques.

2.3 Multiparadigm Systems

We distinguish multiparadigm systems from multiparadigm languages by the fact that systems offer a complete framework for integrating and mixing translators for arbitrary paradigms together. In the following sections we will present such systems by providing a brief overview of each one. The approaches taken are quite diverse; therefore no attempt will be made to categorise them.

2.3.1 Unix

The Unix operating system [RT74] is not often thought of in its multiparadigm facet [Hai86a]. Unix supports multiparadigm programming in the sense that is provides a wide array of tools — supporting many different programming paradigms — which can be combined [KP84] using the facilities of the programmable shell offered. Many of the tools offered such as *awk* [AKW79], *sed* [McM79], *dc* [MC79], and the Bourne shell *sh* [Bou79] are complete languages specialised to deal with specific problems. Others such as *cmp, col, comm, diff, grep, expr, find, indent, join, paste, sort, tee,*

Name	References	Implementation
DSM	[Rum87]	Extension to C
Echidna	[HSS+92]	Implemented on top of Lisp
Educe	[Boc86]	Prolog DBMS
Enhanced C	[Kat83]	Compiler producing C
Fooplog	[GM87]	Language
Icon	[OG87, Gri84, GG83]	Language
KE88	[KE88]	LOOPS and Prolog
Kaleidoscope	[FBB92]	Language interpreter
Lex	[Les75]	C preprocessor
ML-Lex	[AMT89]	C preprocessor
ML-Yacc	[TA90]	ML preprocessor
SB86	[SB86]	Meta-interpreters on Prolog
SPOOL	[FiH86, Yok86]	Implemented on top of Prolog VM
Uniform	[Kah86]	Implemented on top of Lisp
Yacc	[Joh75]	C preprocessor

Table 2.24: Implementations combining the various paradigms

Name	Characteristics												Control
	BR	DT	f	IN	λ	MI	O	R	RT	$\cup$	$\overline{\text{DT}}$	$\overline{\text{RT}}$	
DSM	√			√			√	√				√	
Echidna				√			√		√	√	√		*
Educe	√							√	√	√	√		SLD
Enhanced C	√											√	X
Fooplog		√	√	√	√		√	√	√	√	√		*
Icon	√								√		√	√	X
KE88	√	√	√	√	√		√	√	√	√	√		FR, SLD, X
Kaleidoscope	√	√		√		√	√					√	*, X
Lex	√											√	X, *
ML-Lex		√	√		√				√			√	*, FR
ML-Yacc		√	√		√				√			√	*, FR
SB86	√							√	√	√	√		*
SPOOL	√			√			√	√	√	√	√	√	SLD
Uniform		√	√		√	√	√		√	√			*
Yacc	√											√	X, *

Table 2.25: Characteristics of various paradigm combinations

test, *tr*, *tsort*, and *uniq* [BSD86a] can be viewed as very specialised little languages, programmed by the numerous options that control their operation.

All these languages can be combined together thanks to some important design decisions of the tools, and characteristics of the operating system. Unix supports the painless combination of all these paradigms, by making it easy to create processes (thus invoking multiple paradigms), and by providing an efficient and simple interprocess communication mechanism: the pipe [KP84, p. 31]. The tools themselves are designed in a way that lets them be integrated as components of a working system [KP84, p. 170]. The following conventions are used in the design of the tools [KP84, p. 130–131]:

- Each tool does one job, and does it well.
- The output of each tool, can be used as input for another. It is free of decorative lines, headers, copyright messages, and blank lines.
- All filters write to their standard output the result of processing the argument files, or the standard input, if no arguments are provided.
- Error messages are always written on a special output stream, the *standard error*, so as to avoid confusing the tools reading their input.

New paradigms can be easily added to the system, by making them conform to the above design rules.

2.3.2 MLP

The MLP (Mixed Language Programming) system [HMS88] is based on Unix processes communicating via remote procedure calls. According to the author, multiparadigm programming, is not only useful for utilising specific strengths of various languages (he gives as examples string processing, and arithmetic capabilities), but also because mixed language programs can use existing software libraries irrespective of the language that calls them. The author divides the problems for multiparadigm programming into two categories:

1. problems due to incompatible language definitions, encompassing features suchs as different type systems, parameter passing semantics and exception handling, and
2. problems due to incompatible implementation choices including differences of data representation, storage management, and I/O support.

The system deals only with sequential programs, and no attempt is made to define the semantics and properties of the resulting system. Another simplifying assumption in the design of the system, is the decision to avoid modifications to the operating system, and to keep the address spaces of the cooperating paradigms separate.

The system is implemented as a set of translators that compile interface descriptions from the languages supported into object code. Each process is linked with a special run-time library that handles remote procedure calls using the reliable user

datagram protocol. A purpose-built linker binds the exported procedures to the runtime system, and type checks the procedure uses against the interface descriptions. All interface descriptions are expressed in a type system that is supposed to be a superset of the languages supported: the Universal Type System, UTS. The system supports most common simple types, and the copy in, copy out, and copy in/out parameter passing mechanisms. When the interface descriptions are compiled a server module is generated. That module is linked with the rest of the program modules providing the external interface to the exported functions of the program. The server translates data for all the calls to external procedures into an internal format before calling the remote process, and all the arguments of all incoming remote calls from the internal format into the format expected by the language in which the process is implemented. The system desribed in the paper supports modules written in C, Pascal, and Icon.

The importance of this approach lies in its simplicity of design and implementation. Although simple, the resulting system is efficient and useful.

2.3.3 Compositional Approach

The approach described in [Zav89] defines paradigm composition as a collection of single paradigm programs. The interaction between them is defined at the conceptual level of the participating paradigms. The author maintains that it is important to retain the validation operations of the participating paradigms. The isolated program is validated, by expressing the points of communication with parts written in other paradigms using nondeterminism. Thus for example, when validating a module containing a function defined in another paradigm, that function will be treated as nondeterministic in its range. The author claims, that although isolated validation must be applied with care, it can nevertheless have many uses. Three types on paradigm communication modes are distinguished:

1. Call synchronisation: based on the familiar procedure call, return sequence.

2. Stream synchronisation: the first module produces a stream of characters consumed by the second one. No information is passed back from the second to the first.

3. Event synchronisation: the two modules are synchronised at points where they expect a common event.

The distinction on the synchronisation types is important, because it defines the contractual obligations that can be expected to exist between the communicating modules. Different modes of synchronisation imply different types of nondeterminism and hence, different strategies for isolated validation.

Using this approach, the author designed a small prototype telephone network. The database part of the system was written in Prolog, the billing program in *awk*, and the protocol program in CSP. Finally, the performance simulation part of the system was written in a process-oriented applicative language called Paisley. The Prolog part, interacted with the rest of the system using call synchronisation, the *awk* part using stream synchronisation, and the CSP part using event synchronisation. All communication among the parts described above, happened through the part written in Paisley.

At the implementation stage, the communication between Paisley with CSP and Prolog was realised using remote procedure calls, and with *awk* using an intermediate file.

In summary, the main contributions of this approach lie in the characterisation of the communication modes between the paradigms, and the the proposition for individual validation of the parts using nondeterminism.

2.4 Approach Classification, Analysis, and Evaluation

Having described many different multiparadigm languages and systems, in this section we will attempt to classify their approaches into different categories. We will then outline the basic characteristics of each such approach together with the associated strengths and weaknesses. In this way we hope to gain valuable insight that should lead us towards the best way to tackle the problem of multiparadigm programming. The approaches described in the previous sections can be divided into the following categories:

- new languages,
- language extensions sub-divided into:
  - pre-processors,
  - libraries,
  - meta-interpreters, and
  - expanded implementations,
- theoretical approaches, and
- multiparadigm frameworks.

In the following sections we will examine each of these approaches in more detail.

2.4.1 New Languages

New languages are languages designed with the explicit aim of integrating multiple programming paradigms. Typical examples of this approach are [Bud91, Pla92]. The designer of such a language has full control of all aspects of the system. Therefore, the system can be made internally consistent throughout its design and implementation. Unfortunately, language design is a very difficult area where judgement and good taste are of paramount importance. Another difficulty arises from the implementation aspect of such a system. Many declarative paradigms require very specialised implementation knowledge. These two facts combined call for an exceptionally talented designer or a team with deep knowledge of many diverse areas.

Even when the effort is finished the deliverable is a monolithic block. Adding more paradigms to such a system is very difficult as its modification dynamics are likely to be extremely complex. Finally, in our opinion, the intermingling of syntactic and semantic notions of different paradigms within a single language can easily result in a language that is difficult to implement, learn, and use.

2.4.2 Language Extensions

Language extensions typically start from a single language, and extend it to accommodate different paradigms. The extension of an existing language provides the designer with a solid base for the extensions. This can include the — hopefully sound — design decisions of the original language, available implementations, verification techniques, development, debugging, and performance measuring tools, a range of programs that can be ported with little difficulty, and also a trained user base. However, languages can not be easily extended to handle more than one additional paradigm, and often a language can contain features that will complicate or hinder such extensions. Furthermore, the syntactic support provided by the base language can force the notions from a foreign paradigm to be expressed in a roundabout, non-intuitive style.

There are many ways in which language extensions can be implemented; we will discuss specific advantages and disadvantages of each one in turn.

Pre-processors

A language can be extended by passing the source text of its extended version through a pre-processor that transforms it into the standard version of the language [Ker75]. This approach has been used to implement multiparadigm systems in [MS90, KTMB86]. The preprocessor style of language extension [CHP88] has the advantages of easy implementation, and freedom of syntactic expression. However, good implementations may need to perform a lot of checking of the full language in order to provide useful error messages thus complicating the implementation effort. Furthermore, the development tools associated with the language (debuggers, linkers), are likely to provide an obfuscated picture of the source code to the user, due to the transformations that have been imposed to his source code by the pre-processor.

Libraries

Libraries are pre-compiled sets of modules of a language, that can be used together with other programs. They can offer additional functionality, and some approaches use them in order to provide a multiparadigm programming capability [Mul86]. Libraries offer the easiest way to enhance the power of a programming language. Unfortunately, their scope is limited as they can not affect the syntax or semantics of existing language constructs.

Meta-interpreters

Many declarative interpreter-based languages allow the easy and relatively efficient expression of meta-interpreters: interpreters that implement the original language. These can then be modified to interpret a superset of the original language, and thus used to add more paradigms to it. This approach has been used in [SB86, Bon87, CST87, DFP86]. An implementation based on a meta-interpreter is likely to be easy to implement, but inefficient in its execution. The technique can be utilised for implementing language prototypes.

Enhanced Implementations

Another way to extend a language is to start from an available implementation and modify it to accommodate other paradigms [Cox86]. This is likely to be a more robust approach than a pre-processor while allowing for considerable syntactic change breadth. On the other side, the implementation effort is likely to be an order of magnitude more than that of the other approaches, and the drawbacks of the language-extension approach still hold.

2.4.3 Theoretical Approaches

Many researchers approach multiparadigm programming by attempting to unify the theoretical bases of the paradigms they want to unify [Han90b]. Then, on the theoretically sound groundwork a language is designed and implemented. This approach has the advantage that many verification methods available for specific paradigms are likely to be available for the multiparadigm system. Furthermore, the system should offer a consistent framework where reasoning about programs and program transformations will be possible. In practice, such an approach seems to be extremely difficult. The number of paradigms that can be unified under a theory roof is likely to be limited and adding more paradigms will in most cases completely modify the relevant theory requiring a complete rework of it. The difficulty of the approach often leads to languages tailored after the theory instead of theories tailored after a language. This is — in our opinion — an unlikely approach to language design; most successful languages are pragmatically oriented towards practical needs, rather than derived from a theory — even in areas where the language paradigm is based on strong theoretical groundwork [Har86, REA+86].

2.4.4 Multiparadigm Frameworks

The most promising area in multiparadigm programming, appears to be that of frameworks supporting multiple paradigms. Such frameworks, make no attempt to dictate what paradigms are to be used, but provide the requisite methodology and technology for the integration of the parts. This approach, is open-ended, and has no inherent limitations or disadvantages. However, systems adopting this approach are an order of magnitude more difficult to design and implement, than the systems based on the other approaches, because of their generality and openendedness. We believe, that once a satisfactory solution has been found and adopted, many phases of the design and implementation of multiparadigm systems will be rationalised or automated, in the same way, as is the case today for the design and implementation of programming languages.

In the following sections we will deal with each of the systems surveyed separately, as they all have different advantages and drawbacks.

Unix

The environment provided by the Unix operating system, allows the swift and painless implementation of multiparadigm programs, by combining the different paradigms as processes in a pipeline. Applications developed using this approach however, can be

Functional	•		•	•			•		•
Imperative		•	•			•	•		
Object-Oriented				•	•	•	•	•	
Logic	•	•			•		•	•	•
Distributed								•	
Constraint									•
Number of languages	24	10	9	8	11	7	5	4	5

Table 2.26: Number of languages for the common paradigm combinations

inefficient, as each exchange of information between the parts implemented in different paradigms, requires a context switch and a copy operation through the data space of the operating system's kernel. Furthermore, the information flow is restricted to a one-directional stream, with no possible feedback from the processes downstream to those upstream. More complicated configurations are not supported by the standard user interfaces (shells), and manual implementations of them can lead to deadlocks. This restriction, can make applications with a more complex structure unreliable, or impossible to implement.

MLP

The MLP system allows for complicated interactions between the different components, by the use of remote procedure calls. These however, incur the overhead of the data conversion between the native and global formats, and the passing through the operating system's network layers. Furthermore, the system lacks a conceptual model for extensions to its structure and capabilities.

Compositional Approach

The advantage of the system proposed by [Zav89] lies in the models offered for validating modules written in a single paradigm in isolation, and for characterising the communication modes between different paradigms. Unfortunately, the compositional approach, does not offer a structuring methodology for combining the different paradigms at the implementation level.

2.4.5 Conclusion

We conclude, that multiparadigm system frameworks, offer the greatest promise for usable multiparadigm systems. The ones we examined, have strengths in different areas; all of them however, lack a unifying model for the combination of the paradigms. In the following chapters we hope to offer our readers such a model, and design and implement a system based on it.

2.5 Summary

A programming paradigm is the term used in computer science to define a style of programming expressing the programmer's intent. A number of languages allow pro-

gramming in more than one programming paradigm. The most common paradigm combinations and the number of languages supporting them are summarised in table 2.26. The survey of those languages could not be exhaustive, but we hope that most of the ideas have been touched on. Multiparadigm systems offer a framework for combining arbitrary paradigms. We examined the Unix operating system as a multiparadigm framework, the MLP system based on an RPC process communication mechanism, and the compositional approach which attempts to provide program validation in multiparadigm applications. The approaches to multiparadigm programming can be categorised between new languages incorporating multiple paradigms, the addition of new paradigms in existing languages, multiparadigm theoretical groundwork, and multiparadigm frameworks.

Chapter 3

The Approach

Our approach to the problem of multiparadigm programming will be the following:

- decompose the problem into subparts (section 3.1),
- construct a design prototype for our system by starting from the end-user applications and moving towards the supporting software (section 3.2), and
- based on the basic design directions abstracted from the prototype, design each part of the system starting from its structure and finishing with end-user applications (sections 3.3, 3.4, 3.5, and 3.6)

3.1 Problem Decomposition

The problem of multiparadigm programming can be tackled by realising that it can be decomposed into a hierarchy of four different problems, each with a wider domain:

1. Multiparadigm applications: programs written in more than one programming paradigms.
2. Multiparadigm programming environments: integrated systems or suites of tools that allow the application programmer to develop an application in more than one programming paradigm.
3. Multiparadigm environment generators: systems that can be used in order to design and implement a multiparadigm programming environment.
4. The structure of multiparadigm systems: the general structure suitable for a reliable and efficient implementation of such systems.

The relationship of these four problem areas is presented as an entity-relationship diagram [Che76b] in figure 3.1. The separation of our domain into those areas, enables us to concentrate on each one, clearly identify the specific problems that exist, and propose solutions that can be tested and evaluated in a controlled manner. Attempting to solve the problems in all of the areas above at once, under the title “multiparadigm programming”, may or may not be successful depending on the luck and the intuition of the researcher. Certainly, the end result will be a monolithic system that must be

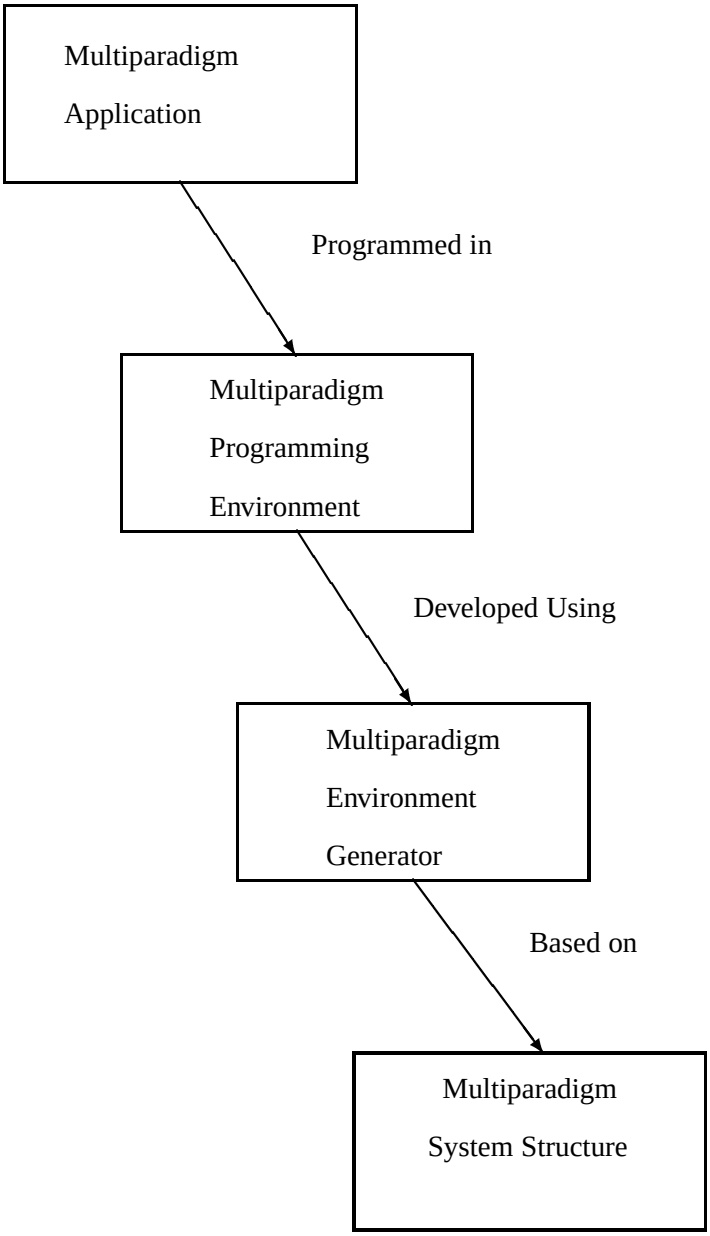

Figure 3.1: Entity-relationship diagram of the separate multiparadigm problem areas

accepted or rejected as a whole. In our approach, different parts of the system can be judged independently, and can be individually refined, re-implemented or even re-designed.

3.2 From Applications to Systems

In this section we will attempt to construct a rough prototype of our design using a synthetic approach. This prototype will then be used as the building base for the system. For each of the multiparadigm problem areas that we have identified in section 3.1, and starting from multiparadigm end-user applications, we will provide a short description, identify the inherent requirements of the area, point-out the various design alternatives, and choose from them a suitable design structure.

3.2.1 Multiparadigm Applications

A multiparadigm application is a system developed using more than one programming paradigm.

Requirements

Multiparadigm programming applications can only be designed and implemented if the following requirements are met[1]:

- support for a number of diverse paradigms,
- inter-operation of these paradigms,
- isolation from unwanted interactions between the paradigms, and
- an efficient application implementation.

Alternatives

These requirements can be met in the following different ways:

1. Write the application in a single language that will offer the facilities of all the paradigms.
2. Try to integrate existing languages supporting the needed paradigms.
3. Implement the application using a number of communicating processes, each using a different paradigm, using the underlying operating system inter-process communication capabilities.
4. Structure the application around modules, each written in the most appropriate paradigm.

[1] See also [KdRB92, p. 8].

Structure

We chose to meet the requirements by structuring applications around modules written in the most appropriate paradigms. This approach is realistic and offers guarantees of paradigm isolation and system efficiency, which the other alternatives lack. Multiparadigm applications can be built by combining existing languages. Such efforts are dependent on the underlying language, linker and operating system technologies. This makes them unportable to other environments.

3.2.2 Multiparadigm Programming Environments

A programming environment is the system in which a multiparadigm application will be written. The structure we proposed for multiparadigm applications makes it necessary to design and implement a specialised programming environment.

Requirements

A multiparadigm programming environment must meet all the requirements set forward by the multiparadigm applications and in addition:

- accommodate the diverse execution models and mechanisms of the various paradigms (e.g. abstract machine interpreters, data structures, signals),
- manage the resources required for implementing the different paradigms (e.g. memory, name-space, process attributes),
- offer an intuitive way to combine code written in different paradigms, and
- offer an orthogonal programming interface to those paradigms.

Alternatives

Given the structure of the multiparadigm applications, multiparadigm programming environments can be structured in a number of different ways. An obvious way is to structure them as a flat structure of different paradigms. A more advanced structure would be a tree-like hierarchy according to similarities between the paradigms, or a graph based on dependency relations between the paradigms. The structural views of the environment user, and the environment implementor need not coincide.

Furthermore, a multiparadigm programming environment can be implemented, either as a single compiler or environment, or as a collection of a number of compilers.

Structure

We have chosen to structure the multiparadigm programming environments implementor's view as a tree of paradigms each compiled by a separate compiler (see figure 3.2 for an example). This structure offers the possibility of reusing part of one paradigm in implementing another. Furthermore the division of the paradigm compilers gives us a more modular and flexible system. This structure is hidden from the multiparadigm programming environment user who views the same system as a flat collection of paradigms (see section 3.3.12).

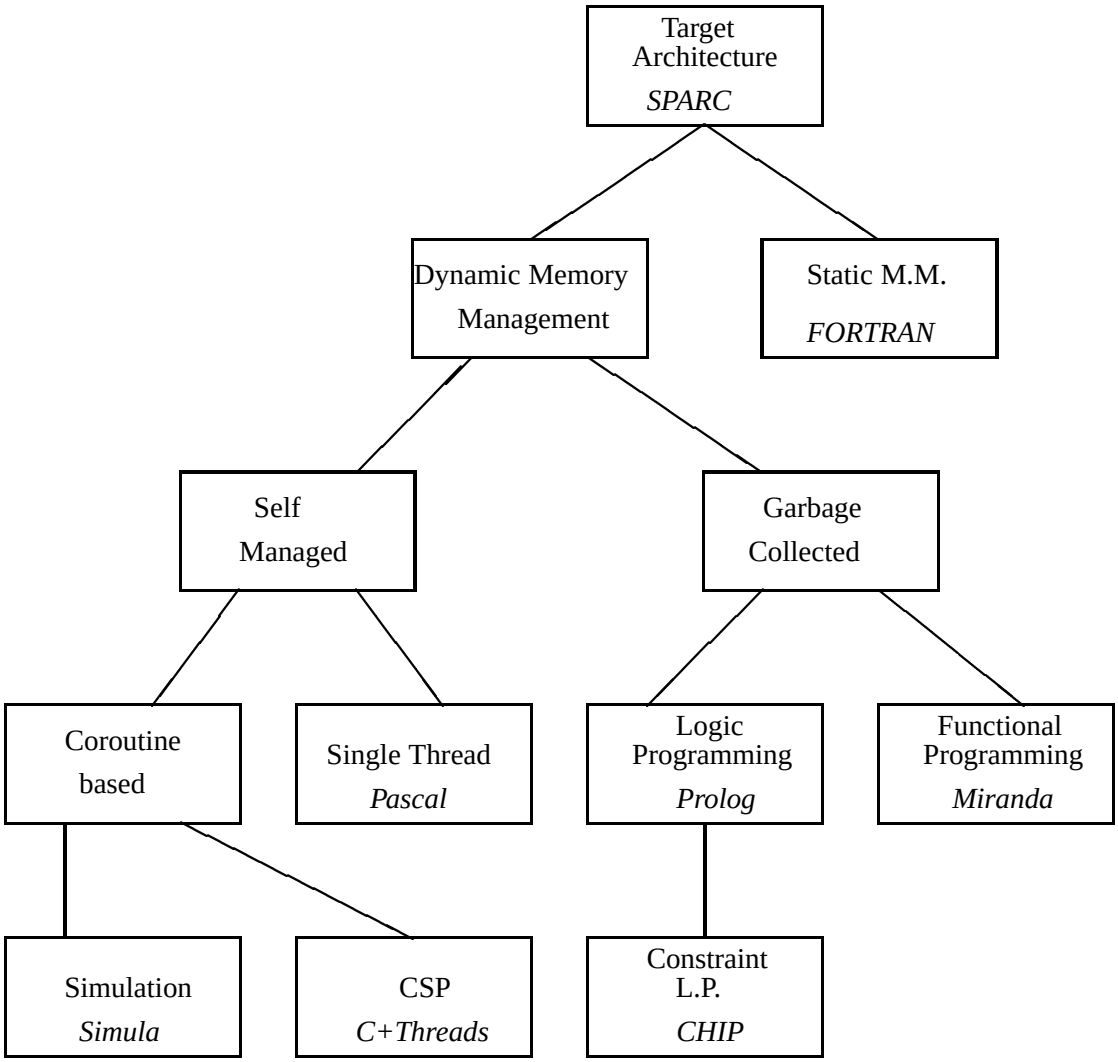

Figure 3.2: Paradigm class tree structure example

3.2.3 Multiparadigm Environment Generators

Multiparadigm programming environments are complex collections of a number of systems, such as compilers, run-time support libraries and the associated utility tools. In order to organise the task of creating such environments we envisage the realisation of *meta-environments*: multiparadigm environment generators. These will offer facilities for making the task of developing multiparadigm programming environments as easy as possible.

Requirements

Any multiparadigm environment generator must satisfy the requirements imposed by the multiparadigm applications and the multiparadigm programming environments and in addition:

- decrease the development time for multiparadigm programming environments,
- promote the creation of efficient and reliable environments, and
- support existing tools.

Alternatives

Given the multiparadigm programming environment structure such systems can be based around two different models:

1. Regard the programming environment as a process that can be decomposed into the paradigms forming the tree hierarchy. This model would lead to a top-down process oriented design.
2. Think of modules written in a given paradigm as objects, paradigms as object classes and use object-oriented design techniques.

Structure

Multiparagm systems change and evolve. The top-down process oriented design entails the *waterfall* life-cycle model [Roy87] with all its shortcomings [LST82]. We decided to use the object-oriented approach. Multiparadigm environment generators are complex structures who would benefit from the robustness, extendibility and reusability offered by the object-oriented design methodology.

3.3 Multiparadigm System Structure

In this section we present the general structure of a multiparadigm system. This is the design structure upon which the concrete designs of the generators, the programming environments, and the applications will be based. We first go through the requirements that this design must satisfy, then list the important ideas on which our design is based, and finally, present the design in detail.

The system structure of a multiparadigm system must be flexible enough to accommodate the different programming paradigms, yet offer a stable foundational structure

for organising the development, at all levels, in a coherent way. Furthermore, it should be possible to implement a system satisfying the above with the minimum of effort. These three requirements can be contradictory, unless the suitable abstractions are chosen. Being contradictory however, they serve as valuable guides and evaluation means for our design. They guide us on the middle way between over and under-specification, with the flexibility requirements forming the upper bound, and structure requirements the lower bound of the specification required. In addition the quality of the design can be judged by how far all three requirement categories are satisfied.

3.3.1 Flexibility Requirements

The system should be as flexible as possible and without arbitrary restrictions. This is important, because programming paradigms have a breadth of syntactic, semantic and implementation-related differences. The flexibility requirements are:

F1 Accommodation of different syntactic notations: Different paradigms are best programmed in different syntactic notations. This will ensure that the most suitable notation for each paradigm will be used and that the programmer will receive the appropriate paradigm related clues when reading the code. Porting code written in a given paradigm will be relatively easy, and in addition, existing tools related each that paradigm can be used.

F2 Accommodation of diverse execution models: Different paradigms have different execution models. Almost[2] all of them can be modelled using a Turing machine; therefore they pose, in theory, no implementation problems. In practice, the execution model of the system must be flexible enough to provide a base for their *efficient* implementation.

F3 Support for different execution mechanisms: Some paradigms are usually implemented by being directly compiled to target machine code, others have their code interpreted, yet others have their code translated to some abstract machine notation and provide the abstract machine execution mechanism as part of the runtime support system. The choice of the execution mechanism depends on the paradigm, the target architecture and space versus time efficiency considerations. The system structure must support all these execution mechanisms to make possible an optimum implementation.

F4 Ability to use existing tools: Implementing a programming paradigm can be a complicated process. If the system structure makes the use of existing tools that handle that paradigm (such as interpreters, compilers or runtime systems) possible, then the resulting system can be implemented in less time, and by capitalising on the existing tool investment, it can be more reliable and efficient.

F5 Arbitrary paradigm mixing and matching: For a multiparadigm environment to be used by the application developers, its internal operation must be transparent, and different paradigms should be easily combined.

[2] An exception would be a paradigm based on real non-determinism such as random number generators using a quantomechanical noise source.

3.3.2 Structural Requirements

The same breadth of paradigm differences that dictated the flexibility requirements for a multiparadigm system structure, combined with the notion that a number of diverse paradigms will be combined in a single development system or application, dictate a different set of structural requirements. These ensure that the system will have a structure to aid its cognitive understanding by environment and application developers, and that the different paradigms will not interfere with each other.

The environment developer must be provided with a conceptual structure on which to base the realisation of the concrete multiparadigm programming environment. This structure will guide his design and implementation steps. Absence of such a structure will require a higher level of effort and possibly training, from the part of the environment developer, in order to create a system that will be easy to understand and use.

Similarly if an ordered system structure is reflected down to the multiparadigm application development level, the application developer will find it easier to envisage how the application will use the different paradigms, and how code using them will function to achieve the intended result. Furthermore, should there be some unwanted interactions between the paradigms, the structure will help the application developer to isolate and correct it.

The structural requirements of multiparadigm systems are:

S1 Isolation of syntactic interactions: As mentioned in the flexibility requirements, different paradigms are best programmed in different syntactic notations. The system structure however, must ensure that these will not interact in unwanted or unexpected ways. Given the complicated parsing rules for some languages such as HASKELL and Prolog, it is also wise to separate the syntactic interactions both at the system definition and at the implementation level, in order to ensure that a multiparadigm programming system definition is easily understood and consistent, and that its implementation is straightforward and correct.

S2 Isolation of execution models: The various execution models of different paradigms call for a system structure where those will not interact in ways that are difficult to specify or implement. Mixing different execution models together is an interesting academic exercise, but from a software engineering point of view, the idea of specifying what happens when, for example, when an exception is raised while backtracking a lazily evaluated function, is obvious.

S3 Isolation of execution mechanisms: In the flexibility requirements we mentioned that many paradigms need some kind of runtime support, either in the form of an interpreter for the real code, or its abstract machine compiled notation, or for handling concepts like dynamic memory allocation and garbage collection or managing execution threads. In a multiparadigm application all support code for different paradigms, will be bundled within a single application. The system structure must be such that the runtime support mechanisms of each paradigm will not interfere with the functioning of the others.

S4 Resource management: Related to the execution mechanisms that must coexist within a single application, is the concept of the finite resources that a number

of paradigms must use in a controlled manner. During the compilation phase of an application, an important resource that must be correctly structured is the application's namespace. During execution the CPU, memory, and peripheral devices required by different paradigms, must be managed in a way that ensures that all paradigms function without interfering. A suitable system structure will make it straightforward to satisfy this requirement.

S5 Documentation: A multiparadigm system is a complicated structure. Precise documentation of its parts is an essential component of the system. Adding it as a design requirement ensures that it will not be treated as an afterthought by the support environment developer and the multiparadigm system implementor.

3.3.3 Efficiency Requirements

The efficiency requirements ensure that the system design will be of real practical value. They serve to guide us towards a system structure that will be practical to implement and use. The three efficiency requirements are:

E1 Efficient paradigm combination: For the application programmers to use multiparadigm programming, code written in different paradigms should be straightforward to combine by an easy to use and efficient mechanism.

E2 Efficient environment creation: We view multiparadigm programming environments as dynamic parts of the program development process. They should be easily adaptable to different technologies, approaches, and projects. For this reason it is important to try to make the design and implementation of multiparadigm programming environments as straightforward as possible. In particular, the addition of a new paradigm to an existing environment, or the modification of an existing paradigm, should be actions efficiently supported by the system structure.

E3 Efficient paradigm implementation: The system structure should make it possible to implement the optimum implementation for each paradigm.

In the following two sections we will describe two concepts that will be used in section 3.3.7 to build up the whole system structure.

3.3.4 Paradigms as Linguistic Transformations

A programming paradigm is really a notation for describing an implementation for a specific problem. This notation can resemble the notation used by the machine that will execute the implementation or it can resemble some other notation suitable for describing implementations in the problem domain. At some point however, the implementation *will* be executed on a real machine and for this reason the semantic gap between the implementation paradigm and the programming paradigm of the target architecture must be bridged. This is usually done by an interpreter, a compiler or a hybrid technique. We regard all these methods as linguistic transformations from the paradigm notation to the target architecture notation. This view although not strikingly impressive provides us with two important insights:

Figure 3.3: T-diagram representations for translator and interpreter.

1. A programming paradigm is nothing *magical*. All programming paradigms can be implemented on all architectures. Furthermore, there is, in principle, no practical or theoretical reason for not being able to combine different paradigms, since they can all be mapped into the same concrete architecture.

2. The target architecture plays an important role when thinking of programming paradigms. The concept of the target architecture should be an integral part of multiparadigm systems and not an externally imposed specification, or an afterthought.

3.3.5 Abstract Description of Multiparadigm Systems

Examining the translation aspect of programming languages, we find that these can be implemented either as a translator[3] or as an interpreter. We use an extended symbolic representation of T-diagrams [ASU85, p. 726] where, for example, the translator ${}_A{}_C{}_B$ denotes a translator from language A to language B, implemented in language C, amd the interpreter D_E denotes an interpreter for language D implemented in language E (figure 3.3). We use the term *source language* for the languages A and D, and the term *target language* for the languages B and E. The language C is an implementation detail of the development environment and will not concertn us at this point. It is apparent, that both types of implementation do not actually provide a complete solution; they merely reduce the problem to another problem. The translator ${}_A{}_C{}_B$ reduces the problem of implementing language A to that of implementing language B, and the interpreter D_E reduces the problem of implementing language D to that of implementing language E. Languages B and E can only be implemented as a translator or as an interpreter; therefore any language implementation will consist of one or more of the four possible different couplings illustrated in figure 3.4. Each coupling in a series represents a level of reduction until the target language of the translator or the interpreter is the machine language of the host machine. At every coupling the *source* and *target* languages must match. The following are examples for each of the four possible coupling types (as shown in figure 3.4):

1. **Translator-translator** Most compilers generate assembly language, which is then passed to the system assembler; this is a translator-translator interface. Some implementation of C++ [Str86b] and Eiffel [Mey88] languages compile into C and use it as a "portable assembler".

[3] We use the term "translator" to denote a compiler, an assembler, or a translator from one language into another.

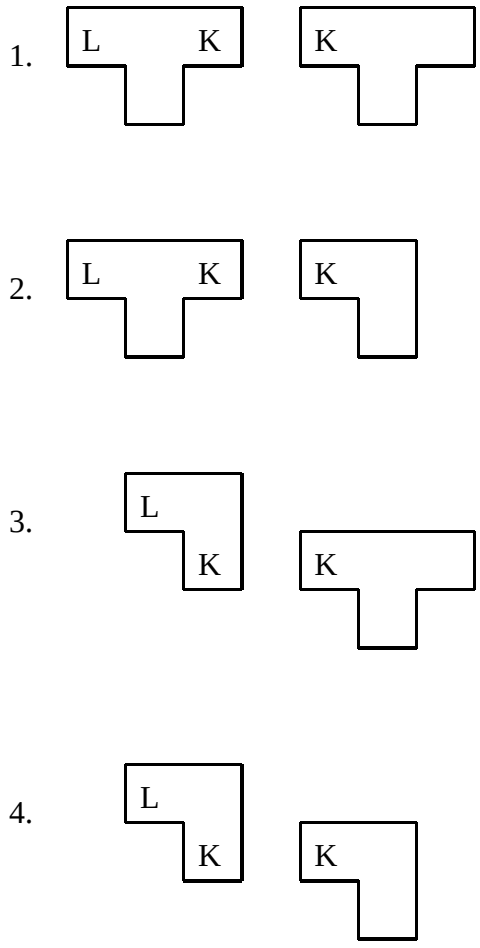

Figure 3.4: Language implementation coupling possibilities.

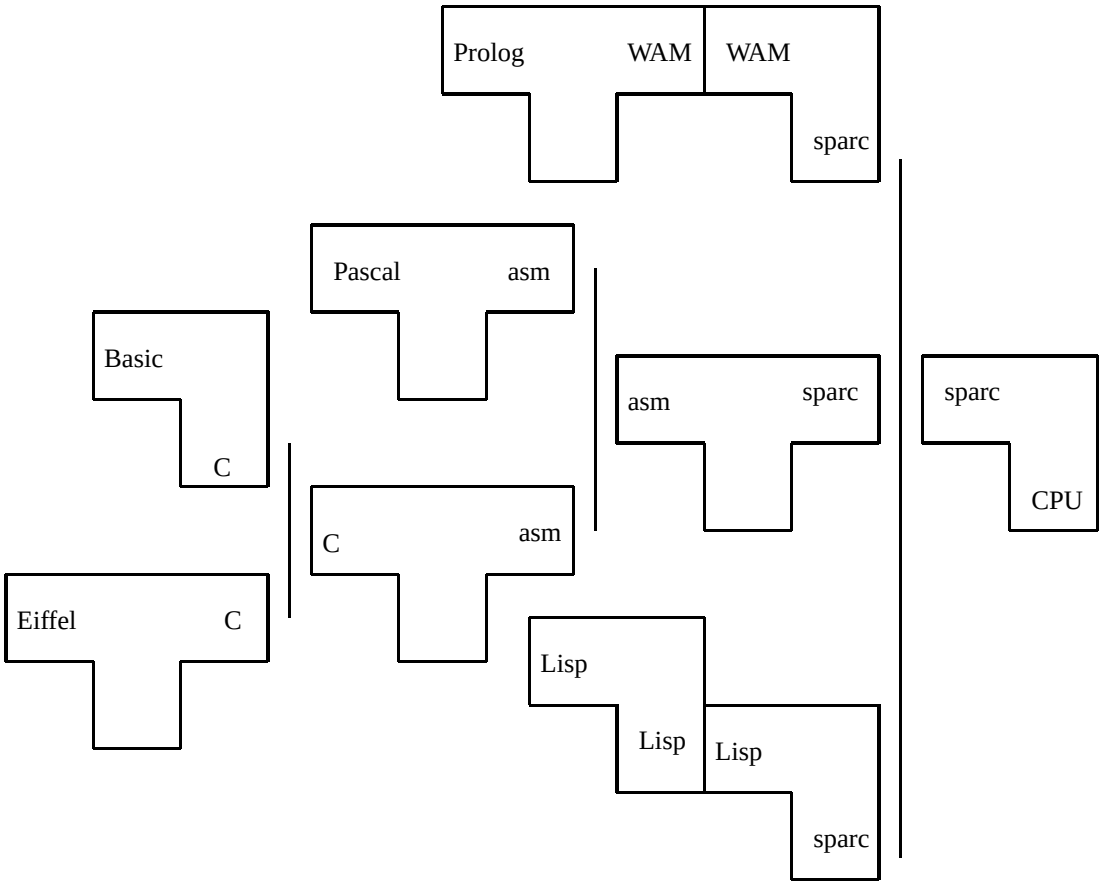

Figure 3.5: A complex language implementation tree.

2. **Translator-interpreter** Many languages are compiled into an abstract machine instruction set which is then interpreted by an abstract machine emulator. Typical examples are most Smalltalk [Gol80] implementation and WAM-based [War83] Prolog systems.

3. **Interpreter-translator** All interpreters imeplemented in a compiled language fall into this category. Typicall examples include the Unix shell [Bou79] and the Perl programming language [WS90].

4. **Interpreter-interpreter** The approach of *meta-interpreters* is often used to extend declarative languages such as Prolog [Coh86] or Lisp [Bon91].

Any combination of languages that run on a given uniprocessor system can be depicted in the form of a tree whose nodes are interpreters and translators. The root of the tree is the system's processor — an interpeter of the system's machine language. All translator that output machine language (e.g. assemblers and linkers) or interpreters implemented in the native language, form the first level branches of the tree. Compilers that output assemble language form the second level of the tree and so on. Figure 3.5 illustrates how the implementation of a number of seemingly unrelated languages

is organised into a tree. Prolog is implemented as a compiler to WAM which is emulated by an interpreter directly compiled into the native code. Lisp also has a native code interpreter, but this is used to run a meta-interpreter [ASS85]. The C and Pascal compilers generate assembly code which is then translated into machine language. C is also used to implement a Basic interpreter and as the target language of an Eiffel compiler.

Ever connection between two nodes is an interface between two different programming languages. As we detailed above, there are four different interfaces betweem two nodes, and each interface can be traversed in two directions, giving a total of eight possible inter-language calling possibilities. If all these possibilities are implmenetable, then any language of the tree can call any other language of the tree — with the help of auxilliary substroutines introduced at every node visited. The following list details how every interface in figure 3.4 can be implemented:

1. **Translator-translator** For a subroutine S_L implemented in L to call a subroutine S_K implemented in K, K must be written to conform with the L calling conventions. As an example, the following SPARC assembly language routine that returns the sum of its two arguments is callable from C[4]:

   ```
   _SK:
           retl
           add       %o0, %o1, %o0
   ```

   For a subroutine S_K implemented in K to call a subroutine S_L implemented in L, K must use the L calling conventions. For example the following SPARC code calls the C *exit* function with 2 as an argument:

   ```
           call      _exit, 1
           mov       2, %o0
   ```

2. **Translator-interpreter** For a subroutine S_L implemented in L to call a subroutine S_K implemented in K, the subroutine K must be implemented using the calling conventions expected by L. The following example shows the Prolog clause *loves(john, mary)* written in SB-Prolog WAM assembler syntax:

   ```
   label((loves, 2, 0)).
   getcon(john, 1).
   getcon(mary, 2).
   proceed.
   ```

   For a subroutine S_K implemented in K to call a subroutine S_L implemented in L, the code in K must be written so as to use the L calling conventions. In the following example the Prolog + predicate is directly called from WAM assembly to move the result of adding $1 + 2$ into variable X.

[4]Note that the SPARC transfer isntructions (jmp,, call, ret) are actually execute after the instruction that follows them.

```
    label((is, 2, 2)).
    getstr(('+', 2), 2).
    uninumcon(1).
    uninumcon(2).
    proceed.
```

3. **Interpreter-translator** For a subroutine S_L implemented in L to call a subroutine S_K implemented in K, the interpeter must provide a way to add new subroutines to the list of routines callable by L. The routine written in K must disassemble the arguments from the interpreter data structures. The following code gragments show how the Unix *curses* [Gro86, curses(3X)] library *raw* function can be added to the Perl [WS90] programming language:

```
int init_curses(void)
{
    struct ufuncs uf;
    char *filename = "curses.c";

    make_usub("raw", US_raw, usersub, filename);
[...]
static int usersub(int ix, int sp, int items)
{
    STR **st = stack->ary_array + sp;
    switch (ix) {
[...]
    case US_raw:
        if (items != 0)
    fatal("Usage: &raw()");
        else {
    int retval;
    retval = raw();
    str_numset(st[0], (double) retval);
        }
        return sp;
[...]
```

   For a subroutine S_K implemented in K to call a subroutine S_L implemented in L, the interpreter to L must provide an entry point for the K routine, and K must use that entry point and provide a suitable environment for the invocation of S_L. The following example illustrates how a C routine can call the Perl routine *myperl* with an argument of 42.

```
    tmpstr = str_make("&myperl(42);", 1);
    sp = do_eval(tmpstr, optype, curcmd->c_stash, FALSE, g, arglst);
```

4. **Interpreter-interpreter** For a subroutine S_L implemented in L to call a subroutine S_K implemented in K, the interpreter must provide an escape function

to move from the meta-level to the K language level. The following example fragment from a Prolog meta-interpreter shows how a term specifying addition (*plus(X,Y,Z)*) is handled by calling the Prolog + predicate:

```
builtin_unify(plus(X, Y, Z), Env, [Binding | Enc]) :-
    lookup(X, Env, X1),
    lookup(Y, Env, Y1),
    lookup(Z, Env, Z1),
    logplus(X1, Y1, Z1, Binding).

logplus(X, Y, Z, bind(X, R)) :-
    is_btrackvar(X),
    is_integer(Y),
    is_integer(Z),
    R is Y + Z.
```

For a subroutine S_K implemented in K to call a subroutine S_L implemented in L, the meta-interpreter must provide an entry point callable from K. In the following example a Prolog program in K calls the *fact* clause which is implemented in the meta-interpreter:

```
fact(N, N1) :-
    rules(Rules),
    metaprolog([fact(N, $N1)], Rules, Findings),
    varval($N1, Bindings, N1).
```

3.3.6 Paradigms as Classes

The second major idea, that will be used in order to design the general system structure, is to regard a programming paradigm as a class. Thus object-oriented programming abstractions can be used when thinking about multiparadigm programming. It turns out that the object metaphor suits the abstraction of a "programming paradigm", and that by using it we can satisfy all the flexibility, structure, and efficiency requirements outlined in sections 3.3.1, 3.3.3, and 3.3.2.

In the following paragraphs we will examine how important aspects of object-oriented programming can be related to programming paradigms and multiparadigm programming. Let us use the equation [Weg87]:

object-oriented = objects + classes + inheritance

and the class definition from [Nel91]:

```
Class <class_name>
  Superclasses:          <superclass_1>, <superclass_2>, ...
  Class Variables:       <class_var_1>, <class_var_2>, ...
  Instance Variables:    <inst_var_1>, <inst_var_2>, ...
  Methods:               <method_name_1>, <method_name_2>, ...
```

Objects

In object-oriented programming an object is an entity having a set of operations and a state that remembers the effect of those operations [Weg87]. Objects are different from functions in that the results from operations on them depend on their state, as that has been defined by previous operations. In our case we use an object as the abstraction mechanism for code written in a given paradigm. Such objects have at least three *instance variables* (figure 3.6):

1. *Source code.* The source code contained in an object is the module provided by the application programmer.

2. *Compiled code.* The compiled code is an internal representation of that specification (generated by the class *compilation method*) that is used by the class *execution method* in order to implement the specification.

3. *Module state.* The module state contains local data, dependent on the paradigm and its *execution method*, that is needed for executing the code of that object.

Every object has at least one *method*:

1. *Instance initialisation method.* The instance initialisation method is called once for every object instance when the object is loaded and before program execution begins. It can be used to initialise the module state variable.

As an example, given the imperative paradigm and its concrete realisation in the form of Modula-2 [Wir85b] programs, an object written in the imperative paradigm can correspond to a Modula-2 module. The *source code* variable of that object contains the source code of the module, the *object code* variable contains the compiled source, and the *module state* variable contains the contents of the global variables. In addition, the *instance intialisation method* is the initialisation code found delimited between `BEGIN` and `END` in the module body.

Classes

In most object-oriented programming definitions a class is defined as the template used to create new objects. Objects created from the same class share the same operations and behaviour [Weg87]. Class variable and methods can be used to implement an interface available to the clients of a class. In our object-oriented multiparadigm programming approach all classes contain at least one class variable (figure 3.6):

1. *Class_state*: contains global data needed by the *execution method* for all instances of that class.

In addition paradigm classes contain at least four *methods*:

1. *Compilation method.* The compilation method is responsible for transforming, at compile-time, the source code of that paradigm into the appropriate representation for execution at run-time.

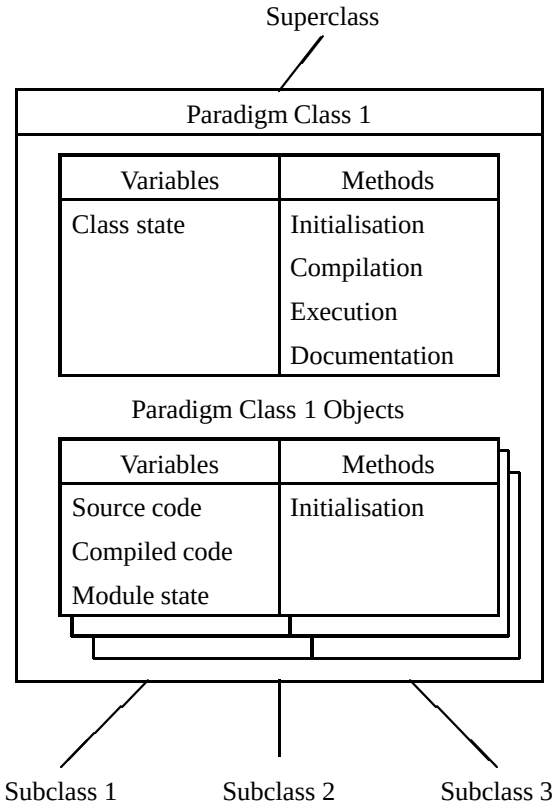

Figure 3.6: Programming paradigm classes and objects

2. *Class initialisation method.* The class initialisation method of a paradigm is called on system startup in order to initialise the class variables of that class. It also calls the instance initialisation method for all objects of that class.

3. *Execution method.* The execution method of a class provides the run-time support needed in order to implement a given paradigm.

4. *Documentation method.* The documentation method provides a textual description of the class functionality. It is used during the building phase of the multiparadigm environment in order to create an organised and coherent documentation system.

The compilation and execution methods contain the machinery needed to implement the import and export call gates described in section 3.3.13.

A nice example of a paradigm class is the logic programming paradigm realised as Prolog compiled into Warren abstract machine instructions [War83]. On this case the *class state* variable contains the heap, stack and trail needed by the abstract machine. In addition, the *compilation method* is the compiler translating Prolog clauses into abstract machine instructions, the class initialisation method is the code initialising the abstract machine interpreter, while the execution method is the interpreter itself.

Inheritance

In object-oriented programming inheritance is used to provide a class with all operations of its superclasses, while allowing its subclasses to use the operations defined by that class. In our approach to multiparadigm programming inheritance is used to bridge the semantic gap between code written in a given paradigm and its execution on a concrete architecture. We regard the programming paradigm of the target architecture as the *root class*. If it is a uniprocessor architecture it has exactly one object instance, otherwise it has as many instances, as the number of processors. The *execution method* is implemented by the processor hardware and the *class state* is contained in the processor registers. The *compiled code* and *module state* variables are kept in the processor's instruction and data memory respectively.

From the root class we build a hierarchy of paradigms based on their semantic and syntactic relationships. Each subclass inherits the *methods* of its parent class, and can thus use them to implement a more sophisticated paradigm. This is achieved because each paradigm class creates a higher level of linguistic abstraction, which its subclasses can use.

3.3.7 General System Structure

Based on the ideas presented on the previous sections we can now build up the general system structure of a multiparadigm system. This structure must satisfy the flexibility, structure, and efficiency requirements outlined in sections 3.3.1, 3.3.2, and 3.3.3. It will be used as the design guide for designing and implementing multiparadigm programming environment generators, multiparadigm programming environments, and multiparadigm applications.

In general a multiparadigm system consists of a number of modules. Code written in different paradigms is put into separate modules. These are separately compiled by

the appropriate paradigm compilers. The paradigm classes are structured in a class hierarchy and are used to encapsulate the properties of each paradigm.

3.3.8 Separate Modules

The smallest linguistic unit that can be used to program in a paradigm, is a module. We loosely define a module as a part of a program that implements some functionality and interfaces to the rest of the program through a set of *import* and *export* declarations. These define the functionality implemented and the functionality required from other modules. Modules can be handled independently in the various phases of program development such as editing, compiling and linking.

A module is, in our opinion, the most appropriate unit for writing code in a given paradigm. The unit granularity stands half-way between that of a single statement, and that of a whole process. A statement, as a paradigm division unit, is too tightly bound to the imperative paradigm to be a general purpose unit. Furthermore, the syntactic differences between the different paradigms would make it impossible to satisfy the requirements F1 and S1. In addition, the name spaces of the different paradigms would interfere, making it difficult to satisfy S4. A process, as a paradigm division unit, would satisfy F1, S1, and S4, but would not satisfy (on a standard operating system) E1, because of the context switching times. Using lightweight processes would be a viable option; for the purposes of this discussion, we regard them as modules.

3.3.9 Separate Compiler for Each Paradigm

As the code in each module is related to a single paradigm, it can be compiled by a special compiler for that paradigm. This satisfies F1, because the compiler can be designed to accept the syntax notation most appropriate for that paradigm, and also S1, because the compiler need not be able to handle syntax notations of other paradigms. For these reasons, existing compilers could be used (using inheritance as explained in a following section), thus satisfying F4. This also satisfies E2 as the use of existing tools cuts development time, and possibly E3 assuming that high quality tools are used.

3.3.10 Class and Object Encapsulation

Each programming paradigm forms a class and each module written in that paradigm is an object of that class. As the execution method of each class is separate from the other paradigm classes, different execution models are supported, satisfying F3 and S2. The execution state of each paradigm is also encapsulated in the form of class and instance variables, preventing interference and satisfying S4. Finally, S5 is satisfied by providing a documentation method for every class.

3.3.11 Tree Class Structure

The classes in a multiparadigm programming environment are structured like a tree. At the root of the tree lies the target architecture. This means that it is *possible* to map any paradigm into it, without any intermediate efficiency burdening agents[5]. Thus F2

[5] In which case of course the tree degenerates into a linear list hanging from the target architecture. In practice a compromise is reached between implementation efficiency and cost.

and E3 are always satisfiable. The paradigms most related to the target architecture are the first subclasses that are layered on top of the root class. Other paradigms that are related to the new paradigms, exposed as "target" architectures, can now be built on top of them. By using subclassing, an existing paradigm can be extended or modified and, in this way E2 is satisfied. Existing tools can be used, satisfying F4, by matching their output to one of the classes available, or by creating a special class to interface their output.

3.3.12 Multiparadigm System Implementor vs. Multiparadigm Application Programmer View

We must note at this point that the class hierarchy is not visible to the multiparadigm application programmer. The hierarchy is useful for the multiparadigm programming environment implementor, as it provides a structure for building the system, but is irrelevant to the application programmer, who only looks for the most suitable paradigm to build his application. This is consistent with the recent trend in object-oriented programming of regarding inheritance as a *producer's mechanism* [Mey90], that has little to do with the end-user's use of the classes [Coo92].

3.3.13 Paradigm Inter-operation

In the following sections we will outline how code written in different paradigms[6] can be used to form a single system. The two main problems to be solved are the transfer of *control*, and the transfer of *data* between the paradigms. Both types of transfer occur when the boundaries between the paradigms are crossed. Control transfer passes the execution control to the runtime machinery of another paradigm, and data transfer passes data between the paradigms.

3.3.14 Control Transfer

Control transfer between the paradigms follows the control transfer conventions of their superclass. This recursive definition is followed until we reach the root class, the target architecture. Using the conventions of the parent paradigm ensures that no unexpected interactions occur. By unexpected interactions we mean implicit and unintended control transfer from one paradigm to the other. The subclasses are coded using the features and caveats of the parent class; this ensures that no unexpected interactions occur. We will attempt to clarify this statement by three examples:

Non-Preemptive Von-Neumann Target Architecture

The basic control transfer mechanism used is the explicit *procedure call*. Implicit calls between other paradigms will be encapsulated within their respective classes and therefore the paradigms remain isolated.

[6] We will use the term paradigm to denote 'paradigm implementation'.

Threads Subclass

We assume now the existence of the *thread* programming subclass with primitives that allow the creation of multiple processes. Since this subclass was coded using the mechanisms of the parent paradigm, the threads created will be non-preemptive and therefore all implicit control transfers will happen *within* the threads paradigm. Therefore no implicit control transfers can occur between paradigms, other than subclasses of the *threads* paradigm. These subclasses will of course be coded to anticipate such control transfers.

Event-Driven Target Architecture

In an event-driven target architecture, all paradigm compilers have to be able to cope with non-deterministic control transfers. For this reason the paradigms provided will — by definition — be able to cope with such transfers. A straightforward way to implement such a system would be to disable interrupts when paradigms that are unsuitable for such an environment are executed, a higher quality implementation would provide a better solution.

Parallel Architecture

Here again the target architecture imposes some stringent requirements regarding synchronisation, memory sharing, message passing etc. All these requirements are inherited, and must be dealt, by all paradigm subclasses.

3.3.15 Data Transfer

Data transfer deals with the way data is transferred between paradigms. Each paradigm can have a different representation for data, or data of a particular paradigm-specific type. This problem is again solved within the class hierarchy. Every paradigm defines mechanisms for transferring data to and from its superclass in the most efficient and intuitive way. When data is transferred between two paradigms it is first converted from the first paradigm, step-by-step, to the format of their common superclass and then converted back to the format of the second paradigm. This is not as inefficient as it sounds for two reasons:

1. Data formats usually do not vary a lot between efficient implementations.
2. We found that most of the transfers are between closely related paradigms.

3.3.16 Paradigm Inter-operation Design Abstraction

The two types of paradigm inter-operation are accommodated by the abstraction of a *call gate*. A call gate is an interfacing point between two paradigms, one of which is a direct subclass of the other. We define two types of call gates, the import gate, and the export gate. In order for a paradigm to use a service provided by another paradigm (this could be a procedure, clause, function, rule, or a port, depending on the other paradigm)[7], that service must pass thought its import gate to be mapped

[7] Static data can be passed between different paradigms using access functions.

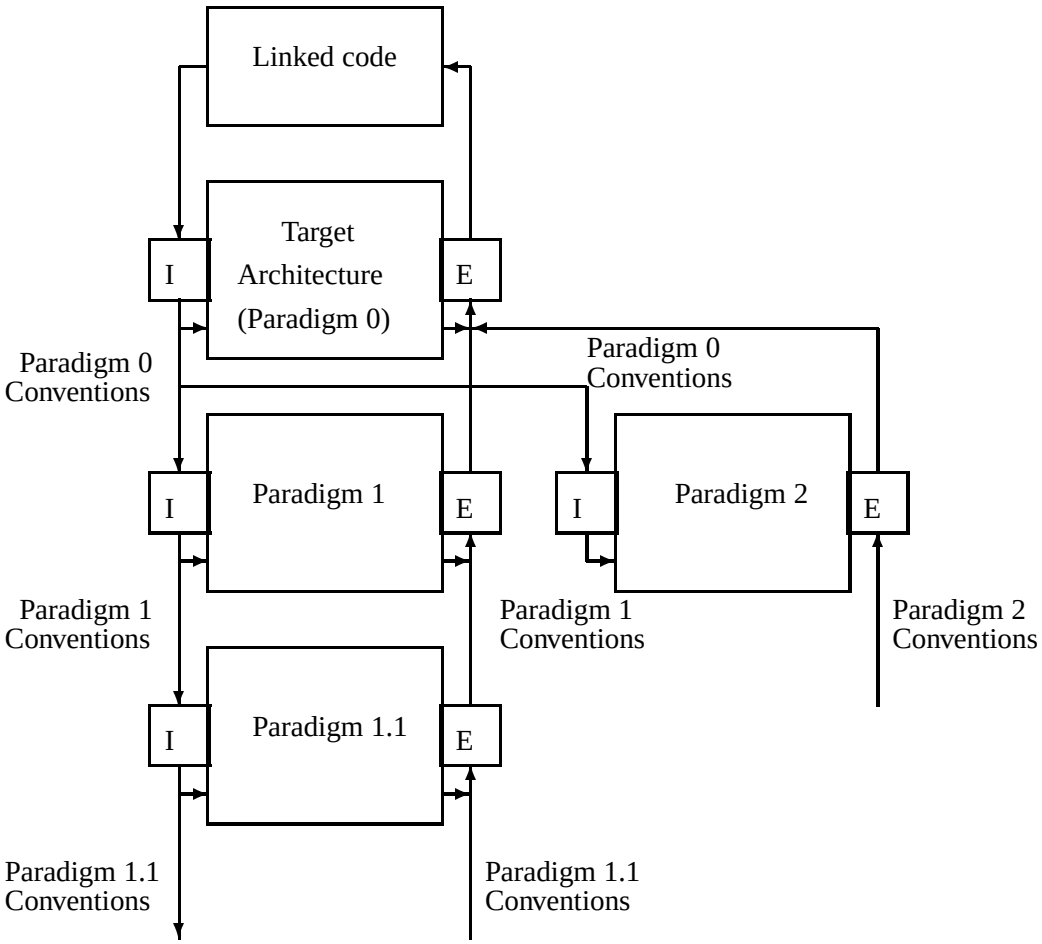

Figure 3.7: Paradigm inter-operation using call gates

from the conventions of the paradigm's superclass to the conventions of that paradigm. Conversely, on the other paradigm the same service must pass through its export gate to be mapped from the conventions of that paradigm to the conventions of its superclass. The call gates are design abstractions and not concrete implementation models. They can be implemented by the paradigm compiler, the runtime environment, or a mixture of the two. The role of the call gate is to implement the control and data transfer functions of the two paradigms. Each paradigm provides an import and an export gate and documents the conventions used and expected.

The input of the export gate, and the output of the import gate follow the conventions of the paradigm, while the output of the export gate, and the input of the import gate, follow the conventions of the paradigms' superclass. The target architecture paradigm combines its import and its export gate using the linked code as the sink for its export gate and the source for its import gate. Figure 3.7 provides an example case. Call gates make the paradigm inter-operation transparent to the application programmer, satisfying requirement F5.

3.3.17 Paradigm Inter-operation Limitations

In theory, any two paradigm implementations can communicate without any limitations on the type of parameters that can be exchanged. Such communication could include the creation and passing of arbitrary data entities from one paradigm to the other — even if one of the paradigms does not support such entities — the modification of the execution strategy of one paradigm, or the invocation of code in one paradigm within an arbitrary execution environment. This theoretical paradigm inter-operation can be implemented, if the following two conditions can be satisfied:

1. Each paradigm has a construct that allows it to access and modify the data space, of the process that is executing the code written in the other paradigm.

2. Full documentation is available on the internal workings of each paradigm, the compilation strategy of its compilers, and the source code of the program being executed.

In our approach, all paradigms are executing as a single process image, therefore the first condition is always satisfied. The second condition must depend on the paradigm implementation used. If the paradigm implementation is a black box, then the paradigm inter-operation facilities that can be offered will be defined by the inputs and outputs of that black box. In most cases however, the breadth of paradigm inter-operation facilities offered will depend on the weighting of the following *language design related* factors:

- their usefulness to the multiparadigm application designer and implementor,
- their implementation difficulty,
- efficiency considerations,
- their effect on the reliability and understandability of the multiparadigm program, and
- semantic distance of the inter-operation facility between the two paradigms.

Consequently, the range of paradigm inter-operation facilities offered, is mostly a design decision of the multiparadigm programming environment implementor.

3.4 Multiparadigm Environment Generators

Multiparadigm programming environment generators are based on, and support the design we outlined in section 3.3. Their existence can induce a multiparadigm system designer to adopt our approach and avoid ad-hocery. Furthermore, the development time and implementation errors can be reduced. We hope that such systems can even be used to create specialised paradigm classes for one specific application. In this way we can provide a solid software engineering foundation for the concept of “little languages” [Ben88, pp. 83-100, 128-131].

3.4.1 Requirements

A multiparadigm programming environment generator must support the design based around objects. We consider the following services to be essential:

- allow the high level description of paradigms as classes,
- convert a system described by paradigm classes into a multiparadigm programming environment, and
- provide support for using existing tools.

The functions of the resulting multiparadigm programming environment can be divided into four areas:

1. paradigm-specific compilers,
2. paradigm combination support,
3. documentation, and
4. run-time support.

Ideally support for all these areas should be provided by the generator.

3.4.2 General Structure

Based on the requirements outlined in the previous section we can now describe the general structure of a multiparadigm programming environment generator. Such a generator will consist of a paradigm description compiler, which takes the class description of a paradigm and creates its compiler, documentation, and run-time support. Other, specialised tools, are used before the paradigm compiler to assist the process of creating paradigms using existing compilers. Generic run-time support must be provided where possible, to implement functionality that will be required for all multiparadigm programming environments: this avoids duplication of effort among multiparadigm programming environment implementors. Finally a 'system wrapper' combines all the tools, documentation and run-time support objects into a single distributable system. The combination of these components can be seen in figure 3.8. In the following sections we will describe the functionality of each of these components in more detail.

3.4.3 Paradigm Description Compiler

The paradigm description compiler takes a class description file containing the elements enumerated in 3.3.6 and converts it to the appropriate representations. Some of the class methods and variables exist during the compilation of a module, others during its run-time; the paradigm compiler must be able to handle both cases. Subclassing and inheritance are important parts of our design; the paradigm compiler must support them. The source and the target code representations of the entities that the paradigm compiler will be required to deal with are dependent on decisions made by the multiparadigm environment generator developer and the target system and will not be dealt with in this chapter. For example, the paradigm compiler can be realised as a

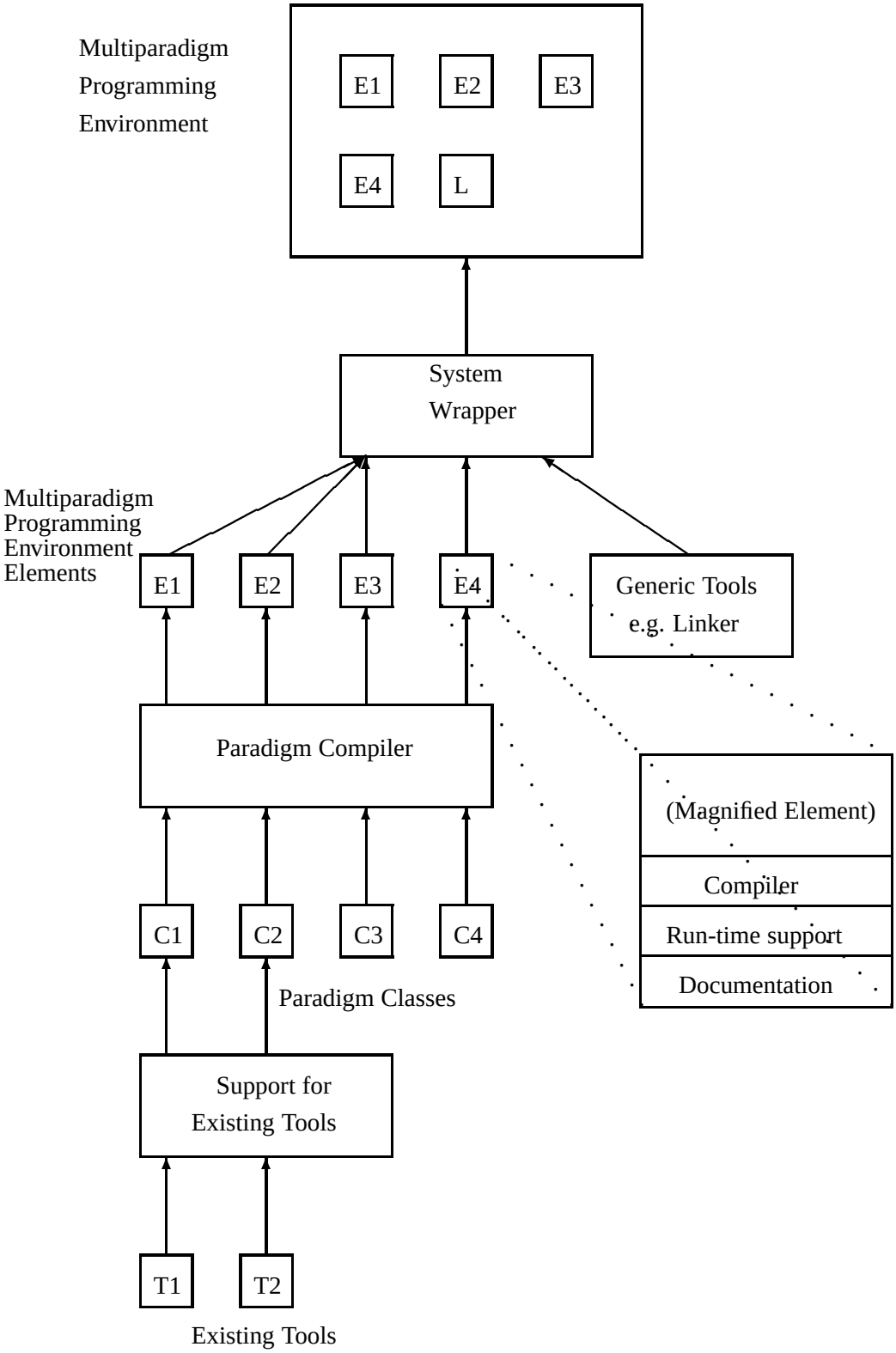

Figure 3.8: General structure of a multiparadigm environment generator

function of an integrated environment, where paradigms are interactively described by filling appropriate slots (e.g. an implementation extension to VOSE [FKG90, NF92], or as a tool dealing with textual descriptions of paradigms. In the first case the target code would be internal descriptions of the paradigm methods and hypertext links to the documentation, in the second the compiler would generate new tools, formated manual pages and run-time support files.

3.4.4 Support for Existing Tools

High quality and efficient implementations of paradigms are costly to implement. For this reason a multiparadigm environment generator must allow existing systems to be used wherever possible. This means that mechanisms must be provided to make such systems conform to the conventions expected by the overall system design. These mechanisms must transform independent systems into encapsulated paradigm objects. Functions of these systems implemented as tools, or commands within an integrated environment, must be converted to methods of the paradigm class. Potential resource requirement clashes (in areas such as name-space, memory usage, and non-reentrant functions) must be identified and dealt with. The format in which the elements of the implementation are provided could be incompatible with the format required by the generator; conversion tools should be provided. The actual support provided will depend on the existing implementations, their common format, problems that are often encountered, and the relative cost factors between re-use and re-implementation.

3.4.5 Generic Run-time Support

Many paradigms need run-time support. All classes contain a class initialisation method and all objects an object initialisation method. These methods must be called in the appropriate order during the startup sequence of a multiparadigm application. This can be complicated (as it deals with system-dependent process initialisation code). This is functionality that will be needed by all multiparadigm programming environments. In order to avoid duplication of effort among the multiparadigm environment implementors, to ensure a correct and portable implementation, and to simplify the task of implementing a multiparadigm programming environment, this functionality can be provided by the environment generator. Environment implementors can then, directly plug it into their systems with minimum additional effort. The design and implementation of these functions is highly dependent on the target system. In an open object-oriented integrated environment (such as Smalltalk) it could be as easy as binding an additional method to the process class constructor. In a file-based system using a link-editor it could require modifying the link-editor, the system startup code, or providing a kernel within which multiparadigm applications are run.

3.4.6 System Wrapper

One further area where the multiparadigm environment implementor can be helped is that of the final packaging. Multiparadigm environments could consist of tens if not hundreds of separate elements, such as specialised tools, libraries, compilers, run-time

support modules, documentation files etc. The process of organising these into a system distribution can be automated thus minimising potential errors and relieving the implementor of this mundane and error-prone activity. The actual design and implementation depends very much on the target system, the distribution method and the range of activities that will be automated. These can reach from a simple system installation procedure in specified directories, to the creation of a distribution tape with installation instructions and printed manuals. Some explorative, rapid-prototyping development environments (such as Smalltalk), make it difficult to separate the product from the environment where its development has taken place. In such cases additional support will be required in order to make this distinction and create an isolated product.

3.5 Multiparadigm Programming Environments

Multiparadigm programming environments are based on the design we outlined in section 3.3. Thus they consist of a number of paradigms organised into a class hierarchy. Each paradigm is a separate class, modules written in that paradigm are instances of that class. The design and implementation of multiparadigm programming environments is based around an environment generator. In the following sections we will describe some features of the design and implementation that are general and do not depend on a particular environment. In section 4.4 we describe the design of a multiparadigm programming environment based on MPSS — an environment generator — and in section 5.3 its implementation.

3.5.1 Delegation of Features Using Subclassing

The object-oriented approach we have adopted supports the powerful notion of *subclassing*: subclasses inherit the methods of their superclass and can provide additional ones, or substitute the existing ones. In the domain of multiparadigm programming this abstraction can be used very effectively by delegating features common among a number of paradigms to a superclass of those paradigms. Thus code duplication is avoided and more effort can be spent in implementing a high quality solution for the common features. When designing a multiparadigm environment it is essential to try to organise the paradigms in a class tree recognising common features that they share. For example most paradigms have a notion of dynamic memory, a class can be created to provide this feature for these paradigms. Two subclasses can be derived from that class, one for programmer-controlled memory allocation and deallocation and another for automatic garbage collection. As another example a simulation paradigm and a communicating sequential processes paradigm could both be subclasses of a coroutine-based paradigm.

Subclassing is not only used for the run-time class execution methods. Syntactic (i.e. compile-time) features of paradigms can be captured with it as well. Many constraint logic languages share the syntax of Prolog, thus it is natural to think of a constraint logic paradigm as a subclass of the logic paradigm providing its own solver method, and extension to the Prolog syntax for specifying constraints.

A paradigm class tree based around these examples is shown in figure 3.7. Assume that a module written in *paradigm 2* imports a facility implemented in *paradigm 1.1*.

The module written in *paradigm 1.1* will export that facility (using the syntax and semantics appropriate to *paradigm 1.1*) to its superclass (*paradigm 1*) throught its export gate, thus converting it to the data types and calling conventions used by paradigm 1. *Paradigm 1* will again pass it through its export gate, converting it to the conventions used by *paradigm 0*, the target architecture. (For example the calling conventions of the Unix system, can include the passing of parameters through a stack frame, and the naming of identifiers with a prepended underscore.) In this form the facility will again be imported from the pool of linked code by *paradigm 1* and made available to its subclasses using its conventions. The facility can then be imported and used by *paradigm 2* which can understand the calling conventions of *paradigm 1*. Although during the path described the facility crossed three paradigm boundaries, in all cases each paradigm just needed to be able to map between its calling conventions and data types and those of its superclass.

3.5.2 Using Inheritance

The fact that a subclass inherits all the methods of the superclass can be put into good use by having the superclasses performing the bulk of the work and each paradigm class perform only its specialised task. Thus a paradigm class method can often deal with the special features of the paradigm and pass the rest to its superclass. This approach, implemented in an ad-hoc way, is quite common in the Unix environment, both in programming and in text processing. In programming there are separate tools that perform macro expansion, regular expression recognition, BNF parser generation. In document processing this approach is even more apparent with a document processed with as many as six tools, each of which understands a specific paradigm (e.g. tables, chemical molecules, graphs, equations), and passes the rest to the next tools in the pipeline [Ker89]. We formalise this approach with the idea of methods inherited by the superclasses and thus provide a structured framework for its wide use.

3.5.3 Implementation Approaches

There are many ways in which a paradigm can be implemented. All of them are supported by our design. In this section we describe them, together with their advantages and drawbacks.

Compile to Target Architecture

The classic way to implement a paradigm is to compile it into target architecture machine code. At run-time these instructions are directly executed by the processor. This method is easy to understand and can result in very efficient implementations. However it requires a lot of effort on the part of the paradigm creator and good knowledge of the target architecture and optimisation techniques. It is obvious that such an implementation of a paradigm will not be portable among different target architectures. If exactly one paradigm will be implemented using this approach (often by making use of existing tools) and other paradigms will use it as a way to access the target architecture, the a good compromise between efficient execution and ease of portability will be achieved.

Interpret

Interpreting some form of the paradigm source code, has the opposite characteristics. The implementation of the paradigm will be relatively straightforward, but the resulting execution speed will be slow. In many cases the memory required by this approach may be a lot less than the memory required by directly compiling to target architecture. The resulting paradigm compilation and execution methods will be portable among all target architectures. This approach may be suitable when initially prototyping a paradigm. At that phase it is important to receive feedback on the paradigm features as they are used, and to be easy to modify the paradigm. Once the paradigm design has settled a more efficient implementation can be chosen. Furthermore there exist cases where the execution speed of the paradigm is not of vital importance (e.g. within the user interface loop). In those cases the advantages of this approach may outweigh its slow speed.

Compile to Superclass

An implementation strategy that lies between the two methods described above is that of compiling to the superclass of a paradigm. Using this strategy the source code of a paradigm is compiled into the source code of the superclass paradigm. Then the compiled method of the superclass paradigm is invoked in order to finish the compilation. If the paradigms are closely related then the compilation effort is likely to be minimal. A good implementation of the superclass will result in a good implementation of the paradigm. In addition, future improvements in the implementation of the superclass will directly positively affect the paradigm quality. This approach is portable, since it only depends on the superclass. Using this approach a whole multiparadigm environment is often structured as a single paradigm class that compiles to the target architecture and a tree of other subclasses that directly, or indirectly compile to that class.

3.6 Multiparadigm Programming Applications

Although this section is preceding the section on the design and implementation of a concrete multiparadigm application (4.5, 5.4), some of its contents are based on lessons learned from that exercise.

3.6.1 Structure

The tree class structure that was apparent to the multiparadigm programming environment environment developer and to the multiparadigm programming environment implementor, is not visible to the multiparadigm programmer. All the paradigms appear as a flat structure and there are no restrictions on how they are mixed and matched. This flat structure, achieved by the use of the call-gates, encourages the selection of the most suitable paradigm for each component. Furthermore the system structure is determined by the actual interrelations of the system components and not by artificial constraints placed by the paradigm tree class structure.

3.6.2 Design

Multiparadigm programming environments developed using our method, will offer access to multiple paradigms via call-gates. Consequently a single-entry, single-exit interface to foreign paradigm code is offered and therefore structured programming design techniques [ES89] are the best suited for designing a multiparadigm application. After the main structural components of the system have been identified, the application developer must examine each one and determine the appropriate paradigm for that component. These components are then designed using a design strategy appropriate for each paradigm. Other paradigms lend themselves to the application of formal method design techniques, other favour experimentation and prototyping, and others massive code reuse. A design environment supporting multiple design methods such as VOSE [FKG90, NF92] can be used in this phase.

An important design maxim that we discovered is that of always using the most appropriate paradigm. In some cases using the best paradigm may only offer minimal advantages compared to implementing that part of the system within another paradigm already used. In a high quality multiparadigm programming environment the cost of using an additional paradigm, both at design and implementation time and at run-time will be negligible. We found that using the appropriate paradigm often resulted in important fringe benefits that we had not anticipated during the original design of the system. For example during the implementation of the *integrator* we decided to use the functional programming paradigm for the numeric evaluation of definite integrals. The same code could probably be written in an imperative style, by using a cookbook routine [PFTV88]. The method used was potentially more efficient and parameterisable. More importantly, we discovered when we decided to add function plotting, that because function evaluation was already coded, evaluating a function was a matter of a single function call.

3.6.3 Implementation

During the implementation phase of a multiparadigm application the implementation methods, techniques and tricks of each paradigm will be used. The existence of multiple paradigms provides the ability to code a part that is particularly well suited for a single paradigm in that paradigm. At later phases of the implementation the system can be instrumented to gather profile data and performance bottleneck parts can be re-implemented in more efficient paradigms.

For larger systems, it may be helpful to implement a custom paradigm, to capture a specific part of the system in the most appropriate notation. This will require the application developer to wear the hat of the multiparadigm environment developer, but may pay back in terms of decreased implementation and maintenance efforts.

3.7 Summary

In this chapter we described our approach to multiparadigm programming. We first decomposed the problem of multiparadigm programming into the four areas of, application development in multiple paradigms, design and implementation of multiparadigm languages, support environments for creating such languages, and a general approach

covering the previous three areas. Then, starting from multiparadigm applications, we examined, how these affect the design of multiparadigm languages, and those in turn, the design of multiparadigm environment generators. Based on those requirements, categorised in the areas of flexibility, structure, and efficiency, we developed the general structure for multiparadigm environments. This is based on treating paradigms as object classes, with paradigm inter-operation handled by the *call-gate* abstraction. Finally, we examined in detail how relying on those principles, multiparadigm environment generators, programming environments, and applications are to be structured.

Chapter 4

System Design

In this chapter, we present the design of the systems implemented, in order to demonstrate the viability of our approach. We outline the objectives of our design, present the relationship of the constituent system parts, and describe each part in detail. In the case of tools, we describe their input and output; in the case of languages we define their lexical elements, syntax, execution semantics, and their built-in library functions. Finally, we outline the design of our prototype multiparadigm application.

4.1 Objectives

The objective of this design and implementation exercise, is to try our ideas outlined in the previous chapter on a real system. Our main goals are:

- provide an experimentation platform, and experiment with multiparadigm programming environment generators, multiparadigm programming environments, and multiparadigm applications,
- locate and rectify any missing details in our approach, and
- hopefully, show the viability of our approach.

We have to stress here, that we are designing and implementing *prototype* systems. They are robust and useful enough to handle the outlined objectives, but will certainly need more work, if they are to be used in real applications.

4.2 System Structure

In order to verify our design objectives outlined in the previous section we designed and implemented three distinct systems corresponding to the bottom three tiers of the hierarchy described in section 3.1. The inter-operation of the three systems forms a concrete version of the top tier i.e. the general structure. Their relationship is illustrated in figure 4.1. The three systems we design are:

1. A multiparadigm programming environment generator, MPSS. This is a tool suite, suitable for designing and implementing multiparadigm programming en-

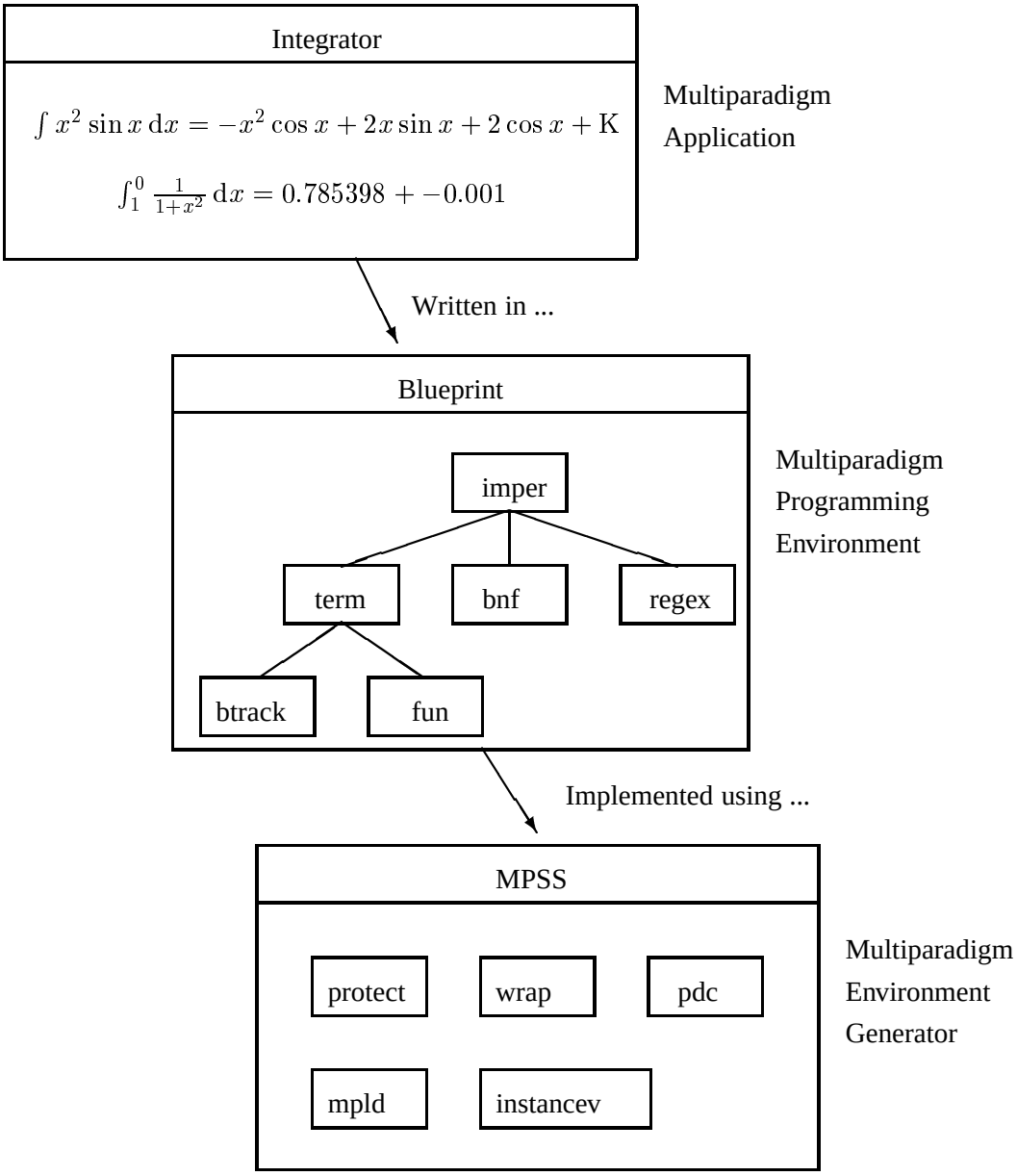

Figure 4.1: Entity-relationship diagram of the implemented systems

vironments. This system corresponds to the general approach described in section 3.3. We will outline its design in section 4.3 and its implementation in section 5.2.

2. A multiparadigm programming environment, *blueprint*, designed and implemented using MPSS. It offers six programming paradigms: imperative, rule-rewrite, BNF grammars, regular expressions, lazy higher-order functions and logic programming. The approach corresponding to this system is outlined in section 3.4. The design will be described in section 4.4 and the implementation in section 5.3.

3. A multiparadigm application, *integrator*, a program allowing the symbolic and arithmetic evaluation of integrals. Its design and theory of operation will be described in section 4.5 and its implementation in section 5.4.

4.3 MPSS: A Multiparadigm Environment Generator

MPSS is a multiparadigm programming environment generator, providing the support functions described in section 3.2.3 (page 44). In the following sections we present the design of MPSS. After a discussion of design alternatives, we outline the general structure of the system, then we detail the design of each of its parts.

4.3.1 Design Alternatives

The functionality of an environment generator can be provided in two different ways: either as a tool suite, or as an integrated environment. The tradeoffs between the two approaches are the following:

User friendliness: An integrated environment will be easier to use, as all its parts will be integrated in a single integrated package. A tool suite, requires the user to familiarise himself with every tool, and the way the tools can be combined.

Consistency, and process enforcement: The integrated environment can enforce the development process, since it will only provide support for the actions the development process describes. In a tool suite the process can be overridden, by using different tools, or combining the existing tools, in non-standard ways.

Flexibility, and extendibility: The tool suite approach provides a more flexible system, since unanticipated requirements can be met by using the tools in different ways. In an integrated environment, problems that were not anticipated in the environment's realisation, can be impossible to solve.

Integration with development platform: Our development platform, Unix, is based on tools. An integrated environment would be awkwardly integrated with the rest of the system, whereas a tool suite would be perfectly matched with the underlying system philosophy.

Implementation difficulty: Implementing an environment is likely to be a lot more difficult, than implementing a set of tools, as Unix provides a lot of support for

tool implementation, by almost no support for integrated environment implementation.

We chose to implement the system as a tool suite, because we found flexibility, extendibility, integration with the development platform, and implementation ease, much more important than process enforcement, and user friendliness. Implementing multiparadigm programming environments is not a trivial task. We expect the users of our generator to know what they are doing.

4.3.2 General Structure

MPSS consists of a number of separate tools that support the implementation task of a multiparadigm programming environment. This is consistent with the Unix philosophy of small individual tools that can be combined with each other [Rit84].

The design philosophy behind MPSS is that of paradigm classes. Every programming paradigm forms a class, with the target architecture paradigm being the root of the class structure tree. The interaction of the different MPSS tools can be seen in figure 3.8, page 63. For every paradigm, the implementor provides a paradigm class description file, that defines the paradigm class. This is then compiled, by the paradigm description compiler provided by MPSS, into a compiler for that paradigm and its manual page. The multiparadigm programming environment (e.g. *blueprint*) user, can use that compiler, to convert source code from the given paradigm into object code. When the source code of all paradigms has been compiled a special link editor, the *multiparadigm link editor* can be invoked to link all the paradigm objects, and associated support libraries together into a runnable system. Two additional tools detect and protect the private variables of each class.

4.3.3 Paradigm Description Compiler

The paradigm description compiler (*pdc*) compiles a paradigm class description file into a compiler for that paradigm and its manual page. Paradigm description files are text files, containing definitions for class variables and methods. Possible variable and method definitions are the instructions for compiling paradigm code, variables that need to be protected, the run-time support library name, and the paradigm compiler's manual page. Some of the variables must be defined for every paradigm, others can be optionally defined if relevant. The class variables currently supported are listed in table 4.1. In addition, the multiparadigm system builder can introduce more variables according to the structure of the system. For example, typechecking support can be added in the form of additional class methods.

Lines beginning with a # character are regarded as comments and ignored. Variables and methods are defined by the method or variable name starting at the beginning of a line, followed by a colon, followed by the value. The value of the variable, or method, can extend to multiple lines, as long as these lines start with a whitespace character. Within the file the programmer can refer to a variable value by using its name mapped to uppercase characters within curly brackets, preceded by a dollar sign, e.g. `${SELF.TOOL}`. The variable `SOURCE` is automatically set within the code generated by the paradigm compiler to reflect the name of the filename source parameter defined by the multiparadigm programming environment user.

Variable name	Meaning
SUPER	Superclass name (TARGET for root class)
TOOL	Name of the compiler to generate
EXTENSION	Paradigm source filename extension
COMPILE	Compilation instructions
INSTANCEV	Class member instance variables
SYSTEM	Multiparadigm environment name (e.g. blueprint)
SYNOPSIS	Summary of the paradigm's operation
DESCRIPTION	Paradigm documentation
SOURCE	User filename parameter

Table 4.1: Class variables supported by the paradigm compiler

The paradigm description compiler performs variable substitution for the variables of the class and its superclass using the usual `SELF.` and `SUPER.` variable prefixes. Using an undefined variable within a class description will produce a compilation error. A sample paradigm description file, is listed in figure 4.2.

4.3.4 Instance Variable Detection

MPSS offers the capability of implementing a the paradigm using existing tools and facilities. One major problem when using such tools is that of *name-space pollution*. Some tools create code with global identifier names (function or variable identifiers) that cannot be specified or changed by the user. Such code can be used only once within a given program. This is for example, the case with the *lex* [Les75] and *yacc* [Joh75] utilities. Having such tools as part of a multiparadigm environment is not possible, as they can only be used to create a single instance of a code object. The instance variable detection tool, *instancev*, when run on the object code that such a utility generates, prints a list of the global variables that are defined in it. These can then be listed in the *instancev* section of paradigm class description file to be automatically converted to private instance variables.

4.3.5 Private Variable Protection

Many of the paradigm translators, either because they are built based on existing utilities, or because of the translator design, or due to features of the implemented language, will contain global variables or procedures that should be private to the class instance. *Protect* is a tool that generates unique identifier names. The names of the private variables and procedures are listed in the *instancev* section of the paradigm class description file. Given this name list, *protect* will create regular expressions, that when applied to the assembly language output of the target architecture paradigm will automatically convert them to class private variables by prepending to them the name of the module in which they occur. A limited version of this tool, *yyhide* [Gli91] is part of the Andrew Toolkit [PHS+88]. *Yyhide* only deals with output generated from yacc and lex, while our tool can handle output from any program generator.

```
# Paradigm description file for the backtracking paradigm
#
# $Id: design.tex,v 1.2 1993/05/03 15:33:55 dds Exp dds $

# Environment name
system: blueprint

# Paradigm name
name: btrack

# Superclass name
super: term

# Source file extension
extension: pb

# Target tool name
tool: btrackpc

# Instance variables and functions.  These variables and functions are
# duplicated across instances of the paradigms runtime machinery
instancev: rules_1 mpss_needlib_lbtrack_0 btrack_3
        tryall_5 solve_4 import_unify_3

# Compilation instructions
compile:
        rm -f ${SOURCE}.${SUPER.EXTENSION}
        bt2term ${SOURCE}.${SELF.EXTENSION} >${SOURCE}.${SUPER.EXTENSION}
        ${SUPER.TOOL} ${SOURCE}.${SUPER.EXTENSION}

# Runtime support modules
runtime: lbtrack.o

# Paradigm description for automatic manual creation
synopsis: backtracking and unification

description:
        \fIBtrack\fP is a pradigm for handling problems that can be
        solved using backtracking and unification.  The \fIbtrack\fP
        programming style intentionaly resembles that of Prolog.  Many
        pure Prolog programs can be ported to \fIbtrack\fP without change
```

Figure 4.2: Sample paradigm description file

4.3.6 Multiparadigm Link Editor

Once the code of every paradigm is converted into target architecture object code, the object modules must be linked together in order to create the final running system. This is handled by the *multiparadigm link editor*, *mpld*. In addition to the duties of the system link editor (resolving references between the modules, and creating executable code) [PW72], the multiparadigm link editor has three additional duties to perform (see also 3.3.6):

- initialise every paradigm class by calling its class initialisation method,
- initialise every separate module in each paradigm, by calling its initialisation method, and
- link in the runtime machinery of each paradigm, by instructing the system linker to include the appropriate libraries.

The class initialisation must be performed before the instance initialisation, as the second one might depend on an initialised class instance. Furthermore initialisation must proceed from the top to the bottom of the class hierarchy, as a class might depend on its superclass for its initialisation.

4.3.7 System Wrapper

A multiparadigm programming environment will typically consist of a number of translators, libraries, preprocessors, manual pages and other auxiliary files. An *mpss* tool, the *system wrapper* will find the parts of the programming environment and organise them into a suitable distribution format. This ensures that the system distribution and its updates can be created reliably and with the minimum effort.

4.3.8 Building a Multiparadigm Programming Environment

We recommend the following approach when building a multiparadigm programming environment using MPSS:

1. Decide the programming paradigms that will be supported, and the names that will be given to their classes.
2. Organise the paradigms into a tree class structure. Paradigms that are extensions of another paradigm should have that other paradigm as their superclass. The target architecture paradigm must be at the root of the class tree. If two paradigms share an important characteristic, try to abstract that characteristic into a separate class which will be the common superclass of the two other classes.
3. Every paradigm needs a class definition file. This, as a minimum, must define:
   - the name of the paradigm,
   - the name of its superclass,
   - the extension of the filenames containing source code for that paradigm,
   - the name of the compiler for that paradigm and,

- instructions for compiling code written in that paradigm into code of the target architecture. These instructions can simply translate the code to the code of the paradigm's superclass and invoke the paradigm's superclass compiler to complete the operation. Interpreter-based implementations must provide at this point any preprocessing needed for the interpreted source code files.

4. For every paradigm create a translator that transforms the code of that paradigm into code of the target architecture. This is normally done by translating into the code of its superclass and invoking the superclass compiler via `SUPER.COMPILE`. The invocation code for that translator is given in the `compile` section of the paradigm class definition file.

5. Each paradigm must have a mechanism for interfacing to the paradigm of its superclass. The two facilities that must be provided are the ability to use functionality provided by the super-class (import call gate), and the ability to make it possible for its super-class to use the functionality *it* provides (export call gate).

6. If an existing tool is used for translating the paradigm code use *instancev* to find out which variables must be made private for every class member instance.

7. If the code generated by the translation process contains some global variables that need to be private to a class member instance, list those in the `instancev` section of the paradigm class definition file.

8. Compile all class definition files to create the compilers for all paradigms.

9. Wrap up the system, using the system wrapper, for distribution or installation.

4.4 *Blueprint*: A Multiparadigm Programming Environment

The MPSS tool suite, described in sections 4.3 and 5.2 forms a system suitable for creating multiparadigm environment generators. *Blueprint* is a multiparadigm programming environment, built using the MPSS philosophy and tools. Its name is derived from the acrostical spelling of the paradigms provided[1], namely:

- **B**NF grammar descriptions (*bnf*),
- **l**azy higher order functions (*fun*),
- **u**nification and backtracking (*btrack*),
- **r**egular expressions (*regex*),
- **i**mperative constructs (*imper*) and,
- **t**erm handling (*term*).

[1] In order to find the name the Unix command "`grep b /usr/dict/words | grep u | grep l | grep r | grep i | grep t`" was executed. *Blueprint* was selected from the 29 words that matched the specification.

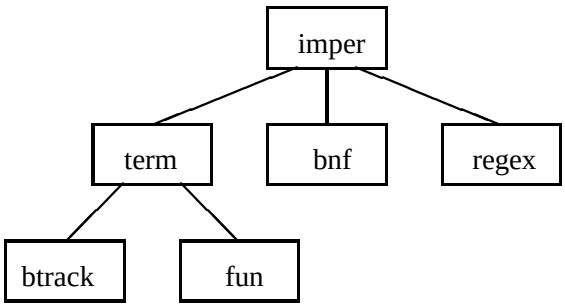

Figure 4.3: *Blueprint* class hierarchy

All these paradigms are provided in the form of *paradigm compilers*: tools that convert the code expressed in a given paradigm, into object code that can be linked and executed together with code from other paradigms.

In the following sections we present the design of the system and its parts.

4.4.1 Design Objectives

Blueprint was designed as an experimental prototype system in order to demonstrate the viability of our approach. Therefore our design was centred around the following objectives:

- realisation of a wide variety of diverse programming paradigms,
- provision of a non-trivial class hierarchy, including the abstraction of common characteristics in a special superclass,
- usage of all the features provided by MPSS and in particular an implementation involving a combination of existing tools, new ones, compilers, and interpreters,
- suitability for the implementation of a useful application, and
- ability to bootstrap the system in order to test and use it as much as possible.

4.4.2 System Structure

Blueprint is designed using the MPSS paradigm class hierarchy notion. The target paradigm is the imperative paradigm provided by the target architecture, which in our case are Sun SPARC computers. The paradigm classes that are implemented can be seen in figure 4.3. Terms are the natural data objects, for both functional and logic languages; the provision of the *term* class is based on this observation and, in addition, provides a practical vehicle for their implementation.

It is important to note, that the tree structure is only used in order to design and implement the system using MPSS. The structure is transparent to a programmer using *blueprint* who is presented with a flat structure of all the paradigms (figure 4.4).

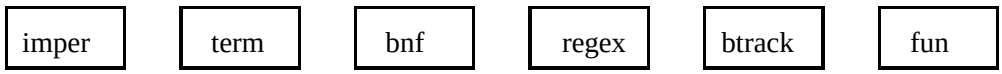

Figure 4.4: Programmer's view of *blueprint*

4.4.3 Imperative Paradigm

The *imper*, imperative construct, paradigm is used for interfacing with the operating system, building user interfaces, accessing the machine at a low level and in cases where efficiency is the first priority. Its syntax and semantics are exactly those of the C programming language [KR78, ANS89].

The paradigm is used for providing the entry function for the *blueprint* program. The entry function is called `mpss_main`. The usual `main` function used in the hosted C programming environments [ANS89, §2.1.2.2.1] should not be used as a duplicate definition error will appear at link time.

4.4.4 Rule-rewrite Paradigm

Term is a term pattern-matching rule-rewrite paradigm modelled after the *struct* programming language [Spi91a]. Its main data object is a *term* which can also be used to construct lists. Pattern matching is used to select a rule from a set of rules that comprise a *term* program. Once a rule is selected, its antecedents are recursively invoked. If one of them fails the next rule whose pattern matches is tried.

Term is suitable for implementing language translation systems. Its deterministic, first-order execution model makes it very efficient compared to higher order or nondeterministic languages, and therefore, a suitable vehicle for succinctly expressing algorithms that do not need the full power of a more advanced paradigm. In our experience, many of the programs found in logic or functional programming textbooks can be expressed in *term* without any important changes.

In the following sections we will describe *term* in more detail.

Language Elements

Term source is organised in files. C-style comments (matching `/*` and `*/` pairs) and white space are ignored when reading a file. The basic data structure of *term* is the term. A term can be an atom, a variable, or a composite term. Atoms can be one of the following:

1. Sequences of alphanumeric characters beginning with a lowercase letter or a dollar sign.

2. Arbitrary text enclosed within single quotes.

3. integers and floating point numbers — expressed as a sequence of digits or as a sequence of digits including a decimal point optionally followed by an exponent respectively.

Operator	Type	Binding	Precedence
$*$	infix	left	1
$/$	infix	left	1
$+$	infix	left	2
$-$	infix	left	2
$<$	infix	left	3
$>$	infix	left	3
$>=$	infix	left	3
$<=$	infix	left	3
$\backslash ==$	infix	left	3
$==$	infix	left	3
$= ..$	infix	left	3
is	infix	left	3
not	prefix	right	4

Table 4.2: *Term* operator list

Variables are written as sequences of alphanumeric characters starting with an uppercase letter or an underscore. A composite term is an atom followed by an open bracket, followed by a comma-separated list of terms, followed by a close bracket character. There are two notations for expressing lists, which are in turn a shorthand for expressing the term "`.(head, tail)`".

1. The *cons* notation where a list is expressed as `[ head | tail ]`.
2. The *element* notation, where a list is expressed as `[e1, e2, ...]`.

An empty list is expressed as `[]`. The operators listed in table 4.2 can be used in order to express some terms in infix notation.

Terms expressed in infix notation can be grouped using brackets.

The following sequences of characters have special meaning when not separated by white space:

```
\==   ==   =..   :-   ->   >=   <=   not   is
```

Syntax

A *term* program consists of a series of rules. A rule consists of a head optionally followed by a comma separated list of terms, followed by a full stop. The special keyword `import` can precede the head of a rule, to indicate that the body of that rule is defined in another module. The head of a rule consists of the rule name, optionally followed by the input and output terms enclosed in brackets. The two comma separated lists of terms, are separated by a `->` sign. The first list contains the terms that are input to that rule, the second, the terms that are output. For example the rule to append two lists can be written in term as follows:

```
append([], L -> L).

append([H | T], L -> [H | L2]) :-
    append(T, L, L2).
```

program:
    EMPTY
    clause program
clause:
    head `:-` *term-list* `.`
    head `.`
head:
    atom `(` *opt-term-list* `->` *opt-term-list* `)`
    atom
opt-term-list:
    EMPTY
    term-list
term-list:
    term
    term `,` *term-list*
term:
    atom
    element-list
    cons-list
    atom `(` *term-list* `)`
    term-expression
    `(` *term* `)`
term-expression:
    term infix-operator term
    prefix-operator term
cons-list:
    `[ ]`
    `[` *term* `|` *cons-list* `]`
    `[` *term* `|` *term* `]`
elem-list:
    `[` *term-list* `]`

Figure 4.5: *Term* BNF grammar

The BNF grammar syntax of *term* can be found in figure 4.5.

Execution

Execution can be understood in terms of a goal. A goal is the name of a rule, together with the values for the input terms of that rule. Given a goal the execution mechanism goes through all the rules with a matching name looking for a rule whose input part of the head matches the input terms of the goal. Matching is performed recursively:

- atoms match identical atoms,
- composite terms match if their names and components match, and
- a variable in the head of a rule will match any term that appears in that position, setting that variable to that term.

When a matching rule is found execution continues with each clause of that rule as a goal. When the last clause of a rule has been successfully executed, the execution of that goal has *succeeded*. At that point, execution continues with the next clauses of the rule that called the initial rule, with any variables that were passed in the output part of the rule head, set to the values given to them during the execution of the rule. If at some point the execution of a rule fails, alternative matching rule bodies are tried. If none of them succeed, then the execution *fails*. The execution can also fail, if no matching rule head is found.

Built-in Predicates

Term has a library with a number of built-in rules. These perform functions that are often needed when writing term programs. A list of them is provided in table 4.3.

Interfacing with *Imper*

The inter-paradigm communication conventions based on the *call gates* abstraction require each paradigm implementation to provide the capability to call functions from its super-class and for programs in its superclass to call functions defined in it. In *term* this is achieved by documenting (in the language) the way to call *term* functions from *imper* and the converse. A more programmer-friendly alternative would have been the implementation of an *interface builder* that would automate this task, given the suitable type specifications of the data to be used between the paradigms. We felt this was not needed in this prototype and evaluation version of the system; it could be added in a production version.

Calling *Term* from *Imper*

All *term* rules are exported by default to the *imper* execution environment. These can be called as C functions of the form term-name_arity. For example the append rule can be called as `append_3`. All the input parameters are pointers of type `struct s_Term` defined in the include file `term.h`. All the output parameters are pointers to variables of type `struct s_Term`. These variables are set on return from *term*. All the *term* rules return `TRUE` on success and `FALSE` on failure. A set of utility functions in the *term* library allow the easy creation of *term* data types to be passed as parameters to *term* rules. These are shown in table 4.4. A complimentary set of functions — shown in table 4.5 allows the disassembly of terms and accessing their parts.

Calling *Imper* from *Term*

Any *imper* function that is coded according to the following guidelines can be called from inside *term*.

- The header `term.h` is included at the start of the source file.
- The name of the *imper* function must end with an underscore, followed by the number of its arguments.

Name	Function
`(<)(A, B ->)`	Succeed if $A < B$
`(>)(A, B ->)`	Succeed if $A > B$
`(\==)(A, B -> )`	Succeed if A does not match B
`(=..)(T -> L)`	Convert term T to a list L
`(==)(A, B -> )`	Succeed if A matches B
`anint(A -> B)`	Round B towards A
`append(L1, L2 -> L3)`	List L3 is list L2 appended to list L1
`arg(N, T -> A)`	Set A to the Nth argument of term T
`atom(A)`	Succeed if A is an atom
`cos(A -> B)`	$B = \cos(A)$
`exp(A -> B)`	$B = e^A$
`fail`	Always fail
`functor(T -> N , A)`	Set N to the name and A to the arity of functor T
`head(L -> H)`	Set H to the the head of a list L
`integer(I ->)`	Succeed if I is an integer
`is(A <- B(C, D))`	Set A to the value $C \otimes_B D$ where B can be '+', '-', '*', or '/'
`length(L -> N)`	N is the length of the list L
`log(A -> B)`	$B = \log(A)$
`makefunctor(L -> T)`	Convert a list L to a functor term T
`member(X, L -> )`	Succeed if X is a member of list L
`nameio(A -> L)`	Convert an atom A to a list L of integers representing the character ordinal values of its name
`nameoi(L -> A)`	Convert a list of integers L representing character ordinal values into an atom
`nl`	Print a new line on the standard output
`pow(X, Y -> Z)`	$Z = X^Y$
`print(T ->)`	Print the name of term T on the standard output
`real(R ->)`	Succeed if R is a real number
`reverse(L1 -> L2)`	List L2 is the reverse of list L1
`sdebugflag`	Succeed if the subclass debugging flag is set
`sin(A -> B)`	$B = \sin(A)$
`tab(N ->)`	Print N tab characters on the standard output
`tail(L -> T)`	Set T to the the tail of a list L
`tan(A -> B)`	$B = \tan(A)$
`termdebug(F ->)`	Set the term debug flag to F
`var(V ->)`	Succeed if V is a variable
`write(T ->)`	Print term T on the standard output

Table 4.3: *Term* built-in rules

Name	Function
`struct s_Term *mktermbyname (int arity, char *name, ... )`	Create an arbitrary term
`struct s_Term *mkint(int i)`	Create an integer term
`struct s_Term *mkdouble(double d)`	Create a floating point term
`struct s_Term *mkvar(char *s)`	Create a variable
`struct s_Term *mkatom(char *s)`	Create an arbitrary atom term
`struct s_Term *TNULL`	The empty list term

Table 4.4: *Term* term creation functions

Name	Function
`char *functorname(struct s_Term *t)`	Return the name of the functor
`stab_handle tgetname(struct s_Term *t)`	Return a handle to the name of a term
`int tgetarity(struct s_Term *t)`	Return the arity of a term
`int tgetarg(struct s_Term *t, int i)`	Return the ith argument of a term
`bool tislist(struct s_Term *t)`	Return TRUE if the term is a list
`struct s_Term *tgethead(struct s_Term *t)`	Return the head of a list term
`struct s_Term *tgettail(struct s_Term *t)`	Return the tail of a list term
`bool tisint(struct s_Term *t)`	Return TRUE if the term is an integer
`int tgetint(struct s_Term *t)`	Return the value of an integer term
`bool tisdouble(struct s_Term *t)`	Return TRUE if the term is a floating point number
`double tgetdouble(struct s_Term *t)`	Return the value of a floating point term
`bool tisvar(struct s_Term *t)`	Return TRUE if the term is a variable

Table 4.5: *Term* term access functions

- Each input argument is of type `struct s_Term*` and each output argument is of type `struct s_Term **`.
- Input terms are accessed using the functions listed in table 4.5 and created using the functions listed in table 4.4.
- The function will return `TRUE` for success and `FALSE` for failure.

Debugging

Term provides a simple trace port debugging facility — in the form of a *tracer* — for code written in *term* and for its subclasses. Two rules, `termdebug` and `sdebugflag` control the operation of the debugging system. The *tracer* is modelled around the Byrd model [Byr80], except that as *term* is deterministic, there is no backtrack port. Specifically in the Byrd model, a predicate can be invoked by calling it, or by backtracking into it, and can terminate either by failure, or by successful exit. In our modified version, shown in figure 4.6, a rule can be invoked only by calling it, and can terminate either by failure, or by successful exit.

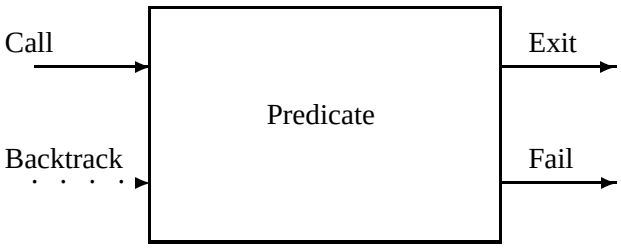

Figure 4.6: Byrd debugging model as modified for *term*

Bibliographical Notes

Our goals for implementing the rule rewrite paradigm language were the provision of data structures found in declarative languages (terms, lists), and compilation into efficient imperative code. The handling of recursive data structures, like the ones used in our language is described in depth in [Hoa73]. Ease of manipulation of tuples and lists is available in the imperative language SETL [Dew79] and its interactive implementation ISETL [Lev89]. Motivation for using our language to implement itself and other parts of the system, was provided by data transformation using sets [Abb89], and the suitability of Prolog for compiler writing [War80], and software engineering tasks [CL93].

On the design side, our first-order approach is corroborated by [Gog90] and the firstification approach described in [Jr.91]; empirical evidence supporting our 'shallow' backtracking implementation can be found in [Smo84, p. 311], and support for the input/output modes in [Smo84, p. 320]. Automatic transformation of logic programs into functional programs (that would be executed in a manner similar to the *term* execution model) is described in [DW89]. Similar choices have been made in the design of Parlog [Cla88], and Strand [SRA89, FT90].

Finally, regarding the implementation, description of how Prolog predicates can be compiled to an imperative language (Pascal) can be found in [Coh85]. The integration mechanism with C is similar to the one used by PIC [BK90].

4.4.5 Regular Expression Paradigm

The *regex* paradigm is used to handle another aspect of programming language implementation, that of lexical analysis. A specification written in the *regex* paradigm contains regular expression descriptions of the input tokens. The description file is in the same format as that expected by the *lex* lexical analyser generator [Les75]. As it is implemented as a paradigm class, an arbitrary number of lexical analysers for different lexical grammars can coexist in the same program.

The generated lexical analyser is invoked by the function `yylex` prefixed by the name of the source file, followed by an underscore. The same rule applies for accessing all the *lex* global variables and functions such as `yyin` and `yytext`.

Bibliographical Notes

Declarative expression of string specifications using regular expression is in most cases a feature found in compiler toolboxes, or stand-alone tools, supporting the generation

of lexical analysers. Compiler toolboxes containing support for specifying strings using regular expressions are described in [GE91, PDC92, DP90]. Stand alone lexical analyser generators based on regular expressions, and the compile-time generation of perfect hash tables are described in [Les75] (*lex*), [HL87] (*mkscan*), [Pax89] (*flex*), [Sch90] (*gperf*), [AMT89] (*ML-lex*), [Heu86], [Gra88] (γ-GLA), and [Gro89] (*rex*).

In addition, regular expression libraries are part of many languages; Snobol4 [GPP71], Icon [GG83],and Perl [WS90] offer string pattern matching. A language with regular-expression-like pattern matching is presented in [Liu86]. Its pattern definitions are built using six basic operations: alternation, concatenation, immediate value, intersection, difference and complement. Our implementation was based on *lex* [Les75], the lexical analyser generator available under Unix.

4.4.6 BNF Grammar Paradigm

The *bnf* paradigm is used for expressing context free grammars. The input file syntax is a small superset of the files accepted by the *yacc* parser generator [Joh75]. It is typically used for building parsers for programming languages, handling complicated user interaction models or even expressing complicated communication protocols. As *bnf* is implemented as a paradigm class, an arbitrary number of grammars can coexist in the same program. This makes the description and implementation of nested grammars (like C with its pre-processor) clean and relatively straightforward.

The one addition to the syntax accepted by *yacc* is the provision of the `%import` keyword. Every *bnf* source file must contain a line with that keyword followed by the name of the function that will be used for providing the input tokens (the default function expected by *yacc* is `yylex`. When a source file is processed by the *bnf* compiler a corresponding header file is generated containing the declarations for all global variables and token identifiers. The global variable names are, as usual, prefixed by the name of the source file, followed by an underscore. The same rule applies for using the generated parser function `yyparse`.

Bibliographical Notes

In the case of the BNF grammar paradigm, support again can be found in compiler toolboxes and stand-alone parser construction tools. Compiler toolboxes containing parser generators are described in [GE91, DP90, PDC92]. Stand-alone tools are described in [Joh75] (*yacc*), [Jon85] (*yacc* in SASL), [TA90] (*ML-yacc*), and [DS88] (*bison*).

An integration scheme for using an arbitrary parser generator in a system (by means of an expert system coordinator) is given in [GHL+92]. A way to isolate the global variables produced by the *yacc* and *lex* generators is described in [Gli91]. Our solution to the same problem uses a more general technique. Finally, a browser for examining the output of such tools is described in [FSO91].

4.4.7 Logic Programming Paradigm

The *btrack* paradigm provides a subset of the Prolog programming language [Kow74, CM84b]. Full backtracking and unification (without *occurs* check) are provided. The

"Edinburgh" syntax [BBP+81] with fixed operator precedence is used.

Btrack is naturally suited for declaratively expressing non-deterministic algorithms. Examples can be found in most Prolog textbooks [SS86b, CM84b, CM84a].

Language Elements

The language elements of *btrack* are the same as those of *term*, described in section 4.4.4, i.e. atoms and variables.

Syntax

The syntax of *btrack* is akin to the "Edinburgh" Prolog syntax. There are no infix operators other than the `:-` which is used to separate the clause head from its body. Clauses are terminated with a full stop. The path-find program is expressed in *btrack* as follows:

```
path(X, X, [X]).

path(X, Y, [X | W]) :-
        edge(X, V),
        path(V, Y, W).

edge(a, b).
edge(c, f).
...
```

At the end of a *btrack* file a single line containing a double percent mark `%%` signifies the end of the code and the start of the import/export section. In that section lines stating with the keyword `import` or `export` followed by the signature of a *term* rule are expected. These can then be used from *btrack* or *term* respectively.

Execution

The Prolog execution and unification mechanism is used. There is currently no way to affect backtracking since the `cut` extralogical feature is not provided.

Built-in predicates

Btrack has a small number of built-in predicates show in table 4.6. All of them have full logical semantics i.e. will behave differently according to the parameters that are non-ground or instantiated. Thus `plus(X, Y, Z)` can be used for adding `Y` and `Z`, or for subtracting `Z` or `Y` from `X`, or finally, for testing if `X` is the sum of `Y` and `Z`.

Interfacing with *Term*

The signatures appearing in the import/export section of the *btrack* source code designate the predicates and rules that can be called from each paradigm. For all exported predicates the export signature specifies the number of arguments and their mode (input or output) for the predicate to be called from *term*, whereas for all imported rules,

Name	Function
`plus(X, Y, Z)`	$X = Y + Z$
`times(X, Y, Z)`	$X = Y * Z$
`integer(X)`	Succeed if X is an integer
`less(X, Y)`	Succeed if X $<$ Y

Table 4.6: *Btrack* built-in predicates

the signature specifies the number of arguments and their mode as expected by the *term* rule. The following fragment of code implements the `cos` function in *btrack* and exports it again to *term*.

```
btcos(A, B) :-
        cos(A, B).
%%

import cos(A -> B).
export btcos(A -> B).
```

When a *term* rule is called from within *btrack* the arguments on the left of the arrow must be ground and the arguments on the right of the arrow must be non-ground. If this is not the case the rule call from *btrack* will fail. When a *btrack* predicate is called from *term* the arguments on the left of the arrow will always be ground. The predicate must succeed with the arguments on the right of the arrow instantiated to non-ground values, otherwise the predicate call from *term* will fail.

Debugging

We provide a simple trace printing debugger. This provides access to the *call* and *fail* ports of the predicates as named in the Byrd model [Byr80].

Bibliographical Notes

The logic programming paradigm [Kow74] is described in introductory form in [Hog84, SS86b]. A number of articles cover implementation details. The unification [Rob65] procedure is described in [Kni89], memory management issues in [Bru82], structure sharing in [BM72], optimisations in [Mel85], and compilation to abstract-machine instructions in [War83]. Our implementation is based on an interpretative scheme implemented our rule-rewrite system *term*. A similar such implementation is described in [Car84].

4.4.8 Functional Programming Paradigm

Fun offers lazy higher order functions and thus programming in a pure functional programming style. The syntax of *fun* resembles that of *Miranda* [Tur85] omitting the guard and pattern matching constructs.

Many of the functional programming examples found in the literature can be written in *fun* with minimal syntactic changes.

Operator	Type	Binding	Precedence
Function application	prefix	left	1
$\wedge$	infix	left	2
$*$	infix	left	3
$/$	infix	left	3
$+$	infix	left	4
$-$	infix	left	5
$<=$	infix	left	5
$==$	infix	left	6

Table 4.7: *Fun* expression precedence rules

Language Elements

When a *fun* program is processed, white space and `/* */` delimited comments are ignored. Identifiers start with an alphabetic character and can optionally be followed by a sequence of alphanumeric characters. Integers are written as sequences of decimal digits; floating point numbers always contain a single decimal point and are optionally followed by a decimal exponent preceded by the `e` character. Infix operators can be expressed as identifiers by enclosing them within brackets e.g. `(*)`.

Syntax

A *fun* program consists of a series of function definitions followed by a single line containing `%%`, followed by the the import/export declarations. Functions are defined by giving the function name, followed by all arguments the function expects, followed by an equal sign and the function value, terminated by a full stop. A function value is an expression. Expressions consist of function applications and can be grouped using brackets. Many built-in function applications can be expressed in shorthand notation using the corresponding infix operator. The factorial function expressed in *fun* looks like the following code fragment:

```
fac x =
        cond    (x == 0)
                1
                (x * fac (x - 1)).
%%
export fac n.
```

The BNF syntax of *fun* can be found in figure 4.7 and the expression precedence rules in table 4.7.

Program Execution

Fun expressions are evaluated using call-by-name [FH88, p. 129] normal-order [FH88, p. 129] evaluation. Function currying [FH88, p. 81] is supported and therefore, programs can be written using the higher-order functional programming style.

module:
    functions `%%` *export-list*
functions:
    EMPTY
    declaration functions
declaration:
    variable variable-list `=` *expression* `.`
expression:
    `(` *expression* `)`
    expression infix-operator expression
    expression expression
    variable
    primitive
    integer-number
    floating-point-number
variable-list:
    EMPTY
    variable variable-list
export-list:
    EMPTY
    export export-list
export:
    `export` *variable variable-list* `.`

Figure 4.7: *Fun* BNF syntax

Name	Description	Infix Operator
(+)	addition	+
(-)	subtraction	-
(*)	multiplication	*
(/)	division	/
(∧)	exponentiation	∧
(==)	equal	==
(<=)	less or equal	<=
cond	two way conditional evaluation	N/A
log2	base 2 logarithm	N/A
anint	round towards n	N/A

Table 4.8: *Fun* primitive functions

Name	Description
app(E1, E2)	Function application
num(X)	Number
var(V)	Variable identifier
prim(P)	Primitive function
lam(V, E)	Lambda expression
let(V, E1, E2)	Let expression

Table 4.9: *Fun* data structure building terms

Built-in Functions

A number of primitive functions are provided. These implement the δ rule reductions [FH88, p. 115] needed in order to write useful programs. Some of the primitive functions have corresponding infix operators. The primitive functions available in *fun* can be found in table 4.8. In addition to the primitive functions, a number of functions, defined in *fun*, are available. Their definition is provided in figure 4.8.

Interfacing with *Term*

A *fun* function of the form `name` $\texttt{arg}_1$ $\texttt{arg}_2 \ldots \texttt{arg}_n$ can be imported into *term* with the signature `name(`$\texttt{arg}_1, \texttt{arg}_2 \ldots, \texttt{arg}_n$`-> Result).` Arguments to and results from *fun* programs must be packaged within special terms. Table 4.9 contains the terms that can be used in order to build the appropriate *fun* data structures.

In the reverse direction, *term* rules of the form `name(`$\texttt{arg}_1, \texttt{arg}_2 \ldots, \texttt{arg}_n$`-> Result)` can be imported into *fun* as function taking n arguments. Again, the terms listed in table 4.9 should be used on the *term* side to disassemble the arguments passed, and to assemble the result.

Debugging

During the expression evaluation, when debugging is turned on, *fun* will provide tracing information on some operations. These can be used in order to locate program

```
true x y = x.
false x y = y.
head l = l true.
tail l = l false.
nil x = true.
cons h t s = s h t.

dec x = (-) x 1.
inc x = (+) x 1.

foldr f x l =
        cond    (l == nil)
                x
                (f (head l) (foldr f x (tail l))).
fac x =
        cond    (x == 0)
                1
                (x * fac (x - 1)).
sum n l =
        cond    (n == 0)
                0
                (head l + (sum (n - 1) (tail l))).

from x = cons x (from (inc x)).
```

Figure 4.8: *Fun* standard library functions

Paradigm	Filename extension	Compiler
Imperative	`.c`	`imperpc`
Rule-rewrite	`.pt`	`termpc`
Logic programming	`.pb`	`btrackpc`
Functional	`.pf`	`funpc`
Rgular expression	`.pr`	`regexpc`
BNF grammar	`.py`	`bnfpc`

Table 4.10: Paradigms, source filename extensions, and their compilers

mistakes. The evaluation of the following expressions will cause a trace line to be printed:

- numbers,
- variables,
- built-in functions and,
- conditional functions.

The evaluation of these expressions usually happens at the bottom of the evaluation tree and thus minimises the amount of information printed.

Bibliographical Notes

Good introductions to the functional paradigm, advocated in [Bac78a], can be found in [FH88, ASS85]. The same references include implementation details. More specialised implementations are described in [Pey87] (compilation based on the abstract G-machine), and [AM87] (the standard ML compiler). Our implementation is based around the eval/apply model first introduced in [McC60]. Other relevant implementation descriptions can be found in [Lic86, Bai85, Lan63].

4.4.9 Using Blueprint

The *blueprint* compilers are used in the same way as the traditional compilers available under the Unix system. The program is implemented as a set of modules in different paradigms. A separate compiler is used to compile each paradigm into object code. The paradigm languages, together with the extensions used, and the names of the compilers, are listed in table 4.10. For example the command `btrackpc sint.pb`, would compile the source file `sint.pb` containing *btrack* source code, into the object file `sint.po`. At the end all the object files are linked together into the single executable file, by runing the multiparadigm link editor *mpld*, with their names as arguments. Again, the command `mpld sint.po main.po`, would link together the two object files `sint.po`, and `main.po` generated by the appropriate paradigm compilers, to create a single executable file, named `a.out`[2].

[2] The name `a.out` is a Unix system convention.

Function	Paradigm
Lexical analysis	regex
Expression parsing	bnf
Symbolic integration	btrack
Numeric integration	fun
Interfacing	term
Expression simplification	term
Graph creation	imper

Table 4.11: Integrator functions and paradigms

4.5 *Integrator*: An Exemplar Multiparadigm Application

Function integration is an interesting aspect of mathematical assistance software packages [gro77, Wol91]. The numeric evaluation of definite integrals is a subject of numerical analysis, while the symbolic evaluation of (typically) indefinite integrals has been an early goal of the AI community. We decided to design and implement *integrator*, a system that would provide both facilities, as an example of a multiparadigm program implemented in *blueprint*.

The paradigms provided by *blueprint* match the needs of a system like *integrator*. Symbolic integration can best be implemented using Prolog-like nondeterministic rules, while a functional language is ideal for describing higher-order numeric evaluation methods. In addition, the user input can best be described as a stream of tokens and a BNF grammar. In the following sections we describe in more detail the design of *integrator*.

4.5.1 Specification

Integrator accepts commands from the standard input. Three commands are implemented:

1. symbolically evaluate an indefinite integral w.r.t. a variable (*sint*),
2. numerically evaluate a definite integral from a to b w.r.t. a variable within a specified precision (*aint*) and,
3. create a graph for an expression given an initial and final value (*plot*).

4.5.2 Paradigm Delegation

As described in the previous section, the design of *integrator* can naturally be divided into different paradigms. Table 4.11 lists the various system modules and the paradigms they were implemented in. When the *integrator* starts, it begins parsing using the input grammar which forms the command interpreter loop. The grammar forms the tokens generated by the lexical analyser into commands. When a command is parsed the relevant *term* rule is called. The *term* rule converts the arguments to the form needed and then calls the function that handles that command. After the command returns the *term* rule prints the result and finally returns to the command interpreter loop.

4.5.3 Numeric Integration

Numeric integration is performed by creating potentially infinite lists of converging series of better approximations as described in [Hug90]. The basic formula used is:

$$\int_b^a f(x)\,\mathrm{d}x = \lim_{\delta x \to 0} \sum_{x=a}^{b} \delta x f(x)$$

We approximate this formula by recursively subdividing the interval. Furthermore, according to [Hug90, p. 30], we can obtain better approximations A from a converging sequence a_n by eliminating the error terms between the approximations using the formula:

$$A = \frac{a_{n+1}2^n - a_n}{2^n - 1}$$

where an estimation of the *order* n can be found by evaluating three consecutive values of the series as follows:

$$n = \text{round } \log_2(\frac{a_0 - a_2}{a_1 - a_2} - 1)$$

These calculations can easily be expressed as function compositions over — potentially — infinite lists, using the features of *fun*.

4.5.4 Symbolic Integration

Symbolic integration can be programmed using a set of heuristic rules since no systematic algorithm for performing it is known. The predicates can be structured as follows:

1. See if the function matches a standard form,
2. If the function can be factored w.r.t. an integer, remove the integer, and integrate the rest. Check if the function is a product or fraction involving the function and its derivative.
3. Try to integrate a product using the "by parts" [BC78, p. 306] integration method using the identity:
$$\int v\frac{\mathrm{d}u}{\mathrm{d}x}\mathrm{d}x = uv - \int u\frac{\mathrm{d}v}{\mathrm{d}x}\mathrm{d}x$$

In the case of symbolic integration, the backtracking and unification features of *btrack* allow for a succinct expression of the rules, as logic predicates.

Bibliographical Notes

A system with similar functionality, but built on top of existing applications and libraries is described in [DR90]. Other multiparadigm applications documented in the literature are a telephone exchange simulator [Zav89], and a system for mainframe peripheral fault diagnosis [Rol87].

An overview of the process of symbolic integration, together with a review of available software and many references can be found in [Sam71]. A description of an

algorithmic approach towards symbolic integration is detailed in [Mos71b]. Risch's algorithm concerning the integration of rational functions containing nested exponentials and logarithms is described in the classic paper [Ris67]. The subject of simplification of algebraic expressions is treated in [Mos71a]. A number of systems that implement symbolic integration are available, some of them are: Reduce [Hea87], MACSYMA [gro77], MAPLE, and Mathematica [Wol91].

Many methods for numerical integration are described in [DR67]. That reference provides a thorough treatment of all methods known at that time, including a chapter on "automatic integration using a computer," and many program listings. More modern treatment of the subject — in the context of the imperative paradigm — can be found in [PFTV88, pp. 111–141], and [Sti92]. Our implementation is based on the evaluation of infinite terms of converging series using the functional programming paradigm. Methods for accelerating the convergence of series are discussed in [PFTV88, pp. 143–146] The applicability of the functional paradigm can be contrasted with problems experienced in a logic programming implementation described in [CMS82]. The suitability of functional languages for numeric work is illustrated in [TP86], which also provided the basis for our approach. A more detailed and in-depth description of the exploitation of the laziness characteristic of the functional paradigm, for numerical work can be found in [HS88].

4.6 Summary

In this chapter we have presented the design of our prototype implementations. We first defined our design objectives, and the systems that we were going to design. We started with the design of the multiparadigm environment generator, MPSS. This included the evaluation of possible design alternatives and the description of its constituent tools, namely the paradigm description compiler *pdc*, the instance variable detector *isntancev*, the private variable protector *protect*, the multiparadigm link editor *mpld*, and the system wrapper *wrap*. We finished our MPSS design description, by giving guidelines on how MPSS would be used to implement a multiparadigm programming environment. The next part we designed, was the *blueprint* multiparadigm programming environment. It supports five paradigms: BNF grammar descriptions, backtracking and unification, lazy higher order functions, regular expressions, imperative constructs, and term handling. For every paradigm not based on existing tools, we described the underlying language lexical elements, syntax, execution semantics, built-in support functions, interfacing with other paradigms, and debugging support. We finished, by describing the design of the *integrator*, a multiparadigm application. The *integrator* was designed to utilise all six paradigms supported by *blueprint*.

Chapter 5

Implementation

In this chapter we describe the implementation of all the system parts. The implementation was done in order to demonstrate the viability of the approach. In particular, our implementation description of *blueprint* can be seen as an experience report on the use of MPSS, and the implementation description of the *integrator* can be seen as an evaluation of *blueprint* for practical applications. We first provide an overall description of the implementation of all the parts. Then, we proceed to describe in detail each of the parts of the three implemented systems. Thus, we will describe the implementation of the tools comprising MPSS, then the implementation of the six *blueprint* paradigms, and finish with the implementation details of the *integrator* application. Novel tools and implementation methods are described in detail, whereas parts that are documented elsewhere (such as the implementations of the declarative paradigms) are only summarised. Detailed notes on our implementation of those systems can be found in appendix A.

5.1 Overall Description

The three systems that we will describe were implemented on top of each other, as described in the previous chapter, and illustrated in figure 4.1, page 72. The bottom layer, the MPSS multiparadigm programming environment generator, was implemented as a set of Unix tools. These were written as small interpreted scripts, using as a basis either the Bourne shell [Bou79], or the Perl programming language [WS90]. Using the tools provided by MPSS we implemented the *blueprint* multiparadigm programming environment. The imperative (*imper*), BNF grammar (*bnf*), and regular expression (*regex*) paradigms, were implemented using existing compilers, with the help of the tool support utilities (*instancev*, and *protect*) provided by MPSS. The term handling (*term*), logic programming (*btrack*), and functional (*fun*) paradigms were all implemented in *term*. Finally, the *integrator* multiparadigm application was implemented by using all six paradigms provided by *blueprint*.

5.2 MPSS: The Multiparadigm Environment Generator

Implementing the MPSS tools as interpreted scripts significantly eased the task of extending and modifying them, during the time they were being used for implementing

Tool	Language and utilities used for implementation
instancev	sh [Bou79], nm(1), and awk [AKW79]
protect	perl [WS90]
mpld	perl, and nm
pdc	perl
wrap	sh

Table 5.1: Implementation languages and Unix utilities used in MPSS

blueprint. Table 5.1, lists the implementation languages and Unix utilities that were used in each one of the MPSS tools. In the following sections we will describe each tool implementation in detail.

5.2.1 *Pdc*: Paradigm Description Compiler

As we described in section 4.3.3, the paradigm description compiler compiles the paradigm description file for a paradigm into a compiler for that paradigm and the associated documentation for that compiler. *Pdc* is implemented in the *Perl* programming language [WS90]. The first step of a paradigm compilation is to set the various variables to their values. This is performed by reading the paradigm description file. *Pdc* variables can refer to other variables in the superclass of the paradigm. In order to be able to resolve references to such variables, *pdc* will read the paradigm class definition module of the paradigm's superclass, unless the paradigm is the target architecture paradigm. After reading the two paradigm description files, *pdc* will replace all references to the variables with their values. The next step is to check for undefined variables. This is simply performed by checking for variable names that have not been replaced. At that point *pdc* can construct the compiler for that paradigm. The compiler created is a Bourne shell script [Bou79, KP84]. An example of a compiler generated by *pdc* is listed in figure 5.1. The shell script will perform the following actions:

- evaluate the base-name of the source file (e.g. `append.pl` become `append`),
- check to see if this is the first compiler in a compilation chain, or it is invoked by the compilation process of a subclass,
- remove any files from previous compilations,
- execute the compilation instructions specified in the `COMPILE` variable (listed in table 4.1),
- protect any instance-specific variables specified in the `INSTANCEV` variable (also listed in table 4.1), and
- if the compiler was not executed by another compiler (as part of the compilcation process of a subclass), run the assembler on the assembly file to create an object file.

```
#!/bin/sh
# Automatically generated file.  Do not modify!
# Generated by /home/dds/bin/pdc fun on 28 June 1992

# Evaluate the base name (source ends in .pf)
SOURCE=`expr $1 : '\(.*\).pf'`

# If some sublass has not set MPSS_COMPILER set it to funpc
MPSS_COMPILER=${MPSS_COMPILER:-funpc}
export MPSS_COMPILER

# Execute the compilation instructions specified by COMPILE
rm -f ${SOURCE}.pt
fun2term <${SOURCE}.pf > ${SOURCE}.pt
termpc ${SOURCE}.pt

# Protect instance-specific variables
protect ${SOURCE} funs mpss_needlib_lfun_0 import_builtin_3 |
perl -pi - ${SOURCE}.s

# If this was not executed by a subclass run the assembler
if test ${MPSS_COMPILER} = funpc
then
     ${CC:-cc} ${CFLAGS} -c ${SOURCE}.s &&
     ${RM:-rm} -f ${SOURCE}.s
fi
```

Figure 5.1: Example of a compiler generated by *pdc*

5.2.2 *Instancev*: Instance Variable Detection

Instancev is used in order to solve name-space pollution problems by identifying global symbols used by various utilities. These symbols are then protected by *protect*. *Instancev* works by examining the object code generated by a compiler looking for symbols with a global scope. It is implemented as a Bourne shell [Bou79] script. First, the `nm` [BSD86b, nm(1)] Unix system command is used to print the global symbols contained in the object file. From these symbols, the text, data and common references are filtered and printed on the standard output by an *awk* [AKW88] command. The symbols printed, are the ones that need to be protected in order to use them as instance variables.

5.2.3 *Protect*: Private Variable Protection

Protect works at the target architecture assembly language level, in order to handle all paradigms up to the root class, and to avoid dealing with complications arising from parsing the structure of the other paradigms. In this way, *protect* can be used as a tool

for all multiparadigm environments, regardless of the paradigms supported. *Protect* is currently used by the code generated by the paradigm description compiler. *Protect* is implemented in the *Perl* programming language [WS90]. Its parameters are the module name on which to operate, and a list of entity names to protect. The output of *protect* is another Perl program that contains instructions which, when applied to the assembly language form of the module, will prepend to all references to those entities the name of the module followed by an underscore. As an example, in the assembly language output of the *yacc* compiler-compiler in a module called `cgram` all definitions and references to the routine `yyparse` will be mapped to `cgram_yyparse`. The script generated will work on any Unix assembly language, thanks to the lexical conventions used by the Unix assemblers.

5.2.4 *Mpld*: Multiparadigm Link Editor

Paradigm classes must be initialised, so that the runtime support of the paradigm will work. In addition, some paradigms require or allow for every module written in that paradigm, some initialisation code to be preformed. These two functions correspond to the class and instance initialisation methods. For example, in the *term* paradigm, provided by the *blueprint* environment, the hash tree used for the global runtime symbol table must be initialised for the whole *term* class (class initialisation method). In addition, for every module written in *term*, the atoms used in it must be individually entered into the symbol table before execution begins (instance initialisation method). *Mpld* solves these two problems by examining the object code that will be linked, for special symbols that indicate that a function is a class or object constructor. These symbols are pushed onto a stack. After all modules have been examined, code for two new functions is created: the one function initialises all classes, while the other initialises all instances. When the resulting code is executed, first all paradigm classes are initialised, and then their instances. This ensures, that instance constructors that depend on the correct functioning of the underlying paradigm will work as expected.

In order to link the runtime machinery (libraries) of each paradigm *mpld* again relies on examining the object code produced by the paradigm compilers. Each paradigm compiler adds a special symbol to the object code, denoting the name of the library that must be loaded. When *mpld* encounters such a symbol, it adds the denoted library to the list of libraries that must be linked, and recursively scans that library for other paradigm support code that must be loaded. This ensures, that a paradigm whose runtime machinery depends on its superclass paradigm, will link correctly, even if no module written in the paradigm language of its superclass is linked into the system. *Mpld* keeps a dictionary of libraries that have been scanned, in order to avoid loops. The operation of *mpld* is illustrated in terms of pseudo-code in figure 5.2.

Mpld is implemented in the *Perl* programming language [WS90]. The associative arrays, stacks, string processing capabilities, easy system interaction and regular expression handling syntax of Perl, significantly eased the task of implementing *mpld*.

```
Push object modules on stack
REPEAT
    FOR every object module DO
        Remove module from stack
        FOR every symbol in module DO
            IF class initialiser THEN add it to class list
            ELSE IF instance initialiser THEN add it to instance list
            ELSE IF library reference AND NOT library used THEN
                Mark library used
                Push library on stack
            END IF
        END FOR
    END FOR
UNTIL stack empty
Create code to:
    Initialise all classes in class list
    Initialise instances in instance list
Invoke system linker to link created code, object modules, and libraries used.
```

Figure 5.2: *Mpld* operation pseudo-code

Paradigm	Implementation Tools and Languages
imper	pdc, cc (existing)
term	pdc, blueprint (imper, bnf, regex, term)
bnf	pdc, yacc (existing)
regex	pdc, lex (existing)
btrack	pdc, blueprint (term)
fun	pdc, blueprint (bnf, regex, term)

Table 5.2: *Blueprint* implementation summary

5.3 *Blueprint*: The Multiparadigm Programming Environment

Blueprint was implemented within the MPSS environment, following the paradigm creation guidelines outlined in section 4.3.8. Every paradigm was implemented as a translator from its source code to the source code of its super class, together with a library containing the run-time support for the paradigm and its individual instances. The paradigm description file containing the description of the paradigm, was automatically compiled by *pdc* into the paradigm compiler and its manual page.

For the paradigms that used existing tools, *instancev* was used to list the per-class-member instance private variables. These were then listed in the paradigm description file, to be automatically protected by the paradigm compiler produced by MPSS. All paradigms were either implemented in *blueprint*, or using existing tools. Table 5.2

Function	Paradigm
Lexical analysis	regex
Parsing	bnf
Code generation	term
Symbol table	imper
Term support	imper
Library routines	term, imper

Table 5.3: *Term* functions and paradigms

provides an overview of the implementation methods used. It should be noted, that *imper*, *bnf*, and *regex* were implemented using existing tools, by creating a suitable paradigm description file. The rest were implemented by writing — in addition to the paradigm secription file — the requisite translators and run-time support. The *blueprint* paradigms used are listed in table 5.2.

5.3.1 *Imper*: Imperative Paradigm

Imper is implemented as a front end to *gcc* — the project GNU C compiler [Sta92] — using special arguments to allow the post-processing of its output. This is needed, in order to provide class encapsulation of the runtime mechanisms. Specifically, the generated output does not consist of object files, but target architecture assembly language source code.

5.3.2 *Term*: Rule-rewrite Paradigm

The *term* paradigm implementation consists of the following parts:

- lexical analyser: converts the source-code input into tokens,
- parser: converts the stream of tokens into a syntax tree,
- code generator: converts the syntax tree into *imper* code,
- symbol table: provides a mechanism for the efficient storage and comparison of strings,
- term support: a set of functions providing support for the basic *term* datatype, the *term*, and
- library routines: a number of routines providing useful *term* functions.

The *term* compiler and the runtime system are implemented using the *blueprint* multi-paradigm programming environment. The implementation paradigms of the various *term* components are summarised in table 5.3. In appendix A.1 (page 183) we describe each of the parts in more detail.

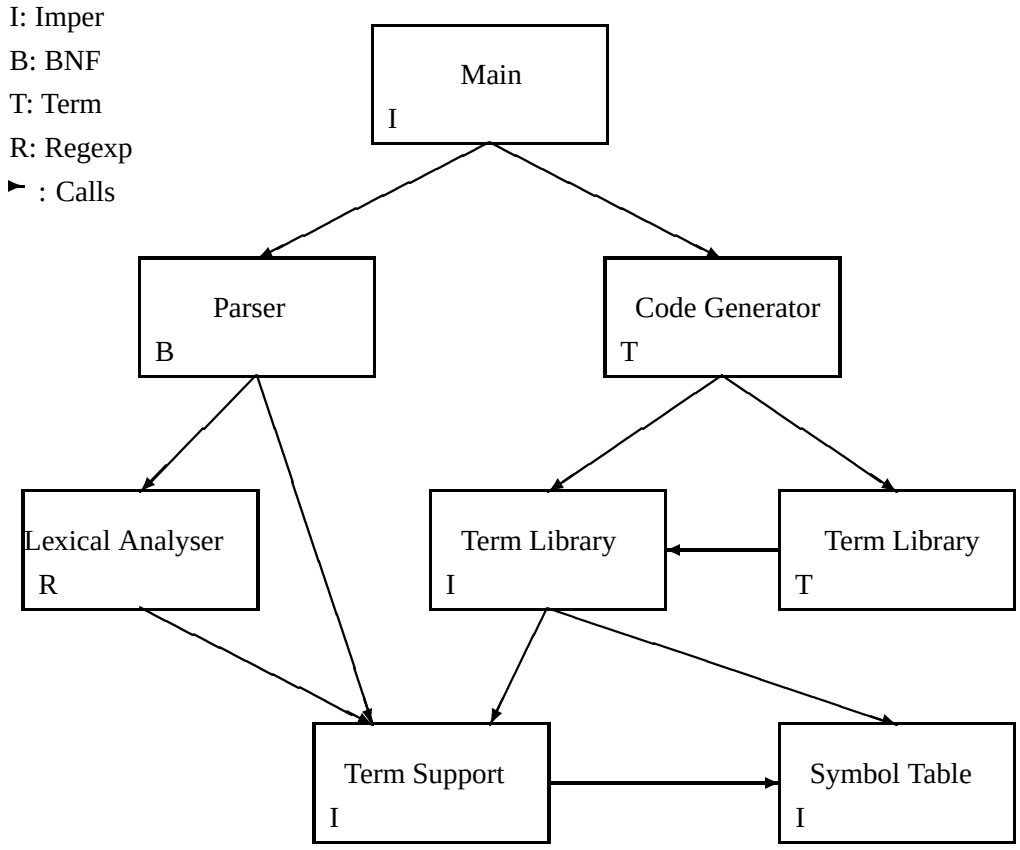

Figure 5.3: *Term* paradigm inter-operation schematic representation

Multiparadigm Structure

It is interesting to examine the way in which the various parts of the *term* implementation work together. Figure 5.3 provides a schematic representation of it. Execution of the *term* compiler starts at the `main` function written in *imper*. That function calls the parser, which in turn calls the lexical analyser. Both call the term handling support (written in *imper*) in order to build the complex terms that will later be passed to *term*. Once the parsing is finished, control returns back to the `main` function, which then invokes the code generator (written in *term*). The code generator makes use of the *term* library routines (written in *term* and *imper*), which in turn rely on the basic term handling routines and the symbol table.

Bootstrapping

Another interesting aspect of the *term* implementation process, is its bootstrapping attribute: the *term* code generator is implemented in *term*. For our discussion we will use the symbolic representation of T-diagrams as explained in section 3.3 (page 48). In our case we will use P to denote Prolog, C for C, and T for *term*.

The first version of *term* was written in Prolog. Care was taken, to avoid using any features of Prolog that would not be available in *term*, such as full unification and

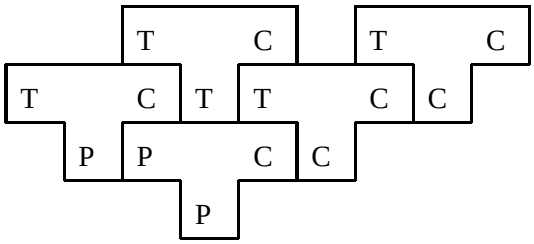

Figure 5.4: *Term* bootstrapping sequence T-diagram

non-determinism. Furthermore the parameters of all rules were split according to their mode (input / output), by a special comment. That first implementation, $P_P C$ in the T-diagram illustrated in figure 5.4, was translating from a subset of Prolog to C and was implemented in that subset of Prolog. The compiler from *term* to C, written in Prolog, $T_P C$, was compiled with it and generated the compiler from *term* to C, written in C $T_C C$. At that point the compiler was hand-translated from Prolog to *term* (a trivial process) and thus the compiler from *term* to C, written in C $T_C C$ was generated. That compiler was compiled by $T_C C$, to create the final $T_C C$: the bootstrapped *term* to C compiler.

5.3.3 *Regex*: Regular Expression Paradigm

Regex is implemented as a front end to *lex* using a paradigm description file. Running *instancev* on some *lex* output, generated a list of 28 instance variables. These were listed in the paradigm description file in order to for them to have their own name-space.

5.3.4 *Bnf*: BNF Grammar Paradigm

Bnf is implemented as a front end to *yacc* using a paradigm description file. Running *instancev* on some *yacc* output generated a list of 21 instance variables. These were listed in the paradigm description file in order for them to have their own name-space. A small *sed* [McM79] script implements the `%import` extension for specifying the name of the lexical analyser function.

5.3.5 *Btrack*: Logic Programming Paradigm

Btrack is implemented via an intepreter that executes *btrack* code in its abstract representation form as a *term* data structure. The syntax of *btrack* is relatively near to that of *term*; for this reason *btrack* is translated into a *term* term data structure by a small Perl [WS90] script. An abstract code interpreter based on a *solve/unify* loop [Coh85, p. 1313], [ASS85, p. 335–380], [SS86b, p. 150], is implemented in *term*. The *btrack* to *term* translation detects all the `import` and `export` declarations and generates the appropriate *term* call gates. Thus, for every exported predicate, an equivalent *term* rule is generated, and for every imported *term* rule, *btrack* access code is generated. Appendix A.2 (page 192) contains a detailed description of the *btrack* implementation.

5.3.6 *Fun*: Functional Programming Paradigm

Fun is also implemented using the *blueprint* multiparadigm programming environment. Lexical analysis is done in *regex*, parsing in *bnf*, intermediate code generation in *imper*, and intermediate code interpretation in *term*. An eval/apply interpreter [FH88, p. 193–195] written in *term* provides the runtime machinery. In appendix A.3 (page 198) we describe the implementation of *fun* in more detail.

5.3.7 Paradigm Inter-operation

The astute reader may have noticed by now, that although we describe the inter-operation of each paradigm with its superclass, we never describe how we implemented the more general facilities that allow any paradigm to call any other paradigm. The reason for this 'omission' is that these facilities were never implemented. Systems built using MPSS need only provide for the inter-operation with each paradigms' superclass. The inter-operation with all other paradigms is performed automatically by MPSS using the call-gate mechanism described in section 3.3.16. As an example, a call of an *imper* function from *fun* (which only interfaces to *term*) is mapped to a call to *term*. The *term* code in turn, will call the *imper* function. The following section contains a more detailed example.

5.3.8 Paradigm Inter-Operation Example

In order to demonstrate how we implemented paradigm inter-operation in the blueprint environment, we created a small artifical example, where all paradigms with non-trivial inter-operation requirements (all except *bnf* and *regex*) call all others. In the example we have implemented in all four paradigms a function that increments its single integer parameter by one. In this way we demonstrate both the passing and the returning of parameters. In addition, each module contains a function that returns the number three by calling the increment function written in the three other paradigms with a starting value of zero. The functions we described together with any needed import and export statements for the *imper*, *term*, *btrack* and *fun* paradigms are listed in figures 5.5, 5.6, 5.7, and 5.8 respectively.

As we described in section 3.3.13, call gates are used to map the functionality of a subclass to the conventions expected by its superclass. In the following paragraphs we will examine how each of the paradigms implements the call gate abstraction.

In order for *term* rules to be available in *btrack*, the *btrack* compiler adds new patterm matching rules to the Prolog engine with the name `import_unify` for all *term* rules that are imported. These look-up any variables in the Prolog environment, call the *term* rule, and create a new environment with the *term* rule result bound to the variable passed. The *btrack* rule for the `fun_add_one` rule is shown in figure 5.9[1]. Similar rules are used for importing the rules from the other paradigms — always using the *term* calling conventions.

The exporting of the *btrack* rule to *term* is handled in a similar way. A *term* rule is created that calles the *btrack* interpreter to solve the query, and looks up the result in

[1] `fun_add_one_t` is the name of `fun_add_one` after passing through the *term* import gate. In general, for manually implemented call gates, we use the convention of *name*`_x` to denote that *name* has been converted to follow the convnentions of paradimg `x`.

```
/* Imper function to be exported to the other paradigms */
int
imper_add_one(int a)
{
        return a + 1;
}

/* Imper function that uses fun, btrack and term */
int
imper_three(void)
{
        return term_add_one(btrack_add_one(fun_add_one(0)));
}
```

Figure 5.5: *Imper* example functions

```
import fun_add_one(a -> b).
import btrack_add_one(a -> b).
import imper_add_one_t(a -> b).

/* A term rule exported to the other paradigms */
term_add_one(A->B) :-
        B is A + 1.

/* A term rule that uses fun, imper and btrack */
term_three(->C) :-
        btrack_add_one(0, A),
        fun_add_one_t(A, B),
        imper_add_one_t(B, C).
```

Figure 5.6: *Term* example functions

```
/* Simple btrack function to be exported */
btrack_add_one(A, B) :-
        plus(B, 1, A).

/* A btrack function that uses fun, imper, term functions */
btrack_three(C) :-
        fun_add_one_t(0, A),
        imper_add_one_t(A, B),
        term_add_one(B, C).
%%
import fun_add_one_t(A -> B).
import imper_add_one_t(A -> B).
import term_add_one(A -> B).

export btrack_add_one(A -> B).
export btrack_three(-> B).
```

Figure 5.7: *Btrack* example functions

```
/* Simple fun function to be exported */
fun_add_one x = x + 1 .

/* Fun function that uses btrack term and imper */
fun_three = btrack_add_one_f (term_add_one_f (imper_add_one_f 0)) .
%%
import btrack_add_one_f x.
import term_add_one_f x.
import imper_add_one_f x.

export fun_add_one x.
export fun_three.
```

Figure 5.8: *Fun* example functions

```
/* Automatically generated import code */
import fun_add_one_t(A -> B).
import_unify(fun_add_one_t(V0, V1), Env -> [bind(LV1, R0) | Env ]) :-
        lookup(V0, Env, LV0),
        lookup(V1, Env, LV1),
        not btrackvar(LV0),
        btrackvar(LV1),
        fun_add_one_t(LV0, R0).
```

Figure 5.9: *Term* code generated for importing a *fun* rule into *btrack*

```
/* Automatically generated export code */
btrack_add_one(A -> B) :-
        rules(Rules),
        btrack([btrack_add_one(A , $B)], Rules, Bindings),
        varval($B, Bindings, B).
```

Figure 5.10: *Term* code generated for exporting a *btrack* predicate to *term*

```
/* Automatically generated import code */
import imper_add_one_f(V0-> R).
import_arity(imper_add_one_f -> 1).
import_builtin(imper_add_one_f, [V0] -> R) :-
        eval(V0, [], V0L),
        imper_add_one_f(V0L, R),
        fundebug(import_built, imper_add_one_f(V0, R)).
```

Figure 5.11: *Term* code generated for importing a *fun* function from *term*

the new environment. The rule for exporting the *btrack* `add_one` predicate to *term* is listed in figure 5.10.

The situation with the functions that *fun* imports and exports is similar, but not identical. As we indicated in section 4.4.8, the programmer is responsible for implementing a part of the call gate functionality, by adhering to the documented *fun* data structuring conventions using *fun* constructors. The *fun* compiled code that imports the `imper_add_one` function is listed in figure 5.11, and the code that exports the `fun_add_one` to *term* is listed in figure 5.12. The manually implemented part of the call-gate functionality is listed in figure 5.13.

The interface between *term* and its superclass *imper* is analogous to the one between *fun* and *term*. The *term* rules are compiled into C programs, but the calls must conform to the *term* calling conventions. The C compiled form of the `term_add_one` rule is listed in figure 5.14.

On the *imper* end, some parts of the *term* call-gate functionality are again manually implemented. Figure 5.15 contains a listing of some interface functions written in *imper* to follow the *term* coding conventions.

Having described the example of arbitrary paradigm interoperation we need to em-

```
/* Automatically generated export code */
fun_add_one(V0-> R) :-
        funs(F),
        feval(rlet(F, app( var(fun_add_one), V0)), [], R).
```

Figure 5.12: *Term* code generated for exporting a *fun* function to *term*

```
/*
 * Manually implemented import call gate functionality
 * Import of term functions into fun
 */
btrack_add_one_f(num(A) -> num(B)) :-
        btrack_add_one(A, B).

/*
 * Manually implemented export call gate functionality:
 * Export of a fun function to term
 */
fun_add_one_t(A -> C) :-
        fun_add_one(num(A), B),
        getnum(B, C).
```

Figure 5.13: Manually implemented parts of the *fun* call-gate functionality

```
/* Automatically generated export code */
bool
term_add_one_2(struct s_Term *i0, struct s_Term **o0)
{
        if (1) {
                struct s_Term *v0;
                if ( is_2(&v0, mkterm(2, et_C5, i0, mkint(1))) && 1) {
                        *o0 = v0;
                        return TRUE;
                }
        }
        return FALSE;
}
```

Figure 5.14: *Imper* code generated for exporting a *term* rule to *imper*

```
/*
 * Manually implemented import call gate of term:
 * Imper_add_one converted to term conventions
 */
bool
imper_add_one_t_2(struct s_Int *a, struct s_Term **b)
{
        *b = mkint(imper_add_one(tgetint(a)));
        return TRUE;
}

/*
 * Manually implemented export call gate of term:
 * Btrack_add_one converted to imper conventions
 */
int
btrack_add_one(int a)
{
        struct s_Int *b = (struct s_Int *)mkint(0);

        (void)btrack_add_one_2(mkint(a), &b);
        return tgetint(b);
}
```

Figure 5.15: Manually implemented parts of the *term* call-gate functionality

phasize that the manually implemented functionality of the call-gates is a result of the design of the various languages. The manually implemented call-gates are needed to overcome language deficiencies in the areas of type systems, and supported language elements.

5.3.9 Implementation Experience

The tools and the design philosophy of MPSS made the implementation of *blueprint* easy and straightforward. Once the structure of the system was decided, the implementation consisted of providing the translation and runtime mechanisms for each paradigm class, together with the relevant paradigm description file. The *paradigm compilers* and manual pages were produced automatically. Details irrelevant to the specific systems, such as the encapsulation of classes, and the class and instance initialisation mechanisms, were automatically handled by MPSS.

The number of diverse paradigms provided and used, convinced us that our underlying philosophy is suitable for implementing multiparadigm systems, irrespective of the paradigms that have to be supported. In addition to the breadth of supported paradigms, our method was successfully used in conjunction with many different implementation methods. The paradigms chosen included some whose compilers already existed (*imper, bnf, regex*), others that were compiled (*imper, term*), and still others that interpreted some semi-compiled or abstract machine code (*bnf, regex*[2], *btrack* [3], *fun* [4]).

5.3.10 Implementation Metrics

To give our reader a rough idea of the implementation effort that went into *blueprint*, the relative size of its parts, and the way different paradigms were used, we summarise in table 5.4 the lines of code written to implement each paradigm. Each row of the table contains a breakdown of the implementation of a specific paradigm. The PDF column contains the number of lines written for the paradigm description file. The number includes any documentation written for that paradigm, that was included in that file. The other columns contain the number of lines of code written in the specific paradigm. All numbers are inclusive of blank lines and comments. Finally, the last rows and columns, contain totals and percentages, for their respective groups. The table shows that the *blueprint* implementation effort was quite modest, at 4259 lines of code. Furthermore, there was a fair distribution among the paradigms used.

5.4 *Integrator*: The Multiparadigm Programming Application

The *integrator* was implemented utilising all six paradigms provided by *blueprint*. The delegation of the various software functions among the different paradigms is described in section 4.5.2 (page 95), and summarised in table 4.11 (page 95). In the

[2] Automata.
[3] Horn clauses.
[4] λ expressions.

Paradigm	PDF	*imper*	*bnf*	*regex*	*term*	*fun*	Total	%
imper	43						70	1.6
term	70	1192	119	84	666		2269	53.3
btrack	60				316		554	13.0
fun	140		305	59	237	43	840	19.7
bnf	95						121	4.3
regex	379						405	9.5
Total	787	1192	424	143	1219	43	4259	100.0
%	18.5	28.0	10.0	3.4	28.6	1.0	100.0	

Table 5.4: *Blueprint* paradigm implementation summary

following sections we will describe the implementation of each part. Where appropriate, we will include small source code excerpts to illustrate the discussed implementation technique or paradigm. We will finish the *integrator* description by presenting a sample session with it and a summary of relevant metric information.

5.4.1 Lexical Analysis

The lexical analyser converts the characters that the user types into the tokens expected by the parsing grammar. Apart from single characters (such as "`+ - ; ( ) * /`"), four other tokens are recognised:

- the commands `sint`, `aint` and `plot` and,
- integers, floating point numbers and function and variable symbols which are returned as *term* atoms.

The code fragment listed in figure 5.16 contains most of the lexical analyser functionality.

The *term* support functions `mkatom` and `mkint` allow the easy interfacing between the imperative rules of *regex* and *term*.

5.4.2 Parsing

The grammar of the *integrator* can be expressed in three rules using operator precedence disambiguifying statements. These are: the definition of an *integrator* session, a command and an expression. The actions for the expression just build the expression tree; the actions for the commands call the relevant *term* rules. The grammar is listed in figure 5.17

5.4.3 Numeric Integration

The following functions create a list of better and better approximations:

```
integrate f a b = integ f a b (f a) (f b).

integ f a b fa fb = cons ((fa + fb) * (b - a) / 2.0)
                        (map addpair (zip2 (integ f a m fa fm)
```

```
                                /* Commands */
"aint"                          { return tAINT; }
"sint"                          { return tSINT; }
"plot"                          { return tPLOT; }

[a-z]([a-zA-Z_]|[0-9])*     {     /* Function and variable symbols */
                                parse_yylval.t = mkatom(yytext);
                                return tATOM;
                            }
[+-]?[0-9]+                 {    /* Handle integers */
                                parse_yylval.t = mkint(atoi(yytext));
                                return tATOM;
                            }
[ \t\n]+                    { /* Eat white space */ ; }
[-;()+*/^]                  { return *yytext; }
```

Figure 5.16: *Integrator* user-interface lexical analyser

```
                                        (integeg f m b fm fb)))
                        where m = (a + b) / 2
                              fm = f m.

addpair l = (head l) + (tail l).
```

We can now define a function that takes an infinite list of better and better approximations and returns the approximation when the difference between it and the next is within a given interval *eps*.

```
within eps l = cond (abs((head l) - (head (tail l))) <= eps)
                        (head (tail l))
                        (within eps (tail l)).
```

Given this function we can define

```
aint fun a b eps = within eps (integrate fun a b).
```

The higher-order error elimination formuli can be expressed in *fun* as:

```
elimerror n l =
        cons (( (head (tail l)) * (2.0^n) - (head l)) / ( 2.0^n - 1.0))
             (elimerror n (tail l)).

order l = anint (log2 (((head l) - (head (tail (tail l)))) /
                       ((head (tail l)) - (head (tail (tail l)))) - 1.0)).

improve s = elimerror (order s) s.
```

```
%left '+' '-'
%left '*' '/'
%left '^'
%%
session : /* EMPTY */
        | session command
        ;
command : error ';'              { /* Error recovery */ yyerrok; }
                                 /* Symbolic integration */
        | tSINT expr tATOM ';'   { seval_2($2, $3); }
                                 /* Numeric integration */
        | tAINT tATOM tATOM expr tATOM tATOM ';'
                                 { aeval_5($4, $5, $2, $3, $6); }
                                 /* Graph plotting */
        | tPLOT tATOM tATOM expr tATOM tATOM ';'
                                 { plot_5($4, $5, $2, $3, $6); }
        ;
expr    : '(' expr ')'           {       $$ = $2; }
        | expr '*' expr          { $$ = mktermbyname(2, "mul", $1, $3); }
        | expr '/' expr          { $$ = mktermbyname(2, "div", $1, $3); }
        | expr '+' expr          { $$ = mktermbyname(2, "add", $1, $3); }
        | expr '-' expr          { $$ = mktermbyname(2, "sub", $1, $3); }
        | expr '^' expr          { $$ = mktermbyname(2, "raise", $1, $3);
        | tATOM '(' expr ')'     { $$ = mktermbyname(2, "fun", $1, $3); }
        | tATOM
        ;
```

Figure 5.17: *Integrator* user-interface grammar

We can now get even more sophisticated, by noting that *improve* takes a series of approximations and, by eliminating the error term, creates a new one. This can be applied to the result of the original *integ* function, but it can also be applied to itself! Taking the second value of these approximations is probably the best action, since it is already computed (by *order*) and is better than the first. We can thus define a higher order method as:

```
super s = map second (repeat improve s).

second l = head (tail l).
```

and in terms of it, the numerical integration function:

```
aint fun a b eps = within eps (super (integrate fun a b)).
```

5.4.4 Symbolic Integration

The general form of the symbolic integration predicate is:

```
sint(FX, X, Loop, DFX) :- ...
```

where `FX` is the function to be integrated, `X` is the variable with respect to which the integration is performed and `DFX` the integrated function. The variable `Loop`, is used to avoid infinite loops in cases where recursion is used. For example in the "by parts" integration, the result contains a new integral. Sometimes it is necessary to apply the "by parts" rule again on that integral, but on other cases recursively applying the rule again and again does not lead anywhere. A heuristic rule is included which increments the `Loop` variable and blocks the search after three recursive applications.

The standard forms are expressed as follows:

$$\int x^n \,\mathrm{d}x = \frac{1}{n+1} x^{n+1} + \mathrm{K}$$

```
sint(raise(X, Y), X, L, mul(div(1, add(Y, 1)), raise(X, add(Y, 1)))) :-
        integer(Y).
```

$$\int \mathrm{e}^x \,\mathrm{d}x = \mathrm{e}^x + \mathrm{K}$$

```
sint(raise(epsilon, X), X, L, raise(epsilon, X)).
```

$$\int \frac{1}{x} \,\mathrm{d}x = \ln x + \mathrm{K}$$

```
sint(div(1, X), X, L, fun(ln, X)).
```

$$\int \cos x \,\mathrm{d}x = \sin x + \mathrm{K}$$

```
sint(fun(cos, X), X, L, fun(sin, X)).
```

$$\int \sin x \, \mathrm{d}x = -\cos x + \mathrm{K}$$

```
sint(fun(sin, X), X, L, mul(-1, fun(cos, X))).
```

Factoring out the integers is expressed, by a predicate for taking the factor out and integrating, and a predicate for factoring.

$$\int nf(x) \, \mathrm{d}x = n \int f(x) \, \mathrm{d}x$$

```
sint(Product, X, L, mul(Integer, Integral)) :-
        getintprod(Product, Integer, Rest),
        sint(Rest, X, L, Integral).

getintprod(mul(X, Y), X, Y) :-
        integer(X).

getintprod(mul(X, Y), Y, X) :-
        integer(Y).

getintprod(mul(X, mul(A, B)), Z, mul(X, W)) :-
        getintprod(mul(A, B), Z, W).

getintprod(mul(mul(A, B), Y), Z, mul(Y, W)) :-
        getintprod(mul(A, B), Z, W).
```

Fractions of the form $\frac{\frac{\mathrm{d}u}{\mathrm{d}x}}{u}$ are integrated using the rule:

$$\int \frac{\frac{\mathrm{d}u}{\mathrm{d}x}}{u} \, \mathrm{d}x = \ln \mid u \mid + \mathrm{K}$$

```
sint(div(A, B), X, L, fun(ln, fun(abs, B))) :-
        ssdiff(B, X, A).
```

The predicate *ssdiff* performs symbolic differentiation on the function, with respect to a variable, and simplifies the result. Although one would think that the same predicates that express the integration rules, could be used for differentiation, this is not the case. The process of symbolic integration is not deterministic. Running the predicates backwards will spawn off an infinite search tree. One example where this would occur is the factoring out of constant integers. Since the function to be integrated would not be instantiated, the pattern match would succeed. Then the factoring predicate would loop factoring the un-instantiated variable again and again producing a tree of the form

```
A == mul(A1, mul(A2, mul(A3 ... B ...
```

(where all the variables would be non-ground) which although mathematically correct is not terribly useful. This can be avoided, by using an extra-logical predicate like *ground*. We opted to define some separate predicates for *sdiff*:

```
sdiff(X, X, 1).

sdiff(raise(X, N), X, mul(N, raise(X, N1))) :-
        integer(N),
        plus(N, N1, 1).
```

a predicate for the simplified differentiation:

```
ssdiff(X, Y, Z) :-
        sdiff(X, Y, Z1),
        simplify(Z1, Z).
```

and a set of predicates for simplifying arithmetic expressions:

```
simplify(mul(1, A), A1) :-
        simplify(A, A1).

simplify(mul(A, 1), A1) :-
        simplify(A, A1).

simplify(mul(-1, mul(-1, A)), A).

simplify(sub(A, mul(-1, B)), add(A1, B1)) :-
        simplify(A, A1),
        simplify(B, B1).

simplify(mul(A, B), mul(A1, B1)) :-
        simplify(A, A1),
        simplify(B, B1).

simplify(raise(X, 1), X1) :-
        simplify(X, X1).

simplify(X, X).
```

The next set of predicates deals with integration of products. First we examine if the function is of the form $\frac{\mathrm{d}u}{\mathrm{d}x}\mathrm{e}^u$ and can thus be integrated using the formula:

$$\int \frac{\mathrm{d}u}{\mathrm{d}x}\mathrm{e}^u \,\mathrm{d}x = \mathrm{e}^u + \mathrm{K}$$

```
sint(mul(DU, raise(epsilon, U)), X, L, raise(epsilon, U)) :-
        ssdiff(U, X, DU).
```

The final predicate embodies the “by-parts” integration mechanism:

$$\int v\frac{\mathrm{d}u}{\mathrm{d}x}\mathrm{d}x = uv - \int u\frac{\mathrm{d}v}{\mathrm{d}x}\mathrm{d}x$$

```
sint(Fun, X, L, sub(mul(U, V), N)) :-
        less(L, 3),                     /* Prevent loops */
        part(Fun, V, DU),               /* Will backtrack here */
        plus(L1, L, 1),
        ssdiff(V, X, DV),
        ssint(DU, X, L1, U),
        ssint(mul(U, DV), X, L1, N).
```

`Part` is simply a non-deterministic predicate which returns the left or right hand side of a multiplication expression. Thus, if an expression can not be integrated by parts using the left hand side as the differentiated function, the search process will backtrack and try the right hand side.

```
part(mul(A, B), A, B).

part(mul(A, B), B, A).
```

5.4.5 Printing the Resulting Expression

After the result is symbolically evaluated, it must to be printed on the standard output. A set of functions written in *term* handle this aspect. The process is deterministic and thus the power of *btrack* is not needed. All of the *term* rules are like the following one:

```
flatprint(raise(X, Y)->) :-
        write('('),
        flatprint(X),
        write(' ^ '),
        flatprint(Y),
        write(')').
```

5.4.6 Graph Generation

Graphs are plotted using the Unix *plot* utility. For it to be used, the output of the program needs to be redirected to the *graph* process. This is accomplished using two functions written in *imper*. Both are written using the conventions needed in order to be called from *term*. The first one saves the current output stream and opens a pipe to the *graph* process:

```
static FILE stdout_back;

bool
startplot_0(void)
{
        stdout_back = *stdout;
        *stdout = *popen("tee values.out | graph | xplot", "w");
        return TRUE;
}
```

The second function waits for the forked process to terminate and restores the standard output:

```
bool
endplot_0(void)
{
        pclose(stdout);
        *stdout = stdout_back;
        return TRUE;
}
```

Both of the functions are called from *term* which loops over the function values:

```
plot(Tfun, Var, X1, X2, Step ->) :-
        mkfun(Tfun, Var, Ffun),
        startplot,
        points(Ffun, Var, X1, X2, Step),
        endplot.
```

`Points` just generates a list of point values on the standard output:

```
points(Fun, Var, X1, X2, Step->) :-
        X1 < X2,
        evalfun(lam(Var, Fun), num(X1), Y),
        getnum(Y, Ny),
        write(X1), write(' '), write(Ny), nl,
        XX is X1 + Step,
        points(Fun, Var, XX, X2, Step).

points(_, _, _, _, _->).
```

The function is, of course, evaluated in *fun* by the sophisticated *evalfun* function:

```
evalfun fun v = fun v. /* Wow ! */
```

5.4.7 Sample Session

In order to demonstrate the capabilities of the integrator, in the following paragraphs we will present a small session with the finished system.

First we symbolically evaluate the integral

$$\int x^2 \sin x \, \mathrm{d}x$$

This is evaluated by twice using the process of integrating by parts [BC78, p. 306]. This is based on the identity:

$$\int v \frac{\mathrm{d}u}{\mathrm{d}x} \mathrm{d}x = uv - \int u \frac{\mathrm{d}v}{\mathrm{d}x} \mathrm{d}x$$

Let

$$\left\{ \begin{array}{rcl} v & = & x^2 \\ \frac{\mathrm{d}u}{\mathrm{d}x} & = & \sin x \end{array} \right. \Rightarrow \left\{ \begin{array}{rcl} \frac{\mathrm{d}v}{\mathrm{d}x} & = & 2x \\ u & = & -\cos x \end{array} \right.$$

Then

$$\begin{aligned} \int v\frac{\mathrm{d}u}{\mathrm{d}x}\mathrm{d}x &= (-\cos x)(x^2) - \int(-\cos x)(2x)\,\mathrm{d}x \\ &= -x^2\cos x + 2\int x\cos x\,\mathrm{d}x \end{aligned}$$

Now we need to evaluate $\int x\cos x\,\mathrm{d}x$, again integrating by parts.

Let

$$\left\{\begin{array}{rcl} v &=& x \\ \frac{\mathrm{d}u}{\mathrm{d}x} &=& \cos x \end{array}\right. \Rightarrow \left\{\begin{array}{rcl} \frac{\mathrm{d}v}{\mathrm{d}x} &=& 1 \\ u &=& \sin x \end{array}\right.$$

and therefore

$$\begin{aligned} \int x\cos x\,\mathrm{d}x &= (\sin x)(x) - \int(\sin x)(1)\,\mathrm{d}x \\ &= x\sin x + \cos x + \mathrm{K} \end{aligned}$$

From the above we can conclude that

$$\int x^2\sin x\,\mathrm{d}x = -x^2\cos x + 2x\sin x + 2\cos x + \mathrm{K}$$

In the following example the same integral is evaluated on the *integrator*:

```
skid% integrator
--> sint x ^ 2 * sin(x) x ;
Integral(((x ^ 2) * sin(x))) =
((-cos(x) * (x ^ 2)) - (2 * -((sin(x) * x) + cos(x))))
```

Factoring the R.H.S. evaluates to the expected result.

Next we try the numeric evaluation of definite integrals. We evaluate

$$\int_1^0 \frac{1}{1+x^2}\,\mathrm{d}x$$

with a precision of 0.001. The result should be $\frac{\pi}{4}$.

```
skid% integrator
--> aint 0.0 1.0 1.0 / (1.0 + x * x) x 0.001 ;
Integral (0, 1) ((1 / (1 + (x * x)))) x = 0.785398 +-0.001
```

Finally, we create a graph of a function $\frac{x^3-x^2}{1+x^2}$ in the range 0.01–1 in 0.01 increment steps:

```
skid% integrator
--> plot 0.01 1 (x ^ 3 - x ^ 2) / (1 + x ^ 2) x 0.01 ;
```

and the graph shown in figure 5.18 appears on our screen.

5.4.8 Implementation Metrics and Paradigm Inter-operation

Integrator was implemented within a period of three days. We believe that each part of the system was implemented in the most suited paradigm. Table 5.5 has a list of the modules and their size, and figure 5.19 provides the paradigm inter-operation call graph of the system. It seems, that implementing the system in the *blueprint* environment resulted in a concise and thus inherently (though always relatively) correct and easily maintainable system. Writing the *implementor* in an imperative language would result in an order of magnitude larger system, while choosing a single declarative language as the implementation vehicle would still make the system at least twice as complicated.

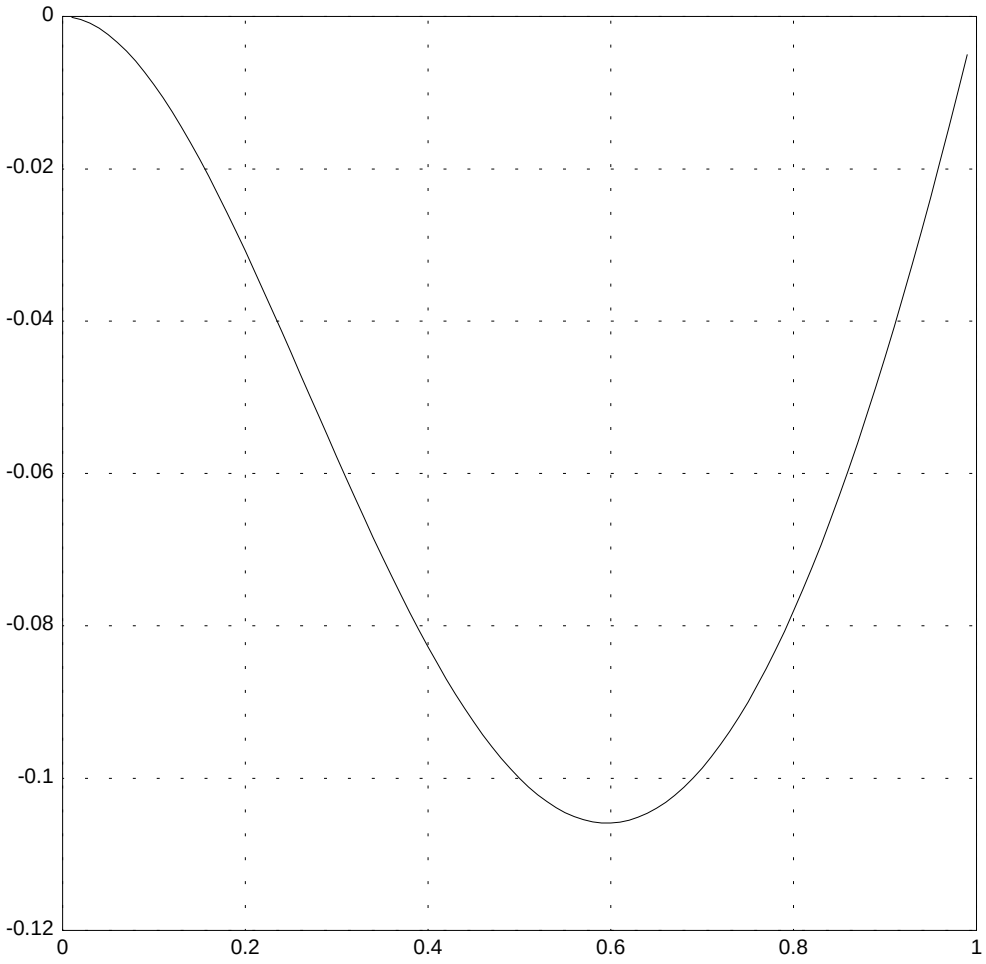

Figure 5.18: Graph created by the *integrator*

5.5 Summary

In this chapter we have presented the implementation details of all our system parts. We first described the implementation of the MPSS tools, — all implemented as interpreted scripts — namely, the paradigm description compiler *pdc*, the instance variable detector *instancev*, the private variable protector *protect*, and the multiparadigm link editor *mpld*. We then described the implementation of each of the paradigms supported by the *blueprint* multiparadigm programming environment. The *bnf* (BNF grammar), *regex* (regular expression), and *imper* (imperative) paradigms were all implemented using existing tools. The *term* (term description), *btrack* (logic programming), and *fun* (functional) paradigms were implemented using the *blueprint* facilities. We finished by describing the implementation of the *integrator* multiparadigm application.

Function	Paradigm	Module	Lines
Symbolic integration	btrack	sint.pb	127
Lexical analysis	regex	scan.pl	47
Expression parsing	bnf	parse.py	76
Numeric integration	fun	aint.pf	75
Interfacing	term	ui.pt	131
Graph creation	imper	main.c	51
Total	blueprint		507

Table 5.5: *Integrator* line count table

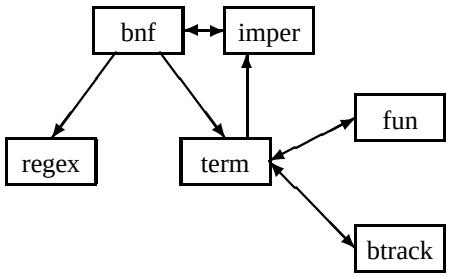

Figure 5.19: *Integrator* paradigm inter-operation call graph

Chapter 6

Critical Analysis

In this chapter we try to provide a critical evaluation of our work. Different aspects of our work must be evaluated using different evaluation criteria. The aspects that we evaluate are: its contributions to multiparadigm research, its standing as a possible programming language offering, the process support available under the MPSS environment, and multiparadigm programming support under the *blueprint* system.

6.1 Multiparadigm Research Contributions

6.1.1 Multiparadigm System Structure

We started our work on multiparadigm programming by describing a suitable structure and approach for developing multiparadigm systems. Existing systems can be divided into multiparadigm applications built by combining single paradigm languages, special purpose multiparadigm languages and general purpose multiparadigm programming frameworks. All three, attack the problem of multiparadigm programming in an *ad hoc* way, at different levels of abstraction. In our work, we defined these levels of abstraction by distinguishing between a multiparadigm application, a programming environment, and an environment generator. Thereby, different approaches can be developed for each abstraction level, and each approach can be designed and evaluated independently.

We based our approach on object-oriented principles: structuring paradigms using a class tree. The compilation of code written in a given paradigm was abstracted as a class method for that paradigm. Class methods, were furthermore used to encapsulate the run-time behaviour of paradigms, thereby isolating them from unwanted interactions. Inheritance was used to provide a basis for common characteristics across paradigms. Thus, in contrast to other object-oriented approaches to multiparadigm programming, we used objects to capture the behavioural, as well as the semantic model of a multiparadigm system. Furthermore, our approach considers the target architecture to be an integral part of our model extending the multiparadigm environment down to the hardware implementation level.

For the communication between modules written in different paradigms we developed the *call gate* abstraction. This is used as the mechanism to convert the tree class model view seen by the multiparadigm programming environment developer into a

flat multiparadigm environment seen be the application programmer. The *call gate* abstraction, is a global inter-paradigm communication mechanism, with only local communication complexity overhead i.e. each paradigm need only implement the calling mechanism for its super-class.

Our approach provides a multiparadigm environment development process model that can be used to engineer multiparadigm environment generators. The object-based encapsulation of each paradigm allows virtually unlimited extensibility of the system. Furthermore, the integration of the hardware architecture in our model, allows efficient, orthogonal implementations.

On the other hand, the rigidity of the development process that we provide may preclude more innovative approaches. Our model, as described, does not provide concrete support for reasoning on a global inter-paradigm scale. Furthermore, the weak coupling of the paradigms might limit the functionality of the system, as possible synergistic effects that could be exhibited between strongly coupled paradigms will not be seen in our system. Also, the incorporation of the hardware architecture into our design model may limit an implemented system's portability, and the *call gate* abstraction can be inefficient in the case of frequent calls between semantically distant paradigms. Since multiparadigm application programmers are unaware of a system's the class structure, this inefficiency may come as a surprise.

We feel that the most important refinement in our system would be stronger support for reasoning about programs. This could be implemented at a local level for each paradigm by developing the equivalent of a *logic call gate*, which would allow the import of external theories into the reasoning model of the logic used by each paradigm.

6.1.2 Multiparadigm Environment Generators

There are not many systems that provide a framework suitable for implementing multiparadigm programming environments. The inter-process communication facilities provided by the Unix operating system can be used as a basis for creating a multiparadigm environment as described in section 2.3.1 (page 30). In addition the MLP system, described in section 2.3.2 (page 30), could be used as a platform for developing multiparadigm programming environments by utilising the support it provides for cross-language procedure calls.

Our system provides a complete framework for integrating existing tools and new paradigm implementations. These are uniformly described using class definition files as outlined in section 4.3.3 (page 74). Special tools are provided to ease the effort of integrating existing implementations of specific paradigms into the integrated system. The multiparadigm link editor allows the linking of arbitrary paradigm classes into a single executable file.

The advantages of our system lie in its efficiency, as the multiparadigm application will be implemented as a single executable file, avoiding expensive kernel context switches, and modularity as the paradigm description files isolate the paradigm implementations from each other. Furthermore, by making it possible to use existing tools, the implementation effort of multiparadigm programming environments can be significantly reduced.

On the negative side, the system as currently implemented, requires from the multiparadigm environment developer, a fair amount of understanding of the paradigm im-

plementation mechanisms. The implementation support provided is relatively low level; support for value mapping, calling convention normalisation, and type checking across paradigms must be implemented by hand. The same lack of automatic support is apparent during the design effort of a multiparadigm environment: no tools are available for checking consistency across paradigms, or browsing the paradigm tree class structure. On the implementation side, the system as currently implemented, is closely tied to the functionality provided by the Unix operating system. Its portability to other platforms is questionable. Finally, having used the system for a single prototype implementation leaves some questions on its robustness and completeness.

Most of the possible refinements of the system could be made possible through its wider and more extensive use. Using the system in more applications would provide us with better insight into the design issues of such systems. Hopefully, a more extensive use of the system would provide the necessary clues for understanding the type of high level design and implementation support tools needed. Furthermore, the system could be enhanced by providing generic (i.e. paradigm and implementation method independent) multiparadigm debugging and program instrumentation support.

6.1.3 Multiparadigm Programming Environment

In contrast to the sparsity of multiparadigm environment generators, there are many systems supporting multiparadigm programming. These were categorised in chapter 2 (page 3) into new languages, language extensions, and support systems.

Multiparadigm programming environments based on our approach provide isolation and weak coupling between an arbitrary number of paradigms. The *blueprint* system allows the application programmer to implement a system using any combination of the six paradigms provided. Some of them were based on tried, existing tools, thus providing efficient and reliable implementation of those paradigms. The model of the system presented to the application programmer is a flat collection of paradigms that can be mixed and matched according to the system's needs. Finally, uniquely in our approach, features common to a number of paradigms were abstracted, and implemented by delegating their realisation to a superclass level.

The system, as implemented, provides a wide range or programming paradigms that can be used to implement a multiparadigm application. Due to the weak coupling between the paradigms, it is easy to reason about any of them, using the logic associated with that paradigm.

On the other hand, as *blueprint* is a prototype implementation, some paradigms are implemented in a suboptimal manner[1]. Thus the execution speed of applications heavily relying on those paradigms will suffer. In addition, the system requires the programmer to know how the paradigms interface with each other. Some interfaces (e.g. *fun-term*) are less transparent than others; the former require from the application programmer to pay attention to the types and values of the parameters used in cross-paradigm calls.

Blueprint could be refined by providing type checking and data mapping support for calls across paradigms. Thus, the programmer would not need to worry about compatibility of data values across different paradigms. Some of the paradigms could

[1] The implementations of *fun* and *btrack* are based on theoretical — non-optimised — interpretation models.

have been implemented an order of magnitude more efficiently by utilising existing industrial-strength compilers that are available for them.

6.1.4 Multiparadigm Programming Applications

We did not find in the literature many descriptions of applications implemented in multiparadigm languages or environments. The few that we found, are described in section 4.5.4, page 96.

Our implementation of the *integrator* application illustrates a practical application using a number of programming paradigms. The *integrator* uses all six paradigms provided by *blueprint*. We feel that each paradigm was used, because that part of the integrator was best expressed in it. However, compared to the commercial mathematical software packages described in sections 4.5.4 and 4.5.4 (page 96), *integrator* is a toy application. Its uses are very limited; it should best be thought as a demonstration of technology, rather than as a mature application program.

We believe that there are many applications that can benefit from being implemented in a multiparadigm programming environment. Mathematical packages form one of the possible application domains, and *integrator* can definitely be extended to handle more interesting problems. Many other applications can also be implemented, in order to further prove the viability of multiparadigm programming.

6.2 Evaluation as a Programming Language

In the following sections we will evaluate our system using classical programming language evaluation criteria [GJ82, pp. 255–265].

Simplicity

Simple languages are easy to learn, and reason about. Our system provides simplicity by separating the different paradigms into different modules, and only providing weak coupling between the paradigms. Of course, a multiparadigm environment will, by its very nature, be more complicated than a language based on a single paradigm. One other source of complexity in our system is the data mapping needed in order to pass values between the interfaces. As we indicated in section 6.1.3 this complicated task can be eased by automating it.

Expressiveness

The provision of many paradigms by our system positively contributes to the expressiveness of it. The programmer can choose the paradigm that is best suited for the part of the application he is trying to express.

Orthogonality

A programming language is orthogonal when its constructing elements can be combined without any arbitrary restrictions. A classic example of a language lacking orthogonality is Pascal, which places a number of arbitrary restrictions on the types of

elements that can be passed and returned from functions, the type declarations, and initialisation constants [Ker81]. Our system is orthogonal on the type of limited the interaction allowed between its paradigm components: the same mechanism is used to communicate between any two paradigms. On the other hand, most of the features provided be each paradigm can not be used in modules expressed in other paradigms; this is a severe lack of orthogonality, which we felt was necessary for the sake of simplicity. A more orthogonal approach should be based on a unifying logic, such as the one described in [GM87], but would be less extensible than ours.

Definiteness

Definiteness is an aspect of the programming language description. Programming languages described in vague and imprecise terms will be difficult to read and implement. We tried to provide support for definiteness in our system, by explicitly requiring paradigm documentation as part of a paradigm class. The weak coupling between the paradigms makes the description of paradigm interactions straightforward. However, the bottom line lies in the definiteness of the actual paradigm description. We feel that this area could be improved.

6.2.1 Readability

We feel that our approach enhances the readability of multiparadigm programs, by isolating the paradigms in different units. In this way, the understanding of a module's functionality only requires following the abstraction model of a single paradigm. Furthermore, the breadth of paradigms provided will allow declarative descriptions of many aspects of the problem, thereby increasing the readability of the overall system. On the other hand, a programmer must be proficient in all the paradigms used by the system.

6.2.2 Reliability

Programs written using our multiparadigm programming approach can be more reliable than programs written in a single paradigm in cases where the utilisation of the multiparadigm approach yields programs that are smaller and consequently contain less errors. The weak coupling between the paradigms should furthermore increase program reliability, by minimising unwanted interactions between paradigms. On the implementation side, reliability is enhanced by support tools like *protect*, and the multiparadigm linker *mpld* described in sections 4.3.5 (page 75), and 4.3.6 (page 77). Furthermore, existing, field tested, paradigm implementations can be utilised. However, the modularity of the system, and the large number of components used, can be a source of unreliability, especially in cases where new versions of existing tools coming from diverse sources are introduced in a stable, functioning system.

6.2.3 Efficiency

We distinguish efficiency in terms of CPU utilisation (time efficiency), and in terms of memory utilisation (space efficiency).

Time

Our system has great potential for compiling programs that will be executed in the least possible time. The separation between the paradigms and the support for existing implementations help by ensuring that each paradigm can be translated without taking into account complexities of other paradigms, and by making it possible to use compilers that generate the most efficient code. The module division between the paradigms allows the implementor to choose classic efficient compilation approaches described in the literature, instead of having to invent new compilation schemes. In addition, the class structure of our system allows for multiple optimisations to be performed at different levels of the paradigm class tree. For example, we can envisage that a declarative logic programming paradigm can be optimised at its class, by providing mode annotations using abstract interpretation, further optimised at the imperative level by doing a global data flow analysis, and optimised a third time at the target machine paradigm class by instruction scheduling.

Space

We do not consider our system to be very efficient in terms of space overhead needed by the various paradigms. There are two sources of this inefficiency: the modularity of our approach which requires a separate run-time mechanism for each paradigm, and the inter-paradigm calls which require space-consuming mapping of variables from the type expected by one paradigm to the type expected by the other. On a brighter side, it is possible to abstract behaviour common to two paradigms into a superclass of the two, thus saving the space that would have been required by two separate implementation mechanisms. We did this in *blueprint*, by providing support for the handling of terms used by the functional and logic programming paradigms, in the *term* paradigm superclass, as described in section 4.4.4, page 80.

In summary, we believe that our approach can be favourably compared to existing programming languages. The programming language evaluation criteria as applied, revealed an expected variety of the tradeoffs that are common in programming language design. We feel that our choices were reasonable and justified.

6.3 MPSS as a Process Support Environment

MPSS can be evaluated as a multiparadigm environment generator and, more generally, as a process support environment. The process that MPSS is supporting, is that of the design, implementation, and evolution of multiparadigm programming environments. Our process support environment evaluation criteria are based on [Ste87]. A more detailed evaluation methodology can be found in [Wei87].

6.3.1 Process Support and Evolution

A process support environment should support the underlying process, and provide a basis for its evolution. MPSS is specifically targeted towards the design and implementation of multiparadigm programming environments. It was built in order to support the object-oriented abstraction based process, that we envisaged as central to

the development of such systems. It is designed to be used together with the other tools available under the Unix operating system, supporting databases, configuration management, and version control. One can criticise it, because it fails to provide an integrated environment where all these facilities can be used together in a coherent way. As MPSS is built as a set of cooperating tools that use the Unix communication mechanisms, it can be easily extended, and thus support the evolution of our process.

6.3.2 Integration with the Conceptual Schema

A process support environment must be properly integrated around a well-defined conceptual schema [Ste87, p. 34]. As we indicated in the previous section MPSS is based on, and very tightly integrated with the conceptual schema (paradigms as object classes) of our process. We found it a great help when implementing the *blueprint* multiparadigm programming environment. Some of the shortcomings of the initial versions were identified in that stage and are now rectified. The support for our process's conceptual schema could be further improved as explained in section 6.1.2.

6.3.3 Evolution

The software process is — like the software itself — a constantly evolving entity. The support environment should therefore be capable of following and supporting this evolution. As explained in section 6.3.1, MPSS depends on the tools available under the Unix operating system for many areas of the process support. Although this means that MPSS does not provide a tightly integrated support environment, it provides the advantage that new and enhanced tools can be easily integrated into the support environment, by simply making them compatible with the Unix and MPSS system conventions.

6.4 *Blueprint* as a Programming Environment

In this section we will evaluate the paradigm support provided by the *blueprint* prototype multiparadigm environment implementation. We will base our analysis on paradigm support evaluation criteria described in [Bob84, KTMB86].

6.4.1 Linguistic Support

A system provides linguistic support for a paradigm when there is a close correspondence between a program's linguistic expression and the programmer's intentions. We believe that in *blueprint*, the correspondence between the program's linguistic expression and the programmer's intentions is comparable with the state of the art for the paradigms implemented. In the paradigms implemented by existing tools (imperative, BNF grammar, and regular expression), the tools provide a close linguistic mapping. The logic programming paradigm closely resembles the Prolog syntax, which is widely used for programming in that paradigm. The functional paradigm, allows the representation of functions as first class objects. It lacks however, many features that are usually found in functional programming languages, such as list comprehensions, polymorphic type checking, and pattern matching.

6.4.2 Program Semantics

The semantics of programs implemented in the various paradigms should be clear and simple. The programs implemented in each paradigm should be easily understandable by humans, and also susceptible to machine analysis and transformation. Again, we feel that the semantics of the paradigms supported are clear, and — where appropriate — susceptible to machine analysis and transformation. In particular, the clear division between the different paradigms allows for standard techniques, documented in the literature, to be applied to the appropriate paradigms. The cases of inter-paradigm communication must be treated in a special way. Some propositions for reasoning in mixed-paradigm systems, were listed in sections 2.2.1 (page 8) (functional with logic programming), 2.2.2 (page 10) (imperative and logic programming), 2.2.3 (page 16) (imperative and functional), and 2.2.7 (page 24) (imperative, functional, and logic programming). General techniques for dealing with this problem have been proposed in [Zav89]. Unfortunately, in our approach, the semantics of the whole system can only be understood in terms of the root class paradigm; in our case the imperative paradigm.

6.4.3 Execution Support

The system must efficiently support the execution of code written in every paradigm. We think that our system supports the rapid execution of the parts written in the imperative, BNF-grammar, regular expression, and rule-rewrite paradigms. The execution speed of programs written in *term* is about the same as that achieved by popular compiled-Prolog systems such as SB-Prolog [Deb88]. The logic programming, and functional paradigms are implemented in a suboptimal (interpreter-based) way, which results in slow program execution. The execution speed of programs written in them is about an order of magnitude slower than that of compiler-technology based declarative systems such as New Jersey ML [AM87] and SB-Prolog [Deb88]. Furthermore, the rule-rewrite paradigm and its subclasses lack a garbage collection mechanism, and therefore consume inordinate amounts of storage space.

6.4.4 Error Reporting, Tracing, and Monitoring

Tracing and monitoring facilities are specifically targeted towards program debugging. The view of the program state should be the one most appropriate to the paradigm. *Blueprint* provides some debugging information on the program being executed. We have tried to use the most appropriate format, for the debugging information associated with each paradigm, and therefore debugging can always be performed at the programming paradigm level, rather than at the implementation paradigm level (initially our debugging for all paradigms was done using a C language symbolic debugger). Still, the debugging support provided for most paradigms is in the form of a trace listing. Higher-level support should be provided in the form of a multiparadigm symbolic debugger.

6.4.5 Analysis and Performance Tuning

Analysis and performance tuning tools are needed in order to improve the performance of real-world applications. Performance analysis tools enable the programmer to make meaningful time-space tradeoffs and to direct his optimisation efforts to the program parts from which the highest performance benefit can be expected. The tracing output mentioned in the previous section, can be used as a primitive instrumentation facility. We have used it in some tests, in order to measure the number of calls made for a given predicate. Although there is a lot of room for improvement, such improvement will only be meaningful in conjunction with more efficient paradigm compilers.

6.4.6 User Interface Tools

A programming environment can be enhanced by the existence of user-interface editors, and automated application builders. As *blueprint* uses a number of standard paradigm representations and the Unix standard calling conventions most user-interface tools (such as [BCH+90]) that generate code compatible with one of the paradigms supported by *blueprint* can be used.

6.4.7 Peaceful Paradigm Coexistence

In a multiparadigm programming environment, each paradigm should be well integrated with the rest, with no unwanted interactions. This is achieved in *blueprint* by the weak paradigm coupling. Thus, within code implemented in a single paradigm there are absolutely no interactions from other paradigms. Functions from other paradigms must be explicitly imported and used. In that case however, the semantics of the paradigm using those functions are diluted by the semantics of the imported paradigm's functions. As an example, importing an *imper* function into *fun* can be used to create functions with side effects. The runtime mechanisms of all paradigms are also well isolated, thanks to the object-oriented class encapsulation provided by MPSS.

6.4.8 Support for New Paradigms

A system supporting the transparent addition of new paradigms can be a significant asset for the evolution of programs implemented in it. *Blueprint* does not provide such support. However, its class-based implementation using the MPSS generator, guarantees that paradigm additions will be easy, and transparent. Arbitrary paradigms can be added to the *blueprint* environment, either as subclasses of existing paradigms, or by making the entire *blueprint* structure a subclass of another paradigm.

6.5 Summary

In this chapter we attempted a critical analysis of our work, and specifically, its contributions to multiparadigm research, its standing as a possible programming language offering, the process support available under the MPSS environment, and multiparadigm programming support under the *blueprint* environment. Our main contributions

to multiparadigm research, are an identifiable multiparadigm system structure, the description of multiparadigm environment generators, the object-based design of multiparadigm programming environments, and the documentation of a multiparadigm application. We evaluated our system using the programming language evaluation criteria of writability, readability, reliability, and efficiency. We examined the process support offered by MPSS, by looking into its scope, integration with the conceptual schema, and evolution. Finally, *blueprint* was evaluated as a programming environment, in the areas of linguistic support, program semantics, execution support, monitoring, and paradigm support.

Chapter 7

Future Work

Multiparadigm programming is a relatively new research area in computer science. Any research in it is bound to raise many new questions; this thesis is no exception. In previous chapters we already hinted at possible enhancements to our work. In this chapter we summarise and categorise these enhancements, and outline some promising new research directions. We will start by describing how our approach can be improved, proceed by outlining possible enhancements to the design of MPSS and *blueprint*, and finish by outlining possible new applications of our work.

7.1 Approach Improvements

Our approach, although completely guiding us in the design and implementation of the MPSS, and *blueprint* prototypes, has only very lightly touched three important aspects of multiparadigm systems. These are:

- a development methodology for multiparadigm applications,
- the formal semantics of multiparadigm systems, and
- type checking across paradigms.

In the following sections we will discuss each one of these aspects in turn.

7.1.1 Development Methodology

A development methodology, is a set of rules that guide the application programmer in the task of designing the application. Most programming paradigms, are associated with a suitable design methodology. Examples are the structured analysis / structured design method which is mainly applicable to the imperative paradigm, and the explorative, or rapid prototyping design method which is associated with the logic programming paradigm, or interpretative implementations of the object-oriented paradigm. There is no methodology, specifically targeted towards multiparadigm programming. Such a methodology would guide the designer in decomposing the problem, choosing the appropriate programming paradigms, and dividing the implementation work across the paradigms. The methodology would have to take into account the breadth of characteristics that would exist among the paradigms. Factors like the abstraction level

of each paradigm, the ability to reason about specific parts of the program, execution efficiency, and the cost of inter-paradigm communication, would have to be part of the methodology's decomposition criteria. The methodology should of course extend throughout the system's life-cycle, and therefore, the relative modifiability and maintainability characteristics of different paradigms, should also be taken into account.

7.1.2 Formal System Semantics

Some of the paradigms used in the *blueprint* environment, are based on an underlying logic that allows one to precisely define the language and program semantics. Unfortunately, the same is not true for the resulting, composite, multiparadigm environment. As we indicated in section 6.4.2 (page 132) the reasoning level of the composite system, is that offered by the root paradigm, in our case the imperative paradigm. Even that, is a theoretical possibility, due to the diverse systems used in creating *blueprint*: none of the systems is described by rigorous semantics. One possible solution to this problem is a unified underlying logic on which the whole system will be based. In section 2.4.3 (page 36) we criticised this approach as lacking modularity and extendibility. An approach compatible with the modular composition of our system, would be a method for *interfacing* logics of different paradigms, in a way similar to the mechanism used for interfacing the inter-paradigm calls, using call gates. As with call gates, only a small number of such interfaces would be needed to cover the whole paradigm tree. However, the procedure for translating a logic theory from one paradigm to another, is likely to be extremely complicated. For this reason, we feel that this method is only suitable for automatic implementation, on a computer aided multiparadigm reasoning system.

7.1.3 Type Checking Support

Our approach does not deal with the type systems of the underlying paradigms. We came away with this omission, by imposing the responsibility for type correctness across paradigm calls on the application programmer. Although this can be acceptable — and indeed practical — for a prototype system, support for a type-system is indispensable for real multiparadigm implementations. The problem of supporting types across paradigms, is due to the diversity of the type systems that are found on different paradigms. Thus, the approach must handle anything, from the typically typeless logic programming languages, to the strong polymorphically typed functional languages, to the run-time dynamic type checking found on some object-oriented implementations. We feel, that this significant challenge should again be solved by a method similar to the one used for call interfacing using call gates. Each paradigm should be able to convert the types of procedures imported from its superclass to the types supported by it, and the the types of its exported procedures to the types supported by its superclass. A database containing the type information from the definition modules of all files, will be implemented at the root paradigm class level, as illustrated in figure 7.1.

The methods for performing this type conversion must exist at compile-time, in the form of specialised *type compilers*, and — where required[1] — at run-time as well.

[1] E.g. dynamic type checking.

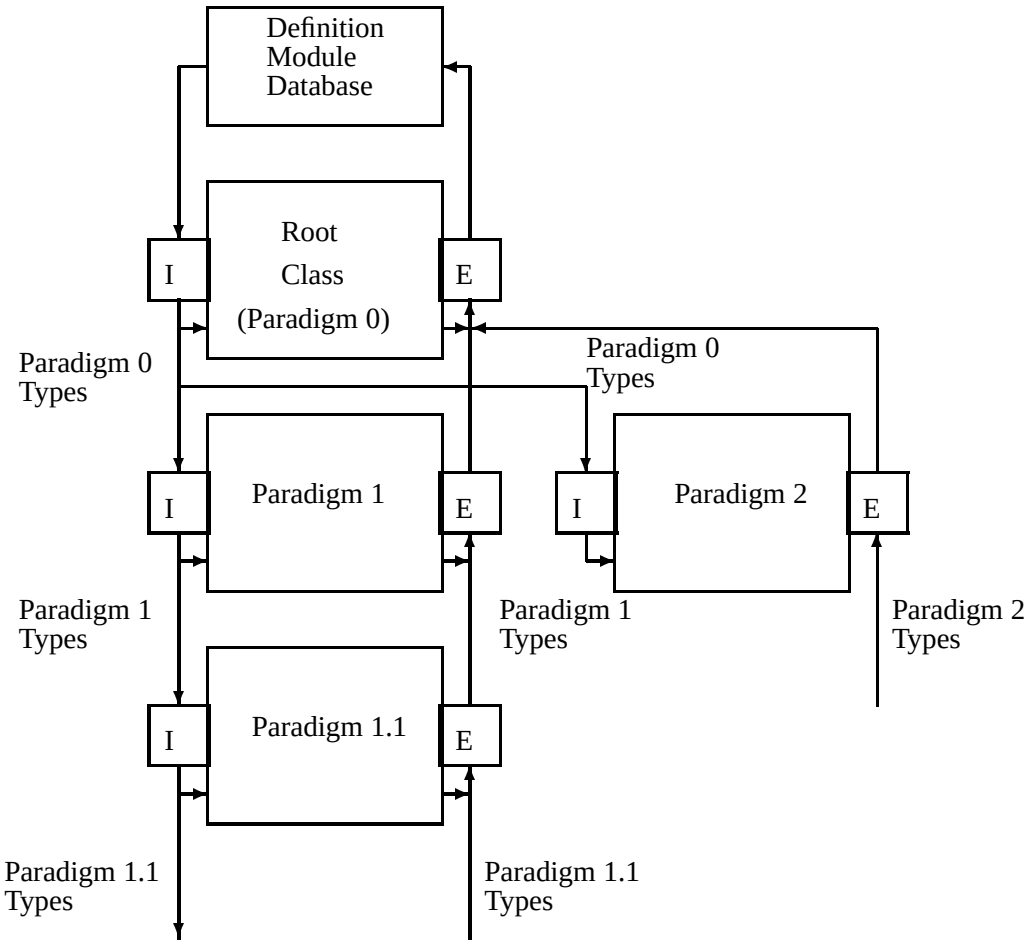

Figure 7.1: Multiparadigm type checking using type gates

Every source file will have a definition module associated with it. All procedures imported and exported from that module will be listed in that file. A paradigm-specific type converter will compile the definition module to conform to the type system expected by the paradigm's superclass. It will also perform type checking for procedures that match at that level.

This approach, although practical, has the disadvantage that the quality of type systems of paradigms between two communicating paradigms affects the quality of the type checking between them. For example, if two strongly typed paradigms have a typeless paradigm as their common superclass, all type checking between them will be eliminated. Similarly, the communication of two paradigms that have as a common superclass a run-time type checking paradigm, will have the overhead of run-time type checking. Judicious arrangement of the paradigms in the class tree is therefore very important.

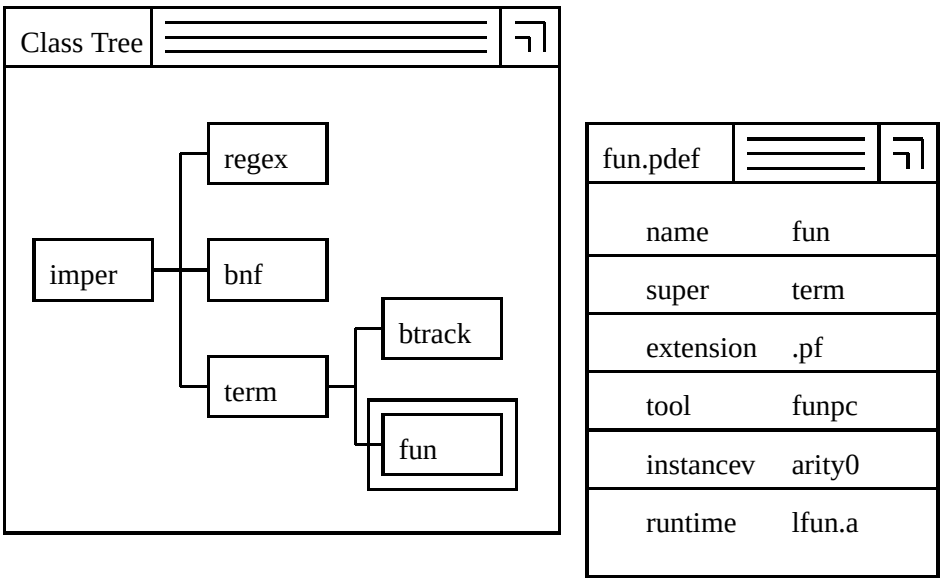

Figure 7.2: Paradigm class browser interface

7.2 MPSS Enhancements

We will now examine the specific improvements that could be made to the prototype implementation of the MPSS generator. MPSS was designed with the narrow scope of proving the practicality of our approach. Therefore, tools that were not absolutely needed for that goal, were not implemented. In the following sections, we will roughly describe some possible tools that would enhance MPSS. These tools, would ease the task of creating multiparadigm environments, and raise the quality of the resulting systems.

7.2.1 Paradigm Class Browser

A multiparadigm programming environment can have a very complicated class structure. Currently the structure of the system, is reflected by the variable relationships as described in the various class definition modules. This can be a source of confusion and errors. High quality object-oriented environments offer a *class browser* [Gol80, pp. 297–307]. We envisage, that such a tool, would be also very useful in the case of MPSS. The paradigm browser should provide a graphical picture of the system's structure, and allow the user to interactively modify the structure and its elements. Figure 7.2 illustrates a hypothetical screen dump of the browser in operation.

The browser would automatically maintain the integrity of the structure, and ensure that the classes were always connected in the form of a tree. The user should be able to view and change the elements of each paradigm class, by identifying the class and the element, using a "point and click" interface. Additionally, the browser could be extended to be the front end of the whole multiparadigm environment generator, offering controlled access to the source code of the paradigm implementations, and the other related tools.

7.2.2 Name-space Verification

The multiparadigm programming environments produced by MPSS create the final executable program file, by linking the object modules of the various source files, together with the run-time libraries of the associated paradigms. The run-time libraries are typically developed independently from the multiparadigm programming environment. Consequently, there is a possibility that symbol names used by one paradigm implementation, may conflict with symbol names used by another. The *instancev* tool handles all such cases, where the symbol names, appear in every module generated by the compiler of a paradigm. It does not however, check for global variables that might conflict across paradigms. This task, must be manually performed by the programmer. When the programmer discoverers such conflicts, he must devise *ad hoc* solutions depending on the type of the problem. This task could be automated, by a special tools that would search all the libraries and object modules containing paradigm support code [2] and report any global name conflicts. An improved version of this tool, could resolve the conflicts by using a suitable renaming strategy.

7.2.3 Type Checking Support

As we indicated in section 7.1.3, our approach would benefit from supporting type checking across paradigms. The approach we outlined in that section, could be performed by MPSS using an enhanced version of the paradigm compiler. The paradigm class definition files would be augmented with a section describing the type system of each paradigm using a sufficiently general notation, possibly with references to strategies implemented as class methods. The paradigm compiler, would then compile the type system descriptions into type call gates, that would provide the interface from the type system of each paradigm to the type systems used by the other paradigms.

7.2.4 Automatic Call Gate Implementation

Currently, the call gates for each paradigm must be hand-crafted by the multiparadigm environment implementor. Using an approach similar to the one outlined in the previous section for the automatic type call gate implementation, the calling convention type mapping, and hence the implementation of call agates, could also be automated. In this case, the paradigm class definition files would be augmented with description of the calling conventions used in each paradigm, again complemented by program code, if needed. The paradigm description compiler, would then use this description, to create the import and export call gates for each paradigm.

7.2.5 Debugging Support

Program debugging in the multiparadigm environment, is currently separately implemented for each environment developed. MPSS does not provide any support for this task. As we hinted in section 6.1.2 (page 126) MPSS could ease the task of creating facilities for multiparadigm debugging, by providing a multiparadigm debugger. This would be a generic support tool, similar in nature to the multiparadigm link editor: it

[2] Compiler-generated object modules are checked by *instancev*.

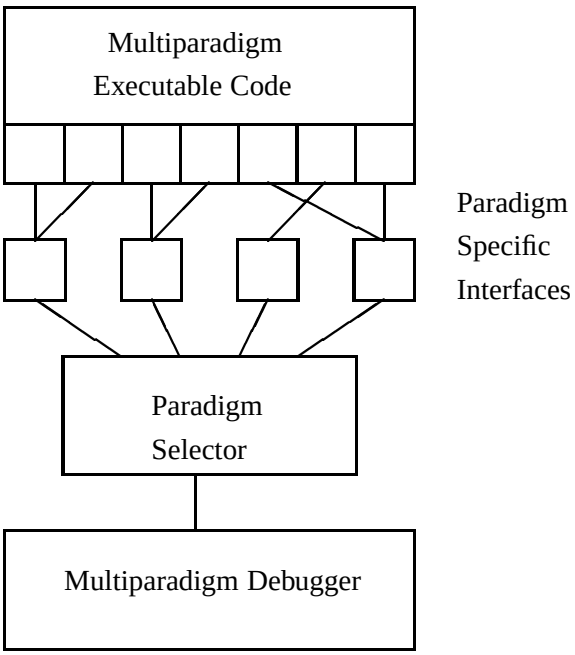

Figure 7.3: Multiparadigm debugger structure

would be provided by MPSS to be distributed with every multiparadigm programming environment developed under it. Such a debugger would offer a common interface for debugging code implemented in any paradigm. A possible structure for such a system is shown in figure 7.3.

Paradigm specific issues, such as data and code representation would be handled by special routines that would be provided by each paradigm (possibly described in the paradigm class definition file). Furthermore, the handling of the code in its compiled form, would be made possible by special debugging hooks provided by the implementations of each paradigm. Such hooks would provide the ability to insert breakpoints into the code, and modify data values. The interfacing between the paradigm specific interfaces, and the multiparadigm debugger, would be handled by a special *paradigm selector* module. That module would intercept requests form the debugger, and direct them to the appropriate paradigm specific module. A special tool would be needed, to map the debugging support provided by existing paradigm implementations, to the format expected by the multiparadigm debugger.

7.2.6 Instrumentation Support

Instrumentation support for the multiparadigm programming environment should also be provided in a form similar to that provided for debugging, i.e. a multiparadigm call graph analyser and profiler. These tools would probably utilise the same hooks provided for debugger support, to create a call graph and a timing profile of the executed code. An interesting feature of such a tool would be the capability to categorise the time among paradigms, and also — where applicable — indicate what percentage of the time was spent executing paradigm code, and what part was used by the paradigm support machinery (e.g. the garbage collector). This could help the multiparadigm

application implementor, decide which parts to implement in which paradigms.

7.2.7 Other Language Tool Support

Programmers often use a number of language-specific tools to enhance their productivity. Specific examples are:

- source file tag generators that create index files for all program public symbols (such as the Unix *ctags* utility),
- pretty-printers and source formaters (such as the Unix *tgrind*, *cb*, and *indent*) utilities, and
- language-specific editor modes (e.g. the *c-mode* of the *Emacs* [Sta84] editor.

All these can be handled in a multiparadigm environment by adding new methods to the paradigm description class. Thus, every paradigm description will be augmented with specific methods for the features indicated. The method descriptions can be declarative (e.g. a list of keywords that must be emboldened for pretty-printing, or a regular expression for locating public symbols), or imperative for more complex tasks (e.g. Emacs-lisp code for an editor mode).

7.3 *Blueprint* Enhancements

The *blueprint* prototype multiparadigm programming environment can be improved in three different ways:

1. more efficient paradigm implementations,
2. additional programming paradigms, and
3. incorporation of MPSS enhancements.

In the following sections we will offer more details on these possible enhancements.

7.3.1 Efficiency

The paradigm implementations provided by *blueprint* are implemented, as mentioned in section 6.1.3 (page 127), in a suboptimal way. The implementations of the functional and logic programming paradigms, are merely technology demonstrations, and are not suitable for heavy-duty work. The execution speed of the code generated for those two paradigms is very slow, because the implementation is based on an interpreter. The execution speed can be vastly improved, by implementing a new compiler, or using one of the existing compilers, adapted for *blueprint*. In addition, the implementations of *btrack*, *fun*, and *term*, consume inordinate amounts of memory space, which are never returned to the application memory pool. This situation could be improved by an improved implementation of *fun* and *btrack*. For *term*, we had identified during its design, that many of the memory allocations were superfluous, and could thus be eliminated with some additional management overhead. Furthermore, there are many optimisations that are possible on the *term* code generation, such as tail recursion elimination, will also result in significant memory savings.

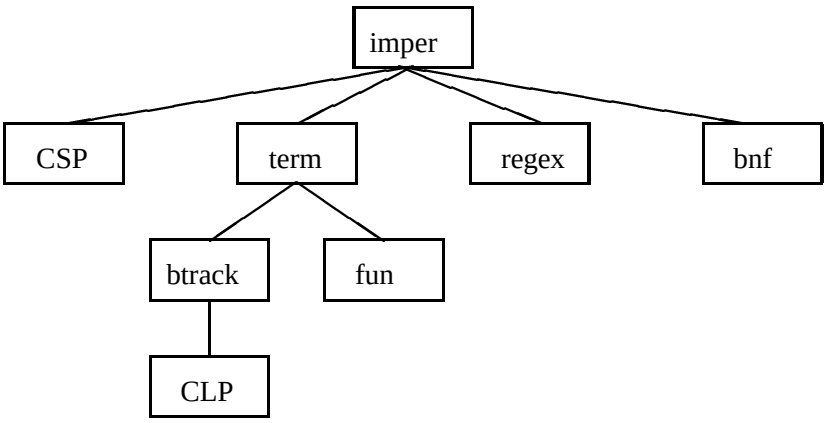

Figure 7.4: Blueprint paradigm extensions

7.3.2 Additional Paradigms

Another way to make *blueprint* a more attractive programming environment would be the provision of additional programming paradigms. Two paradigms that would easily fit the model and scope of the system, are a paradigm based on communicating sequential processes (CSP), and a constraint logic programming paradigm (CLP). The class structure of such a system is illustrated in figure 7.4. In the following sections, we will provide a short description of the way each paradigm could be added to the system.

Communicating Sequential Processes

A communicating sequential process [Hoa78] paradigm, would probably be a subclass of the imperative paradigm, in the case of a single processor machine. Under the SunOS operating system it would be implemented by using the lightweight process (LPW) library and the associated routines. The call gate for communication with the other paradigms should be provided by the root process, in order to ensure that no unwanted interactions result, from the reentrancy limitations of the underlying C library. By this approach, we can ensure that whenever the call gate is used, the system is in a stable state, and it is not processing a C library function within an execution thread.

Constraint Logic Programming

Constraint logic programming would naturally be provided as a subclass of the logic programming paradigm. The constraint logic programming implementation would offer additional methods for constraint solving, inheriting from the logic programming superclass all other methods.

7.3.3 Integration of MPSS Improvements

Most of the MPSS enhancements that were mentioned in section 7.2, will apply to the *blueprint* implementation, making *blueprint* a higher quality programming environment. The most significant improvements on *blueprint* will be the provision of type

checking, data mapping, and debugging.

Type Checking

Currently type checking within *blueprint* is only performed at the level of the *imper* paradigm. The types used in the calls among the other paradigms are not checked. During the development of the *integrator* application, we spent time chasing errors, that a type checking system would have identified at compile time. If MPSS had the support needed for easily creating such systems, then the better paradigm implementations used within *blueprint*, would be integrated with the inter-paradigm type checking support mechanism, so that full type checking would be available throughout the system.

Automatic Paradigm Call Data Mapping

The *blueprint* environment, as implemented, requires from the application programmer to convert all values passed between the paradigms using functions, such as those listed in tables 4.4 (page 85), and 4.9 (page 92). Using the automatic inter-paradigm call data mapping facilities of an improved MPSS, data types would be transparently mapped across calls. The application programmer would call functions from foreign paradigms, using the normal paradigm data values, and these would automatically be converted to the appropriate format for that paradigm.

Debugging and Instrumentation

Finally, the multiparadigm debugging and instrumentation facilities provided by the enhanced version of MPSS would be used on the code created by *blueprint*. The data needed for the multiparadigm debugger is already provided by the tracing output functions of all paradigms, as detailed in pages 85, 92, and 89. The multiparadigm debugger and profiler, would interpret that data to provide a user-friendly, and consistent multiparadigm debugging interface.

7.4 Interesting Applications of our Approach

In this final section of our chapter, we examine other possible applications of our approach.

7.4.1 Parallel Processor Target Architecture

One possible application of our approach would be the development of a multiparadigm programming environment, targeting a MIMD parallel processor architecture. Such architectures, are supposed to be ideal execution vehicles for declarative paradigms such as the logic programming and the functional paradigms, but also traditionally offer industrial-strength compilers for the imperative paradigm (typically for the Fortran language). It should be apparent, that basing a multiparadigm environment on such an architecture, can be a rewarding exercise. Such an environment, would

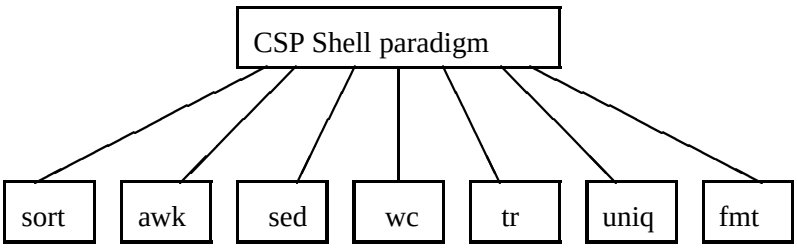

Figure 7.5: Multiparadigm Unix tool composition

be based on a class abstracting the hardware capabilities of the architecture, and providing process scheduling and communication mechanisms. On top of that class, we would then place the imperative paradigm. The declarative paradigms, would — as in the *blueprint* design — share a common superclass. That superclass would handle the allocation of processes to processors, by virtualising the processor pool into an infinite number of processors, thereby potentially simplifying the task of the declarative paradigm implementations.

7.4.2 Multiparadigm Language Development Systems

One area where multiparadigm programming is a long tradition, is the implementation of language processors. There, the lexical analysis is typically specified using a regular expression based paradigm, the parsing using a BNF grammar based paradigm, and the code generation phase is often described in terms of operations on trees [AG85, Tji86, Rin89]. [GHL+92] describe Eli, a system that is supposed to integrate such tools, with the help of an expert system. We believe that our approach, can also be used for such a purpose. The *blueprint* prototype system, already contains support for lex-like lexical analysers and yacc-like parsers. It could be extended to provide support for tree pattern matching, and other paradigms that would increase the productivity of the language processor implementor. Possible candidates would be, a machine description notation paradigm, and a language offering a backtracking capability for register allocation.

7.4.3 Multiparadigm Unix Tool Composition

We noted in section 2.3.1 (page 30), that Unix can be thought off as a multiparadigm programming environment. Our approach can be applied, to integrate many of the Unix system tools, together with the Bourne shell [Bou79], in the form of a single multiparadigm programming language. This integrated language, would be much more efficient than the schemes based on the interprocess communication mechanism of Unix, since all process context switches, would be eliminated.

The superclass of that system, would be a paradigm offering a communicating sequential process abstraction, with a syntax similar to that of the Bourne shell. All other paradigms, would be implemented as direct descendants of that superclass, following the traditional Unix parameter mechanism syntax. A sample class structure for such a system is illustrated in figure 7.5. This system could be implemented, using the MPSS multiparadigm programming environment generator. The *instancev* and *protect* tools would be very valuable for integrating the code of all the Unix tools into a single

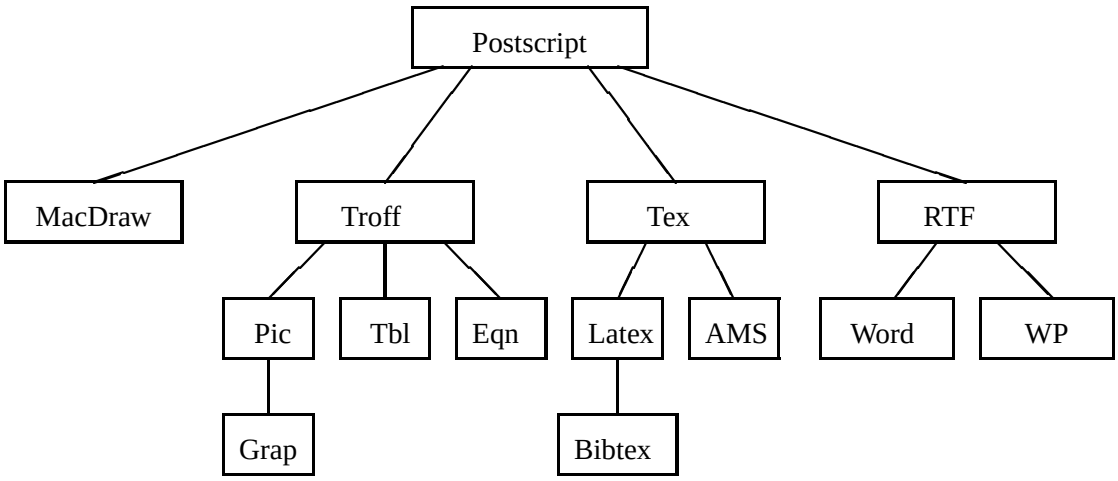

Figure 7.6: Multiparadigm document processing

executable program.

7.4.4 Multiparadigm Document Processing

Programming languages are not the only computer-related area that can benefit from our multiparadigm approach. One other area where a number of paradigms can be found, and multiparadigm programming would be useful, is that of document processing. Exactly as in programming, we can distinguish in document processing declarative document preparation languages such as *pic* [Ker82] , *tbl* [Les79], *eqn* [KC74], and macro packages such as LaTeX [Lam85], and *mm* [MM884], as well as imperative ones such as *MacDraw*, *troff* [Oss79], TeX [Knu89], and *Postscript* [Inc85] . In addition, [MK89] describe *Monk*, a high level text compiler for the sort of declarative constructs implemented by macro packages. Just as in programming different paradigms are suited for different tasks. In addition — in the domain of document processing — it is common for documents to be created by teams whose members use different document preparation tools. A multiparadigm document processing system, would allow the seamless integration of the output of all the paradigms into a final product, such as a book.

A sample class structure of such a configuration is shown in figure 7.6. The root class of such a system, would be a low level page description language, similar to the one used for describing the end product. A possible example would be the Postscript language. The other paradigms would be organised as a class tree in the usual fashion, with lower level languages such as TeX, *troff* and *RTF* [3] acting as the superclasses of higher level languages such as LaTeX, *mm*, *pic*, and *Microsoft Word*. Paradigm communication, in this case means the integration of the output of the different paradigms into the final document. The call gates provided would allow for metric information to be passed from one paradigm to the other, in order to communicate the size of objects expressed in different paradigms across them. Many of the paradigms described,

[3] Rich Text Format: a common inter-application document exchange format.

offer a facility for incorporating Postscript code, this is one more reason for choosing Postscript as the common superclass.

7.4.5 Reduced Feature Languages

We finish our discussion of future work, with a personal view of programming language design issues, as related to multiparadigm programming. While implementing *blueprint*, we found that the modularity of our approach, significantly eased our implementation task. We therefore propose the notion of *reduced feature languages*, languages that do only one thing, well. As each language will be very simple, significant effort can be put into optimising its implementation, by creating a very high quality compiler, producing very efficient code. The small size of each language, could even afford us the luxury of formally defining the semantics of the language, and building a formally proved correct compiler for it. These languages can then be combined, using our multiparadigm programming approach, to create a programming environment, offering all the programming paradigms, as correct and efficient implementations.

7.4.6 Application Specific Paradigms

Finally, we would like to mention the possibility of application specific paradigms. This was already hinted in [Hoa83, p. 37]. With a system like MPSS, the implementation, and integration of new paradigms, can be made part of the application programming process. Thus, the application programmer can develop paradigms, that contain the abstractions relevant to the application domain. Such paradigms can for example be, a menu hierarchy description language, for applications using a menu user-interface system, or an operating system function interface description language, for a system call instrumentation facility like the SunOS *trace* command [SUN90, trace(1)]. The application programmer, would design the language and, using rapid prototyping tools, craft a compiler for that language. The language would probably be compiled into the superclass paradigm of it; therefore, the compilation will be a relatively easy process. The new language can then be painlessly integrated into the programming environment, by writing the suitable paradigm class definition module.

7.5 Summary

In this chapter, we summarised and categorised some possible enhancements to our work, and outlined promising new research directions. Our approach can be improved by describing a precise development methodology, providing formal semantics to multiparadigm environments, and type checking across paradigms. These enhancements can be utilised in the MPSS implementation. Additionally, MPSS can be enhanced by providing a paradigm class browser, automatic *call-gate* implementation, and debugging support. Similarly *blueprint* can be improved by the incorporation of the new MPSS capabilities. Furthermore, the *blueprint* paradigms can be more efficiently implemented, and additional paradigms such as communicating sequential processes, and constraint logic programming can be added to it.

We outlined possible new research directions based on our approach, by describing how it can be used to, create multiparadigm environments on parallel architectures,

provide a conceptual basis for the design of language development systems, implement an efficient Unix tool composition mechanism, and integrate diverse document preparation systems. Finally, we proposed the notions of *reduced feature languages*, based on a single programming paradigm, and *application specific paradigms*, developed to support the implementation of a single application.

Conclusions

Multiparadigm programming allows the programmer to express the implementation of a system in a number of different paradigms. The use of multiparadigm programming techniques, can lower implementation costs, and result in more reliable and efficient applications. In this thesis, after describing the work related to this area, we presented our approach to multiparadigm programming, based on a multiparadigm system developed using object-oriented principles. The problems of multiparadigm programming were separated into the areas of application development in multiple paradigms, design and implementation of multiparadigm environments, and generators for creating such environments. This separation, allowed the methodical study of the different issues and solutions that exist on each level.

For each area, we identified the problems and requirements, and proposed our solution. Based on our proposed approach, we then designed and implemented, the multiparadigm programming environment generator MPSS, providing tools for implementing multiparadigm environments, the multiparadigm programming environment *blueprint* which supports programming in six different programming paradigms, and the *integrator* application utilising the paradigms provided by *blueprint*. The critical evaluation of the implemented systems, and the many exciting extensions of our work, lead us to believe that our novel approach is sound and practical.

Our main contributions to the area of multiparadigm programming have been the clear identification of the problem areas, the study organised along that identification, the use of object-oriented principles in the design of multiparadigm environments, the *call-gate* abstraction for the inter-operation of arbitrary paradigms with linear implementation overhead and flat paradigm structure representation, and the proposal for multiparadigm environment generators based on our approach. These contributions were made concrete, by the design and implementation of systems based on our proposed approach.

The area of multiparadigm programming is still young and under development. We hope that our contribution will provide a basis for new, interesting, and exciting work.

Bibliography

[Abb89] Russell J. Abbott. Set notation as a language to specify data transformation programs. *Software: Practice & Experience*, 19(6):593–606, June 1989.

[Abr86] Harvey Abramson. A Prological definition of HASL: A purely functional language with unification-based conditional binding expressions. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 73–129. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[AG85] A. V. Aho and M. Ganapathi. Efficient tree pattern matching: An aid to code generation. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 334–340, January 1985.

[Aka86] Kiyoshi Akama. Inheritance hierarchy in prolog. In Eiiti Wada, editor, *Logic Programming '86, Proceedings of the 5th Conferene*, pages 12–21, Tokyo, Japan, June 1986. Springer-Verlag. Lecture Notes in Computer Science 264.

[AKN86] Hassan Aït-Kaci and Roger Nasr. Logic and inheritance. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 219–228, St. Petersburg Beach, Florida, USA, Januray 1986. Association for Computing Machinery.

[AKW79] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. Awk — a pattern scanning and processing language. *Software: Practice & Experience*, 9(4):267–280, 1979.

[AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[All83] Elizabeth Allen. YAPS: A production system meets objects. In *AAAI 83: Proceedings of the National Conference on Artificial Intelligence*, pages 5–7, Washington D.C., USA, August 1983.

[AM87] Andrew W. Appel and David B. MacQueen. A standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987. Lecture Notes in Computer Science 274.

[Ame89] Pierre America. Issues in the Design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, October 1989.

[AMT89] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for standard ML. Part of the New Jersey Standard ML distribution, December 1989.

[And90] P. B. Andersen. *A Theory of Computer Semiotics: Semiotic Approaches to Construction and Assessment of Computer Systems*. Cambridge University Press, 1990.

[ANS89] American National Standard for Information Systems — programming language — C: ANSI X3.159–1989. Published by the American National Standards Institute, 1430 Broadway, New York, New York 10018, December 1989. (Also ISO/IEC 9899:1990).

[AR88] Norman Adams and Jonathan Rees. Object-oriented programming in Scheme. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 277–288, Snowbird, Utah, USA, July 1988. Association for Computing Machinery.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Jullie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1985.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[AW76] E. A. Ashcroft and W. W. Wadge. Lucid — a formal system for writing and procing programs. *SIAM Journal of Computing*, 5(3):336–354, September 1976.

[aYCK93] Gao Yaoqing andd Yuen Chung Kwong. A survey of implementations of concurrent, parallel, and distributed smalltalk. *ACM SIGPLAN Notices*, 28(9):29–35, September 1993.

[Bac78a] John Backus. Can programming be liberated form the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bac78b] John Backus. The history of FORTRAN I, II and III. *ACM SIGPLAN Notices*, 13(8):165–180, August 1978.

[Bai85] D. Bailey. The University of Salford Lisp/Prolog system. *Software: Practice & Experience*, 15(6):595–609, June 1985.

[Ban86] Robert G. Bandes. Constraining-unification and the programming language unicorn. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 397–410. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[BBLM84] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the integration of logic programming and functional programming. In *1984 International Symposium on Logic Programming*, pages 160–166, Atlantic City, New Jersey, USA, February 1984. The Computer Society of the IEEE, IEEE Computer Society Press.

[BBLM86] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. Leaf: A language which integrates logic, equations and functions. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 201–238. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[BBP+81] D. L. Bowen, L. Byrd, L. M. Pereira, F. C. N. Pereira, and D. H. D. Warren. *PROLOG on the DECSystem-10 User's Manual*. Department of Artificial Intelligence, University of Edinburgh, Edinburgh, United Kingdom, 1981. Technical Report.

[BC78] L. Bostock and S. Chandler. *Pure Mathematics*. Stanley Thorner (Publishers) Ltd., 1978.

[BCH+90] Len Bass, Brian Clapper, Erik Hardy, Rick Kazman, and Robert Seacord. Serpent: A user interface environment. In *Proceedings of the Winter 1990 USENIX Conference*, pages 245–257, Washington D.C., USA, January 1990. Usenix Association.

[BDG+88] Daniel G. Bobrow, L. G. DeMichiel, R. P. Gebriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp object system specification, X3J13 document 88–002R. *ACM SIGPLAN Notices*, 23, September 1988. Special Issue.

[BDL82] Marco Bellia, Pierpaolo Degano, and Giorgio Levi. The call by name semantics of a clause language with functions. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 281–295. Academic Press, 1982.

[Bea92] Brian W. Beach. Connecting software components with declarative glue. In *14th International Conference on Software Engineering*, pages 120–136, Melbourne, Australia, May 1992. ACM Press.

[Ben88] Jon Louis Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, 1988.

[Ber88] F. Berthier. Using chip to support decision making. Technical Report LP TR-LP-30, ECRC, Arabellastr. 17, 8000 Munich 81, West Germany, February 1988.

[Bey86] Meurig Beynon. Paradigms for programming. Technical report, Department of Computer Science, University of Warwick, Coventry, UK, 1986.

[BK90] Joanne L. Boyd and Gerald M. Karam. Prolog in 'C'. *ACM SIGPLAN Notices*, 25(7):63–71, July 1990.

[BKK+86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. *ACM SIGPLAN Notices*, 21(11):17–29, November 1986. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '86 Conference Proceedings, September 29 – October 2, Portland, Oregon, USA.

[BL86] Marco Bellia and Giorgio Levi. The relation between logic and functional languages: A survey. *Journal of Logic Programming*, 3(3):217–236, October 1986.

[BM72] R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 7*, chapter 6, pages 101–116. Edinburgh University Press, 1972.

[BMPT90] A. Borig, P. Mancarella, D. Pedreshi, and F. Turini. Logic programming within a functional framework. In *Programming Language Implementation and Logic Programming International Workshop PLILP 90 Proceedings*, pages 372–386, Linkoping, Sweden, August 1990.

[Bob84] Daniel G. Bobrow. If Prolog is the answer, what is the question. In *Fifth Generation Computer Systems 1984*, pages 138–145, Tokyo, Japan, November 1984. Institute for New Generation Computer Technology (ICOT), North-Holland.

[Boc86] Jorge Bocca. EDUCE a marriage of convenience: Prolog and a relational DBMS. In *1986 Symposium on Logic Programming*, pages 36–45, Salt Lake City, Utah, USA, September 1986. The Computer Society of the IEEE, IEEE Computer Society Press.

[Bol86] Herold Boley. RELFUN — a relational / functional integration with valued clauses. *ACM SIGPLAN Notices*, 21(12):87–98, December 1986.

[Bon87] Pierre E. Bonzon. An environment model for the integration of logic and functional programming. In John McDermott, editor, *IJCAI 87: Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 18–23, Milan, Italy, August 1987.

[Bon91] Pierre Bonzon. Processing functional definitions as declarative knowledge: A reduced bytecode implementation of a functional logic machine. In H. Boley and M. M. Richter, editors, *Processing Declarative Knowledge: International Workshop PDK '91 Proceedings*, pages 271–278, Kaiserslautern, Germany, July 1991. Springer-Verlag. Lecture Notes in Computer Science 567.

[Bou79] S. R. Bourne. An introduction to the UNIX shell. In Unix Programmer's Manual [Uni79].

[BP93] Doug Bell and Mike Parr. Spreadsheets: a research agenda. *ACM SIGPLAN Notices*, 28(9):26–28, September 1993.

[Bru82] Maurice Bruynooghe. The memory management of Prolog implementations. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 83–98. Academic Press, 1982.

[BSD86a] *UNIX Programer's Reference Manual*. Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.

[BSD86b] *UNIX User Reference Manual*. Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Bud91] Timothy A. Budd. Blending imperative and relational programming. *IEEE Software*, 8(1):58–65, January 1991.

[Byr80] L. Byrd. Understanding the control flow of Prolog programs. In *Logic Programming Workshop*, Debrecen, 1980.

[CAC93] Communications of the ACM. special issue: Concurrent object-oriented programming. 36(9), September 1993. Optional.

[Car84] Mats Carlsson. On implementing Prolog in functional programming. In *1984 International Symposium on Logic Programming*, pages 154–159, Atlantic City, New Jersey, USA, February 1984. The Computer Society of the IEEE, IEEE Computer Society Press.

[Cas92] Rommert J. Casimir. Real programmers don't use spreadsheets. *ACM SIGPLAN Notices*, 27(6):10–16, June 1992.

[CDG+92] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[Che76a] T. E. Cheatham Jr. Programming language design issues. In John H. Williams and David A. Fisher, editors, *Design and Implementation of Programming Languages, Proceedings of a DoD Sponsored Wokshop*, pages 397–435, Ithaca, USA, October 1976. Springer-Verlag. Lecture Notes in Computer Science 54.

[Che76b] Peter Pin-Shan Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Che80] Daniel Chester. HCPRVR: An interpreter for logic programs. In *Proceedings of the First Annual National Conference on Artificial Intelligence*, pages 93–95. American Association for Artificial Intelligence, August 1980.

[CHP88] James R. Cordy, Charles D. Halpern, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings IEEE 1988 International Conference on Computer Languages*, Miami, USA, October 1988. IEEE Computing Society.

[CL93] Paolo Ciancarini and Giorgio Levi. What is logic programming good for in software engineering. Technical Report UBLCS-93-9, Laboratory of Computer Science, University of Bologna, Piazza di Porta S. Donata 5, 40127 Bologna, Italy, April 1993.

[Cla88] Keith L. Clark. Parlog and its applications. *IEEE Transactions on Software Engineering*, December 1988.

[CM84a] Keith L. Clark and Frank G. McCabe. *micro-PROLOG: Programming in Logic*. Prentice Hall, 1984.

[CM84b] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer Verlag, second edition, 1984.

[CMS82] Keith L. Clark, William M. McKeeman, and Sharon Sickel. Logic program specification of numerical integration. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 123–139. Academic Press, 1982.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Coh85] Jacques Cohen. Describing Prolog by its interpretation and compilation. *Communications of the ACM*, 28(12):1311–1324, December 1985.

[Coh86] Shimon Cohen. The APPLOG language. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 239–276. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[Coh90] Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Col90] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[Coo92] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. *ACM SIGPLAN Notices*, 27(10):1–15, October 1992. Seventh Annual Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '92 Conference Proceedings, October 18–22, Vancouver, British Columbia, Canada.

[Cox86] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[CP90] James R. Cordy and Eric Promislow. Specification and automatic prototype implementation of polymorphic objects in Turing using the TXL

dialect processor. In *Proceedings IEEE 1990 International Conference on Computer Languages*, Miami, USA, March 1990. IEEE Computing Society.

[CST87] Roberto Chislanzoni, Luca Spampinato, and Giorgio Tornielli. Reflection as a tool for integrations: An exercise in procedural introspection. In John McDermott, editor, *IJCAI 87: Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 44–47, Milan, Italy, August 1987.

[DAT91] Claudio Delrieux, Pablo Azero, and Fernando Tohmé. Toward integrating imperative and logic programming paradigms: A WYSIWIG approach to Prolog programming. *ACM SIGPLAN Notices*, 26(3):35–44, March 1991.

[DE91] Mireille Ducassé and Anna-Maria Emde. Opium: A debugging environment for Prolog development and debugging research. *ACM Software Engineering Notes (SIGSOFT)*, 16(1):54–59, January 1991. Demonstration presented at the Fourth Symposium on Software Development Environments.

[Deb88] Saumya K. Debray. *The SB-Prolog System, Version 3.0: A User Manual*. University of Arizona, Department of Computer Science, Tucson, AZ 85721, USA, September 1988.

[Dew79] Robert B. K. Dewar. The SETL programming language, 1979.

[DFP86] J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 37–70. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[DGP91a] John Darlington, Yi-ke Guo, and Helen Pull. The design of constraint functional logic programming. Technical report, Department of Computing, Imperial College, London, UK, February 1991.

[DGP91b] John Darlington, Yi-ke Guo, and Helen Pull. Introducing constraint functional logic programming. Technical report, Department of Computing, Imperial College, London, UK, February 1991.

[DGP92] John Darlington, Yike Guo, and Helen Pull. A new perspective on integrating functional and logic languages. In *The 4th International Conference on the Fifth Generation Computer Systems*, pages 682–693, Tokyo, Japan, June 1992. ICOT.

[DL86] Doug DeGroot and Gary Lindstrom, editors. *Logic Programming, Functions, Relations and Equations*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1986.

[DP90] H. Dobler and K. Pirklbauer. Coco-2: A new compiler compiler. *ACM SIGPLAN Notices*, 25(5):82–90, May 1990.

[DR67] Philip J. Davis and Philip Rabinowitz. *Numerical Integration*. Blaisdell Publishing Company, 1967.

[DR90] M. C. Dewar and M. G. Richardson. Reconciling symbolic and numeric computation in a practical setting. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems: International Symposium DISCO'90 Proceedings*, pages 195–204, Capri, Italy, April 1990. Springer-Verlag. Lecture Notes in Computer Science 429.

[Dra87] Włodzimierz Drabent. Do logic program resemble programs in conventional languages. In *Proceedings 1987 Symposium on Logic Programming*, pages 389–396, San Francisco, CA, USA, August–September 1987. Computer Society Press of the IEEE.

[DS88] Charles Donnelly and Richard Stallman. BISON: The YACC-compatible parser generator. Distributed by the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, October 1988.

[Duc92] Mireille Ducassé. *Un analyseur de trace extensible pour l' automatisation du débogage*. PhD thesis, l' Université de Rennes I, Rennes, France, June 1992. In English.

[DW89] Saumya K. Debray and David S. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, July 1989.

[EGM89] U. Engelmann, Th. Gerneth, and H. P. Meinzer. Implementation of predicate logic in APL2. *APL QuoteQuad*, 19(4):124–128, August 1989. Conference Proceedings APL89.

[EH80] Douglas Lee Eckberg and Lester Hill, Jr. The paradigm concept and sociology: A critical review. In Gary Gutting, editor, *Paradigms and Revolutions*, pages 117–136. University of Notre Dame Press, Notre Dame, London, 1980.

[Ein84] Bo Einarsson. Mixed language programming. *Software: Practice & Experience*, 14(4):383–395, April 1984.

[ES89] Susan Eisenbach and Chris Sadler. *Program design with Modula-2*. International computer science series. Addison-Wesley, 1989.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[FBB92] Bjorn N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In O. Lehrmann Madsen, editor, *ECCOP '92 Europen Conference on Object-Oriented Programming*, pages 268–286, Utrecht, The Netherlands, June/July 1992. Springer-Verlag. Lecture Notes in Computer Science 615.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. In N. Jones, editor, *3rd European Symposium on Programming*, pages 134–151, Copenhagen, Denmark, May 1990. Springer-Verlag. Lecture Notes in Computer Science 432.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[FiH86] Koichi Fukunaga and Shin ichi Hirose. An experience with a Prolog-based object-oriented language. *ACM SIGPLAN Notices*, 21(11):224–231, November 1986. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '86 Conference Proceedings, September 29 – October 2, Portland, Oregon, USA.

[FKG90] Anthony Finkelstein, Jeff Kramer, and Michael Goedicke. Viewpoint oriented software development. In *Third International Workshop on Software Engineering and its Applications*, pages 357–351, Toulouse, France, December 1990.

[FL86] Antony A. Faustini and Edgar B. Lewis. Towards a real-time dataflow language. *IEEE Software*, 3(1):29–35, January 1986.

[Fle90] A. C. Fleck. A case study comparison of four declarative programming languages. *Software: Practice & Experience*, 20(1):49–65, January 1990.

[Fri91] Linda Wieser Friedman. *Comparative Programming Languages: Generalizing the Programming Function*. Prentice Hall, 1991.

[Fro87] Bertram Fronhöfer. PLANLOG: A language framework for the integration of procedural and logical programming. In John McDermott, editor, *IJCAI 87: Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 15–17, Milan, Italy, August 1987.

[FSO91] Richard Furuta, P. David Stotts, and Jefferson Ogata. Ytracc: a parse browser for Yacc grammars. *Software: Practice & Experience*, 21(2):119–132, February 1991.

[FT90] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.

[Gab90] D. M. Gabbay. Labeled Deductive Systems. Technical Report 90 – 22, Zentrum für Informations und Sprachverarbeitung, Universität München, Leopoldstr. 139, 8000 München 40, Germany, December 1990. Draft Version 5, To be published by Oxford University Press.

[Gal86] Hervé Gallaire. Merging objects and logic programming: Relational semantics. In *AAAI 86: Fifth National Conference on Artificial Intelligence*, pages 754–758, Philadelphia, PA, USA, August 1986.

[GE91] J. Grosch and H. Emmelmann. A tool box for compiler construction. In D. Hammer, editor, *Compiler Compilers : Third International Workshop, CC '90*, pages 106–116, Schwerin, Germany, October 1991. Springer-Verlag. Lecture Notes in Computer Science 477.

[Gem87] Peter A. Gemtos. *Methodology of Social Sciences*. Papazisis, Athens, Greece, third edition, 1987. In Greek.

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall Software Series. Prentice-Hall Inc., Englewood Cliffs, NJ, USA, 1983.

[GHL+92] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[GJ82] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language and Evolution*. Jon Wiley and Sons, 1982.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 28–38, Cambridge, Massachusetts, USA, August 1986. Association for Computing Machinery.

[GLDD90] Yi-ke Guo, H. Lock, John Darlington, and R. Dietrich. A classification for the integration of functional and logic languages. Technical report, Department of Computing, Imperial College and GMD Forschungstelle and der Universität Karlsruhe, March 1990. Deliverable for the ESPRIT Basic Research Action No.3147.

[Gli91] Bob Glickstein. YYHIDE: A post-processor for Yacc and Lex files. Part of the Andrew System, 1991.

[GM79] Hamish I. E. Gunn and Ronald Morrison. On the implementation of constants. *Information Processing Letters*, 9(1):1–4, July 1979.

[GM86a] Joseph A. Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 295–363. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[GM86b] Joseph A. Goguen and José Meseguer. Extension and foundations of object-oriented programming. *ACM SIGPLAN Notices*, 21(10):153–162, October 1986. Proceedings of the Object-Oriented Programming Workshop held at Yortown Heights, June 9–13, 1986.

[GM87] Joseph A. Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[Gog90] Joseph A. Goguen. Higher-order functions considered unnecessary for higher-order programming. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 12, pages 309–351. Addison-Wesley, 1990. Written in 1987.

[Gol80] Adele Goldberg. *Smalltalk 80: The Language and its Implementation*. Addison Wesley, 1980.

[GPP71] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The Snobol 4 Programming Language*. Prentice Hall, 2nd edition edition, 1971.

[Gra88] Robert W. Gray. γ-GLA. In *Proceedings of the Summer 1988 USENIX Conference*, pages 147–160, San Francisco, CA, USA, June 1988. Usenix Association.

[Gre80] John C. Greene. The Kuhnian paradigm and the Darwinian revolution in natural history. In Gary Gutting, editor, *Paradigms and Revolutions*, pages 297–320. University of Notre Dame Press, Notre Dame, London, 1980.

[Gri84] Ralph E. Griswold. Expression evaluation in the Icon programming language. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 177–183, Austin, Texas, USA, August 1984. ACM, SIGPLAN, SIGACT, SIGART.

[gro77] MACSYMA group. MACSYMA reference manual. Technical report, MIT, Massachusets, USA, 1977.

[Gro86] Computer Systems Research Group. *UNIX Programmer's Reference Manual*. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Distribution.

[Gro89] J. Grosch. Efficient generation of lexical analysers. *Software: Practice & Experience*, 19(11):1089–1103, November 1989.

[GTC+90] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz, and Xavier Pintado. Class management for software communities. *Communications of the ACM*, 33(9):90–103, September 1990.

[Hai86a] Brent Hailpern. Multiparadigm languages. *IEEE Software*, 3(1):6–9, January 1986.

[Hai86b] Brent Hailpern. Multiparadigm research: A survey of nine projects. *IEEE Software*, 3(1):70–77, January 1986.

[Ham76] Richard G. Hamlet. High-level binding with low-level linkers. *Communications of the ACM*, 19(11):642–644, November 1976.

[Han90a] M. Hanus. Compiling logic programs with equality. In *Programming Language Implementation and Logic Programming. International Workshop PLILP 90 Proceedings*, pages 387–401, Linkoping, Sweden, August 1990. Springer-Verlag. Lecture Notes in Computer Science 456.

[Han90b] Michael Hanus. A functional and logic language with polymorphic types. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems: International Symposium DISCO'90 Proceedings*, pages 215–224, Capri, Italy, April 1990. Springer-Verlag. Lecture Notes in Computer Science 429.

[Han91] Michael Hanus. The ALF system: An efficient implementation of a functional logic language. In H. Boley and M. M. Richter, editors, *Processing Declarative Knowledge: International Workshop PDK '91 Proceedings*, pages 414–416, Kaiserslautern, Germany, July 1991. Springer-Verlag. Lecture Notes in Computer Science 567.

[Har86] Robert Harper. Introduction to standard ML. LFCS Report Series ECS–LFCS–86–14, University of Edinburgh, Department of Computer Science, Edinburgh EH9 3JZ, UK, November 1986.

[Hea87] Anthony C. Hearn. *Reduce User's Manual*. The RAND Corporation, Santa Monica, CA, USA, version 3.3 edition, July 1987.

[Heu86] V. P. Heuring. The automatic generation of fast lexical analysers. *Software: Practice & Experience*, 16(9):801–808, September 1986.

[HHT82] Å. Hansson, S. Haridi, and S.-Å. Tärnlund. Properties of a logic programming language. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 266–280. Academic Press, 1982.

[HJJ83] P. Henderson, G. A. Jones, and S. B. Jones. The LispKit manual. Technical monograph PRG-32(1), Oxford University Computer Laboratory, Programming Research Group, 1983.

[HKW85] F. Hattori, K. Kushima, and T. Wasano. A comparison of Lisp, Prolog and Ada programming productivity in AI area. In *Proceedings of the 8th International Conference on Software Engineering*, pages 285–291, London, United Kingdom, September 1985. IEEE Computer Society Press.

[HL87] R. Nigel Horspool and Michael R. Levy. Mkscan — an interactive scanner generator. *Software: Practice & Experience*, 17(6):369–378, June 1987.

[HMS88] Roger Hayes, Steve W. Manweiler, and Richard D. Schlichting. A simple system for constructing distributed, mixed-language programs. *Software: Practice & Experience*, 18(7):641–660, July 1988.

[Hoa73] C. A. R. Hoare. Recursive data structures. Technical Report AIM-223 STAN-CS-73-400, Stanford University, Computer Science Department, October 1973.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa83] C. A. R. Hoare. Hints on programming language design. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*, pages 31–40. Computer Science Press, 1983. Reprinted from Sigact/Sigplan Symposium on Principles of Programming Languages, October 1973.

[Hog84] Christopher John Hogger. *Introduction to Logic Programming*. Academic Press, 1984.

[HS88] Matthew Halfant and Gerald Jay Sussman. Abstraction in numerical methods. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 1–7, Snowbird, Utah, USA, July 1988. Association for Computing Machinery.

[HSE90] Brian Henderson-Sellers and Julian M. Edwards. The object-oriented systems life cycle. *Communications of the ACM*, 33(9):142–159, September 1990.

[HSS+92] William Havens, Susan Sidebottom, Greg Sidebottom, John Jones, and Russel Ovans. Echidna: A constraint logic programming shell. In *Proceedings of the 1992 Rim International Conference on Artificial Intelligence*, pages 165–171, Seoul, Korea, September 1992.

[Hug90] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 2, pages 17–42. Addison-Wesley, 1990. Also appeared in the April 1989 issue of The Computer Journal.

[Hug91] John G. Hughes. *Object-Oriented Databases*. Prentice Hall, 1991.

[IK87] Herman Iline and Henry Kanoui. Extending logic programming to object programming: The system LAP. In John McDermott, editor, *IJCAI 87: Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 34–39, Milan, Italy, August 1987.

[Inc85] Adobe Systems Incorporated. *Postscript Language Reference Manual*. Addison-Wesley, 1985.

[IT86] Yutaka Ishikawa and Mario Tokoro. A concurrent object-oriented knowledge representation language orient84/k: Its features and implementation. *ACM SIGPLAN Notices*, 21(11):232–241, November 1986. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '86 Conference Proceedings, September 29 – October 2, Portland, Oregon, USA.

[JGM86] Michael A. Jenkins, Janice I. Glasgow, and Carl D. McCrosky. Programming styles in Nial. *IEEE Software*, 3(1):46–55, January 1986.

[JLM86] Joxan Jaffar, Jean-Louis Lassez, and Michael J. Maher. Logic programming language scheme. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 441–467. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[Joh75] Stephen C. Johnson. Yacc — yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA, July 1975.

[Joh88] Stephen C. Johnson. Yacc meets C++. *Computing Systems*, 1(2):159–167, Spring 1988.

[Jon85] Simon L. Peyton Jones. Yacc in Sasl — an exercise in functional programming. *Software: Practice & Experience*, 15(8):807–820, August 1985.

[JP91] Radha Jagadeesan and Keshav Pingalis. A fully abstract semantics for a first-order functional language with logic variables. *ACM Transactions on Programming Languages and Systems*, 13(4):577–624, October 1991.

[Jr.91] George Charles Nelan Jr. *Firstification*. PhD thesis, Arizona State University, Arizona, USA, December 1991.

[JS86] Bharat Jayaraman and Frank S. K. Silberman. Equations, sets, and reduction semantics for functional and logic programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 320–331, Cambridge, Massachusetts, USA, August 1986. Association for Computing Machinery.

[JSG86] Bharat Jayaraman, Frank S. K. Silberman, and Gopal Gupta. Equational programming: A unifying approach to functional and logic programming. In *IEEE Computer Society 1986 International Conference on Computer Languages*, pages 47–57, Miami, Florida, USA, October 1986. IEEE Computer Society, IEEE Computer Society Press.

[JW75] Kathleen Jensen and Niklaus Wirth. *PASCAL User Manual and Report*. Springer Verlag, second edition, 1975.

[Kah82] Kenneth M. Kahn. Intermission — Actors in Prolog. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 213–228. Academic Press, 1982.

[Kah86] Kenneth M. Kahn. Uniform: A language based upon unification which unifies (much of) Lisp, Prolog, and Act 1. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 411–438. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[Kat83] Jacob Katzenelson. Introduction to enhanced C (EC). *Software: Practice & Experience*, 13(7):551–576, July 1983.

[KC74] Brian W. Kernighan and L. L. Cherry. A system for typesetting mathematics. Computer Science Technical Report 17, Bell Laboratories, Murray Hill, NJ, USA, May 1974.

[KdRB92] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1992.

[KE88] Timothy Koschmann and Martha Walton Evens. Bridging the gap between object-oriented and logic programming. *IEEE Software*, 5(4):36–42, July 1988.

[Ker75] Brian W. Kernighan. Ratfor — a preprocessor for a rational Fortran. *Software: Practice & Experience*, 5(4):395–406, 1975.

[Ker81] Brian W. Kernighan. Why Pascal is not my favorite programming language. Computer Science Technical Report 100, Bell Laboratories, Murray Hill, NJ, USA, July 1981. Available online at http://cm.bell-labs.com/cm/cs/cstr. (Reprinted in Comparing and Assessing Programming Languages Ed. A. Feuer N. Gehani Prentice-Hall 1984).

[Ker82] Brian W. Kernighan. PIC — a language for typesetting graphics. *Software: Practice & Experience*, 12:1–21, 1982.

[Ker89] Brian W. Kernighan. The UNIX system document preparation tools: A retrospective. *AT&T Technical Journal*, 68(4):5–20, July/August 1989.

[KHDS87] James Kempf, Warren Harris, Roy D'Souza, and Alan Snyder. Experience with CommonLoops. *ACM SIGPLAN Notices*, 22(12):214–226, December 1987. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '87 Conference Proceedings, October 4–8, Orlando, Florida, USA.

[Kin89] Roger King. My cat is object-oriented. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 2, pages 23–30. Addison-Wesley, 1989. ACM Press Frontier Series.

[KM90] Tim Korson and John D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–60, September 1990.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[Knu89] Donald E. Knuth. *The TeXbook*. Addisson-Wesley, 1989.

[Kor86a] William A. Kornfeld. Equality for Prolog. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 279–293. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[Kor86b] Henry F. Korth. Extending the scope of relational languages. *IEEE Software*, 3(1):19–28, January 1986.

[Kos87] Yoshiyuki Koseki. Amalgamating multiple programming paradigms in Prolog. In John McDermott, editor, *IJCAI 87: Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 76–86, Milan, Italy, August 1987.

[Kow74] Robert Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Information Processing 74, Proceedings of IFIP congress 74*, pages 569–574, Stockholm, Sweden, August 1974. International Federation of Information Processing, North-Holland.

[Kow79] Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, July 1979.

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, first edition, 1978.

[KTMB86] Kenneth Kahn, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. Objects in concurrent logic programming languages. *ACM SIGPLAN Notices*, 21(11):242–257, November 1986. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '86 Conference Proceedings, September 29 – October 2, Portland, Oregon, USA.

[KTMB87] Kenneth Kahn, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. Vulcan: Logical concurrent objects. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 75–112. MIT Press, 1987.

[Kuh70] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago and London, 2nd, enlarged edition, 1970. International Encyclopedia of Unified Science. 2:(2).

[Lam85] Leslie Lamport. *LATEX: A Document Preparation System*. Addisson-Wesley, 1985.

[Lan63] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1963.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, May 1966.

[LC86] Guy Lapalme and Suzanne Chapleau. Logicon: an integration of Prolog into Icon. *Software: Practice & Experience*, 16(10):925–944, October 1986.

[Les75] Michael E. Lesk. Lex — a lexical analyzer generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, USA, October 1975.

[Les79] Michael E. Lesk. TBL — a program to format tables. In Unix Programmer's Manual [Uni79].

[Lev89] Gary Marc Levin. An introduction to ISETL. Technical report, Clarkson University, Dept. of Math and Computer Science, Potsdam NY, USA, 1989. Available via ftp from grape.ecs.clarkson.edu.

[Lic86] Zavdi L. Lichtman. The function of T and NIL in LISP. *Software: Practice & Experience*, 16(1):1–3, January 1986.

[Lin85] Gary Lindstrom. Functional programming and the logical variable. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 266–280, Januray 1985.

[Liu86] Ken-Chih Liu. A string pattern matching extension to Pascal and some comparisons with SNOBOL4. *Software: Practice & Experience*, 16(6):541–548, June 1986.

[LM81] Henry Ledgard and Michael Marcotty. *The Programming Language Landscape*. Science Research Associates, Chicago, USA, 1981.

[LMT89] Wilf R. LaLonde, Jim McGugan, and Dave Thomas. The real advantages of pure object-oriented systems, or why object-oriented extensions to C are doomed to fail. In George J. Knafl, editor, *Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, COMPSAC '89*, pages 344–350, Orlando, Florida, USA, September 1989. IEEE Computer Society, IEEE Computer Society Press.

[LST82] Meir M. Lehman, Vic Stenning, and Władysław M. Turski. Another look at software design methodology. *ACM SIGSOFT Software Engineering Notes*, 9(2):38–53, April 1982.

[LV73] D. Lurié and C. Vandoni. Statistics for FORTRAN identifiers and scatter storage techniques. *Software: Practice & Experience*, 3:171–177, 1973.

[Man91] Andrei V. Mantsivoda. Flang: a functional logic language. In H. Boley and M. M. Richter, editors, *Processing Declarative Knowledge: International Workshop PDK '91 Proceedings*, pages 257–270, Kaiserslautern, Germany, July 1991. Springer-Verlag. Lecture Notes in Computer Science 567.

[MC79] R. H. Morris and L. L. Cherry. DC — an interactive desk calculator. In Unix Programmer's Manual [Uni79].

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(12):184–195, December 1960. Part I.

[McC92] Francis G. McCabe. *Logic and Objects*. Prentice Hall, 1992.

[McM79] L. E. McMahon. SED — a non-interactive text editor. In Unix Programmer's Manual [Uni79].

[MD88] P. Van Hentenryck M. Dincbas, H. Simonis. Solving the car-sequencing problem in chip. Technical Report LP TR-LP-32, ECRC, Arabellastr. 17, 8000 Munich 81, West Germany, February 1988.

[Mel75] Lucio F. Melli. the 2.PAK language: Goals and descriptions. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, pages 549–555, Tbilisi, Georgia, USSR, Sepetember 1975.

[Mel85] C. S. Mellish. Some global optimizations for a Prolog compiler. *The Journal of Logic Programming*, 2(1):43–66, April 1985.

[Met87] Rand Methfessel. Implementing an access and object oriented paradigm in a language that supports neither. *ACM SIGPLAN Notices*, 22(4):83–93, April 1987.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Mey90] Bertrand Meyer. Lessons from the design of the Eiffel libraries. *Communications of the ACM*, 33(9):68–88, September 1990.

[Mey92] Betrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[MH88] John C. Mitchel and Robert Harper. The essence of ML. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 28–46, Januray 1988.

[MHMA89] Robin Milner, Robert W. Harper, David B. MacQueen, and Andrew W. Appel. *Standard ML Reference Manual*. Bell Laboratories, Murray Hill, New Jersey 07974, 1989. Part of the mid-1989 distribution of the Standard ML of New Jersey compiler.

[Mil85] Robin Milner. The standard ML core language (revised). Technical report, University of Edinburgh, 1985.

[MK89] Sharon L. Murrel and Thaddeus J. Kowlaski. Monk: A high-level text compiler. *AT&T Technical Journal*, 68(4):45–60, July/August 1989.

[MM884] *Unix System V Documenters Workbench*. Indianapolis, Indiana, software release 1.0 edition, 1984. CIC No. 307-152.

[MMW84] Yonathan Malachi, Zohar Manna, and Richard Waldinger. TABLOG: The deductive-tableu programming language. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 323–330, Austin, Texas, USA, August 1984. ACM, SIGPLAN, SIGACT, SIGART.

[MMW86] Yonathan Malachi, Zhar Manna, and Richard Waldinger. TABLOG: A new approach to logic programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 365–394. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[Moo86] David A. Moon. Object-oriented programming with flavors. *ACM SIGPLAN Notices*, 21(11):1–8, November 1986. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '86 Conference Proceedings, September 29 – October 2, Portland, Oregon, USA.

[Mos71a] Joel Moses. Algebraic simplification: A guide for the perplexed. *Communications of the ACM*, 14(8):527–537, August 1971.

[Mos71b] Joel Moses. Symbolic integration: The stormy decade. *Communications of the ACM*, 14(8):548–560, August 1971.

[MR89] Scott Meyers and Steven P. Reiss. Representing programs in multiparadigm software development environments. In George J. Knafl, editor, *Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, COMPSAC '89*, pages 420–427, Orlando, Florida, USA, September 1989. IEEE Computer Society, IEEE Computer Society Press.

[MS90] Gerhard Mehldau and Robert A. Schowengerdt. A C-extension for rule-based image classification systems. *Photogrammetric Engineering and Remote Sensing*, LVI(6):887–892, June 1990.

[Mul86] Carlo Muller. Modula — Prolog: A software development tool. *IEEE Software*, 3(6):39–45, November 1986.

[MW91] Vincent D. Moynihan and Peter J. L. Wallis. The design and implementation of a high-level language converter. *Software: Practice & Experience*, 21(4):391–400, April 1991.

[Nak84] Hideyuki Nakashima. Knowledge representation in Prolog/KR. In *1984 International Symposium on Logic Programming*, pages 126–130, Atlantic City, New Jersey, USA, February 1984. The Computer Society of the IEEE, IEEE Computer Society Press.

[Nak85] Hideyuki Nakashima. Term description: A simple powerful extension to Prolog data structures. In Aravind Joshi, editor, *IJCAI 85: Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 708–710, Los Angeles, CA, USA, August 1985.

[Nar85] Sanjai Narain. A technique for doing lazy evaluation in logic. In *1985 Symposium on Logic Programming*, pages 261–269, Boston, Massachusetts, USA, July 1985. IEEE Computer Society, IEEE Computer Society Press.

[Nel91] Michael L. Nelson. An object-oriented tower of Babel. *OOPS Messenger*, 2(3):3–11, July 1991.

[NF92] Bashar Nuseibeh and Anthony Finkelstein. Viewpoints: A vehicle for method and tool integration. In *International Workshop on CASE (CASE '92)*, Montreal, Canada, July 1992.

[Nie89] Oscar Nierstrasz. A survey of object-oriented concepts. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 1, pages 3–22. Addison-Wesley, 1989. ACM Press Frontier Series.

[OG87] J. O'Bagy and R. Grisworld. A recursive intepreter for the Icon programming language. In R. Wexelblat, editor, *Intepreter and Intepretive Techniques. SIGPLAN '87 Symposium*, pages 138–149, St. Paul, MI, USA, June 1987.

[Old92] Ernst-Rüdiger Olderog. Interfaces between languages for communicating systems. In W. Kuich, editor, *Automata, Languages and Programming: 19th International Colloquium*, pages 641–655, Wien, Austria, July 1992. Springer Verlag. Lecture Notes in Computer Science 623.

[Omo91] Stephen M. Omohundro. The Sather language. Available by anonymous ftp from icsic.berkeley.edu:sather, June 1991.

[Oss79] J. F. Ossanna. NROFF/TROFF user's manual. In Unix Programmer's Manual [Uni79].

[Pax89] Vern Paxson. *Flex: Fast Lexical Analyzer Generator*. Real Time Systems, Bldg, 46A, Lawrence Berkeley Laboratory, Berkeley CA, USA, 1989.

[PDC92] T. J. Parr, H. G. Dietz, and W. E. Cohen. PCCTS reference manual. *ACM SIGPLAN Notices*, 27(2):88–165, February 1992.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.

[PHS+88] Andrew J. Palay, Wilfred J. Hansen, Mark Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader, and Thom Peters. The Andrew toolkit — an overview. In *Proceedings of the Winter 1988 USENIX Conference*, pages 9–22, Dallas, TX, USA, January 1988. Usenix Association.

[PK88] Dewayne E. Perry and Gail E. Kaiser. Models of software development environments. In *Proceedings of the 10th International Conference on Software Engineering*, pages 60–68, Singapore, April 1988. IEEE Computer Society Press.

[Pla89] John R. Placer. *G: A Language Based on Demand-Driven Stream Evaluations*. PhD thesis, Oregon State University, Oregon, USA, June 1989.

[Pla91a] John Placer. The multiparadigm language G. *Computer Languages*, 16(3/4):235–258, 1991.

[Pla91b] John Placer. Multiparadigm research: A new direction in language design. *ACM SIGPLAN Notices*, 26(3):9–17, March 1991.

[Pla92] John Placer. Integrating destructive assignment and lazy evaluation in the multiparadigm language G-2. *ACM SIGPLAN Notices*, 27(2):65–74, February 1992.

[Pra83] Vaughan Pratt. Five paradigm shifts in programming language design and their realization in Viron, a dataflow programming environment. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–9, Austin, TX, USA, Januray 1983.

[PW72] Leon Presser and John R. White. Linkers and loaders. *ACM Computing Surveys*, 4(3):149–167, September 1972.

[Rad90] Atanas Radensky. Toward integration of the imperative and logic programming paradigms: Horn-clause programming in the Pascal environment. *ACM SIGPLAN Notices*, 25(2):25–34, February 1990.

[REA+86] Jonathan Ress, William Clinger (Editors), H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, R. K. Dybcig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, and M. Wand. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986. Dedicated to the Memory of ALGOL 60.

[Red85] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *1985 Symposium on Logic Programming*, pages 138–151, Boston, Massachusetts, USA, July 1985. IEEE Computer Society, IEEE Computer Society Press.

[Red86] Uday S. Reddy. On the relationship between logic and functional languages. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 3–36. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[Ren82] Tim Rentsch. Object oriented programming. *ACM SIGPLAN Notices*, 17(9):51–57, September 1982.

[Rey70] John C. Reynolds. GEDANKEN — a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, May 1970.

[Ric83] Elaine Rich. *Artificial Intelligence*. McGraw-Hill, 1983.

[Rin89] Jussi Rintanen. Preliminary version of ML-twig user manual. Part of the New Jersey Standard ML distribution, June 1989.

[Ris67] R. H. Risch. The problem of integration in finite terms. *Transactions of the American Mathematical Society*, 139, 1967.

[Rit84] Dennis M. Ritchie. Reflections on software research. *Communications of the ACM*, 27(8):758–760, 1984.

[RL77] Frederic Richard and Henry F. Ledgard. A reminder for language designers. *ACM SIGPLAN Notices*, 12(12):73–82, December 1977.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rol87] David W. Rolston. A multiparadigm knowledge-based system for diagnosis of large maniframe peripherals. In *Proceedings: The Third Conference on Artificial Intelligence Applications*, pages 150–155, Orlando, Florida, USA, February 1987. IEEE Computer Society, IEEE Computer Society Press.

[Roy87] Winston W. Royce. Managing the development of large software systems. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–338, Monterey, California, USA, March 1987. IEEE Computer Society Press. Reprinted from the Proceedings, IEEE WESCON, August 1970, pages 1–9, published by TRW.

[RS82] J. A. Robinson and E. E. Sibert. Loglisp: Motivation design and implementation. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 299–313. Academic Press, 1982.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[Rum87] James Rumbaugh. Relations as semantic constructs in an object-oriented language. *ACM SIGPLAN Notices*, 22(12):466–481, December 1987. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '87 Conference Proceedings, October 4–8 Orlando, Florida, USA.

[Sam71] J.E. Sammet. Software for nonnumerical mathematics. In John R. Rice, editor, *Mathematical Software*, chapter 6, pages 295–330. Academic Press, 1971. ACM Monograph Series.

[SB86] Leon Sterling and Randall D. Beer. Incremental flavor-mixing of meta-interpreters for expert system construction. In *1986 Symposium on Logic Programming*, pages 20–27, Salt Lake City, Utah, USA, September 1986. The Computer Society of the IEEE, IEEE Computer Society Press.

[SBK86] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1):10–18, January 1986.

[Sch90] Douglas C. Schmidt. Gperf: A perfect hash function generator. In *USENIX C++ Conference*, pages 87–100, San Francisco, CA, USA, April 1990. Usenix Association.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[Shr86] Bruce D. Shriver. Software paradigms. *IEEE Software*, 3(1):2, January 1986.

[Smo84] Gert Smolka. Making control and data flow in logic programs explicit. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 311–322, Austin, Texas, USA, August 1984. ACM, SIGPLAN, SIGACT, SIGART.

[Smo86] Gert Smolka. Fresh: A higher-order language with unification and multiple results. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 469–524. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[Sny86] Alan Snyder. CcommonObjects: An overview. *ACM SIGPLAN Notices*, 21(10):19–28, October 1986. Proceedings of the Object-Oriented Programming Workshop held at Yortown Heights, June 9–13, 1986.

[Spi90] Diomidis Spinellis. An implementation of the Haskell language. Project report, Imperial College, Department of Computing, London, UK, June 1990.

[Spi91a] Diomidis Spinellis. Application examples for Struct, July 1991.

[Spi91b] Diomidis Spinellis. Type-safe linkage for variables and functions. *ACM SIGPLAN Notices*, 26(8):74–79, August 1991.

[SRA89] *Strand 88 User Manual*, June 1989. Admiralty Release.

[SS86a] Masahiko Sato and Takafumi Sakurai. Qute: A functional language based on unification. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 131–155. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[SS86b] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, USA, 1986.

[ST83] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in concurrent Prolog. *New Generation Computing*, 1:25–48, 1983.

[Sta84] R. M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. In D. R. Barstow, H. E. Shrobe, and E. Sandwell, editors, *Interactive Programming Environments*, pages 300–325. McGraw-Hill, 1984.

[Sta92] Richard M. Stallman. Using and porting GNU CC. Free Software Foundation, 675 Mass Ave, Cambridge, MA, USA, May 1992.

[Ste87] Vic Stenning. On the role of an environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 30–34, Monterey, California, USA, March 1987. IEEE Computer Society Press.

[Sti92] Mike Stimpson. Numerical integration using adaptive quadrature. *The C Users Journal*, pages 31–36–167, May 1992.

[Str83] Bjarne Stroustrup. Adding classes to the C language: An execrise in language evolution. *Software: Practice & Experience*, 13(2):139–161, February 1983.

[Str84] Bjarne Stroustrup. Data abstraction in C. *Bell System Technical Journal*, 63(8):1701–1732, October 1984.

[Str86a] Rob Strom. A comparison of the object-oriented and process paradigms. *ACM SIGPLAN Notices*, 21(10):88–97, October 1986. Proceedings of the Object-Oriented Programming Workshop held at Yortown Heights, June 9–13, 1986.

[Str86b] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10–20, May 1988.

[SUN90] Sun Microsystems Inc., Mountain View, California. *SunOS Reference Manual*, 1990. Release 4.1.

[SY86] P. A. Subrahmanyam and Jia-Huai You. Funlog: A computational model integrating logic programming and functional programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 157–198. Prentice Hall, Englewood Cliffs, NJ, USA, 1986.

[TA90] David R. Tarditi and Andrew W. Appel. ML-Yacc, version 2.0: Documentation for release version. Part of the New Jersey Standard ML distribution, April 1990.

[Tho90] Ken Thompson. A new C compiler. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 41–51, London, UK, July 1990. UKUUG.

[Tic92] Walter F. Tichy. Programming-in-the-large: Past, present and future. In *14th International Conference on Software Engineering*, pages 362–367, Melbourne, Australia, May 1992. ACM Press.

[Tji86] Steven W. K. Tjiang. Twig reference manual. Computer Science Technical Report 120, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, January 1986.

[TN89] D. C. Tsichritzis and O. M. Nierstrasz. Directions in object-oriented research. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 20, pages 523–536. Addison-Wesley, 1989. ACM Press Frontier Series.

[TOO86] Ikuo Takeuchi, Hiroshi Okuno, and Nobuyasu Ohsato. A list processing language TAO with multiple programming paradigms. *New Generation Computing*, 4(4):401–444, 1986.

[TP86] Hai-Chen Tu and Alan J. Perlis. FAC: A functional APL language. *IEEE Software*, 3(1):36–45, January 1986.

[Tre90] P. C. Treleaven, editor. *Parallel Computers: Object-Oriented Functional, Logic*. John Wiley & Sons, 1990.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice & Experience*, 9(1):31–49, January 1979.

[Tur85] David A. Turner. Miranda — a non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, September 1985. Springer-Verlag. Lecture Notes in Computer Science 201.

[Uni79] *UNIX Programmer's Manual. Volume 2 — Supplementary Documents*. Bell Telephone Laboratories, Murray Hill, NJ, USA (also available online http://plan9.bell-labs.com/7thEdMan/), seventh edition, 1979.

[Uus92] Tarmo Uustalu. Combining object-oriented and logic paradigms: A modal logic programming approach. In O. Lehrmann Madsen, editor, *ECCOP '92 Europen Conference on Object-Oriented Programming*, pages 98–113, Utrecht, The Netherlands, June/July 1992. Springer-Verlag. Lecture Notes in Computer Science 615.

[VLM88] Jean Vaucher, Guy Lapalme, and Jacques Malenfant. SCOOP: Structured concurrent object oriented prolog. In S. Gjessing and K. Nygaard, editors, *ECCOP '88 Europen Conference on Object-Oriented Programming*, pages 191–211, Oslo, Norway, August 1988. Springer-Verlag. Lecture Notes in Computer Science 322.

[Wan89] Yair Wand. A proposal for a formal model of objects. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 21, pages 537–539. Addison-Wesley, 1989. ACM Press Frontier Series.

[War80] David H. D. Warren. Logic programming and compiler writing. *Software: Practice & Experience*, 10:97–125, 1980.

[War83] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Computer Science and Technology Division, 333 Ravenswood Ave., Menlo Park, CA, USA, October 1983.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.

[Weg87] Peter Wegner. Dimensions of object-based language design. *ACM SIGPLAN Notices*, 22(12):168–182, December 1987. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '87 Conference Proceedings, October 4–8, Orlando, Florida, USA.

[Weg89] Peter Wegner. Guest editor's introduction to special issue of computing surveys. *ACM Computing Surveys*, 21(3):253–258, September 1989. Special Issue on Programming Language Paradigms.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[Wei87] Nelson Weiderman. Evaluating software development environments. In *Proceedings of the 9th International Conference on Software Engineering*, pages 292–293, Monterey, California, USA, March 1987. IEEE Computer Society Press.

[Wel89] M. Wells. Multiparadigmatic programming in Modcap. *Journal of Object-Oriented Programming*, 1(5):53–60, January/February 1989.

[Wex76] Richard L. Wexelblat. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 331–336, San Fransisco, CA, USA, October 1976. IEEE Computer Society Press.

[Wic84] Brian W. Wichmann. Is Ada too big? a designer answers the critics. *Communications of the ACM*, 27(2):98–103, February 1984.

[Wir74] Niklaus Wirth. On the design of programming languages. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of IFIP Congress 74*, pages 386–393, Stockholm, Sweden, August 1974. International Federation for Information Processing, North-Holland Publishing Company.

[Wir85a] Niklaus Wirth. From programming language design to computer construction. *Communications of the ACM*, 28(2):159–164, February 1985.

[Wir85b] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, third edition, 1985.

[Wit60] Ludwig Wittgenstein. Philophische Untersuchungen. In *Schriften*, volume I. Suhrkamp Verlag, Frakfurt a.M., Germany, 1960. In German.

[WK89] Mark B. Wells and Barry L. Kurtz. Teaching multiple programming paradigms: A proposal for a paradigm-general pseudocode. *SIGCSE Bulletin*, 21(1):246–251, February 1989.

[Wol91] Stephen Wolfram. *Mathematica : A System for doing Mathematics by Computer*. Addison-Wesley, second edition, 1991.

[WP77] David H. D. Warren and Luis M. Pereira. Prolog — the language and its implementation compared with Lisp. *ACM SIGPLAN Notices*, 12(8):109–115, August 1977. Proceedings of the Symposium on Artificial Intelligence and Programming Languages.

[WS90] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, USA, 1990.

[Wu91] Shaun-inn Wu. Integrating logic and object-oriented programming. *OOPS Messenger*, 2(1):28–37, January 1991.

[WW88] J. H. Williams and E. L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 169–179, Januray 1988.

[Yok86] Shinji Yokoi. A Prolog based object oriented language SPOOL and its compiler. In Eiiti Wada, editor, *Logic Programming '86, Proceedings of the 5th Conferene*, pages 116–125, Tokyo, Japan, June 1986. Springer-Verlag. Lecture Notes in Computer Science 264.

[YS86] Jia-Huai You and P. A. Subrahmanyam. Equational logic programming: An extension to equational programming. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 209–218, St. Petersburg Beach, Florida, USA, Januray 1986. Association for Computing Machinery.

[Zan84] Carlo Zaniolo. Object-oriented programming in Prolog. In *1984 International Symposium on Logic Programming*, pages 265–270, Atlantic City, New Jersey, USA, February 1984. The Computer Society of the IEEE, IEEE Computer Society Press.

[Zav89] Pamela Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 6(5):15–25, September 1989.

Glossary

blueprint: The prototype multiparadigm programming environment — we developed using MPSS — allowing programming in the imperative, rule-rewrite, BNF grammar, regular expression, logic programming, and functional paradigms.

btrack: The implementation of the logic programming paradigm under the *blueprint* environment.

call gate The abstraction used for describing the calling convention mapping between the imported and exported routines of a paradigm and its superclass paradigm. See also *import call gate*, and *export call gate*.

class definition file: A file defining the variables and methods of a paradigm. This file is compiled by *pdc* — the paradigm description compiler of MPSS — to create the compilers and documentation for that paradigm.

class initialisation method: The method that is used to initialise the runtime machinery of a whole paradigm class. For example, for a paradigm supporting dynamic memory this method would initialise the memory allocation heap.

existing tool support: Support for tools already available in the programming environment. These tools are typically interpreters or compilers for some programming paradigms. The support provided, allows the seamless integration of those tools into the multiparadigm programming environment.

export call gate: An abstraction used for describing a paradigm's conversion of exported routines calling conventions, to conform with the conventions expected by its superclass paradigm.

fun: The implementation of the functional programming paradigm under the *blueprint* environment.

generic run-time support: Support libraries and tools provided our a multiparadigm programming environment generator, for integration into a multiparadigm programming environment, that do not depend on the paradigms that will be implemented. In the case of MPSS, this support includes the multiparadigm link editor, and the class and instance initialisation libraries.

imper: The implementation of the imperative paradigm under the *blueprint* environment.

import call gate: An abstraction describing the calling convention conversion, of routines that are imported from the superclass paradigm, to the conventions used within the importing paradigm.

input / output mode: Referring to the arguments of a predicate, specifies whether an argument is an input variable or a return variable.

instance initialisation method: The method that is used to initialise each separate module of a paradigm. For example, each `BEGIN` / `END` block in a Modula-2 program would be implemented as a separate instance initialisation method.

instance variable: A variable related to the state of a paradigm's implementation machinery. Every module implemented in a given paradigm needs a distinct set of instance variables.

instancev: A tool — part of MPSS — that identifies which of the variables that appear in the output of an existing compiler, should be made paradigm instance specific.

integrator: The example multiparadigm application allowing the symbolic and numeric integration of expressions. It is implemented in the *blueprint* multiparadigm environment.

mpld: See *multiparadigm link editor*.

mpss: Multiparadigm Programming Support System. A prototype tool suite for building multiparadigm programming environments. Our implementation of a multiparadigm environment generator.

multiparadigm application: A program written in more than one programming paradigm.

multiparadigm environment generator: A system that can be used in order to design and implement a multiparadigm programming environment.

multiparadigm framework: A system providing a methodology, and the technological support for creating multiparadigm applications consisting of *arbitrary* paradigms. MPSS provides such a framework.

multiparadigm link editor: A tool — part of MPSS — that links together object code from different programming paradigms. It automatically handles paradigm class and instance variable initialisation, and runtime library integration.

multiparadigm programming environment: A programming language, integrated environment, methodology, or tool suite that allows the application programmer to develop an application using more than one programming paradigm.

multiparadigm programming system: A combination of some of the following: multiparadigm environment, multiparadigm framework, multiparadigm environment generator, and multiparadigm application.

multiparadigm system structure: The overall design on which multiparadigm programming systems and multiparadigm applications are based. This includes the design strategy, the components and their combination, and the way paradigms inter-operate.

object-oriented paradigm: A programming paradigm based on a combination of objects containing methods and state, organised in classes, and structured using an inheritance mechanism.

paradigm description compiler: A tool — part of MPSS — that converts a paradigm class definition files into a compiler for that paradigm, together with its documentation.

paradigm class: A set of methods and variables defining the implementation of a programming paradigm. Paradigm objects, are the modules implemented in that paradigm.

paradigm description file: A file containing the source description of a paradigm class. It is compiled by the paradigm description compiler to make the paradigm described part of a multiparadigm programming environment.

paradigm: A set of laws, theories, applications, and experimentation relevant to a scientific theory. See section 2.1, page 4. In all other sections of this thesis we use *paradigm* to denote *programming paradigm*.

pdc: See *paradigm description compiler*.

primitive function: A function defined within the language, and not in terms of other language elements.

programming paradigm: An abstraction used to express aspects of a system's implementation. A way of expressing the programmer's intents. See section 2.1, page 4.

protect: A tool — part of MPSS — that given a list of variables that should be instance specific within the implementation of a paradigm, modifies the output of an existing compiler to isolate those variables.

regex: The implementation of a programming paradigm based on character regular expressions under the *blueprint* environment.

signature: A declaration of the arity, and possibly input output modes, of a function or procedure.

system wrapper: A tool — part of MPSS — that creates a distribution package out of all components of a multiparadigm environment.

term: The implementation of a term-based rule-rewrite paradigm under the *blueprint* environment.

variable protection: The task of hiding the names of some global variables before the linking process, typically, by prepending to their names, the name of the object file.

Appendix A

Implementation Notes

In this appendix we provide detailed implementation notes on *term*, *btrack*, and *fun*. Code examples have been provided where appropriate.

A.1 *Term*: Rule-rewrite Paradigm

The *term* paradigm implementation consists of the following parts:

- lexical analyser: converts the source-code input into tokens,
- parser: converts the stream of tokens into a syntax tree,
- code generator: converts the syntax tree into *imper* code,
- symbol table: provides a mechanism for the efficient storage and comparison of strings,
- term support: a set of functions providing support for the basic *term* datatype, the *term*, and
- library routines: a number of routines providing useful *term* functions.

In the following sections we will describe each of the parts in more detail.

A.1.1 Lexical Analysis

The lexical analyser is implemented in *regex*. Its functionality is simple: it ignores white-space, converts special character sequences into the single tokens, and the rest of the character sequences into atoms of the appropriate type (integer, character, variable, floating point). The special character sequences converted into appropriate tokens are the following:

```
not   is   import   \==   ==   =..   :-   ->   >=   <=
```

The atom character sequences are converted into *term* atoms using the functions `mkvar`, `mkatom`, `mkint`, and `mkdouble` listed in table 4.4, page 85. As an example, integers are recognised by the following code fragment:

```
append([], L -> L).

append([H | T], L -> [H | L2]) :-
    append(T, L, L2).
```

Figure A.1: The `append` rule in *term*

```
[
  imply(
    head(append, [ [], L ], [ L ]),
    [  ]
  ),
  imply(
    head(append, [ [ H | T ], L ], [ [ H | L2 ] ]),
    [ append(T, L, L2) ]
  )
]
```

Figure A.2: The `append` rule as a *term* term

```
[+-]?[0-9]+       {       /* Integer */
                          yylval.t = mkint(atoi(yytext));
                          return T_ATOM;
                  }
```

After an integer is scanned, the value of `yylval.t` will contain a pointer to an integer *term* atom.

A.1.2 Parsing

The parser is implemented in *bnf* with a grammar description similar to the one listed in table 4.5, page 82. The *term-expression* part of the grammar is expanded, to list one-by-one all *term* infix operators whose precedence is set by special precedence disambiguifying rules. After parsing, the global variable `module` is set to point to a term containing the whole parse tree of the program. The parse tree, is of the form of a list containing all the clause definitions. Each list element is a term with the name `imply` and two elements: the head of the clause as a single element of a term named `head`, and the rest of its elements in the form of a list. As an example parsing the `append` implementation in figure A.1 will generate the *term* term listed in figure A.2.

The term is created using the interface functions `mkterm`, `mktermbyname`, `mkfunctor`, and `mkatom` listed in table 4.4, page 85. For example the *bnf* rule for an infix multiplication term is:

```
    | term '*' term     { $$ = mktermbyname(2, "*", $1, $3); }
```

rule-name(input-arg$_{1.1}$, ..., input-arg$_{1.i}$ $->$ output-arg$_{1.1}$, ..., output-arg$_{1.o}$) :-
    body-clause$_{1.1}$(args),
    ...
    body-clause$_{1.m}$(args).

...
rule-name(input-arg$_{n.1}$, ..., input-arg$_{n.i}$ $->$ output-arg$_{n.1}$, ..., output-arg$_{n.o}$) :-
    body-clause$_{n.1}$(args),
    ...
    body-clause$_{n.m}$(args).

Figure A.3: An arbitrary *term* rule

A.1.3 Code Generation

The code generator, written in *term*, converts the syntax tree term into *imper* code. In order to understand this process, we must first explain how a *term* rule can be mapped into *imper* code. Taking a set of *term* rules of the form listed in figure A.3 (where $1.i = n.i$ and $1.o = n.o$) we can map it into an *imper* function of the form listed in figure A.4. As a concrete example, compiling the `append` implementation listed in figure A.1, will generate the *imper* code listed in figure A.5.

The process of code generation is performed in the following way:

1. A pass is made through the parse tree to create a list of rules defined, or imported, together with their respective parameter types (input or output).

2. For each rule defined in the parse tree, the parse tree is separated into the rules that have the same name and arity and the rest.

3. Code is generated for the first set of rules.

4. The process repeats with the rest of the rules.

Code for a set of rules with the same name and arity is generated in the form of a single *imper* function as follows:

1. An *imper* function is declared according to the name and arity as shown in figure A.4.

2. For every rule in the set, code is generated to check if the input parameters passed to the *imper* function match with those specified in the rule head.

3. Code is generated, so that if a match is found each one of the body clauses will be called while they return TRUE. This is done, by merging the calls in a boolean-and expression, which is short-circuit evaluated according to the C programming language specification [ANS89, §3.3.13].

```
bool
rule-name_(i + o)(struct s_Term *i0, ... struct s_Term *i1.i,
        struct s_Term *o0, ... struct s_Term **o1.o)
{
        if (matches i0 – i1.i against input-arg1.1 – input-arg1.i ) {
                declare a local variable for every output variable
                of body-clause1.1 – body-clause1.m
                if ( body-clause1.1 && ... && body-clause1.m) {
                        set o0–o1.o to the appropriate values
                        return TRUE;
                }
        }
        ...
        if (matches i0 – in.i against input-argn.1 – input-argn.i ) {
                declare a local variable for every output variable
                of body-clausen.1 – body-clausen.m
                if ( body-clausen.1 && ... && body-clausen.m) {
                        set o0–on.o to the appropriate values
                        return TRUE;
                }
        }
        return FALSE;
}
```

Figure A.4: An arbitrary *term* rule compiled into *imper*

```
bool
append_3(struct s_Term *i0, struct s_Term *i1, struct s_Term **o0)
{
    if (i0->type == et_Null && 1) {
        /*
         * L is i1
         */
        if (1) {
            *o0 = i1;
            return TRUE;
        }
    }
    if (i0->type == et_C1 && i0->arity == 2 && 1) {
        struct s_Term *v0;
        /*
         * L2 is v0, L is i1, H is i0->t[0], T is i0->t[1]
         */
        if (append_3(i0->t[1], i1, &v0) && 1) {
            *o0 = mkterm(2, et_C1, i0->t[0], v0);
            return TRUE;
        }
    }
    return FALSE;
}
```

Figure A.5: `Append` compiled into *imper*

4. Code is generated so that if all body clause evaluations succeed, the output variables of the rule are set to the values defined in the head, and the function succeeds, returning TRUE.

5. The final piece of generated code, is executed when all rules of the set fail: the function fails, returning FALSE.

The head matching code is generated, by recursively matching the elements of each term defined in the head, against the input variables passed in the function. This matching, is performed from the root of the term tree to the branches, also using the logical and operator of C, so that if the root of two term trees do not match, no further checks are made. Thus, matching of a term and a parameter is performed as follows:

1. If the term is a variable, ignore it.

2. If the term is an atom, check that the atom name of the term matches that of the parameter.

3. If the term is a number, check that is has the same value as the parameter.

4. If the term is a composite structure, check that the name and arity of the term matche that of the parameter.

5. If the term is a composite structure, recursively invoke the matching procedure for every element of that term.

If a *term* variable is used twice within the input part of a rule head, a run-time matching function `fct_unifies_2` is called, to verify that both *imper* variables point to terms that are equivalent. For example the code for `equals(X, X ->).` is:

```
bool
equals_2(struct s_Term *i0, struct s_Term *i1)
{
    if (fct_unifies_2(i0, i1)) {
        return TRUE;
    }
    return FALSE;
}
```

Scanning for variables with the same name is performed recursively through all the input terms of the rule.

The clauses in the body of the rule are called by creating their input arguments *in situ* at the place of call. If the head of a rule matches, local variables are defined to be used as result holders for the output parameters of the body clauses.

For reasons of run-time and compilation efficiency, there is a direct mapping between *term* atoms and *imper* identifiers. All atom names that are valid as *imper* identifiers are used by prepending the prefix `et_` to them. Sometimes though, a *term* atom will not form a valid *imper* identifier. [1] In those cases, a special routine will transform it into a *coded identifier* consisting of a string of the ASCII character codes of the original identifier. Thus for example, the *term* atom 'hello world' (which is not a valid *imper* identifier, as it contains a space) is converted to the identifier `et_X1041011081081113211911111410810 0`. The printing routine of *term* is responsible (at compile-time), to map the printing of such atoms to their real names.

A.1.4 Symbol Table

All the atom names that are used in *term* are stored in exactly one place in memory. Therefore, comparing two string atoms (or term names) for equality, can be done by a single pointer comparison operation. This is made possible, by a symbol table, which is used to store all the atom names. At program load time, when each paradigm object is initialised, the symbol table is used to initialise the variables used to refer to atoms, to point to the respective atom name strings in memory. The same symbol table, is used by all modules within a program, and therefore comparisons between terms generated in different modules (or paradigms) behave as expected. The symbol table, — based on the implementation described in [Spi90] — is organised as a hash

[1] A *term* identifier must start with an alphabetic character or an underscore and can only contain alphanumeric characters including the underscore. In addition, C keywords can not be used as identifiers [ANS89, §3.1.2].

```
#define et_C1 handles[0]

stab_handle handles[1];

void
mpss_add_constructor_instance_term(void)
{
        handles[0] = stab_findadd(atoms, ".");

}
```

Figure A.6: `Append` initialisation code

table of binary trees. From data presented in [LV73], it appears that a hash table is a viable technique for organising a symbol table, if another mechanism is available for resolving hash collisions. The first character of the symbol is used as the index into the hash table. A special opaque data type the `stab_entry` has been defined as the handle in conjunction with symbols. In addition, the data type `stab_table` provides the opaque data type definition of the symbol table. Access procedures for adding new symbols, accessing existing symbols, walking through the table using a higher order function, and getting the symbol values are defined.

The class initialisation function of *term*, initialises the hash entries of the table, while the instance initialisation method of each module, will add to the symbol table all the atoms used in that module. All the code in the modules, refers to the atom names by the handle returned by the symbol table `findadd` routine. In this sense, the atom names are similar to the notion of ”dynamic constants” proposed in [GM79]. In our *append* example, the initialisation code generated for the append module instance, contains the statements listed in figure A.6. The identifier `et_C1` is used throughout the code generated to refer to the “.” atom name, which is the list constructor functor[2]. According to the above, the initialisation code of `append`, adds to the global symbol table `atoms`, the list constructor functor “.”, and initialises the first element of the `handles` array, with the address of that symbol in the symbol table. If that symbol had already been added to the table by another class, then the address returned, would be that of the symbol first added.

A.1.5 Term Support

Term support (implemented in *imper*) supplies the basic functions needed to handle the `term` data-type. A `term` is defined as the following *imper* structure:

```
struct s_Term {
        stab_handle type;               /* Term name */
        short arity;                    /* Number of sub-terms */
        struct s_Term *t[1];            /* Sub-terms */
```

[2] A list $[a, b]$ is just a shorthand for the term $.(a, .(b, NULL))$.

```
};
```

The first two elements of the structure, the type and the arity, are common for all terms. The third element can vary across terms; for example it is an integer on terms of integer type, and a floating point number in terms of floating point type. Furthermore, there can be an arbitrary number of term pointers in the array denoted by the element `t`; the structure is dynamically allocated with a real size depending on the terms arity. Variables are implemented as special terms whose name is `__VARIABLE__`, arity is 0, but which contain a single element in the `t` array: the name of the variable.

The term support module implements all term creation functions listed in table 4.4, page 85, and all term access functions listed in table 4.5, page 85. Their implementation is simple and straightforward. The creation functions allocate the memory required to store the term structure, and set the members of the structure to the appropriate values. Most of the access functions are coded as macros to access the members of the structure. The only slightly complicated function, is that returning the name of the functor. That function first needs to examine the type of the functor. If the functor is of type integer, or floating point number, then it calls the system print routine to print the value of the number into a static buffer and returns the address of the buffer, otherwise it returns the name of the symbol as stored in the symbol table.

A.1.6 Library Routines

A *term* program can use a number of rules that are built-in to the language. These rules are listed in table 4.3, page 84. Many are implemented in *imper*, as they provide very basic functions that need special access to the contents of the term structure. Other, higher level ones, are coded in *term* and provide functionality often needed by *term* programs, in order to reduce code duplication. The split of the implementation code between *imper* and *term*, reflects both genuine technical decisions and development history[3]. Table A.1 provides a summary of the size[4] and implementation paradigm of all the library routines.

A.1.7 Debugging

A simple debugging capability has been implemented by means of a *tracer*. The *term* compiler will generate debugging code when the debugging flag is set. This code will at run-time, print the calls and exits from the rules. A sample output of the *tracer*, for the invocation of the clause `t :- append([a,b], [c], L).` is listed in figure A.7. Each line consists of the following parts:

- the name of the port (`Call`, `Exit`, or `Fail`),
- an increasing sequence number,
- the name of the rule, and

[3] The ability to use *term* modules together was added fairly late at the *term* development cycle, and therefore, up to that point all library routines had to be coded in *imper*.

[4] Lines of code without comments.

Name	Paradigm	L.O.C.
`(<)(A, B ->)`	*imper*	40
`(>)(A, B ->)`	*imper*	38
`(\==)(A, B -> )`	*imper*	49
`(=..)(T -> L)`	*imper*	14
`(==)(A, B -> )`	*imper*	48
`anint(A -> B)`	*imper*	6
`append(L1, L2 -> L3)`	*term*	4
`arg(N, T -> A)`	*imper*	19
`atom(A)`	*imper*	12
`cos(A -> B)`	*imper*	6
`exp(A -> B)`	*imper*	6
`fail`	*imper*	7
`functor(T -> N , A)`	*imper*	9
`head(L -> H)`	*term*	1
`integer(I ->)`	*imper*	12
`is(A <- B)`	*imper*	52
`length(L -> N)`	*term*	5
`log(A -> B)`	*imper*	6
`makefunctor(L -> T)`	*imper*	9
`member(X, L -> )`	*term*	4
`nameio(A -> L)`	*imper*	16
`nameoi(L -> A)`	*imper*	16
`nl`	*imper*	8
`pow(X, Y -> Z)`	*term*	4
`print(T ->)`	*imper*	8
`real(R ->)`	*imper*	12
`reverse(L1 -> L2)`	*term*	5
`sdebugflag`	*imper*	7
`sin(A -> B)`	*imper*	6
`tab(N ->)`	*imper*	11
`tail(L -> T)`	*term*	1
`tan(A -> B)`	*imper*	6
`termdebug(F ->)`	*imper*	8
`var(V ->)`	*imper*	12
`write(T ->)`	*imper*	40
Unary function support	*imper*	17
Total	*imper*	500
Total	*term*	24

Table A.1: *Term* library routines implementation summary

```
Call 0: t_0()
Call 1: test_append_3([a,b],[c],OUT)
Call 2: test_append_3([b],[c],OUT)
Call 3: test_append_3([],[c],OUT)
Exit 4: test_append_3(IN,IN,[c])
Exit 5: test_append_3(IN,IN,[b,c])
Exit 6: test_append_3(IN,IN,[a,b,c])
Exit 7: t_0()
```

Figure A.7: `Append` sample debug output

- its input parameters in the case of a `Call` or the output results in the case of an `Exit`.

The tracer is implemented by inserting suitable code at all the entry and exit points of all rules. This code is compiled, only when debugging is enabled, and therefore no performance cost is associated with it. Furthermore, the printing of tracing information can be disabled even when debugging *is* enabled. The code calls the special function `debug_real` with the name of the rule, its parameter modes (input / output), and the actual parameters as arguments. This function prints the tracing information, and can be used as a hook to provide more advanced debugging support.

A.2 *Btrack*: Logic Programming Paradigm

The syntax of *btrack* is relatively near to that of *term*; for this reason *btrack* is directly translated to a term, by a small Perl [WS90] script. An abstract code interpreter based on a *solve/unify* loop [Coh85, p. 1313], [ASS85, p. 335–380], [SS86b, p. 150], is implemented in *term*. The code that does the *btrack* to *term* translation detects all the `import` and `export` declarations and generates the appropriate *term* call gates. Thus, for every exported predicate, an equivalent *term* rule is generated, and for every imported *term* rule, *btrack* access code is generated.

A.2.1 Translation to *Term*

The *btrack* interpreter is a *term* clause of the form `btrack(Goal, Rules -> Bindings)`. *Btrack* will try to solve the goal using the rules, and return the generated variable bindings. Variables in the internal form of *btrack* are designated by a name starting with a "$" sign. The predicates of a *btrack* program are represented as a *rules* list. Each predicate of the *rules* list, is a list. The first element of that list is the head of the predicate; the goal — if any — is represented by the remaining elements of the list. The whole list is packaged within the *term* rule `rules(-> Rules)` for retrieval. This is illustrated by the path-find predicate [SS86b, p. 220] listed in figure A.8 and its associated *term* rule listed in figure A.9

```
path(X, X, [X]).

path(X, Y, [X | W]) :-
        edge(X, V),
        path(V, Y, W).

edge(a, b).
edge(c, f).
edge(a, c).
edge(c, g).
edge(a, d).
edge(f, h).
edge(a, e).
edge(e, k).
edge(d, j).
edge(f, i).

%%

export path(From, To -> Path).
```

Figure A.8: Path finding predicate in *btrack*

```
rules(->[
[path($X, $X, [$X])],
[path($X, $Y, [$X | $W]) ,
        edge($X, $V),
        path($V, $Y, $W)],
[edge(a, b)],
[edge(c, f)],
[edge(a, c)],
[edge(c, g)],
[edge(a, d)],
[edge(f, h)],
[edge(a, e)],
[edge(e, k)],
[edge(d, j)],
[edge(f, i)],
[]
]).
```

Figure A.9: Path finding predicate rules as translated to *term*

```
/* Succeed for an empty goal */
solve([], Prog, env(_, Bindings) -> Bindings).

/* Solve for built-in predicates */
solve([Goal | Goals], Prog, env(Level, Bindings)-> Result) :-
        builtin_unify(Goal, Bindings, Newenv),
        solve(Goals, Prog, env(Level, Newenv), Result).

/* Solve for rules in Prog */
solve(Goal, Prog, env(Level, Bindings)-> Result) :-
        Newlevel is Level + 1,
        tryall(Goal, Prog, env(Newlevel, Bindings), Prog, Result).
```

Figure A.10: *Btrack* evaluator: the `solve` rules

A.2.2 Execution

As mentioned above, the execution of a *btrack* predicate involves finding the variable bindings, that will satisfy a given goal for the set of rules specified. This is implemented by the `solve` rule, that takes as parameters the goal, the program, and the variable bindings, and returns the new variable bindings. The variable bindings are represented as an *environment*. The *environment* is a list of terms of type `bind(variable, value)`, representing bindings of variables to values. This approach is inefficient, resulting in O(n^2) search time. More efficient approaches based on structure-copying and structure-sharing [BM72] were not implemented in this prototype version. Within the `solve` rule this list is packaged within a term of type `env(level, environment)`, where `level` is the current level of the search tree. This is used in order to rename variables and avoid name clashes.

The implementation of `solve` is shown in figure A.10. We distinguish three cases in its operation:

1. The list of goals is empty: `solve` succeeds.

2. The head of the list of goals unifies with a built-in predicate: the environment is updated according to the unification with the built-in predicate and `solve` is recursively invoked for the rest of the goals.

3. If the goal does not fall in any of the first two cases, then *solve* will invoke the `tryall` rule which will try to match the goal with any of the goals within the program, returning a new, updated environment.

`Tryall` (listed in figure A.11) works by trying to unify the head of the goals passed as the first parameter, with the head of the predicate list passed as the second parameter. If this unification succeeds, the rest of the goals passed as the first parameter are appended to the subgoals of the matched predicate from the predicate list (with the appropriate variable renaming), and `solve` is invoked with a new set of goals. If the goal does not unify with the head of the first predicate in the list, then `tryall` is

```
tryall([Goal | Goals], [[Subgoal | Subgoals] | Rules],
                        env(Level, Env), Prog-> Result) :-
        copy(Subgoal, Level, Nsubgoal),
        lunify(Goal, Nsubgoal, Env, Newenv),
        copy(Subgoals, Level, Nsubgoals),
        append(Nsubgoals, Goals, New),
        solve(New, Prog, env(Level, Newenv), Result).

tryall(Goal, [Rule | Rules], Env, Prog-> Result) :-
        tryall(Goal, Rules, Env, Prog, Result).
```

Figure A.11: *Btrack* evaluator: the `tryall` rules

recursively invoked with the rest of the predicate list. The `copy` rule, creates a new copy of the composite structure passed, with all variables renamed according to *level*, by appending to them, an underscore, followed by the character representation of *level*.

A.2.3 Unification

Unification is performed by first looking up the values of variables passed in the environment (`lunify`), and then invoking the real unification rule (`unify`). The real unification is based on [SS86b, p. 150] and is listed in figure A.12. The implementation of the `lookup` rule is not trivial. When `lookup` finds a variable binding within the environment, it has to check the whole environment again for possible bindings of that variable. `Lookup` is therefore implemented by calling a recursive version of lookup with two copies of the environment: one to check for the current variable binding, and one to use on recursive invocations.

A.2.4 Built-in Predicates

Btrack built-in predicates (listed in table 4.6, page 89) are handled by the special *term* rule `builtin_unify`. This tries to match the head of the goal, with one of the built-in predicates, and then invokes the appropriate predicate implementation. The implementation is suitably coded in order to provide logical semantics. This is done by checking if the parameters passed are variables, or constants. A sample implementation of the `plus` built-in predicate is listed in figure A.13.

A.2.5 Inter-operation with *Term*

Inter-operation with the super-paradigm of *btrack*, *term* (and through that, by means of call-gates, with all other paradigms), is achieved by creating *term* compatible rules for all predicates that are exported, and *btrack* compatible predicates for all *term* rules that are imported. The *term* interface rule for each exported *btrack* predicate, performs the following functions:

- get a copy of the module's rules,

```
/* Unify after looking up the variables in the environment */
lunify(X, Y, Env -> Newenv) :-
        lookup(X, Env, X1),
        lookup(Y, Env, Y1),
        unify(X1, Y1, Env, Newenv).

/* Unify simple terms */
unify(X, Y, Env -> [bind(Y, X) | Env]) :-
        btrackvar(X), btrackvar(Y).

unify(X, Y, Env -> [bind(X, Y) | Env]) :-
        btrackvar(X), not btrackvar(Y).

unify(X, Y, Env -> [bind(Y, X) | Env]) :-
        not btrackvar(X), btrackvar(Y).

unify(X, Y, Env -> Env) :-
        not btrackvar(X), not btrackvar(Y),
        atom(X), atom(Y),
        X == Y.

unify(X, Y, Env -> Newenv) :-
        not btrackvar(X), not btrackvar(Y),
        not atom(X), not atom(Y),
        term_unify(X, Y, Env, Newenv).

term_unify(X, Y, Env -> Newenv) :-
        functor(X, FX, NX), functor(Y, FY, NY),
        FX == FY, NX == NY,
        unify_args(NX, X, Y, Env, Newenv).

unify_args(0, X, Y, Env -> Env).

unify_args(N, X, Y, Env -> Env2) :-
        unify_arg(N, X, Y, Env, Env1),
        N1 is N - 1,
        unify_args(N1, X, Y, Env1, Env2).

unify_arg(N, X, Y, Env -> Newenv) :-
        arg(N, X, ArgX), arg(N, Y, ArgY),
        lunify(ArgX, ArgY, Env, Newenv).
```

Figure A.12: *Btrack* evaluator: the `unify` rules

```
/* X = Y + Z */
logplus(X, Y, Z -> bind(X, R)) :-
        btrackvar(X), integer(Y), integer(Z),
        R is Y + Z.

logplus(X, Y, Z -> bind(Y, R)) :-
        integer(X), btrackvar(Y), integer(Z),
        R is X - Z.

logplus(X, Y, Z -> bind(Z, R)) :-
        integer(X), integer(Y), btrackvar(Z),
        R is X - Y.

logplus(X, Y, Z -> nobind) :-
        integer(X), integer(Y), integer(Z),
        R is Y + Z,
        X == R.
```

Figure A.13: *Btrack* evaluator: `plus` with logical semantics rules

- execute the *btrack* evaluator with the *term* rule as a goal (with suitably mapped variable names), getting a new environment, and
- set the appropriate *term* output variables from the returned environment.

For example the *term* interface code for the `path` predicate is the following:

```
path(From, To  -> Path) :-
        rules(Rules),
        btrack([path(From, To  , $Path)], Rules, Bindings),
        varval($Path, Bindings, Path).
```

Importing *term* rules from *btrack*, is made possible by adding another term rule `import_unify`, which is handled in a way similar to the `builtin_unify` rule. One such rule is added for every imported *term* rule. The rule is called from `solve` as another special case. It performs the following functions:

- looks up the variables of the predicate passed in the current environment,
- verifies that the variables are of the correct type (i.e. that the variables on the left of the arrow are bound within the environment to constants and that the variables on the right of the arrow are not bound in the current environment),
- calls the *term* rule with the correct values, and finally,
- returns back to the *btrack* evaluator with an updated environment containing the output variables bound to the *term* return values.

As an example a *btrack* import line like

```
call: edge(a,$V_1)
call: path(b,i,$W_1)
call: path(b,i,$W_1)
call: path(b,i,$W_1)
call: edge(b,$V_3)
call: edge(b,$V_3)
call: edge(b,$V_3)
fail: edge(b,$V_3)
fail: path(b,i,$W_1)
call: path(c,i,$W_1)
call: path(c,i,$W_1)
```

Figure A.14: *Btrack* sample debug output for the solving of `path`

```
import cos(A -> B).
```

will result in the following *term* code being generated:

```
import cos(A -> B).
import_unify(cos(V0, V1), Env -> [bind(LV1, R0) | Env ]) :-
        lookup(V0, Env, LV0),
        lookup(V1, Env, LV1),
        not btrackvar(LV0),
        btrackvar(LV1),
        cos(LV0, R0).
```

A.2.6 Debugging

Debugging is handled by a tracer following the Byrd [Byr80] model. Is is implemented, by calling a debug information printing rule, at suitable points of the *btrack* evaluator. In order to avoid intermixing *term* debug output from the execution of the *btrack* evaluator, together with actual debug out from the *btrack* evaluator, the *term* debugging output is switched off, when executing *btrack*. In this way, the programmer debugging a multiparadigm application, will only see debugging output from his application, and not "noise" output from the execution of the *btrack* evaluator. A part of the debugging output from evaluating the clause `path(a, i, P)` is listed in figure A.14.

A.3 *Fun*: Functional Programming Paradigm

Fun is also implemented using the *blueprint* multiparadigm programming environment. Lexical analysis is done in *regex*, parsing in *bnf*, intermediate code generation in *imper*, and intermediate code interpretation in *term*. An eval/apply interpreter [FH88, p. 193–195] written in *term* provides the runtime machinery.

Fun Construct	Description	Intermediate Code
name v_1 ... v_n = *E*	Function definition	val(N, lam(V1, ... lam(VN, E) ...)
expr$_1$ *expr*$_2$	Function application	app(E1, E2)
expr$_1$ *infix*$_{prim}$ *expr*$_2$	Infix primitive	app(app(prim(P), E1), E2)
number	Number	num(X)
variable	Variable identifier	var(V)
primitive	Primitive function	prim(P)

Table A.2: *Fun* program representation, as a *term* environment

```
fac x =
        cond    (x == 0)
                1
                (x * fac (x - 1)).
```

Figure A.15: A factorial implementation in *fun*

A.3.1 Lexical Analysis

The lexical analyser for *fun*, written in *regex*, converts the input stream of characters into tokens. During the scanning process, all white space and comments are ignored. Apart from the single character tokens, four token categories are returned:

1. primitive function identifiers: the names of the *fun* primitive functions (listed in table 4.8, page 93),

2. variable names: all other identifiers starting with an alphabetic character,

3. integers, and

4. floating point numbers.

In addition, the character sequences $==$ and $<=$ are scanned as a single token, as are the identifiers `import` and `export`.

A.3.2 Intermediate Code

Fun programs are converted into a *term* representation of sugared lambda calculus. Their *term* representation, is an environment: a list of terms named `val`, each of the terms containing the name of a variable, and its value. The way *fun* language elements are represented as an environment is summarised in table A.2. In order to avoid variable name clashing, all variables are given unique names, by appending an underscore followed by a function definition serial number to their names. As an example, the environment representation of the `fac` factorial function listed in figure A.15 as a *term* term, is listed in figure A.16.

```
val(fac,
    lam(
        x_9,
        app(
            app(
                app(
                    prim(cond),
                    app(
                        app(
                            prim('=='),
                            var(x_9)
                        ),
                        num(0)
                    )
                ),
                num(1)
            ),
            app(
                app(
                    prim('*'),
                    var(x_9)
                ),
                app(
                    var(fac),
                    app(
                        app(
                            prim('-'),
                            var(x_9)
                        ),
                        num(1)
                    )
                )
            )
        )
    )
)
```

Figure A.16: The *fun* factorial function as a *term* term

A.3.3 Execution

The execution of a *fun* function is implemented by an *eval/apply* interpreter implemented in *term*. The implementation of the interpreter is closely modelled after the one described in [FH88, p. 205–211]. Three term constructors, not listed in table A.2, are used for the internal operation of the interpreter:

1. `Closure(Expr, Env)` is used to package a weak head-normal form expression `Expr`, together with its environment `Env`, at the point of the evaluation of a lambda expression.

2. `Susp(Expr, Env)` is used when evaluating a function application to suspend the second expression evaluation, which is passed in that form to *apply*. In this way, the interpreter evaluates the expressions in a lazy manner.

3. `Op(Prim, Arity, Args)` represents a primitive in curried form. `Arity` is the number of arguments is still needs, while arguments already collected are contained in the list `Args`. As an example the primitive "+" is internally represented before any arguments have been collected as `op('+', 2, [])`; when it is ready for δ reduction it will be of the form `op('+', 0, [x, y])`.

The whole *fun* interpreter together with the implementation of the "+" primitive is listed in figure A.17.

A.3.4 Inter-operation with *Term*

Inter-operation with the super-paradigm of *fun*, *term* (and through that, by means of call-gates, with all other paradigms), is achieved by creating *term* compatible rules for all functions that are exported, and *fun* compatible primitive functions for all *term* rules that are imported.

Whenever a *fun* function of the form `name` $\texttt{arg}_1$ $\texttt{arg}_2 \ldots \texttt{arg}_n$ is exported to *term*, a *term* rule with the signature `name(`$\texttt{arg}_1, \texttt{arg}_2 \ldots, \texttt{arg}_n$`-> Result)` is created. That rule retrieves the environment where all the function definitions are stored, and evaluates a *recursive-let* of those functions to the application of the name of the function to the arguments passed. The result of the evaluation is also the result of the term rule. For example, exporting the `fac` function would result in the following *term* rule being generated:

```
fac(V0-> R) :-
        funs(F),
        feval(rlet(F, app( var(fac), V0)), [], R).
```

The reverse direction, of importing *term* rules of the form `name(`$\texttt{arg}_1, \texttt{arg}_2 \ldots, \texttt{arg}_n$`-> Result)` is implemented by adding a special `builtin` and `arity` term pair, similar to the one used to implement primitive functions, called `import_arity` and `import_builtin`. The `import_arity` term, specifies the number of input parameters expected by the *term* rule. The `import_builtin` rule, has as a head, the name of the imported *term* rule, followed by a a list containing as many variables, as the *term* rule has input parameters. When that head matches (by a call from the `apply` function of the *fun* interpreter), the arguments passed in the list, are evaluated

```
eval(app(E1, E2), Env -> Result) :-
        eval(E1, Env, R1),
        apply(R1, susp(E2, Env), Result).
eval(num(X), _ -> num(X)).
eval(var(V), Env -> X) :-
        envlookup(V, Env, Val),
        eval(Val, Env, X).
eval(prim(P), _ -> op(P, Ap, [])) :-
        arity(P, Ap).
eval(op(A, B, C), _ -> op(A, B, C)).
eval(let(V, E1, E2), Env -> Result) :-
        eval(E2, [val(V, susp(E1, Env)) | Env], Result).
eval(lam(A, B), Env -> closure(lam(A, B), Env)).
eval(closure(A, B), _ -> closure(A, B)).
eval(susp(E, Env), _ -> R) :-
        eval(E, Env, R).
eval(rlet(Defs, E), Env -> Result) :-
        append(Defs, Env, Newenv),
        eval(E, Newenv, Result).

apply(closure(lam(V, B), Env), A -> Result) :-
        eval(B, [val(V, A) | Env], Result).
apply(op(P, 1, Args), A -> Result) :-
        builtin(P, [A | Args], Result).
apply(op(P, N, Args), A -> op(P, N1, [A | Args])) :-
        N1 is N - 1.

arity('+' -> 2).

builtin('+', [A, B] -> num(C)) :-
        eval(A, [], A1),
        getnum(A1, A2),
        eval(B, [], B1),
        getnum(B1, B2),
        C is B2 + A2.
```

Figure A.17: The *fun* eval/apply interpreter

```
var: fac
num: 0
var: x_9
num: 5
num: 0
var: x_9
num: 5
built: equal(num(0),num(5),0)
var: fac
num: 0
var: x_9
num: 1
var: x_9
num: 5
built: sub(5,1,4)
num: 0
```

Figure A.18: *Fun* sample debug output for evaluating `fac`

in an empty environment, and then the *term* rule is called with the result variable being the result of the `import_builtin` rule. For example assuming that there is a *term* rule with the signature `termmult(A, B -> C)`, the resulting *term* interface rule would be:

```
import_arity(termmult -> 2).
import_builtin(termmult, [V0, V1] -> R) :-
        eval(V0, [], V0L),
        eval(V1, [], V1L),
        termmult(V0L, V1L, R).
```

A.3.5 Debugging

Debugging, is implemented by calling a debug information printing rule, at suitable points of the *fun* evaluator. A part of the debugging output from evaluating the function `fac(5)` is listed in figure A.18. The `num` and `val` lines, signify evaluation of numbers, and variables; they are implemented by modifying the two respective `eval` rules. The `built` lines refer to evaluation of built-in primitive functions (δ reductions); they list as their parameters, the arguments and the result of the primitive function. In order to avoid intermixing *term* debug output from the execution of the *fun* evaluator, together with actual debug out from the *fun* evaluator, the *term* debugging output is switched off, when executing *fun*. In this way, the programmer debugging a multiparadigm application, will only see debugging output from his application, and not “noise” output from the execution of the *fun* evaluator.

Index

M

N

O

P

Q

R

S

T

Appendix B

Trademarks

Intel, 386 and iAPX386 are trademarks of Intel Corporation.
Microsoft and MS-DOS are trademarks of Microsoft Corporation.
Miranda is a trademark of Research Software Ltd.
PDP-11 and VAX are trademarks of Digital Equipment Corporation.
TeX is a trademark of the American Mathematical Society.
Unix, is a registered trademark of USL/Novell in the USA and some other countries.
Colgate is a registered trademark of Procter and Gamble Inc.
Ada is a registered trademark of the U.S. department of defence.
Eiffel is a trademark of Interactive Software Engineering Inc.
Objective-C is a trademark of Productivity Products International.
Simula 67 is a trademark of Simula AS.
Smalltalk is a trademark of Xerox Inc.
All other trademarks are property of their respective owners.