The Generation of Concurrent Code for Declarative Languages

Nicholas John Rothwell

Ph D

University of Newcastle upon Tyne

1986

Abstract

This thesis presents an approach to the implementation of declarative languages on a simple, general purpose concurrent architecture. The safe exploitation of the available concurrency is managed by relatively sophisticated code generation techniques to transform programs into an intermediate concurrent machine code. Compilation techniques are discussed for $\mathcal{F}$–HYBRID, a strongly typed applicative language, and for $\mathcal{L}$–HYBRID, a concurrent, nondeterministic logic language.

An approach is presented for $\mathcal{F}$–HYBRID whereby the style of programming influences the concurrency utilised when a program executes. Code transformation techniques are presented which generalise tail-recursion optimisation, allowing many recursive functions to be modelled by perpetual processes. A scheme is also presented to allow parallelism to be increased by the use of local declarations, and constrained by the use of special forms of identity function.

In order to preserve determinism in the language, a novel fault handling mechanism is used, whereby exceptions generated at run-time are treated as a special class of values within the language.

A description is given of $\mathcal{L}$–HYBRID, a dialect of the nondeterministic logic language Concurrent Prolog. The language is embedded within the applicative language $\mathcal{F}$–HYBRID, yielding a combined applicative and logic programming language. Various cross-calling techniques are described, including the use of applicative scoping rules to allow local logical assertions.

A description is given of a polymorphic typechecking algorithm for logic programs, which allows different instances of clauses to unify objects of different types. The concept of a *method* is derived to allow unification information to be passed as an implicit argument to clauses which require it. In addition, the typechecking algorithm permits higher-order objects such as functions to be passed within arguments to clauses.

Using Concurrent Prolog's model of concurrency, techniques are described which permit compilation of $\mathcal{L}$–HYBRID programs to abstract machine code derived from that used for the applicative language. The use of methods allows polymorphic logic programs to execute without the need for run-time type information in data structures.

Acknowledgements

I would like to thank **Harry Whitfield** for his patient supervision of this project.

Many people have influenced this research. Foremost amongst these is **Luca Cardelli**, with whom I had many informal discussions, and who gave me a feel for the style and elegance of very high level languages.

Phil Treleaven provided some much-needed motivation, and made me aware of the Fifth Generation Project and its implications for academic research.

Many people at the universities of Newcastle and Edinburgh offered ideas and solutions to some of my many technical problems. **Richard Hopkins** gave me the benefit of numerous technical discussions concerning concurrent architectures.

I am indebted to **David Rees** for the facilities and opportunity to complete this thesis. **Richard Marshall** helped me master TEX, and avoid the most common formatting blunders.

I am extremely grateful to **Kevin Mitchell**, who volunteered a good deal of his time to helping me structure this thesis, and did a lot of the proof-reading.

I humbly thank **Jesus Christ** for His guidance and infinite patience during the four years of this research.

This work was funded by a research studentship from the Science and Engineering Research Council.

Declaration

I declare that this thesis has been composed by myself, and that the work it describes is entirely my own.

N. J. Rothwell

Nicholas John Rothwell
November 5, 1986

Table of Contents

1. Introduction **1**

1.1 Motivation 1

1.2 Concurrent Systems 3

1.3 This Thesis 7

2. Concurrent Compilation of $\mathcal{F}$–HYBRID **11**

2.1 Overview 11

2.2 The SECD Machine and the FAM 15

2.3 The $\mathcal{F}$–HYBRID Abstract Machine 16

2.3.1 The Bounded Stack Model 18

2.3.2 Profiling and Stack Determination 20

2.3.3 Indirection Vectors 21

2.4 Compilation Details 22

2.4.1 Simple Expressions and Declarations 22

2.4.2 Structures and Local Declarations 24

2.4.3 Functions and Function Calls 26

3. Optimisation and Concurrency Extraction for $\mathcal{F}$–HYBRID **31**

3.1 Optimisation Strategy 32

3.2 Concurrent Structure Assignment 33

3.3 Structural Tail Recursion 35

3.4 Application Optimisation 37
3.5 Control of Concurrency 40
3.5.1 Abstracted View of Optimisation 41
3.5.2 Increasing Concurrency 42
3.5.3 Decreasing Concurrency 47

4. The Language $\mathcal{L}$–HYBRID **58**
4.1 Introduction 58
4.1.1 Data Types 59
4.1.2 Unification 60
4.1.3 The Structure of $\mathcal{L}$–HYBRID 61
4.2 The Logical Computational Model 63
4.3 Nested Systems and Higher Orderness 66
4.4 Cross Calling 67
4.4.1 Behaviour of Logical Variables 69

5. A Static Typechecking Algorithm for Logic Programs **70**
5.1 Introduction 70
5.2 Basic Typechecking 71
5.2.1 Clauses and Clause Declarations 71
5.2.2 Unit Clauses 72
5.2.3 Non-Unit Clauses 74
5.2.4 Disjunctive Clauses 76
5.2.5 Recursive Clauses 77
5.2.6 Query Expressions 77
5.2.7 Query Declarations 79
5.2.8 Free Logical Variables 79
5.3 Type-Secure Unification 80

5.3.1 Type Determination 82

5.3.2 Methods 84

5.3.3 Method Determination 85

5.3.4 Logical Variables 86

5.4 Type Labelling 87

5.4.1 Method Construction 87

5.4.2 Properties of Labelled Types 89

5.4.3 Occurrence Rules 89

5.4.4 Unification Rules 90

5.5 Disjunctive Clauses 92

5.6 Non-Generic Clauses 92

5.7 Method Propagation 93

5.8 Recursive Declarations 95

5.9 Typechecking Logical Variables 96

6. Concurrent Implementation of $\mathcal{L}$–HYBRID **99**

6.1 Introduction 99

6.2 Synchronisation Instructions 100

6.3 Compilation of Clauses and Queries 104

6.3.1 Entry Sequences 106

6.3.2 Storage of Logical Variables 112

6.3.3 Query Expressions and Query Declarations 115

6.3.4 Logical Variables 117

6.4 Unification 126

6.4.1 Active and Passive Structures 126

6.4.2 Structure and Value Unification 127

6.5 Method Construction 132

7. Conclusions and Future Work **140**

Bibliography **147**

A. A Tutorial for $\mathcal{F}$–HYBRID **152**

A.1 Introduction 152

A.2 Using the System 153

A.3 Lexical Conventions 153

A.4 Simple Expressions 154

A.5 Structured Types 158

A.6 Declarations and Bindings 164

A.7 Functions 171

A.8 Exceptions and Fault-Values 179

B. A Tutorial for $\mathcal{L}$–HYBRID **185**

B.1 Introduction 185

B.2 Notation 185

B.3 Simple Clause Definitions 186

B.4 Queries 191

B.5 Complex Clauses and Constructors 195

B.6 Goal Structure 197

B.7 Higher Order Functions and Higher Order Clauses 200

C. HYBRID Syntax **208**

D. The HYBRID Abstract Machine Code **211**

D.1 The Abstract Machine 212

D.1.1 Memory Structure 212

D.1.2 Fault-Values 213

D.1.3 Registers **215**

D.2 Abstract Instruction Set **216**

Chapter 1

Introduction

1.1 Motivation

In the past few years, trends in computer technology have drastically altered the engineering trade-offs which have to be made in the design and realisation of computer systems. The traditional von Neumann architecture (single processor, flat global address space) is based upon the trade-off made by the designers of the first computers, which comprised a large, complicated, expensive processor attached to a (by modern terms) small memory. As technology has progressed, processors have become smaller and faster, and memory capacities have increased by several orders of magnitude.

With the advent of Very Large Scale Integration (VLSI), the traditional von Neumann trade-off has come under increasing attack. Processing power has become much cheaper, but the cost of communication between the various parts of a computing system has become proportionally greater. As a result, the von Neumann machine is now far from the ideal architecture to be realised in VLSI, and researchers are striving for machine architectures which better suit the design and implementation resources now offered.

Due to the relative cheapness of processing power and expense of communication, VLSI systems are best realised as large numbers of small, simple computing elements exhibiting local communication [Mead 80]. Whereas a traditional architecture encourages a sequential computational model (corresponding to a single processor executing a program in a flat, linear address space), a VLSI architecture should encourage a distributed computational model, in which sections of program are executed concurrently by a number of processing elements which communicate with one another. A concurrent model of computation can efficiently exploit the physical resources of a VLSI system, whereas a sequential model of computation will become progressively more unsuitable as technological developments continue.

Another argument can be made for concurrent systems, independently of such engineering issues. Whereas the first von Neumann machines were machine-code programmed, and executed instructions at the rate of a few hundred per second, modern machines execute bodies of software compiled from, in some cases, millions of lines of source code. Hardware systems based on conventional architectures are approaching their limits in terms of improvements in performance to support such sophisticated software systems. Applications are being identified [Fuchi 83] for which improvements in performance of orders of magnitude will be required, and this increase will not be achieved through continued miniaturisation and technological improvements. Continued increases in performance can only be achieved by exploiting VLSI components much more efficiently than via conventional architectures.

It has also been suggested [Ashcroft 77] that the lack of suitable hardware drastically inhibits our ability to conduct modern software research. Without appropriate machines, we are restricted in the expressiveness and power of our programming languages, and this impairs our ability to develop algorithms for modern applications. The greater the gap between architecture and application, the larger and more complex the application software will be, and the larger and more complex the language used to implement it. The von Neumann machines' commitment to sequential concepts will become more and more of a liability as developments on the hardware side make parallel processing increasingly attractive.

From these points, it can be deduced that adequate performance for higher level languages can only be achieved by exploiting concurrency in computer systems.

1.2 Concurrent Systems

Recently there has been a great deal of research into concurrent architectures [Treleaven c]. Many of these research projects have been based on the premise that the von Neumann computer architecture should be replaced by some radical new machine organisation. The search for new computational models to deal with concurrent systems has instigated research into novel concurrent architectures [Moto-Oka 83].

Early research into novel concurrent architectures yielded the ideas behind dataflow machines [Dennis 79]. Research based upon principles of VLSI design yielded various types of tree machine [Browning 80], as well as novel approaches to pipelined architectures [Kung 78]. New concepts about memory organisation yielded machines capable of performing message passing [Wilner 80] and reduction [Mago 80]. Such machines, however, fail to provide facilities for truly general purpose computing. By contrast, one point overlooked by the opponents of the traditional control flow architecture is its general purpose nature [Treleaven a]. The issues faced with such a general concurrent machine are the complexity of programming it safely, and of exploiting the available concurrency efficiently.

Traditionally, such complexity is handled by abstraction [Dahl 72]; we must therefore apply abstraction to the design and programming of concurrent architectures. Hardware does not abstract easily; architectures which have a particular abstraction tend not to be very general. Abstraction should therefore be handled in a familiar manner, by means of software; this allows the massive investment in software technology to be exploited, whilst allowing flexibility. The approach of this thesis is to use a sophisticated compiler to utilise the unsafe, concurrent hardware. The compilation techniques are hidden from the end user, so that their complexities do not interfere with the programming of applications, and the compiler provides a level of abstraction to protect the user.

A problem still to be addressed is the exploitation of concurrency. For the same reasons as above, this should be done by software abstraction. However, concurrency exploitation cannot be done automatically by a compiler, since static analysis often cannot determine the run-time behaviour of a program. It is therefore necessary for the programming language to provide the means to control the concurrency (or, at least, not let it be totally uncontrolled), without sacrificing abstraction. Is it possible to design a software system which is truly general-purpose, without being over-complex to use or unsafe to program in, and can it be made to handle the concurrency exploitation? The aim is a programming system where concurrency can be controlled, without the danger associated with concurrent programming.

Considered at a low enough level, computer systems are inherently concurrent, since they comprise a large number of asynchronous hardware components. However, this concurrency is hidden by the traditional "machine level", where a sequential instruction set is provided, together with some extremely unstructured mechanisms for dealing with nondeterministic events (interrupts). On top of this level is generally built an operating system to provide "false" concurrency in the form of multitasking. Although the abstraction is reasonable, it occurs at a fairly coarse level, allowing a computer system to be regarded as a number of processors each executing a (predominantly) isolated task. This trend continues upwards into the development of computer networks, where a simple abstraction (such as the paradigm of a distributed file system [Brownbridge 82]) sits over a coarse level of concurrency (a relatively small number of large discrete processors).

Utilising finer concurrency causes loss of abstraction. Languages designed to describe concurrent systems (such as OCCAM [INMOS 84b], and its precursor, CSP [Hoare 78]) are inherently unsafe because they make a very low level of concurrency directly visible to the programmer. There have been sequential languages extended to allow some concurrency to be expressed [Welsh 79], but these only serve to provide traditional operating system functions wrapped in a more sanitary package; in any case, programmers faced with any sort of traditional language tend to program in it sequentially, deliberately avoiding any features provided for concurrency. Attempts to provide a general form of implicit concurrency in traditional sequential languages (rather than the specialised concurrency found in array processing) have failed due to the complete unsuitability of the languages themselves. A traditional sequential language would, in any case, not provide any facility to control the concurrency of its execution, since the concurrency would be totally implicit.

A better approach is to formalise concurrent systems directly, and then build a language which captures that formalism. The language CCS [Milner 80] is a notation for a calculus which describes communicating agents in terms of state transitions. As such, the language has a high level of abstraction, since it encapsulates the calculus directly, without any awareness of lower level features.

At the highest level of abstraction are to be found denotational languages which are free from any notion of execution, whether sequential or concurrent. Since such languages are referentially transparent, there is no need to guarantee the order of execution of the parts of a program, since the outcome will be the same regardless. Such languages are, therefore, ideal for execution on a concurrent architecture, despite their abstraction over any computational concepts [Turner 81]. The problem remains, however, of controlling the concurrency of execution of a declarative language.

This problem has already been solved in the logic language PROLOG [Clocksin 81], where the *cut* operator is used to control the search path of an executing program, as well as to force termination of potentially infinite computations. The notion of sequential backtracking found in PROLOG has been attacked by many researchers in the field of artificial intelligence [Sussman 72], and its replacement by constructs less dependent on sequential execution makes PROLOG an ideal language for concurrent implementation [Kowalski 82]. In addition, such a construct provides the additional facility of allowing the concurrency in a program to be expressed by the programmer. Some dialects of PROLOG adopt a CCS-like notation to describe communication between concurrent parts of a program [Monteiro 84]; others allow the dependencies between the variables of the program to dictate the concurrency [Shapiro 83b]. The logic language presented in this thesis is a dialect of Shapiro's Concurrent Prolog, which embodies the control of concurrency by expressing dependencies between logical variables.

Functional languages have traditionally been free from such annotations, since they lend themselves to straightforward implementation on conventional architectures [Landin 64]. For a concurrent architecture, however, it is necessary to address the issue of controlling the concurrency utilised by an executing program. Many implementors have introduced annotations into functional languages to this end [Hudak 86, Halstead 85]. Such an approach destroys the abstraction gained by the use of a functional language in the first place, by giving the programmer dangerous insights into the way his program is being executed. In addition, such annotations are generally architecture dependent.

The ideal aim, when implementing a functional language on a concurrent architecture, is to allow some intimation of the required concurrency within the language, without sacrificing the declarative view of computation. The principle of separation of concerns suggests that the means to control the concurrency of a program should be a totally separate mechanism from the program itself, for example in the form of *pragmas*. However, the concise nature of functional languages means that such pragmas would have to occur often and with local scope, in order to provide sufficient control over the computation; it may be necessary to control the execution of individual function calls, for example. This use of pragmas would merely demote them to the level of annotations.

The functional language Crystal [Chen 86] takes a laudable step in this direction by allowing the arguments passed to functions to dictate the topology of the executing program. The topology can be altered by restructuring some of the functions which comprise the program. However, there is still a lingering awareness of the topology of the architecture, since arguments to functions are assumed to denote indices of processors[1]. The functional language presented in this thesis generalises the approach adopted by Crystal. As a result of optimisation techniques used in the compiler, it is possible to intimate the required concurrency within a program by writing sections of the program in a particular *style*. Advantages of this technique are that fine control over concurrency is achieved without annotations, and that this control is performed within the language itself. The compilation techniques presented make use of an abstract model of concurrency analogous to data-driven computation [Treleaven 82].

1.3 This Thesis

We begin (chapter 2) by presenting the functional language $\mathcal{F}$–HYBRID, together with an implementation technique for $\mathcal{F}$–HYBRID on a concurrent architecture. $\mathcal{F}$–HYBRID is closely related to a dialect of ML [Cardelli 83b], but does not feature the abstract type mechanism, although it has a polymorphic typechecker of the style intimated in [Milner 77]. The $\mathcal{F}$–HYBRID compiler generates code for an abstract machine similar to the functional abstract machine used for ML [Cardelli 83a], but featuring facilities for multiprocessing. The code generated by the compiler will execute on a concurrent architecture, but has no regard for efficiency (it generates a process for every function call), and allows no control over the concurrency from within the program.

In order to maintain determinism in the language, the exception mechanism devised for ML [Milner 85] has been replaced by a system which treats faults as first class objects which can be passed as arguments, returned as results and embedded in data structures. This concept (based upon ideas presented in [Mycroft 81]) is better suited to concurrent execution of programs, but requires the compiler to have some awareness of fault-values when code is generated. It is

[1]Crystal allows integer arguments only

also necessary to perform some run-time tagging of objects to determine whether or not they represent faults.

Chapter 3 presents techniques to transform and optimise the code generated by the compiler, in order to remove concurrency where it is redundant (ie. gives rise to idle processes in the system), and to perpetuate existing processes in the system. As a consequence of the optimisation techniques, it becomes possible to restructure programs to vary the concurrency utilised in their execution. Techniques are derived for controlling the concurrency exploitation of programs by using different language constructs, and for constructing identity functions which impose concurrency constraints.

An optimisation is presented which generalises tail-recursion optimisation, allowing many recursive functions to be modelled by perpetual processes. This technique allows certain classes of applicative program to be executed by a set of processes communicating systolically via streams of data values.

Chapter 4 introduces the logic language $\mathcal{L}$–HYBRID, which is a dialect of Concurrent Prolog [Shapiro 83b]. A primary aim in the implementation of $\mathcal{L}$–HYBRID has been to confer total type security on the Concurrent Prolog system, rather than to use the typeless (or dynamically typed) system found in conventional PROLOG. This type security allows $\mathcal{L}$–HYBRID to perform unification over objects of a variety of types, rather than over typeless functors.

This aim has been achieved by *embedding* the logic language within the applicative language, and altering the semantics of the polymorphic type system to enforce security on PROLOG's unification process. Due to this embedding, it has been necessary to strictly define the semantics of the interface between the two languages, and to reconcile the determinacy of the applicative language with the nondeterminacy of the logic language.

The language HYBRID is presented as a combination of $\mathcal{F}$–HYBRID and $\mathcal{L}$–HYBRID, in the spirit of Loglisp [Robinson 80]. The properties of a combined functional and logic language have been presented elsewhere [Sato , Barbuti], and such an approach yields a number of benefits. Logical inferences can be initiated from applicative programs, and functions can be called from logic programs. A clause declaration is treated as a special type of function definition, and so clauses can be declared under the same scoping rules as functions. As a result, it is possible to make *local* assertions, and perform local queries. Due to the higher order nature of the language, functions and clauses can be passed as arguments to functions and clauses, returned as results, or embedded in data structures. Clauses can be expressed in a form analogous to the lambda-expression form of a function, and the block structure of the language is respected in the bodies of clauses. $\mathcal{L}$–HYBRID can be regarded as an extension of $\mathcal{F}$–HYBRID which imparts facilities such as nondeterminism and pattern matching to the language, whereas $\mathcal{F}$–HYBRID allows manipulation of data values by means other than unification.

Chapter 5 describes the typechecking algorithm for $\mathcal{L}$–HYBRID. Typechecking not only ensures the type validity of programs, but also serves to provide information to the compiler, so that unification across objects of various discrete types can be performed. Such unification is totally secure, in that the structure of objects need not be inspected to determine their type in order to perform the unification. The concept of a *method* is derived as a means to allow distinct instances of polymorphic logical clauses to unify objects of different types.

Due to rigid type verification, it is possible to permit some unification of functional objects (functions or clauses); as a result, the language allows higher-order logical inferences. This facility, coupled with an extension of the Concurrent Prolog guard construct, allows meta-inferences and negation-by-failure.

Chapter 6 details the compilation process for $\mathcal{L}$–HYBRID programs. Unification operations are compiled directly into sequences of abstract machine instructions (in the spirit of [Warren 77]), rather than being left for interpretation by the machine. The nondeterminism of $\mathcal{L}$–HYBRID is a direct consequence of the nondeterminism of the underlying (abstract) architecture. Due to the variety of types of object supported by the enclosing applicative language, the compiler needs to generate type information which can be passed as argument to a clause; this information allows clauses to unify structures of arbitrary type.

The abstract machine architecture presented for $\mathcal{F}$–HYBRID is extended to cope with certain logical concepts (for instance, creation and instantiation of logical variables).

Chapter 7 draws conclusions and identifies topics for further research.

Appendices A and B present tutorials for the programming languages $\mathcal{F}$–HYBRID and $\mathcal{L}$–HYBRID, and describe language features not covered in earlier chapters. Appendix C presents the syntax for the HYBRID language. Appendix D details the abstract machine used by the compilers. The concurrency and communication primitives of the machine are presented and a description is given of the instruction set.

Chapter 2

Concurrent Compilation of $\mathcal{F}$–HYBRID

2.1 Overview

This chapter describes a compilation technique for the functional language $\mathcal{F}$–HYBRID. Programs written in $\mathcal{F}$–HYBRID are compiled into code to be executed by several cooperating processes. Since $\mathcal{F}$–HYBRID is a deterministic, referentially transparent language, the fact that a program can be executed concurrently does not in any way affect the meaning of the program, but only its execution time.

The language $\mathcal{F}$–HYBRID bears close similarity to a dialect of ML [Cardelli 83b, Milner 85], and familiarity is assumed with applicative languages. Appendix A comprises a tutorial-style description of the language.

Compilation is viewed as a two stage process. The first stage is the transformation of the program into an abstract intermediate code, consisting of instructions for a language-specific *abstract machine.* The second stage of the compilation process is the further processing ("assembly") of the abstract code into code for a physical architecture. Concern shall be given only to the first stage of the process; the abstract intermediate code, which is executable in its own right, is therefore regarded as the final, target code for the compiler.

The transformation process from programs to abstract machine code can itself be broken down into a number of stages. The *first pass* is responsible for transforming a program into abstract machine code. The *second pass* attempts to make the abstract code as efficient as possible. The code generated by the first pass is executable in its own right; the inclusion of the second pass does not affect the results obtained by executing the code, but may drastically improve the efficiency of its execution.

The second pass is responsible for making sequences of instructions execute more quickly, but also serves a much more important purpose. The compilation phase generates code without regard for how efficiently it can be executed by several cooperating processes. As a result, unoptimised code will make bad use of the multiprocessing resources available. The optimiser attempts to transform the code into a form which can be used efficiently by cooperating processes.

It is necessary to define the notion of "efficiency" with respect to a multiprocessing environment. An efficient program (or a program compiled to efficient code) will exploit the facilities of a multiprocessor to a greater extent than an inefficient program. The fundamental resource to be exploited in a multiprocessor is the individual processing elements. Efficiency is therefore a measure of how well the constituent processors are being exploited.

The multiprocessing environment under consideration allows processes to be created and destroyed dynamically, and imposes no virtual limit on the number of processes in existence at any time. It is therefore necessary to allocate processes to physical processors during the execution of a program. This can either by done according to some predetermined algorithm during the execution of a program, or can be performed by some "operating system" which services requests for processes at run-time. In either case, the efficiency of a program can be measured in terms of how many of the physical processing elements are performing some useful function at an arbitrary point in the computation.

For the purposes of this thesis, it is assumed that the cost of communication between processes is not inordinately expensive. It is hoped that techniques could be found on a closely coupled architecture to minimise this cost to the level of, say, a global store access.

From the point of view of the abstract machine architecture, the efficiency of the program is defined in terms of the number of processes created to execute the program, and the proportion of those processes kept busy during their lifetime.

Two aims can be identified for the optimisation process:

- *The creation of an optimal number of active processes* to cooperate in the execution of a program. This allows the time-complexity of a computation to be reduced (if there are enough physical processors to be allocated to virtual processes), or allows the target machine to be utilised to best effect (by keeping every constituent processor busy);

- *The minimisation of the number of idle processes in the system.* This aim may be disregarded for a multiprocessing system where idle processes can be "swapped out" in favour of runnable processes. However, it may be impossible to guarantee that an idle process never occupies a processor, or there may be a considerable overhead in maintaining idle processes in the system (for example, if the context switch time is considerable). It is therefore considered a valid aim to statically minimise, by optimisation, the creation of processes which will remain largely idle.

The use of the word "optimal" in the paragraph above requires some qualification. There is no theoretical reason why a functional program should not be reduced to a set of processes at a very fine level, with a process corresponding to each primitive operation performed in the language. If there is a great number of active processes in the system, then all the physical processors will be kept busy executing a subset of these processes. The assumption is made, however, that process creation and deletion are relatively expensive tasks in proportion to the tasks that an individual processor performs. To quantify, the cost of a process creation is considered to be equivalent to the cost of a full procedure call on a conventional architecture. It is therefore possible to outline a third aim for the optimisation process:

- *The perpetuation of existing processes in the system.* With the assumption that process creation is relatively expensive, it can be concluded that the perpetuation of processes is desirable. A process should not be created for a trivial task, since the execution of that task will complete relatively quickly, resulting in termination of the process; such a task is better performed by an existing process. If the creation of a process for a particular task is going to be immediately followed by the deletion of another process, the original process should be preserved for the new task. A process may therefore perform a series of tasks before finally terminating.

It is conceivable that the system could easily deal with the situation where the termination of one process is followed by the creation of another, but the two events may take place on separate processors, in which case little correlation can be drawn between them. In this case it is desirable to statically optimise this situation. A saving is also made if the two processes have some commonality (for example, register values or references to the heap), since termination and creation actions such as garbage collection need not be performed.

Since function application is the basis for the execution of functional programs, and since the evaluation of a function call will generally be a non-trivial task (the function will probably contain (possibly recursive) calls to other functions), the function call is treated as the operation for which a process creation is generated. Built-in functions (such as arithmetic operators) are assumed to denote trivial tasks, so that, for example, the evaluation of

$$expr_1 \ * \ expr_2$$

does not generate a process to perform the multiplication (although $expr_1$ and $expr_2$ may be evaluated concurrently). The unoptimised code generated by the compilation stage will contain a process creation for every call to a non-primitive function. It is the task of the optimisation stage to maximise the efficiency of these operations with regard to the points mentioned above.

2.2 The SECD Machine and the FAM

The abstract machine architecture used to implement $\mathcal{F}$-HYBRID is related to the functional abstract machine (FAM) used as an intermediate stage in the interpretation of ML [Cardelli 83a]. The FAM itself is based upon the abstract SECD machine used in the interpretation of Lisp [Landin 64].

The SECD machine is defined in terms of four registers denoted by the names **stack**, **environment**, **control** and **dump**. Since each register may contain a structured value (s-expression), the state of the machine can be defined purely in terms of these registers, without recourse to a global memory.

The **stack** is used for intermediate values during the evaluation of expressions; for example, all arithmetic operators take arguments from, and return results to, the stack. The **environment** is used to store the values of variables in the environment of the function being executed. The **control** register contains a list of instructions to be executed (the remainder of the currently executing function). The **dump** is used to save sets of registers. The execution of programs on the SECD machine is given detailed consideration elsewhere [Henderson 80].

The FAM is a version of the SECD machine optimised for fast function application. ML programs compiled into FAM machine code are not interpreted directly, but assembled into native code for the machine running the ML interpreter [Cardelli 83b]. As a consequence, the use of Lisp data structures to represent the machine state is inappropriate, and true stacks are used instead. The FAM contains three stack pointers denoted by **AP**, **SP** and **TP**; each points to a portion of main store. **AP** is the argument pointer, and refers to a stack used for intermediate results (like the SECD machine's **stack** register). **SP** points to a stack containing return addresses and closures of active functions; it performs the same function as the SECD machine's **dump** register. **TP** points to a stack of trap frames used to resume control of a function after an exception has been activated.

2.3 The $\mathcal{F}$–HYBRID Abstract Machine

The $\mathcal{F}$–HYBRID abstract machine is the target machine for the $\mathcal{F}$–HYBRID compiler. Programs written in $\mathcal{F}$–HYBRID are transformed into abstract code which can either be assembled into code for a physical architecture, or interpreted by an emulator for the abstract machine.

The $\mathcal{F}$–HYBRID abstract machine is a parallel control flow machine. There is a global address space which is accessible by each of an indefinite number of processes. Each cell in the address space may contain a simple scalar value, or a pointer to (address of) another cell. Each process executes a portion of code, and updates the store. A process may create any number of sibling processes, any of which may outlive its parent. The total number of processes running on the abstract machine at any time is considered unlimited, and it is assumed that an attempt to create a new process will always succeed[1].

There is a solitary mechanism for communication and synchronisation between processes: a simple message passing procedure is supported on memory locations in the heap area. Memory cells can be assigned a unique value (henceforth called *empty*). Any subsequent attempt by a process to access such a location causes the process to be suspended until the location is assigned a non-empty value (by some other process). This mechanism allows processes to communicate with one another, and also acts as a synchronisation primitive: one process can suspend itself, and be resumed by another process performing an assignment to memory.

[1]Some consideration is given to this point in chapter 7.

With the exception of this communication mechanism, no guarantees are made against interference between processes accessing the same location in the heap, and it is up to the compiler to maintain the integrity of data structures created on the heap (section 2.3.1). Since the language under consideration is purely applicative, this task should not present any great problems.

The address space is partitioned into a number of areas. The *code area* contains all the code generated for the program being executed, together with the code for *top-level* functions (those in the top-level environment). Every process has read access to the code area, but none has write access. The *heap area* is used to build data structures; any process may claim areas of store from the heap. An object claimed from the heap will initially contain the *empty* value in each constituent cell, but the cells may be accessed by any number of processes. The heap area is assumed to be garbage collected. The *stack area* is used to allocate stack space for each process. Each process is allocated a portion of stack (of bounded size) from the stack area, and generally uses it for local calculations. When a process terminates, the stack space is immediately reclaimed for subsequent use; in addition, some instructions cause a process to discard its stack and claim a new stack (generally of a different size). The stack area need not be garbage collected, since stacks are freed explicitly, but needs to be compacted to avoid fragmentation.

A process can pass a pointer to an element of its stack to another process, allowing the second process write values for the first. When this occurs (section 2.4.3), the stack must not be reclaimed, and the stack locations reserved for the second process to write to must not be overwritten by the first process.

Heap storage is claimed in blocks of predetermined size during the execution of a program. A heap allocation by a process results in that process being given a pointer to a fixed size block of store, all elements of the block being *empty*. These elements can be assigned simple or pointer values, and the address of the block can be passed between different processes. A block may be reclaimed when no process possesses any reference to it.

The state of the $\mathcal{F}$–HYBRID machine is regarded as the value of the memory locations of the global store, together with the state of each of its processes. Each process has a set of private registers each capable of holding a scalar or pointer value. Registers cannot be made *empty*, and it is impossible to assign a register the *empty* value from a store location (the process attempting to do so would be suspended). Although they are referred to by descriptive names, the registers are functionally equivalent, with two exceptions: one register is reserved for use as a stack pointer, and another is regarded as the program counter. Upon creation, a proportion of the register values of the child process are derived from the register values of the parent. This facility allows a child process to communicate with its parent, and a parent process to create several children which can communicate with one another.

2.3.1 The Bounded Stack Model

Since the abstract architecture provides no protection against interference between processes accessing store locations, a computational model is adopted which minimises the complexity of communication between processes. The simplicity of the model is due to the fact that the $\mathcal{F}$-HYBRID language is applicative in nature; there is no assignment statement, and therefore no notion in the language of updatable memory.

Initially, an $\mathcal{F}$-HYBRID program is executed by a single process, referred to as the *outermost* process. The outermost process will generally create child processes to assist in the computation; a child process is created for each call to a non-primitive function, although many of these process creations are removed from the final code by optimisation. A child process is passed a closure to execute, together with an argument, and the address of an empty cell in which to return the result of calling the function.

It is necessary to guarantee that every process will return a result, in all but two circumstances. The first of these is for the outermost process, which need not return a result if a top-level declaration is being established. The second is when a computation is non-terminating; an infinite computation will result in any processes waiting for the outcome to be suspended indefinitely.

The compiler guarantees that all execution paths through the code of a function will encounter a `Result` instruction, causing a result to be passed to the caller, before causing the process to terminate. Initially, all instances of `Result` are followed immediately by `Stop`, although optimisation can result in code containing a `Result` followed by, in some cases, considerable computations.

In unoptimised cases, a parent process allocates an element of its stack to be used as a result location by a child process. The code generator must guarantee that such a location will be examined by the parent (causing its suspension, if necessary), before the location is overwritten or the stack is reclaimed. It might initially be thought that an expression such as

```
let x = e in 7
```

would violate such a constraint; however, the storage scheme used for local variables (section 2.4.2) avoids stack interference in such cases. An expression of the form

```
let (_) = e in 7
```

(where `(_)` represents a "don't-care" value) results in the code for *e* being removed by optimisation, again avoiding interference on the stack.

In optimised cases, a parent may create a child process which will return a result into some structure on the heap. In this case, any *empty* location in a structure is either assigned by the process that claimed the structure, or is a result cell to be assigned by some child process.

It is guaranteed in the compilation process that:

- every process (except the outermost) will return a result, or else proceed indefinitely;

- any result returned to an element of a stack will be examined before the element is overwritten or the stack reclaimed;

- every *empty* location in structures claimed from the heap will be assigned by the claimant, or is the result destination for some descendant of the claimant;

- no (non-*empty*) value assigned to a heap element will be overwritten by another value at some later time.

Note that stack values may be overwritten, simply in the process of the stack being collapsed and re-used.

The top-level environment is implemented as a stack which grows indefinitely during use. Only the outermost process has direct access to the environment stack, and increases the stack size explicitly by means of the `InflateEnv` instruction (section 2.4.1). Values in the top-level environment may be overwritten when a top-level declaration *fails* due to a pattern-matching failure. In this case, the identifiers established in the binding are unconditionally bound to the fault-value

```
(fault :  {decl})
```

regardless of any previous values for these identifiers present in the environment (section A.6). This overwriting is guaranteed to be safe, since only the outermost process has direct access to these locations.

2.3.2 Profiling and Stack Determination

An executing program consists of a number of cooperating processes, each of which has a section of store to use as a local stack. For each newly created process, the size of the stack it requires must be determined before the process begins execution. Since every function is executed on a separately allocated portion of stack, the stack space required for each individual function invocation can be determined statically.

The abstract instructions `Code` and `Fork` (section 6.3) contain a size field which represents the maximum amount of stack space required by a process executing a function (in the case of `Code`) or a branch of a clause (in the case of `Fork`). When the compiler generates these instructions, the size field is left blank; the size is determined by a separate *profiling* stage which examines the generated code.

Associated with each instruction is an *effect* value, which is the amount by which the execution of the instruction will alter the size of the stack. The effect of each instruction can be determined statically. The profiler examines the code of the function, taking into account all possible execution paths through the code, and derives the maximum stack size by examining the effect of each instruction.

This process imposes some restrictions on the structure of the code generated. If an instruction can be encountered by more than one path (for example, as the destination of a jump) then the stack level at that instruction must be the same for each possible path. This allows the stack size to be determined within a loop, without knowledge of how many times the loop is executed[2].

2.3.3 Indirection Vectors

The syntax of the various instructions described below should be fairly self-explanatory. However, the notion of an *indirection vector* requires explanation.

An indirection vector represents the address of some cell in the global address space. It consists of a register name together with a number of non-negative integer offsets. The register is assumed to contain the address of some cell generated by, for example, claiming some storage space from the heap. The integer offsets are interpreted as address indices. The first offset denotes the cell addressed by indexing the register address by the integer value. For each

[2]This restriction is lifted in the case of terminating instructions such as `Stop`, in the interests of code efficiency.

subsequent offset, the cell referenced by the previous offset is expected to contain an address to be indexed by the offset. Any indirection vector will contain at least one offset.

An indirection vector will be written in one of the following forms. The form $reg/x_1/\ldots/x_i$ represents an indirection vector comprising a register *reg* and a series of integer offsets x_1 to x_i ($i > 0$, $x_i \geq 0$). In the form *reg/x'*, the term *x'* represents a series of offsets $x_1/\ldots/x_i$. The term *x.x'* represents a series of offsets the first of which is *x*, and the rest of which are represented by *x'* ('.' represents "cons"). The term *x'::y'* represents a series of offsets *x'* followed by a series of offsets *y'* ('::' represents "append").

2.4 Compilation Details

A program written in $\mathcal{F}$–HYBRID may either be an expression to be evaluated, or a declaration to be established. In either case, the program is initially executed by a single process which will generally cause the creation of other processes to assist in the computation. The $\mathcal{F}$–HYBRID interpreter considers the execution of a program to be *finished* when the process set is empty (ie. all processes have terminated). However, it is possible to conceive a system where expressions and declarations can be entered while previous ones are still being evaluated. The compiler is trusted to generate code which will not result in deadlock, based on the concurrency model outlined in section 2.3.1.

2.4.1 Simple Expressions and Declarations

A program is initially executed by a single process. This process is created with the register `R_Result` pointing at the top of the global environment stack. For a simple top-level expression (one without calls to non-primitive functions), a single process calculates the value using its local stack space for intermediate results (figure 2–1). The instruction `Stop` terminates the process. The instruction

```
::    1 + 2 + 3 + 4;

L1: Int 1;
    Int 2;
    Plus;
    Int 3;
    Plus;
    Int 4;
    Plus;
    Result;
    Stop;

>   10 :  int
```

Figure 2–1: A Simple Expression

`Result` copies the value from the top of the local stack into the cell pointed at by the register `R_Result` (in this case, the top of the top-level environment stack). The instructions `Result` and `Stop` generally occur together, but optimisation can result in code where a `Result` is executed by a process which can then go on to perform other computations before being terminated by `Stop`.

Top-level declarations are stored in a top-level *environment*. Values in the environment are made available to any process requiring access to them in the following manner. The outermost process created to evaluate an expression or declaration is given a pointer to the environment in the register `R_Result`. This process may directly access values in the environment by indirection vectors of the form `Result/`i (for $i \geq 1$). Any other process which requires an environmental value must be passed the value within a closure, since its `R_Result` register will point to its parent's stack (section 2.4.3) or some cell claimed from the heap (section 3.2).

The interpreter expects the value of a top-level expression to be placed at `Result/0` by the outermost process, using the instruction `Result`. The values generated for a top-level declaration are placed in `Result/`i for $i \geq 1$, by means of `Move` instructions; `Result/0` cannot be used because top-level declarations

```
::     let x = 1
           and y = false
           and z = nil;

L7: InflateEnv 3;                                      For x, y, z
    Int 1;
    Move Result/1;                                     Store x
    False;
    Move Result/2;                                     Store y
    Nil;
    Move Result/3;                                     Store z
    Stop;

>    x = 1 :  int
>+   y = false :  bool
>+   z = [] :  α List
```

Figure 2–2: A Top-Level Declaration

may reference the last top-level expression. The instruction `InflateEnv` (used by the outermost process) informs the interpreter that the top-level environment size is to be increased to make space for new bindings (figure 2–2).

2.4.2 Structures and Local Declarations

Structures are created in an unconventional manner. The block of store to be used for the structure is initially claimed from the heap. Elements of the structure are then assigned explicitly. This method of structure building allows a structure to be assigned by a number of concurrent processes (section 3.2), without the processes interfering with one another. For untagged structures such as tuples and lists, each location in the block is assumed to be *empty*. For tagged structures such as variants, the tag field (as specified in the instruction) is assumed to be assigned (figure 2–3).

Values declared within a local declaration are not stored directly on the local stack as might be expected, but are instead stored within a block of store

```
::    (1, nil, (2, 3));

L1: TupleCell 3;                              Structure for result
    Int 1;
    Move Local/1/0;                           First element
    Nil;
    Move Local/1/1;                           Second element
    TupleCell 2;                              Structure for (2, 3)
    Int 2;
    Move Local/1/0;
    Int 3;
    Move Local/1/1;
    Move Local/1/2;                           Third element
    Result;
    Stop;

>   (1, [], (2, 3)) :  (int, α List, (int, int))
```

Figure 2–3: Simple Structures

claimed from the heap. Although the stack might be used in simple cases where the bound value is calculated by the process establishing the binding, the heap must be used when a bound value may be created by some other process. For an expression of the form

```
let x = e₁
    and y = e₂
    and ...
in  7
```

any processes created for e_1, e_2, ... must not interfere with the parent's stack, since the values attached to the bound variables are immediately discarded. The use of a section of heap for variable bindings is considered preferable to forcing the parent process to await all the bindings, since evaluation of the bound expression (in this case, the 7) can be performed independently of the variable bindings. The use of processes to establish local bindings is discussed in section 3.2.

```
::    x + y where x = 1 and y = 2;

L7: Block 2;                                        For x, y
    Int 1;
    Move Local/1/0;                                 Store x
    Int 2;
    Move Local/1/1;                                 Store y
    From Local/0/0;
    From Local/1/1;
    Plus;
    Deflate 1, 1;
L6: Result;
    Stop;

>   3 :  int
```

Figure 2–4: A Local Declaration

The instruction `Block` claims a piece of store of the appropriate size; the block has the same form as that used for structures. Subsequent `Move` instructions assign the individual cells of the block. `From` instructions are used to retrieve the values of the identifiers (figure 2–4).

2.4.3 Functions and Function Calls

An expression denoting a function evaluates to a *closure*. A closure contains the code of the function, together with a set of values denoting the free variables of the function. The instruction `Code` allocates a block of store from the heap and assigns the appropriate location of the block with a pointer to the code. The environment of the function (containing its free variables) is built explicitly using a `Block` instruction followed by assignments to the block. The block is then incorporated into the closure by means of a `Closure` instruction. The instruction `NullClosure` may be used for functions which do not contain free variables.

The code generated for a function body assumes that the argument to the function will be passed in the register `R_Arg`. The parent's stack is not used; this allows the parent to immediately continue execution, since the value in `R_Arg` is guaranteed to be passed immediately to the child process. The function's first action is generally to remove the argument from `R_Arg`, since this register will be used to pass arguments in subsequent function calls. It is also assumed that the free variables of the function are pointed to by the register `R_Global`; `Global/0` is assumed to be the value bound to the first free variable, `Global/1` to the second, and so on. The body of the function will, in general, contain a number of branches, each with a distinct argument pattern. The code generated will contain a number of sections, one for each branch of the function. Each such section will contain code to decompose the argument according to the structure of the formal parameter. Failure will cause control to pass to the next branch, after the stack pointer (`R_Local`) has been repositioned to point to the bottom of the stack[3]. It is assumed that a process executing a function is required to return the result to the cell pointed at by the register `R_Result`. Each branch of the function is compiled into a code sequence terminated by the instructions `Result` and `Stop` (figure 2–6).

A process is created for every call to a non-primitive function. The first stage of the function call is the assignment of the closure to be activated, and the argument to be passed, to the registers `R_Func` and `R_Arg` respectively. A process is then created to execute the function by means of the instruction `PushProcess` (figure 2–5).

The assignment to `R_Func` and `R_Arg` is done in a particular order to avoid corruption of these registers during curried and nested application. If the register values are created and assigned one after the other, an expression such as

```
f(g(x))
```

will result in code of the form

[3]Study of the efficient compilation of patterns is considered beyond the scope of this thesis.

```
::    g nil;

L1: From Result/1;                              Retrieve g
    Nil;                                        Argument
    MoveReg R_Arg;
    MoveReg R_Func;
    PushProcess;                                Process for g nil...
    Result;                                     Return g nil
    Stop;

>   1 :  int
```

Figure 2–5: A Simple Function Call

```
Retrieve f;
Move R_Func;
Retrieve g;
Move R_Func;
Retrieve x;
Move R_Arg;
PushProcess;
Move R_Arg;
PushProcess;
```

resulting in corruption of R_Func.

The child process created by PushProcess is given initial register values as follows:

- The program counter (R_Program) references the first instruction of the activated function;
- The environment register (R_Global) points to a block containing the global variables of the function;
- The stack pointer (R_Local) points to the bottom of a newly allocated area of store to be used for local stack space;
- The result register (R_Result) points to a memory cell to be used to pass back the result of the function;

- The argument register (`R_Arg`) is inherited directly from the parent.

The execution of a `PushProcess` instruction causes the parent's local stack to rise by one element, and the new element to be assigned *empty*. This cell is the one pointed to by the child's `R_Result` register. When the child process terminates, therefore, the result of the function application will have been placed on the top of the parent's stack. Any attempt in the meantime by the parent to access this value will cause the parent to suspend until the value arrives. Note that the parent can perform other computations before accessing the result of the function call. If both arguments to a binary operator are function applications, then a process will be created for each before the parent attempts to access either result cell. In addition, many occurrences of `PushProcess` are removed by optimisation, so that the child process can assign the result of its computation directly to a structure on the heap without any intervention from the parent.

Allowing a child process to directly access its parent's stack is avoided in the case of local declarations (section 2.4.2), since the parent process is not guaranteed to examine all the results computed by its children. In the case of `PushProcess`, however, it is guaranteed that the parent will *always* examine the stack cell allocated for the child, before attempting to re-use that portion of the stack. For example, in the case of a binary operation such as

```
f(x) * g(y)
```

the parent process will allocate a stack cell for, and create a process to evaluate, `f(x)`, before going on to evaluate `g(y)`. The result of `f(x)` will then be accessed by the parent, causing suspension if necessary.

```
::    let g(a : a') = a    |
          g(_) = x;

L7: InflateEnv 1;                                      For definition of g
L6: Code 2 [                                           Code for g
       L5: Block 2;                                    Storage for a, a'
           FromReg R_Arg;                              First branch
           |
           |                                           Decompose argument,
           |                                           jump to L3 if impossible
           |
           |                                           Retrieve a
           |
           Result;
           Stop;                                       End of First Branch

       L3: Fall 2;
           Triv;
           FromReg R_Arg;
           |
           |                                           Second branch:
           |                                           Ignore argument, retrieve
           |
           Result;
           Stop;
    ];
    |
    |                                                  Closure for g
    |
    Move Result/1;
    Stop;

>   g = λ :  (int List → int)
```

Figure 2–6: A Function Definition

Chapter 3

Optimisation and Concurrency Extraction for $\mathcal{F}$–HYBRID

This chapter describes the optimisation phase of the $\mathcal{F}$–HYBRID compiler. Optimisation is performed on the abstract machine code based purely on the code itself, regardless of the language constructs the code represents.

Although the code generated by the compilation phase of the compiler is executable in its own right, it will make very bad use of the concurrent architecture used to execute it. In particular, the program will execute almost totally sequentially, since a process which executes a `PushProcess` instruction to activate a function will, in general, immediately demand the result of the function, and so wait for the result to arrive without doing any useful work. In addition, the fact that structure assignment is performed by means of `Move` instructions means that no concurrency would be exploited in assigning the elements of a structure; for example, in an expression such as

```
(f(), g())
```

the result of `f()` is sought for the first element of the structure before evaluation of `g()` commences.

This chapter describes a series of optimisations, some of which are actually performed during the generation of the abstract code, and some of which are applied once the code has been generated. The purpose of these optimisations is to increase the efficiency of an executing program by increasing its concurrency utilisation.

Optimisation results in occurrences of `PushProcess` being removed from the code, and generally being replaced by some other form of process creation. The unoptimised code is concurrent in the sense that a number of processes are created to execute a program, but very few of the processes are active at any one time. An executing program will consist of a small number of active processes together with a large number of processes awaiting function results. The optimiser does not increase program efficiency by increasing the number of process creations occurring in the code; rather, it attempts to alter the code generated so that each process can be kept active whilst its offspring is executing. The result of such alteration is that the same number$\overset{\text{of}}{\wedge}$(or fewer) processes are overlapped to a greater extent in the execution of a program, increasing the number of processes active at any one time. If any process cannot be kept busy whilst its offspring is executing, then the process creation is removed from the code, and the parent process performs both tasks. As a result of these optimisations, the number of processes in the system at any time may increase, since a single parent process is free to create a number of offspring without waiting for each one to terminate.

Subsequent examples show code which has been optimised.

3.1 Optimisation Strategy

Simple tail-recursion optimisation is performed during code generation. If a function (or top-level expression) performs a function application as its last action, then the function call can be performed as a tail-recursion, rather than by the creation of another process. The instruction `TailApply` causes control to pass to the function pointed to by `R_Func`. `TailApply` has the effect of replacing the process's local stack with a new stack of the correct size for the function to be executed. However, a simple optimisation can result in the stack being used

```
::    explode "Foo";

L1: From Result/5;                                    Retrieve explode
    Str "Foo";
    MoveReg R_Arg;
    MoveReg R_Func;
    TailApply;

>    [70; 111; 111] :  int List
```

Figure 3–1: Simple Tail Recursion

for a number of consecutive functions. It is necessary for the process to maintain a record of the size of the stack currently allocated; if a function invoked by tail-recursion requires a larger stack, then the current stack is replaced, otherwise the same stack may be used (with R_Local reset accordingly). Using such a system, a process would eventually allocate a stack large enough for any set of functions featuring mutual tail-recursion. Figure 3–1 illustrates simple tail-recursion.

All the optimisations performed by the compiler are peephole optimisations. The notation

$$x_1; \ldots; x_n \Longrightarrow y_1; \ldots; y_m$$

with $n > 0$, $m > 0$ represents the replacement of the instructions x_i by the instructions y_j $(1 \leq i \leq n, 0 \leq j \leq m)$.

3.2 Concurrent Structure Assignment

Structures are created in two stages. The first stage is the allocation of an empty structure from the heap, and the second stage is the assignment of the elements of the structure. Typically, a process which is creating a structure will assign the elements of the structure one by one; if one of the elements is the result of a function call, then the process will be suspended until the result of the function has been returned. With the structure assignment optimisation below, function results within structures can be assigned by the process executing the function, leaving the calling process free to continue execution.

The structure assignment optimisation is represented thus:

```
PushProcess;      ⟹ VecProcess -> Local/(x-1).x'
Move Local/x.x'

PushProcess; ⟹ VecProcess -> reg/x'
Move reg/x'
```

The instruction

```
VecProcess -> vec
```

creates a process to execute the function in R_Func in the same manner as PushProcess, but the result register of the child process points at the cell denoted by *vec*, rather than the top of the parent's stack. The parent's stack size remains unchanged.

The first optimisation deals with a structure accessed through the local stack. The VecProcess instruction assigns the element of the stack directly. Note that the structure offset is reduced by one in the optimised code, since the parent's stack does not inflate. The second optimisation deals with a structure referenced through some other register (for example, R_Result). The function calls in the example (figure 3–10) result in the creation of processes which assign the structure directly. Each VecProcess creates a process which is responsible for assigning the appropriate part of the structure.

Since the values bound within local declarations are held in structures, the same optimisation results in concurrency in declarations where the right hand side is a function call (figure 3–11). In this case, each call to the function f results in a process being created by VecProcess which is responsible for assigning the appropriate part of the declaration block.

3.3 Structural Tail Recursion

The optimisations presented in this section are performed on the assumption that "re-using" a process is preferable to the creation of a new process. This is justified by aim of keeping the number of idle processes to a minimum; in addition, the re-use of stack areas afforded by tail-recursion avoids fragmentation of store.

Structural tail-recursion is a generalisation of conventional tail-recursion. A function call is defined to be structurally tail-recursive if the enclosing function performs no further action apart from assigning the function result within a structure. For example, a function declaration such as

```
F(x) = 1 :  g(x)
```

contains a structural tail-recursive call to g, since the only action performed after the activation of g is an assignment to the list cell.

There are two optimisations associated with the extraction of structural tail-recursion. For the first, it should be observed that each execution path through the code for a function or top-level expression is terminated by the instructions `Result` and `Stop`. The optimisation

```
PushProcess; ⟹ TailApply
Result;
Stop
```

is performed during code generation, so this sequence need not be considered for optimisation. The first optimisation is

```
VecProcess -> vec; ⟹ Result;
Result                VecProcess -> vec
```

The `Result` can be brought forward since the top of the stack is unaffected by the `VecProcess` (the code generator guarantees that *vec* is not `Local/0`).

Intuitively, this optimisation allows a function or top-level expression to return a structured result before assigning the final element (if it is a function call).

This optimisation will generally result in the sequence of instructions

```
VecProcess -> vec; Stop
```

since the `Result` instruction has been brought forward. This sequence, however, contains a redundant process creation, since the parent process creates the child process and immediately terminates. Therefore, a second optimisation is applied:

```
VecProcess -> vec;  ⟹ SetResult vec;
Stop                   TailApply
```

This optimisation allows the parent process to perform the task initially intended for the child. The `SetResult` instruction reassigns the parent's `R_Result` register to point at the cell referred to by *vec*. The `TailApply` then begins execution of the function.

As a result of the optimisations above, a function which returns a structure as result will be made tail-recursive if the last element of the structure is a function call. This allows, for example, list generating functions to be tail-recursive. In the example of figure 3–12, the function `To` executes iteratively to generate a list. The final actions performed before the iteration (the `TailApply`) are to return the current list element (using `Result`), and to set the result pointer to reference the tail of this list element, ready for the next iteration (using `SetResult`).

The above optimisation deals with the situation where the last element of a structure is assigned the result of a function. Another optimisation can be applied in the more general case where the function call may be quite deeply nested within a structure, as long as the call occurs as the last evaluation before the structure is constructed:

```
VecProcess -> Local/0.x';  ⟹ Move Local/y.y';
Move Local/y.y'               VecProcess -> Local/(y-1).y'::x'
```

```
::    1 : 2 : 3 : 4 : To 5;

L1: ConsCell;
    Int 1;
    Move Local/1/0;
     |
     |                                        Rest of list
     |
    From Result/1;                            Retrieve To
    Int 5;
    MoveReg R_Arg;
    MoveReg R_Func;
    Move Local/1/1;
    Move Local/1/1;
    Move Local/1/1;
    Result;                                   Return [1; 2; 3; 4 | ε]
    SetResult Local/0/1/1/1/1;
    TailApply;                               Activate To

>   [1; 2; 3; 4; 5; 4; 3; 2; 1] :  int List
```

Figure 3–2: Deep Structural Tail Recursion

This optimisation acts on a function call followed by a series of `Move` instructions. It attempts to propagate the function call to the end of the execution path, so that it can be transformed into a tail-recursion. The indirection vector associated with the `VecProcess` gets longer at each optimisation, since the result cell is further embedded within the structure (figure 3–2). The optimisation is intended to be followed by an application of `VecProcess` $\Longrightarrow$ `TailApply`.

3.4 Application Optimisation

If a function call is required immediately before the computation can continue, then the result of the function call will be placed on the top of the stack, rather than being stored within some structure. In this case, the initial optimisation `PushProcess` $\Longrightarrow$ `VecProcess` cannot be applied, and the instruction performing the function application remains a PushProcess (for example, the Fibonacci function of figure 3–13). In many such situations, it is unnecessary to create a process to perform the function call, since the parent process will immediately wait for a result from the child process. There are circumstances where the process creation should be preserved, however. One of these is when the parent process can do useful work while the child is calculating a result. In an expression such as

```
f(x) * g(y)
```

the code generated has the form

```
Assign x to R_Arg, f to R_Func
PushProcess;
Assign y to R_Arg, g to R_Func
PushProcess;
Times;
```

The first PushProcess creates a process to evaluate f(x); the parent process can then proceed to the evaluation of g(y). However, a process creation for g(y) is unnecessary, since the parent process will immediately await the result returned in order to perform the multiplication.

A final optimisation is therefore performed to remove occurrences of PushProcess which will cause suspension of the parent until the child returns its result. Instead of increasing the concurrency in an executing program, this optimisation serves to remove idle processes in the system.

```
PushProcess;  ⟹  Apply;
[stack access]     [stack access]
```

[*stack access*] represents any instruction which attempts to read the top element of the stack (Local/0). The creation of a new process to perform the function call is considered unnecessary, since the parent process will immediately wait for the child process.

The instruction Apply is used to invoke a function without the overhead of creating a process, although a new portion of stack is allocated for the function.

The process executing the Apply instruction saves its current state, and then begins execution of the function. The saved state is restored when the function terminates.

Each process has a register R_Caller which is used exclusively for saving process states. When a process is initially created, the value of R_Caller is *nil*. Each time an Apply is encountered, R_Caller is assigned a pointer to a portion of store which is then used to save the state (registers) of the current process. Before the current process state is saved, the stack size is increased by one; the top of the stack can then be used for the result of the function call (the function being executed on the newly allocated portion of stack). Since each process state suspended by Apply has its own piece of local stack space, a process which has executed a series of Apply instructions can be viewed as having a single stack consisting of a number of segments, one attached to each saved process state.

Several executions of Apply will result in a list of saved process states. When a process terminates (by executing a Stop), any previously saved state (stored in R_Caller) is resumed, with the program counter (R_Program) advanced by one instruction. Each Stop instruction encountered will cause a previously suspended state to be resumed; if there is none, then the process terminates.

As a result of this optimisation, the expression

```
f(x) * g(y)
```

will be transformed into a sequence of instructions of the form

```
Assign x to R_Arg, f to R_Func
PushProcess;
Assign y to R_Arg, g to R_Func
Apply;
Times;
```

The second PushProcess instruction (followed immediately by the multiplication) is replaced by Apply. In the example of figure 3–13, the function Fib performs two recursive calls to itself. Because the first call is followed by an unrelated activity (in this case, setting up the second call), a PushProcess is used. Because the second call is followed immediately by `Plus`, which requires the result of the call, the second call can be performed using `Apply`.

3.5 Control of Concurrency

Although the 𝓕-HYBRID language contains no explicit constructs for controlling the manner in which a program executes, there must be some means to control the concurrency exploitation for any given program. The program may have undesirable properties when it is executed (for example, the creation of large numbers of suspended processes), and these properties may not be detectable statically when the program is compiled. For example, consider a expression such as

```
    if f(x) then g(x) else h(x)
```

Under some circumstances, it may be desirable to generate the results of `g(x)` and `h(x)` before the completion of `f(x)` (for example, if `f`, `g` and `h` all represent considerable computations, *and* there are sufficient computing resources to execute them in parallel[1]). Under other circumstances, it may be desirable to await the result of `f` before invoking either `g` or `h` (for example, if one of the latter represents a non-terminating computation).

The concurrency exploited by an executing program reflects on the optimisations that have been performed on the code of the program. The concurrency utilisation may therefore be affected by altering the *style* of the program to make it more or less amenable to optimisation.

[1]This expression may be invoked a considerable number of times within, say, a recursive function.

3.5.1 Abstracted View of Optimisation

The code generation stage of 𝓕-HYBRID compilation results in code which will execute concurrently, but will make very bad use of the available resources. The optimisation stage of the compiler results in code with the following properties:

- A process will be created for a function call which appears as an element of a structure;
- A function call which occurs as the last operation (other than structure building) within a function or top-level statement will be executed by the calling process;
- Any function call whose result is immediately required by the caller is performed directly *by* the caller, using `Apply`;
- Other function calls are performed by a new process created with the `PushProcess` instruction.

The concurrency exploitation of a program can therefore be viewed as being dependent on the number of function calls embedded within structures or local declarations.

The optimisation techniques presented earlier in this chapter should be considered in a general context, and not specifically in the context of sections 3.5.2 and 3.5.3. It is quite possible that different types of application would bring about ways of controlling the concurrency of programs, using various coding techniques quite different from those presented here.

Section 3.5.2 presents techniques for increasing the concurrency utilisation within a program, or for ensuring that the processes created are kept busy. Section 3.5.3 presents techniques for decreasing concurrency. It should be noted, however, that any technique applied to increase the concurrency of an application may be explicitly avoided to deliberately decrease the concurrency, and vice versa.

3.5.2 Increasing Concurrency

This section identifies two techniques for increasing the concurrency in a program. The first technique involves the use of local declarations to indicate that certain expressions should be executed in parallel with the context in which they are referenced; this technique follows as a consequence of the optimisation presented in section 3.2. The second technique attempts to remove unnecessary dependencies between bound variables and the values bound to them, by delaying as long as possible (or omitting completely) the pattern matching process performed for the binding.

Local Declarations

The use of a local declaration for the result of a function call will force a process to be created for the call (if this does not happen already). This increases the concurrency of the program in two ways:

- The calling process can continue execution until the value of the declared identifier is required;
- If the identifier is bound to a structured object, the child process can continue to create the object whilst the parent object "consumes" it.

If it is desirable to initiate a computation sometime before the result is required, then the computation should be bound to a local variable. Recall (section 2.4.2) that values for local declarations are stored in a *block* which is claimed from the heap. Assignments to such a block may therefore be performed by `VecProcess` instructions, when the value being assigned is the result of a function call. In figure 3–3, the identifiers `f1` and `f2` are used to begin calculation of `Fib 10` and `Fib 12` before evaluation of the condition. Note however that the computations bound to `f1` and `f2` both run to completion regardless of the outcome of the conditional expression.

```
::     let
          f1 = Fib 10                 -- Create computations
          and f2 = Fib 12             -- "f1" and "f2"
       in
          if Fib 8 > 100 then f1 else f2;

>    144 :  int
```

Figure 3–3: Background Computations

An argument to a function is passed in the register R_Arg. The calling process (or process context[2]) places the argument into R_Arg before creating a new process (or process context) to execute the function. This implies that a function cannot be invoked until its argument (or the outermost structure of its argument, if structured) is present, since the R_Arg register of a process cannot be assigned remotely by some other process.

If a function returns a structure as result, then the result may be returned before all elements of it are defined; the undefined elements correspond to processes still involved in the execution of the function (see section 2.3.1).

For an expression of the form

```
f(g(x))
```

the optimiser uses Apply to evaluate g(x), and again to activate f, since both arguments (x and g(x)) are needed by the parent process, which places the value in R_Arg. The optimisation is valid if g returns a single scalar result, since the outermost process would in any case be forced to await the result of g(x) before applying f. However, if g returns a structure as result, it would be possible for f to commence execution while g was still executing (because the child will be optimised to return a result and then continue execution); f could commence as soon as the outermost structure of g's result was returned. Consider

[2]Use of Apply results in a new context being generated for the *current* process.

```
::    let rec Incr nil = nil    |
              Incr(x : x') = (x + 1) : Incr x';

>   Incr = λ :  (int List → int List)
```

Figure 3–4: The function Incr

a list processing function such as Incr (figure 3–4). If a call to Incr occurs as a function argument, then Apply will be used instead of a process creation (figure 3–14). In this case, each of the outermost calls to Incr is suspended until the next innermost call has executed to completion. It would be possible to overcome this problem by preserving the PushProcess instructions in such cases; the optimisation

PushProcess $\Longrightarrow$ Apply

could be applied only when the function being activated returned an unstructured result[3]. However, such an approach reduces the degree to which the concurrency of the function call can be controlled. In this situations, the program can be restructured in order to reclaim the concurrency removed by the Apply optimisation.

In order to gain the latent concurrency in an expression featuring nested function calls (by allowing each function to generate partial results for consumption by its parent), a local declaration can be used within the argument (figure 3–15). Each local identifier is allocated a Block of one element, which can then be assigned using a VecProcess instruction, allowing the separate incarnations of Incr to execute in parallel.

The predefined composition operation "o" contains such a local declaration, being defined by

[3]A property which can be derived from the type of the function.

```
(f o g) x = f(y where y = g x)
```

This forces the creation of a process for each function of the composition. The example of figure 3–15 could have been written

```
(Incr o Incr o Incr) [1; 2; 3]
```

with no loss of concurrency.

Parameter Decomposition

A function will, upon invocation, attempt to decompose its argument in order to bind parts of the argument to lambda-bound identifiers. The execution of a function will therefore be suspended if any part of the argument to be bound to an identifier is not present (is *empty*).

Concurrency will be limited by:

- Having a function decompose its formal parameter to a greater extent than necessary;
- Having a function access a *component* of a formal parameter identifier before necessary.

These points present two ways to increase the concurrency within a function call. Each involves the degree to which formal parameters are decomposed, and where in the body of the function the decomposition takes place. The following two functions are equivalent:

```
F1(bind) = ..., f(e), ...                          [1]

F2(id) = ..., f(e where bind = id), ...            [2]
```

Function [1] will, however, attempt to decompose its actual argument immediately, possibly suspending the execution of the entire function. Function [2] will delay the decomposition until it is necessary.

The second improvement comes about by delaying reference to the argument as long as possible. Of the two functions

F1(*id*) = (*e* where *bind* = *id*) + f() [1]

F2(*id*) = f() + (*e* where *bind* = *id*) [2]

function [2] is likely to result in greater concurrency, since the decomposition of the argument is delayed until after the evaluation of f().

It may also be advantageous to re-order arguments to commutative binary operators when one argument is a parameter decomposition. Of the two expressions

(e_1 where *bind* = *id*) + e_2 [1]

e_2 + (e_1 where *bind* = *id*) [2]

expression [2] evaluates (or creates a process to evaluate) e_2 before attempting to decompose *id*.

Since a function attempts to decompose its argument according to identifiers in, and the structure of, its formal parameter, the nature of the formal parameter will also affect the concurrency by altering the degree to which a function's actual parameter must be present before the function can execute. Consider the functions defined by

```
F1(x : _) = x          [1]
F2(x : _ : _) = x      [2]
F3(x : y : z) = x      [3]
```

Function [2] performs greater decomposition than function [1]. An argument of the form

e_1 : ϵ

(where ϵ represents *empty*) will suspend function [2], although function [1] will immediately execute and return the result e_1. Function [2] will execute when presented with an argument

$$e_1 : \epsilon : \epsilon$$

However, function [3] will suspend, since it will attempt to assign the empty parts of the structure to the local variables y and z.

3.5.3 Decreasing Concurrency

In many circumstances, the assumptions made by the optimiser as to the concurrency desired within a program may be incorrect. In some such circumstances, the optimiser may attempt to extract excess concurrency from a program. The repercussions of this will be

- A large number of idle processes in the system (which the optimiser has assumed can be kept busy, when this is not the case);

- An excessive number of active processes in the system (a combinatorial explosion arising from unnecessary or undesired process creation).

In these cases, it is necessary to restrict the concurrency extracted from the program. Again, this is done by altering the *style* of the program.

This section presents an approach to concurrency control whereby functions written in a particular way can be used to control the concurrency of a program without affecting its meaning. Such functions might be viewed as annotations; however, two points serve to differentiate the approach presented here with the use of annotations:

- The `WAIT` and `FLATTEN` functions presented below are written in the 𝓕–HYBRID language itself; this shows that coding techniques can be used to reduce concurrency without recourse to "ad-hoc" features;

- The concurrency control comes about *naturally* as a consequence of the optimisation techniques presented above, rather than being a property of the annotations themselves.

An additional advantage to the approach presented here is that functions such as `WAIT` and `FLATTEN` can be written in $\mathcal{F}$–HYBRID as required to have any desired effect.

The Functions `WAIT` and `FLATTEN`

Concurrency results from function calls which occur in elements of structures or local declarations, since the elements of a structure or declaration block may be assigned independently by separate processes. A way of reducing the concurrency is force the creation of one process to await the completion of another. This will, for example, give complexity savings within recursive functions. In an expression such as

```
let id = e₁
in  e₂
```

some method is required for forcing termination of the processes evaluating e_1 before commencing evaluation of e_2.

The initial approach is to delay the evaluation of e_2 until *id* is assigned by the processes evaluating e_1. This can be achieved by the introduction of a curried function of two arguments:

```
    ::     let WAIT (_) x = x;

    >    WAIT = λ :  (α → (β → β))
```

A call to `WAIT` with one argument evaluates to the identity function. Therefore, in the expression

```
let id = e₁
in  WAIT id e₂
```

```
::    let rec Fib 1 = 1   |
              Fib 2 = 1   |
              Fib n = let x = Fib(n - 1)
                      in  WAIT x (x + Fib(n - 2));

>   Fib = λ :  (int → int)
```

Figure 3–5: Fibonacci Using Minimal Concurrency

the value of *id* is required in the R_Arg register for the call to WAIT, in order to yield a function to be applied to e_2. The evaluation of e_2 is therefore delayed until *id* is present.

Use of the function WAIT allows the Fibonacci function of 3–13 to be rewritten so as to allow almost sequential execution (figure 3–5).

When a declaration introduces several identifiers, various uses of WAIT can be used to selectively control the concurrency required. It is usually necessary to rewrite the declaration using enc (section A.6), together with some renaming to avoid name clashes, so that each individual declaration can express its dependency on the termination of some previous one. Once this has been done, a declaration can be modified in a number of ways. Figure 3–6 illustrates three declarations with different dependencies between the expressions being evaluated. In declaration [1], evaluation of e_1, e_2 and e_3 occurs in completely parallel. In declaration [2], evaluation of e_2 awaits the value of e_1. In declaration [3], e_1 and e_2 are evaluated in parallel, but evaluation of e_3 only commences when *both* values have arrived. The use of WAIT does not affect the meaning of the program, since

```
WAIT e
```

is an identity function, for any (terminating) value of *e*.

The WAIT function suspends evaluation of its second argument until the first argument becomes instanced; however, complete evaluation of the first argument

```
let id₁ = e₁
    and id₂ = e₂
in  e₃                                              [1]

let id₁ = e₁
    enc id₂ = WAIT id₁ e₂
in  e₃                                              [2]

let id₁ = e₁
    and id₂ = e₂
in  WAIT id₁ (WAIT id₂ e₃)                          [3]
```

Figure 3–6: Various Uses of WAIT

```
::     let rec LongList 0 = [1]    |
               LongList 1 = [1]    |
               LongList n = let x = LongList(n - 1)
                            in  WAIT x (x :: LongList(n - 2));

>   LongList = λ :  (int → int List)
```

Figure 3–7: An Inadequacy of WAIT

may be unfinished when evaluation of the second argument begins[4]. In the function LongList of figure 3–7, the use of WAIT will have little effect, since it will only cause the evaluation of LongList(n - 2) to pause until the first element of LongList(n - 1) arrives.

In order to overcome this problem, it is necessary to await all the values in a structure, rather than the outermost structure itself. This can be achieved by the use of a function for *flattening* structures.

An identity function

FLATTEN : $\alpha \rightarrow \alpha$

[4]In particular, this will happen if the first argument is structured.

```
::    let {rec FLATTEN_List nil = nil    |
               FLATTEN_List L = WAIT (FLATTEN_List(tail L)) L
          -- Flattens a list

          enc FLATTEN_List_List =
                 FLATTEN_List o (map FLATTEN_List)}
          -- Flattens a list of lists

         and FLATTEN_Tuple3(x, y, z) =
                WAIT x (WAIT y (WAIT z (x, y, z)));
         -- Flattens a tuple of three elements

>    FLATTEN_List = λ :  (α List → α List)
>+   FLATTEN_List_List = λ :  (β List List → β List List)
>+   FLATTEN_Tuple3 = λ :  ((γ, δ, ε) → (γ, δ, ε))
```

Figure 3–8: Various FLATTEN functions

is written to take a structure as argument and return an identical structure as result. However, FLATTEN is defined so that the result is returned only when the argument is complete.

In practice, a number of FLATTEN functions need to be defined, one for each distinct structure of object[5]. Figure 3–8 illustrates FLATTEN functions for objects of type α List, α List List and (α, β, γ). Using one of these, the LongList function can be rewritten in a sequential manner (figure 3–9).

Used in conjunction with each other, WAIT and FLATTEN can have wide ranging effects on the concurrent behaviour of 𝓕–HYBRID programs.

[5]See chapter 7 for a brief discussion of this topic.

```
::    let rec LongList 0 = [1]    |
              LongList 1 = [1]    |
              LongList n = let x = FLATTEN_List(LongList(n - 1))
                           in  WAIT x (x :: LongList(n - 2));

>   LongList = λ :  (int → int List)
```

Figure 3–9: A use of FLATTEN

```
::    (f 1, f 2, f 3) where f x = (x, ~x);

L7: |
    |                                            Code for f
    |
   TupleCell 3;
   From Local/1/0;
   Int 1;
   MoveReg R_Arg;
   MoveReg R_Func;
   VecProcess -> Local/0/0;                      Process for f 1

   From Local/1/0;
   Int 2;
   MoveReg R_Arg;
   MoveReg R_Func;
   VecProcess -> Local/0/1;                      Process for f 2
    |
    |
    |
   VecProcess -> Local/0/2;                      Process for f 3
   Stop;

>   ((1, ~1), (2, ~2), (3, ~3)) :  ((int, int), (int, int), (int, int))
```

Figure 3–10: Structure Assignment

```
::     let f x = ~x
       ins x = f 1
           and y = f 2
           and z = f 3
           and w = f 4
       in  x * y + z * w;


L15:Block 5;
L6:  |
     |                                              Code for f
     |
    Move Local/1/0;

    From Local/0/0;
    Int 1;
    MoveReg R_Arg;
    MoveReg R_Func;
    VecProcess -> Local/0/1;                        Process for f 1

    From Local/0/0;
    Int 2;
    MoveReg R_Arg;
    MoveReg R_Func;
    VecProcess -> Local/0/2;                        Process for f 2
     |
     |
     |
    VecProcess -> Local/0/3;                        Process for f 3
     |
     |
     |
    VecProcess -> Local/0/4;                        Process for f 4
     |
     |                                              Calculate x * y + z * w
     |
    Result;
    Stop;

 >   14 :  int
```

Figure 3–11: Local Declaration Assignment

```
::     let rec To n = if n > 0 then n : To(n - 1)
                      else nil;

L7: InflateEnv 1;
L6: Code 5 [
       L5: Block 1;                                  For n
           FromReg R_Arg;
           Move Local/1/0;                           Save n
           From Local/0/0;                           Retrieve n
           Int 0;
           GT;                                       Test n > 0
           TestFault 0, L2;
           FalseJump L3;                             False: return nil

           ConsCell;
           From Local/1/0;
           Move Local/1/0;                           n :  ...
           From Global/0;                            Retrieve To
           From Local/2/0;
           Int 1;
           Minus;                                    Evaluate n - 1
           MoveReg R_Arg;
           MoveReg R_Func;
           Result;
           SetResult Local/0/1;
           TailApply;                                ... :  To(n - 1)

       L2: |
           |                                         Return (fault :  {bind})
           |

       L3: |
           |                                         Return nil
           |
    ];
    |
    |                                                Closure
    |
    Stop;

.>  To = λ :  (int → int List)
```

Figure 3–12: Structural Tail Recursion

```
::    let rec Fib 1 = 1    |
              Fib 2 = 1    |
              Fib n = Fib(n - 1) + Fib(n - 2);

L6: InflateEnv 1;
L5: Code 5 [
       L4: |
           |                                      Deal with Fib 1, Fib 2
           |
       L2: Fall 2;
           Block 1;
           FromReg R_Arg;                         Store n
           Move Local/1/0;
           From Global/0;                         Retrieve Fib
           From Local/1/0;                        Retrieve n
           Int 1;
           Minus;
           MoveReg R_Arg;
           MoveReg R_Func;
           PushProcess;                           Call Fib(n - 1)

           From Global/0;
           From Local/2/0;
           Int 2;
           Minus;
           MoveReg R_Arg;
           MoveReg R_Func;
           Apply;                                 Call Fib(n - 2)

           Plus;
           Result;
           Stop;
    ];
    |
    |                                             Closure
    |
    Stop;

>   Fib = λ :  (int → int)
```

Figure 3–13: PushProcess and Apply

```
::    Incr(Incr(Incr [1; 2; 3]));

L1: From Result/1;                                        Retrieve Incr
    From Result/1;                                        three
    From Result/1;                                        times
     |
     |                                                    Generate [1; 2; 3]
     |
    MoveReg R_Arg;
    MoveReg R_Func;
    Apply;

    MoveReg R_Arg;
    MoveReg R_Func;
    Apply;

    MoveReg R_Arg;
    MoveReg R_Func;
    TailApply;

>   [4; 5; 6] : int List
```

Figure 3–14: An Inefficient Nested Application

```
::    Incr(
         x where x = Incr(
            y where y = Incr [1; 2; 3]));

L9: From Result/1;
    Block 1;
    From Result/1;
    Block 1;
    From Result/1;
     |
     |                                              Generate [1; 2; 3]
     |
    MoveReg R_Arg;
    MoveReg R_Func;
    VecProcess -> Local/0/0;
    From Local/0/0;                                 Execution continues as soon
                                                    as start of list arrives
    Deflate 1, 1;

L7: MoveReg R_Arg;
    MoveReg R_Func;
    VecProcess -> Local/0/0;
    From Local/0/0;                                 Similarly
    Deflate 1, 1;

L8: MoveReg R_Arg;
    MoveReg R_Func;
    TailApply;

>   [4; 5; 6] :  int List
```

Figure 3–15: A More Efficient Nested Application

Chapter 4

The Language $\mathcal{L}$–HYBRID

4.1 Introduction

This chapter is an introduction to a logical (or relational) subset of the declarative language HYBRID. HYBRID is a language which supports both logical and functional programming styles, either separately or in combination. The name $\mathcal{L}$–HYBRID will be used to refer to the logical subset of HYBRID.

HYBRID can be viewed as a functional programming language with facilities for declaring logical objects (clauses and variables), and executing logic programs (ie. performing deductions). $\mathcal{L}$–HYBRID is embedded into the functional language in such a way that logical inferences can be initiated from applicative programs, and functions can be called from logic programs. A program can therefore be written partly in a functional form, and partly in logical form. Functional expressions can contain declarations of clauses and calls to clauses (queries); as a consequence, programs written in $\mathcal{F}$–HYBRID can access program fragments written in $\mathcal{L}$–HYBRID. Logical expressions can contain functional expressions; as a consequence, programs written in $\mathcal{L}$–HYBRID can access program fragments written in $\mathcal{F}$–HYBRID. A clause declaration is treated as a special type of function definition, and so clauses can be declared under the same scoping rules as functions. Due to the higher order nature of the language, functions and clauses can be passed as arguments to functions and clauses, returned as results, or embedded in data structures. Clauses can be expressed in a form analogous to the lambda-expression notation for a function; a clause declaration simply serves to associate a name with such an expression, in the same way that a function declaration associates a name with an expression denoting a function.

The syntax for clauses and deductions bears some similarity to various dialects of PROLOG; some familiarity with PROLOG is therefore assumed. However, $\mathcal{L}$–HYBRID adopts lexical conventions appropriate to a functional language; this implies, in particular, that logical variables are introduced by declaration, and not determined by case distinction.

$\mathcal{L}$–HYBRID is statically scoped and strongly typed. This follows as a consequence of the fact that it is embedded within a functional language with these properties. Logic programs are polymorphically typechecked at compile-time (chapter 5); this ensures that, for example, a logical deduction cannot fail due to an inappropriate type of data object being manipulated at run-time.

A detailed description of the language $\mathcal{L}$–HYBRID is presented in tutorial form in appendix B. This chapter serves to introduce the concepts of the language needed in order to describe the typechecking algorithm (chapter 5) and the scheme used for code generation (chapter 6).

4.1.1 Data Types

$\mathcal{L}$–HYBRID features all the data types to be found in $\mathcal{F}$–HYBRID (viz. integers, booleans, strings, lists, disjoint sums, variants and functions). Programs written in $\mathcal{L}$–HYBRID can manipulate objects conforming to these types. In addition, there is a new type corresponding to the logical notion of a clause. A clause can be viewed as a special kind of function. It is "called" from within a query in the same way that a function is called in an expression. A clause is passed an argument (or tuple of arguments), which may be a simple expression, or may be some pattern containing logical variables. The type of a clause determines the type of argument which it may be passed. Whereas a function returns a result, which is a data object of a particular type, a clause either *succeeds* or *fails.* Any computation performed by a clause is communicated to the calling context by means of logical variables.

Logical variables behave in much the same manner as their counterparts in PROLOG. A logical variable is initially undefined (or *uninstantiated*), and is manipulated by the process of unification, which causes it to be instantiated to some data value. This value may itself contain uninstantiated values corresponding to other logical variables. The declaration of a logical variable simply serves to associate a name with a data object which behaves in a particular manner. Logical variables are not given any special type denoting their logical nature; the type of a logical variable is simply the type of any object to which it may become instantiated.

4.1.2 Unification

$\mathcal{L}$–HYBRID supports the notion of unification found in PROLOG. When an instance of a clause is activated, it attempts to unify the argument passed to it (the actual parameter) according to the structure and value of its argument. This process is different to that used for passing parameters to functions; within a function, the formal parameter serves merely to denote the structure of the argument, and specify which parts of that structure should be bound to identifiers. Within a clause, however, the formal parameter is a structure which determines, amongst other things, the effect the unification process has upon logical variables passed within the argument. However, the formal parameter sections of functions and clauses have an identical syntax. Unification can be viewed as a generalisation of the parameter passing method used in function calls, since the unification process may instantiate one or more logical variables appearing in the argument, in addition to assigning variables appearing in the head (formal parameter section) of the clause. The unification process provides another generalisation: variables may be *repeated* in the head of a clause, thus providing a facility to perform implicit comparison on parts of an argument, at no extra cost to the complexity of the language.

4.1.3 The Structure of $\mathcal{L}$–HYBRID

The underlying computational model for the execution of $\mathcal{L}$–HYBRID programs is based on that used by Concurrent Prolog. Both languages incorporate guarded-command indeterminacy, and dataflow-like synchronisation achieved by means of shared logical variables. This can be contrasted with the computational model for $\mathcal{F}$–HYBRID programs, which can be viewed as a process of reduction of expressions to the values they denote.

The functional computational model is totally deterministic; the value of an expression is dependent solely upon the form of the expression (subject to the bindings of a particular environment). The outcome of a logical deduction, however, may be nondeterministic, meaning that the outcome depends on properties of the system which are not accessible within the language itself. This property, together with the ability to invoke logical deductions from within functional expressions, provides the means to access nondeterminism from functional programs, should this be desired.

The computation of a logic program is equivalent to the construction of a proof of an existentially quantified goal from a set of axioms. In the case of $\mathcal{L}$–HYBRID, the scoping rules of the language determine the axioms used in the proof derivation. Such a computation may either *succeed* or *fail*; if it succeeds, then the output of the computation resides in the state of the logical variables presented to the program.

Logic programming languages provide facilities to control the proof process, at the expense of completeness. Such facilities may be used to perform computations not possible within first order logic (for example, negation), or to improve the efficiency of the proof procedure. It is also generally necessary to perform operations which are not practical within a simple relational computational model (for example, arithmetic operations). In either case, the completeness of the logical model is compromised in the interests of efficiency and practicality.

The computational model used by PROLOG is tailored to the execution of useful PROLOG programs, rather than to the general proof of first order logical deductions. PROLOG adopts purely sequential semantics, and can thus give an operational definition of the various non-logical features provided. PROLOG models logical indeterminacy by means of sequential search and backtracking; this means that a proof which requires the correct choice to be made by a number of conjunctive goals will eventually succeed.

$\mathcal{L}$–HYBRID has a computational model based upon that used by Concurrent Prolog. As with PROLOG, this model is incomplete with respect to first order logic. The computational model is nondeterministic, which implies that certain proofs will fail where, in terms of an indeterminate logic model, they would be expected to succeed. There is no notion of backtracking; a clause will perform a particular unification nondeterministically, regardless of the suitability of this unification for the deduction as a whole. To counteract this nondeterminism, there are facilities for explicitly controlling the proof procedure. The *guard* construct provides a means of restricting the nondeterministic behaviour of a proof, and the *commit* operator allows a clause to direct the progress of a proof according to the outcome of a guard.

Because the logic computational model is nondeterministic, there is no requirement for consistency to be enforced between a number of goals; such consistency is expressed explicitly by means of the guard construct. It is therefore possible to utilise parallelism in the execution of a query. Conjunctive (or and-parallel) goals may be executed in parallel, since the outcome of any one goal is not dependent on the success or failure or any other, and since nondeterminism implies that there need not be any collaboration to achieve a consistent set of bindings to logical variables. It is also possible to execute disjunctive (or-parallel) goals in parallel, since only one such goal will succeed, depending on the conditions of the guard, and not on the consistency requirements of the rest of the system.

Because the computational model is nondeterministic, there is no need to introduce primitive features to provide nondeterminism. In particular, arbitration (such as the merging of two streams whenever elements become available) can be achieved directly in the language, without resorting to some primitive construct which makes the arbitrary choice.

The computational model is essentially one of process reduction. A query is represented by a set of processes. Each process, executing a goal, attempts to nondeterministically reduce itself to a set of conjunctive sub-processes. Processes corresponding to unit clauses terminate immediately. The outcome of the query is *success* if all conjunctive processes and sub-processes succeed, and *failure* otherwise.

4.2 The Logical Computational Model

Although $\mathcal{L}$–HYBRID is a dialect of the language Concurrent Prolog, $\mathcal{L}$–HYBRID does not completely contain Concurrent Prolog (unification is not guarded), and Concurrent Prolog does not completely contain $\mathcal{L}$–HYBRID (there is no scoping or typechecking, and no higher-order facilities). However, the languages have a common computational model, based upon process reduction.

A query term or declaration contains a set of conjunctive goals, all of which must succeed in order for the query to succeed. The goals may be viewed as processes which collaborate to determine the outcome of the query. Communication is by means of logical variables shared between the goals. Synchronisation is achieved by specifying which logical variables may be instantiated by which goals; a variable marked as *read-only* within a goal may not be instantiated by the process executing the goal, and the process must wait for the variable to be instantiated by some other goal. The read-only annotation, therefore, serves to control the deduction by restricting the order in which goals can be reduced.

A particular query term activates the clause denoted by the clause part of the term. Consider a simple clause of the form clause *formals* :- G_1, ..., G_i

This clause will immediately *fail* if it cannot unify its actual argument with its formal parameter. Otherwise, it will *reduce* to the goals $G_1, \ldots, G_i$, which must all succeed, as before. A *unit clause* (a clause with no body) will succeed immediately upon unification. A literal goal (a boolean expression as goal) will succeed if the expression evaluates to true, and fail otherwise. It is the activation of unit clauses and literal goals which allow a logical computation to finally terminate, since these do not reduce to other goal processes.

A clause with several alternative branches denotes a disjunction:

```
clause formals₁ :- ...   |
       formals₂ :- ...
```

The clause will fail if no unification to the formal parameters succeeds (ie. if *all* alternatives fail). Otherwise, one of the successful alternatives is nondeterministically chosen to reduce to a conjunction (or to succeed, if it is a unit clause). Each alternative branch will attempt to unify the actual parameter against its formal parameter, but only one will be permitted to succeed or reduce.

The most general form of a clause body consists of two sets of goals, separated by the symbol '\':

```
clause formals₁ :- ...   |
                   ...
       formalsₙ :- G₁, ..., Gᵢ \ T₁, ..., Tⱼ   |
                   ...
```

The goals $G_1, \ldots, G_i$ constitute the *guard*, and the goals $T_1, \ldots, T_j$ the *tail*, of the clause body. Given a set of disjunctive clause alternatives, each will attempt head unification. For each alternative for which this is successful, the guard will be activated as a conjunctive system. Only if this succeeds will the process attempt to *commit* (exclude other disjunctive processes from reduction), and then reduce to the tail. Only one process will be permitted to commit successfully; any other process attempting to commit will simply terminate.

If there is no explicit commit symbol ('\'), then the guard is assumed to be null. In this case, a process attempts to commit when it has completed its head unification.

Not all process failures in the system are fatal; the failure of a head unification, or of a guard system, only fails that alternative. If all alternatives within a clause fail, then the clause invocation itself fails. Any query failure which is not within a guard system fails the entire query. A top-level query term or declaration succeeds if *all* the (non-guard) processes within it succeed, and fails if *any* (non-guard) process fails.

The guard system may consist of the single keyword `otherwise`; this may only occur within a clause which has more than one alternative branch, and only occur once within this clause. Such a guard will succeed if and only if all other guard systems fail. The `otherwise` notation allows negation to be expressed by failure without resorting to a higher-order or nested query. In a clause of the form

```
clause formals₁ :- G₁, ..., Gᵢ \ T₁, ..., Tⱼ
       formals₂ :- otherwise   \ U₁, ..., Uₖ
```

the tail system $U_1, \ldots, U_k$ will be invoked on the *failure* of the guard $G_1, \ldots, G_i$. In addition, the use of `otherwise`, together with the facility to "add" definitions to a clause (by defining a new instance of it in terms of an old one), makes it possible to define a clause in which alternatives are sought in a strict sequential order (for an example, see section B.3).

There are several important points to note about the logical computational model, which are not immediately apparent from the above description:

- There is *no* remote process termination. If a clause alternative commits, it does *not* immediately terminate the other disjunctive process sets. Rather, the other alternatives attempt to commit, fail to do so, and terminate. In a conjunctive system, a failure does not terminate the other goals in the system; rather, they continue, discover the failure condition, and terminate.

However, if a top-level query fails, the execution of the program continues without waiting for termination of the other processes;

- All unifications have irreversible, global effect (in particular, unifications performed by the alternative branches of a clause, even within a guard).

It is a consequence of these two points that a query can complete whilst there are still processes active (and which will eventually fail) capable of performing further unifications. Although the active processes will eventually terminate, they may have the side-effect of instancing variables visible in the result of the query.

4.3 Nested Systems and Higher Orderness

A logical query is a conjunctive set of goals. Each goal is executed by a logical process which attempts to reduce nondeterministically to other processes. A query terminates when the (non-guard) process set is empty. Execution of a logic program is essentially "flat"; either the entire process set succeeds (and the query succeeds), or one or more processes fails (and the query fails). Corresponding to this model of execution, there is a single level of synchronisation and communication, achieved by means of logical variables. There is, however, a level of nesting possible in Concurrent Prolog (and $\mathcal{L}$-HYBRID) programs. An alternative branch of a clause does *not* reduce to the guard system, since the tail must still be dealt with. Hence, a guard system is invoked as a separate query, and the clause invoking the guard awaits completion of the guard before continuing.

$\mathcal{L}$-HYBRID features another "layering" facility. From within logical queries it is possible to activate functional program fragments. Each such fragment can contain other clause definitions and queries. It is, therefore, possible have a deductive system parts of which are comprised of other deductive systems. Note that it is possible to perform non-local unifications within such layered queries;

there is no safeguard against a goal instancing a variable which is declared in a more global deductive system.

Operationally, a clause is treated as a special kind of function. It takes a special kind of argument (which may be a value, or may be some pattern of logical variables), and returns a special kind of result (*success* or *failure*). $\mathcal{L}$–HYBRID allows clauses to be treated as first-class objects; they may be passed as arguments, returned as results, and embedded in data structures. Functions may be higher-order over clauses, in the sense that they may takes clauses as arguments, or return clauses as results. Clauses themselves may be higher-order, since they may be passed argument patterns containing functions or clauses.

Appendix B contains numerous examples of clauses-as-objects, in particular clauses being passed as arguments to, and results of, functions.

4.4 Cross Calling

The HYBRID language is a combination of the applicative language $\mathcal{F}$–HYBRID, and the non-deterministic logic language $\mathcal{L}$–HYBRID. Sections of program written in either "sub-language" can access sections of program written in the other, and an entire program may consist of $\mathcal{F}$–HYBRID and $\mathcal{L}$–HYBRID mixed in any desired manner. The final application will dictate which parts of the program are written in which sub-language, and at what "granularity" the mixing will occur. The HYBRID language is, strictly speaking, functional, since any statement presented to the interpreter is expected to be a functional expression or declaration. However, the logic system is immediately accessible; the top-level expression may be a query-expression, and the top level declaration may be a clause declaration. At the lowest level, also, the HYBRID language should be considered functional, since literals and constructors represent functional values (as opposed to *functors*), and are evaluated in a functional way. In addition, all of the basic operations other than comparison (ie. unification) are only available as primitive functional operators.

Various constructs of $\mathcal{L}$–HYBRID can, therefore, be categorised in a functional sense. A clause is a special sort of function, since it is passed an argument, returns a result (*success/failure*), and may invoke other queries (including itself). A clause declaration is equivalent to a functional declaration which binds identifiers to clause-expressions. A query declaration serves to bind identifiers to values. A query expression represents a value. Logical variables can be thought of as bindings of names to objects with a particular behaviour. However, some constructs can only be explained with reference to the logical computational model (for example, nondeterminism), and can therefore not be regarded functionally.

Many similarities can be draw between the HYBRID language and the combined applicative and logic language Loglisp [Robinson 80]. The most powerful feature claimed for Loglisp is the ability to have the answer to a query be delivered as a functional data object, so that it can be subjected to arbitrary manipulation; HYBRID provides this feature in precisely the same manner. In addition, the ability to build primitive predicates applicatively and then invoke them from a logical context makes the logic system effectively open-ended.

The major difference between Loglisp and HYBRID is the treatment of clauses. Loglisp assertions are stored in a global database which may be interrogated from any part of an applicative program, regardless of its context. Side-effecting primitives are provided to add assertions to the database. By contrast, HYBRID has no notion of a global database of assertions; the assertions available in any context are comprised of the clause objects which are in scope at that context. This does not imply that HYBRID cannot implement a global database; a simple top-level declaration of the form

```
let clause C1 :- goals1
           and C2 :- goals2
           and ...
           and Cn :- goalsn;
```

establishes a set of assertions visible in any context. However, the fact that HYBRID treats clauses as first class objects allows much more complex structures of assertions to be manipulated. For example, local assertions of the form

```
let clause C :- goals
in  query G1, ..., Gn
```

provide a semantically clean alternative to the **assert** and **retract** primitives found in PROLOG.

4.4.1 Behaviour of Logical Variables

The behaviour of a logical variable cannot be explained in purely functional terms, since the unification process has no functional counterpart. However, logical variables (or the values contained within them) may be accessible to a functional part of a program. In this case, a logical variable is treated as a binding of a name to an object with a rather strange behaviour.

In most cases, the behaviour of a logical variable is unimportant, and the variable can just be regarded as a data value. However, it is possible for a query term or declaration to leave logical variables uninstantiated. In functional terms, a free variable is treated as the fault-value

```
(fault : {uninst})
```

As with other fault-values, it may be passed unaltered as argument and embedded in data structures, and will cause fault propagation if an attempt is made to access its value. For further discussion on the treatment of free logical variables in a functional context, see appendices B and D.

Chapter 5

A Static Typechecking Algorithm for Logic Programs

5.1 Introduction

This chapter describes a typechecking algorithm for logic programs. Programs are presented in the language $\mathcal{L}$–HYBRID, a logical subset of the combined logic and functional programming language HYBRID. Typechecking is performed statically, and a common polymorphic typechecking algorithm is used for both the logical and functional parts of a program. The approach taken is a largely pragmatic one, driven by two aims:

1. To infer type security on logic programs;

2. To provide the type information necessary to perform unification.

A more rigorous approach to the typechecking of logic programs is given in [Mycroft 83].

5.2 Basic Typechecking

Firstly, a presentation is given of the basic typechecking rules for $\mathcal{L}$–HYBRID. The type domain is that used by $\mathcal{F}$–HYBRID, augmented by a new type used to represent clauses. In addition, logical variables are dealt with in a special manner.

5.2.1 Clauses and Clause Declarations

The *clause* is the $\mathcal{L}$–HYBRID equivalent of a function. A clause may be "called" with an argument as a logical goal, and "returns" success or failure as a result. Initially, clauses which only have one alternative (disjunctive) branch will be considered. A *clause declaration* has a special syntax, in which the keyword `clause` acts as a declaration operator, thus:

`let clause` *Id(formals)* `:-` *body*

A clause declaration may introduce several clauses, using declaration operators such as and. Recursive clauses may be declared using `rec`. In each case, the declaration merely serves to associate each of one or more identifiers with a *clause expression*, which is a syntactic construct representing a clause. A declaration of the form above is equivalent to

`let` *Id* `= clause` *formals* `:-` *body*

It is therefore only necessary to consider the typechecking rules for clause expressions, since clause declarations are not (semantically) special constructs in themselves.

Every clause has a type which is an instance of the general type

α `Clause`

where α is a type variable, and `Clause` is a (postfix) unary type operator. The type `Clause` is unary since a clause does not return a typed result, but merely a success or failure condition. Instantiations of the type variable reflect the type of argument which can be passed to the clause. The type of the clause is constrained by the form of its formal parameter (as with lambda-expressions), which may contain bound identifiers. The type of the clause is further constrained by the types imposed on these identifiers, determined by the clause body.

5.2.2 Unit Clauses

A *unit clause* contains no body, but merely a *head* comprising a formal parameter, or *pattern*. The type of a unit clause is therefore determined solely by its formal parameter. The determination of the type of a unit clause is almost identical to the determination of the argument type of a function. Given two expressions of the form

```
    lambda bind. e
```

and

```
    clause bind
```

the typechecking conditions imposed by *bind* are identical. The only difference in the typechecking rules between lambda- and clause-bound identifiers is in the treatment of repeated variables. A repeated lambda-bound identifier constitutes an error, whereas a repeated clause-bound variable is allowed.

A structured argument (or *pattern*) serves to decompose, by unification, an actual parameter according to a particular structure, and bind parts of the parameter to identifiers for subsequent use. Determining the type of the formal parameter is equivalent to determining the most general type of object which can be passed to the clause as actual parameter.

A formal parameter consists of primitive objects such as constants and identifiers composed into patterns by means of constructors. The type of the formal

```
::     let clause A 3
                  and B nil
                  and C([true; false], in_lft "");

>   A = ψ :  int Clause
>+  B = ψ :  α List Clause
>+  C = ψ :  (bool List, (str + β)) Clause
```

Figure 5–1: A Simple Clause Declaration

```
::     clause(x);

>   ψ :  α Clause
```

Figure 5–2: A Simple Unit Clause

parameter is determined by the types derived for the lowest level components, and the conditions imposed by the constructors.

A constant (or nilary constructor such as **nil**) has an immediately derivable type. A functional expression has the type derived for the expression. Constructors impose constraints on the argument type according to their corresponding structured types (figure 5–1). An occurrence of an identifier which is *free* within constructors in the formal parameter serves to declare that identifier locally within the clause (figure 5–2).

Formal clause identifiers are declared in a similar manner to formal function identifiers. They are declared in a context which is visible only within the body of the clause. Identifiers are allocated non-generic[1] free types which may then be instanced according to context.

[1]A *non-generic* type is a type such that different instances of it share the same type variables. By contrast, each instance of a generic type is given a different set of type variables. See [Milner 77] for a detailed explanation.

```
::    clause(x, y, x);

>   ψ :  (α, β, α) Clause
```

Figure 5–3: A Repeated Head Variable

```
::    clause(_, _, _);

>   ψ :  (α, β, γ) Clause
```

Figure 5–4: Anonymous Variables

Such an identifier does not represent a value, but serves to name a *logical variable* local to the clause. The logical variable is unified to the corresponding part of an actual parameter in a similar manner to the binding of an identifier to part of a function argument. It is important to note, however, that an identifier may be repeated within the head of a clause. Each occurrence of such an identifier denotes the same logical variable, and so the types determined for each occurrence of an identifier must be unified (figure 5–3).

The reserved identifier '_' represents the anonymous logical variable. Each occurrence of '_' is therefore given a unique, free type (figure 5–4).

5.2.3 Non-Unit Clauses

A non-unit clause is a clause with a body containing one or more goals. The simplest form of non-unit clause has as body a list of goals:

clause *formals* :- *goal*$_1$, ..., *goal*$_n$

Each goal may reference the logical variables declared implicitly in the head of the clause. The type of the clause is therefore affected by the types determined for the variables from the goals.

```
::    clause x :- (head x? > 0);

>   ψ :  int List Clause
```

Figure 5–5: A Literal Goal

There are two forms of goal. The *literal goal* is a functional expression which returns a boolean value. The *clause goal* represents an activation of a clause with an actual parameter.

A literal goal is signified by enclosure within brackets; additionally, a primitive expression which cannot be interpreted as a clause goal is considered to be a literal goal. A literal goal is interpreted as an expression, and typechecked in the usual manner. The type of the expression is expected to be boolean, and is therefore unified against the type `bool`. The logical variables declared locally to the clause are visible within a literal goal, and have non-generic types. In addition, the typechecker ensures that they are referenced as read-only variables (figure 5–5).

A clause goal consists of a *clause part* and an *argument part*. The clause part denotes the clause to be activated, and the argument part denotes the argument to be passed to the clause. The clause part does not merely *name* a clause to be activated; rather, it is an expression which evaluates (in a functional sense) to a clause object which can then be activated. The clause part is therefore typechecked as an expression in the usual manner. For the purposes of typechecking, the argument is also regarded as a functional expression. Identifiers within the argument are sought in the environment of the body of the clause. An argument part may also contain the anonymous variable '_'; each occurrence of '_' is granted a unique free type. The goal is considered to be well-typed if the type of the clause is an instance of α `Clause`, where the type instanced to the variable α unifies with the type determined for the argument (figure 5–6).

A clause body may consist of a *guard part* and a *tail part*, separated by the

```
::    let clause Is_Zero 0;

>   Is_Zero = ψ :  int Clause
```

Figure 5–6: A Simple Clause

```
::    clause x :- Is_Zero y with y;

>   ψ :   α Clause
```

Figure 5–7: Use of "with"

commit symbol, '\'. In this case, the goals comprising the guard and the goals comprising the tail are each treated in the above manner.

A clause body may declare local logical variables for use in the goals of the body:

```
clause formals :- goal₁, ..., goalₙ
                  with with-vars
```

The variables specified after the `with` are declared locally for the clause body, and are therefore visible within the goals comprising the body (figure 5–7). These variables are given unique non-generic free types.

5.2.4 Disjunctive Clauses

A clause may have several disjunctive (or *or*-parallel) branches, each of which has a formal parameter or head, and an optional body:

```
clause formals₁ :- body₁    |
       formals₂ :- body₂    |
       ...
```

For the purposes of typechecking, each alternative branch is considered to be a clause in its own right, and typechecked independently of the others. Each

```
::    clause("", _, _)    |
            (_, false, _)    |
            (_, _, a) :- (a? > 0);

>    ψ :  (str, bool, int) Clause
```

Figure 5–8: A Disjunctive Clause

branch will therefore yield a type which is an instance of α `Clause`. The type of the entire disjunctive clause is obtained by unifying together the types obtained from each of the branches, as in figure 5–8.

5.2.5 Recursive Clauses

A recursive clause may be declared in a number of ways. A standard recursive declaration, introduced by the keyword `rec`, may contain a clause declaration (introduced by `clause`); in this case, the clauses will be established within the mutually recursive context of the declaration (figure 5–9). A clause declaration (introduced by `clause`) may contain recursive declarations introduced by `rec` (figure 5–10). In both cases, however, the clause declaration is simply an abbreviation for a normal recursive declaration using clause expressions. The typechecking rules for recursive clause declarations are therefore identical to those for normal recursive declarations. The identifiers being declared are established with non-generic free types, and the new environment is used to typecheck the right-hand sides of the recursive declaration. The types of the identifiers are then made generic. Again, with the exception of repeated variables, this procedure is identical to the treatment of recursive function declarations.

5.2.6 Query Expressions

A query expression is a set of conjunctive goals optionally qualified with a list of local logical variables:

```
::     let rec Fact 0 = 1     |
               Fact n = n * Fact(n - 1)
               and clause Foo x :- eq(x, Fact x?)     |
                          Foo x :- Foo(x? - 1);

>    Fact = λ :  (int → int)
>+   Foo = ψ :  int Clause
```

Figure 5–9: A Mixed Recursive Declaration

```
::     let clause Is_Zero 0
                  and rec Foo x :- Foo x;

>    Is_Zero = ψ :  int Clause
>+   Foo = ψ :  α Clause
```

Figure 5–10: A Recursive Clause Declaration

query *goals* with *with-list*

The goals, which may be clause goals or literal goals, are typechecked in the same manner as goals in a clause body. They do not, however, directly determine the resultant type of the query expression, although they may instance the types of the local logical variables. If there are no local variables, then then resultant type of the query expression is boolean, regardless of the type conditions imposed by the goals. If there is a single local variable, then the type of the query expression is the type determined for that variable. If there are several local variables, then the type of the query expression is a tuple, with each element type being the type of the corresponding logical variable. The type of a query expression

query *goals* with V_1, ..., V_n

is a tuple of the form

(t_1, ..., t_n)

where each t_i is the type derived for the variable V_i.

The logical variables have non-generic types whilst the goals are being typechecked, but the resultant query expression is given a generic type.

5.2.7 Query Declarations

A query declaration comprises a set of conjunctive goals and a list of local logical variables; it introduces (bindings to) the logical variables. The declaration is made within a particular environment, and modifies that environment to include the names of, and types assigned to, the logical variables. The logical variables are initially declared with non-generic free types, and the goals are typechecked as before. The types of the variables can then be made generic for use in the new environment.

5.2.8 Free Logical Variables

In the functional language $\mathcal{F}$–HYBRID, it is possible to create an object whose type is totally free. The constructor `fault`, when applied to an object of type

```
{apply, bind, cond, decl, infer,
 prim : str, signal : str, uninst} List
```

creates a fault-value with free type (section A.8). In practice, this causes no problem, since fault-values are printed in a particular manner regardless of their type, and because all the primitive operations in the $\mathcal{F}$–HYBRID language cater for being presented with fault-values (section D.1.2).

However, it is also possible for the typechecking of a goal within a query expression or declaration to derive a free type for a logical variable. This provides another opportunity for an object to be created whose type is an isolated type variable. Since logical variables have non-generic type whilst their goals are being typechecked, a variable whose resulting type is totally free must never be logically unified by any of the clauses within the goals (otherwise, its type would have been unified with the corresponding part of the clause type). Such an identifier emerging from a query must therefore be bound to an uninstantiated logical variable. When used in a functional context, the variable behaves as a particular kind of fault-value, and so may be permitted a free type.

5.3 Type-Secure Unification

Having presented a typechecking algorithm for ensuring type security in logic programs, the basis for the statically scoped, strongly typed logic language $\mathcal{L}$–HYBRID has been established. Since $\mathcal{L}$–HYBRID and $\mathcal{F}$–HYBRID comprise a common language, they both operate over the same data structures, and objects in both languages have a common type domain.

In the context of a strongly typed functional language such as $\mathcal{F}$–HYBRID, some type determination needs to be done when dealing with overloaded operators; since each such operator represents a possibly infinite set of functions, the correct one of which must be chosen according to context.

It is clear that such an overloaded operator cannot be applied to objects with polymorphic type, since the correct function cannot be determined from context. This makes functions such as

```
lambda (x, y).  x = y
```

illegal, since the precise meaning of '=' is unknown. Although this is an irritating restriction, functional languages can overcome this problem (by, for example, forcing the arguments to '=' to have monomorphic type, or by passing the correct comparison function explicitly as a parameter).

Since logic languages return results by unification, a scheme of this nature would impose the restriction that all results returned from clauses be monomorphic, and would make the logic language no more powerful than a functional language which disallowed polymorphic functions. This problem must therefore be solved in order to make a strongly typed logic language usable.

Conventional logic programming languages (such as PROLOG) are typeless. There is a universal data constructor (the *functor*), and unification is performed over data structures composed from functors. $\mathcal{L}$–HYBRID must, however, perform unification over objects of a wide variety of types, and unification over objects of a particular type must respect the structural properties of that type.

Two general approaches are outlined to the problem of unifying objects of various types. The first approach is to have a single, general unification algorithm which embodies all of the different kinds of unification required. This approach requires that all dynamic data structures be marked with their type, or at least with enough information so that the unification algorithm can determine how to perform the correct kind of unification. The unification algorithm incorporates run-time checks on the objects being dealt with to determine their structure.

This approach has a number of disadvantages. The major disadvantage is that the unification algorithm is comparatively complicated, since the manner in which it operates depends on the structure of objects encountered at run-time, and this structure must be determined dynamically. Since structures (and their types) may be arbitrarily complicated, the unification algorithm must in addition be recursive, even if the objects being unified are not.

Another disadvantage is that such an approach makes demands on the (run-time) structure of data objects. In essence, it must be possible to tell, from the representation of an object, how to correctly unify it. Although data objects can be easily tagged with such information, such a system does not fit well into an environment where all other operations can be performed regardless of the structure of data objects.

A third disadvantage of such a system is that it does not provide safeguards against attempts to unify objects of inappropriate type (for example, functions). Although it would be possible to tag such objects as being non-unifiable, such tags would only be encountered at run-time. This scheme in effect amounts to performing run-time typechecking. This disadvantage of dynamic unification can in fact be overcome by making modifications to the polymorphic typechecking algorithm, so that unifyable objects can be statically distinguished from non-unifyable ones [Milner 86]. However, a modified typechecking algorithm can, in fact, totally guarantee that a particular unification algorithm only acts on objects of the correct type, and a second approach adopting such an algorithm is the one adopted.

Since the $\mathcal{L}$–HYBRID language provides a potentially infinite set of unifyable types, it is chosen to regard unification as an infinite set of algorithms, one for each unifyable type. Each unification operation featured in $\mathcal{L}$–HYBRID is performed by one such algorithm.

This approach means that attempts to unify inappropriate objects can be rejected at compile-time, since no correct unification algorithm exists. Note also that the unification process is simplified, since a particular algorithm may "assume" that it is acting on objects of the correct type, and need not inspect their structure. An algorithm may therefore be optimised by making assumptions about its arguments. It is also unnecessary to select the correct algorithm at run-time, since a static approach guarantees that the correct algorithm will be used for the appropriate argument.

5.3.1 Type Determination

Having advocated a scheme whereby a particular unification algorithm is selected according to the argument's type, it is necessary to statically determine the type of argument to each such unification. The typechecking algorithm must therefore perform two tasks. The first of these is the standard typechecking procedure, which ensures statically that all possible operations within a program will be performed on objects of appropriate type. The second task is one of *type determination*. For the purposes of $\mathcal{L}$–HYBRID, type determination can be viewed as the static selection of the correct unification algorithm in all contexts where a unification by type occurs. By the same token as above, such a unification could not be specified over objects with polymorphic type, since the correct unification algorithm cannot be determined.

An ideal type determination algorithm is one which combines the advantages of dynamic typing (the types of objects are unspecified in unifications) with the advantages of static typing (type security, plus selection of the correct unification algorithm in all cases). Observe that, although a function such as the example above contains an overloaded operator applied to a polymorphic argument, the function is well defined as long as each instance (call) of the function attempts to apply the operation to a distinct monotype. The correct function denoted by '=' would be selected according to the types established at the call of the function, rather than those in its definition. An attempt to apply the operation to an inappropriate type of argument can be detected at a particular call to the function, since this is where the meaning attached to the overloaded operator is established[2]. In a similar manner, a clause definition which attempts to unify polymorphic objects can be considered well defined, since each instance of the clause within a goal establishes monotypes for the objects to be unified, and thus establishes the correct unification algorithm.

Although unification is a symmetrical operation, some asymmetries must be introduced into the typechecking algorithm. For example, the expression

```
query (clause nil) [lambda x.  x]
```

is well-defined, since the clause can perform a structural unification regardless of the type of objects in the argument list. However, the expression

```
query (clause [lambda x.  x]) nil
```

is invalid, since the clause cannot unify the literal value of its formal parameter against `nil`. This point is better illustrated by a declaration of the form

```
let clause BadClause [lambda x.  x];
```

Although the unification associated with this clause can be performed in the query

[2]Precautions need to be taken for lambda-bound functions, however (section 5.6).

```
    query BadClause nil
```

an attempt to activate BadClause with an instantiated list is clearly illegal, since unification over functions is not well defined. For this reason, non-unifyable literals (such a lambda-expressions) must be disallowed within formal parameters to clauses.

5.3.2 Methods

This chapter goes on to present a type determination algorithm which performs static typechecking according to the scheme outlined above. The type determination of arguments for unification in a clause definition is deferred, if necessary, until a particular invocation of the clause within a goal. Such a scheme is type secure, since any type determination which cannot be done when a clause is defined (since the relevant types are polymorphic) is done for every goal featuring the clause.

An implication of this scheme is that some unification information (an indication of the correct algorithm) must be passed to a clause for each invocation of the clause from a goal. This information reflects the instances of types which were polymorphic when the clause was defined, and will generally be different for each goal featuring the clause. This unification information will be termed a *method.* Any clause which defines unification over polymorphic entities will "expect" to be passed a method when it is invoked. Each goal featuring such a clause will pass a distinct method to the clause. This method comprises (informally) some identification of the correct unification algorithm or algorithms to be used for the (type of) argument being passed.

Note firstly that a method needs to be determinable from the type of a clause. A clause which requires a method may be re-bound to other identifiers or embedded within data structures; the only medium in which the method information can be conveyed is its type. Moreover there has to be a specific type semantics for dealing with such clauses, since the types of clauses may be propagated by type unification. The type semantics must deal with such eventualities as unifying the types of two clauses each of which requires a distinct method. For example in

```
query (if cond
       then clause₁
       else clause₂) arg
```

the resulting clause type must represent a method which is consistent for both of the constituent clauses (in effect, a "union" of the information required by each clause); a consequence of this is that a clause may be passed method information which it does not require.

5.3.3 Method Determination

A method can be seen to be dependent in some way on the type variables occurring within the type of a clause. A method which consisted of unification information for each type variable instanced in the goal would be sufficient for any unification performed when invoking that clause. However, this approach has two drawbacks:

- Method information will be lost if the types of clauses are unified together. For example, a clause with a resulting type α `Clause`, and which requires a method, can have its type unified with a clause of type `int Clause`, in which case the method information is lost;

- A free variable occurring within the type of a clause does not specifically mean that the the clause requires a method to unify the appropriate (part of the) argument. For example, a clause with the anonymous logical variable '`(_)`' occurring within its formal parameter does not imply that a unification needs to occur. However, the appropriate part of the argument type will still be polymorphic. Similarly, the object denoted by

  ```
  clause nil
  ```

has a polymorphic type α `List Clause`, although no type information is needed to perform the appropriate unification.

A method can be seen, therefore, to comprise type information for performing unification of the *values* of polymorphic formal parameters, as opposed to unification by their structure, as in the example above.

It now remains to determine which unifications within a clause head can be considered as value unifications.

5.3.4 Logical Variables

In operational terms, a logical variable occurring within a clause head is initially *free*, and becomes instantiated by unification to an appropriate part of the actual parameter. For the purposes of typechecking, therefore, logical variables behave in a similar manner to updatable references (section 5.9). If there is only one occurrence of a logical variable within the head of a clause, the outcome of that particular unification is predetermined (a binding of the logical variable to the argument value), and there is no need to perform any kind of unification. A unification of values need occur, therefore, only for a repeated logical variable.

Note that such unifications may be monomorphic. A clause such as

```
clause (x, x) :- (x? > 0)
```

requires no method because the type of the unification is known when the clause is defined. Therefore, only such unifications over polytypes need be considered.

It is chosen to ignore the case of polymorphic expressions within clause heads. Such expressions are infrequent, since most polymorphic items within a clause head contain constructors and therefore represent active unification. Any remaining polymorphic expressions are unlikely to have their types instanced within the goals. To conclude, method information is to be provided for logical variables occurring in the head of a clause which are repeated, and have polymorphic type.

5.4 Type Labelling

Some way is needed of representing, within the type of a clause, the fact that some parts of a formal parameter are unified by value. This is not necessary for monomorphic types, since the type of the unification is already determined when the clause is defined. It is however necessary for polymorphic types, since the type information will be passed within the method. Note that a clause may attempt to unify several objects with distinct free types within one goal, or one object containing a number of distinct polytypes. A method may therefore provide several unification algorithms for these distinct types.

A *type label* is defined to be a positive integer associated with a (possibly structured) type. A type t labelled with an integer n will be written t_n; for example, the type $\texttt{int}$ labelled with the value 3 will be written $\texttt{int}_3$. The process of *labelling* a type is to associate a type label (1, 2, ...) with every free type variable in that type.

The method required for a clause is determined by labelling some of the free types within the type of the clause. If a logical variable is repeated within the head of a clause, then any free types within the type of the logical variable are labelled. Note that no labelling will occur, therefore, if the type of the variable is monomorphic. The method required by the clause is therefore a set of unification algorithms, one for each labelled free type of the clause type. If a logical variable has several type variables in its type, then several algorithms from the method will be used to unify it. When a clause is invoked, it expects to be passed a method for unifying its repeated polymorphic logical variables.

5.4.1 Method Construction

A method is constructed for every (non-literal) goal which attempts to activate a clause with labelled type. The clause type will be instanced according to the argument type (in the normal process of typechecking), resulting in a set of type labels attached to instanced types. These types are used to generate the method. A method is a list of unification algorithms, one for each label in the (instanced) clause type. Since clause types are usually generic, a particular clause can be passed different methods in different goals.

For the purposes of typechecking, a method can be considered to be a list of numbered types, of the form

$$[n_1 = t_1;\ \ n_2 = t_2;\ \ \ldots]$$

Each type t_i is expected to be a type appropriate for unification. Each entry $t_i = n_i$ denotes a unification algorithm for objects of that type.

The process of constructing a method for a goal is as follows:

- For each labelled type t_n, remove the label n, and create a method entry $n = t$;
- Instantiate the clause type according to the type of the argument in the goal. This will have the effect of instancing the free types within the method.

As an example, consider the clause defined by the declaration

```
let clause eq(x, x)
```

The type initially assigned to the clause by the typechecking process will be

```
(α, α) Clause
```

The type of the repeated logical variable x will be labelled, resulting in a type for eq of `(`α_1`, `α_1`) Clause`.

For every goal which invokes eq, each argument, with a type of the form `(`t`, `t`)`, will give rise to a method $[1 = t]$ (figure 5–11).

```
::    query eq(nil, [1]);

   ▷     query
            eq (nil, [1])                          [1 = int List]


>  false :  bool
```

Figure 5–11: A Call to eq

5.4.2 Properties of Labelled Types

It is necessary to extend the semantics of polymorphic types to encompass labelled types. A labelled type inherits all the properties of the type itself, but is, in addition, subject to a set of rules governing the behaviour of the type according to its labels. Rules may be divided into *occurrence rules*, which govern the circumstances under which a labelled type may occur, and *unification rules*, which govern the way it behaves when manipulated by the typechecker.

5.4.3 Occurrence Rules

The occurrence rules for type labels are as follows:

- Labelled types must be generic (and non-generic types unlabelled);

- A valid type will have labels contained solely within the type operator `Clause`. This implies that the type $\texttt{int}_1$ `Clause` is a valid type, whereas $\texttt{int}_1$ is not. It should be noted that, by this criterion, methods may contain invalid types;

- A particular label may occur several times within a type, but each occurrence refers to the same subtype (one with the same structure and the same type variables);

```
::    lambda x. clause [a; a; lit x];

Type clash in:  [a; a; lit x]
Attempt to label non-generic type:  α(0) List
```

Figure 5–12: A Restriction on Type Labelling

- No labelled type may contain another type with the same label. This condition follows from the condition above, together with the occur-check which is performed on every type unification.

The first condition must be strictly enforced by the typechecker. Any attempt to label a non-generic type is considered to constitute a type fault, as is any attempt to unify a labelled type to a non-generic type (figure 5–12). The second condition always holds true since the only types to be labelled are the types of arguments to clauses. The labels cannot be propagated out of context since labelled types are guaranteed to be generic.

A label will be repeated if a free type, which occurs several times within a clause type, is labelled. In this case the labelled types are, trivially, identical. The unification rules for labelled types (section 5.4.4) guarantee that this condition is maintained.

A further occurrence rule for labelled types is that they may not occur within the type of an argument to a clause. This avoids confusion between the labels of a clause type and any labels occurring within its argument. This rule implies that, for example, the clause eq defined above cannot be passed as argument to another clause.

5.4.4 Unification Rules

There are a number of rules governing the behaviour of labelled types under type unification.

- Labels are propagated by unification. A type variable unified to a labelled type variable becomes labelled. Alternatively, an unlabelled type variable will always unify to a labelled one, and not vice versa;

- Differing labels will not unify. For example, an attempt to unify α_1 with β_2 will produce a type fault;

- Labelled free types may be instanced;

- When two types are unified, the types attached to any corresponding labels in the two types are unified as well.

As an example of the second rule, the clause `eq` may occur in a context which instances its type to `(int₁, int₁) Clause`. This property in itself implies a change in the way that monotypes behave. Whereas conventionally a type constant such as `int` represents the single type of all integers, it is now necessary to cater for the existence of an arbitrary number of distinct `int` types, some of which may adopt labels. Consider the following example:

```
::    let clause Both_Zero(0, 0)
      in  [Both_Zero; eq];

>   [ψ; ψ]  :  (int₁, int₁) Clause List
```

There are several occurrences of the type `int`, some of which acquire the label '1' through unification. Such unification must, however, not affect the original type derived for `Both_Zero`. In practice, this rule is implemented by completely copying the (generic) type of any identifier accessed by name (recall that non-generic types cannot acquire labels), and by giving each primitive type-generating construct in the language a completely new type structure.

The last rule maintains the condition that a label occurring several times within a type is attached to the same type. In the example of figure 5–13, the general type of $(\alpha + \beta)$ derived for both `Foo` and `Goo` is further unified due to the type labels present. This rule may cause unifications to cascade, since the

```
::    let clause Foo(in_lft x, in_lft x)
                 and Goo(in_rht y, in_rht y)
      in  [Foo; Goo];

>   [ψ; ψ] :  ((α₁ + α₁), (α₁ + α₁)) Clause List
```

Figure 5–13: Label Propagation and Unification

types attached to corresponding labels may themselves contain corresponding labels whose types must be unified.

5.5 Disjunctive Clauses

A clause may contain several alternative (or-parallel) branches. For the purposes of typechecking, each branch is considered a separate clause, and typechecked in isolation. Type labels are also assigned in isolation. The resultant clause type is determined by unifying the types of the individual branches (with reference to the unification rules which must be enforced for the unification of labelled types, as detailed above).

5.6 Non-Generic Clauses

It is permissible to have clauses passed as parameters to, and invoked from, other clauses or functions (figure 5–14). Such a clause cannot be passed a method within the goal, since the type of the clause is unlabelled (in fact, free) when the goal is analysed. However, an attempt might be made to pass a labelled clause (one requiring a method) as argument. Some way must be found, therefore, of ensuring that an attempt to activate a labelled clause passed as parameter, is faulted.

```
::     let Try c = query c x with x;

>   Try = λ :  (α Clause → α)
```

Figure 5–14: A Clause as Argument

```
::     let clause Eq_0 x :- eq(x, 0);

   ▷      clause Eq_0 x :-
             eq (x, 0)                          [1 = int]

>   Eq_0 = ψ :  int Clause
```

Figure 5–15: A Cascaded Method

The notation $t_{\oslash}$ Clause denotes the type of a clause which takes argument of type t, and which may not unify with a labelled clause type. The '$\oslash$' denotes the fact that the clause argument type may not acquire labels.

When a goal is analysed, the clause type is examined. If it is free and non-generic, then it is instanced to $t_{\oslash}$ Clause, where t is the type derived for the argument. The type $\alpha_{\oslash}$ Clause will unify with β Clause, in which case the "do-not-label" property is propagated. Any attempt to unify $T1_{\oslash}$ Clause with $T2$ Clause will fail if the type $T2$ contains labels.

5.7 Method Propagation

The type determination algorithm described allows clauses which perform unifications of polymorphic values to be called from goals which instance the types of their arguments. Such a goal may be contained in the body of another clause, in which case this clause is responsible for constructing the method for the goal (figure 5–15).

```
::    let clause Foo(x, y) :- eq(·[x], y : _);

   ▷    clause Foo (x, y) :-
           eq ([x], y :  _)                      [1 = α₁ List]

>   Foo = ψ :  (α₁, α₁) Clause
```

Figure 5–16: Label Propagation

This system has one major weakness. The argument type of the goal may still be polymorphic, and dependent on the argument type of the containing clause, as in the example of figure 5–16. In this case, the method derived for eq will contain an entry of the form $[1 = \alpha]$, where the α is contained in the type of Foo.

This weakness is circumvented by the following procedure:

- If a method for a goal within a clause contains free type variables, and those type variables occur within the type of the containing clause, then the free types are labelled.

In the example above, Foo acquires a labelled type, since it requires method information which can then be passed to eq (figure 5–16).

Note that the free types within the method must occur in the type of the containing clause, otherwise the method information cannot be propagated into the clause. In the example of figure 5–19, an attempt is being made to label the type of x. Since x is local to the clause, and does not appear in the head of the clause, the label attached to the type of x cannot be propagated to the type of the clause itself.

Note also that the method used for the inner goal may be of a completely different structure to that passed to the clause. In figure 5–20, the method information required for each call to eq depends on a structured type derived from free types of x and y. In this case, the method constructed for a goal is

```
::    let rec clause Foo(x, x) :- Foo(x, x);

   ▷     clause Foo (x, x) :-
            Foo (x, x)                             [RecCall]

>   Foo = ψ :  (α1, α1) Clause
```

Figure 5–17: Use of [*RecCall*]

not generated from a set of unification algorithms, but must instead be *derived* from the method passed to the activation of Foo.

There are therefore two criteria for labelling the type of a clause. A clause type is labelled according to repeated logical variables in the head, and according to free types in any of its contained methods.

5.8 Recursive Declarations

Recursive declarations featuring clauses need to be dealt with in a special way. A recursively declared clause may require its type to be labelled due to free types in methods occurring within it. However, such methods may be determined by the eventual labelled type of the clause. This circularity is overcome by adopting a convention that all the clauses defined by one recursive declaration have exactly the same method.

Goals which feature recursive clause calls are marked with a special method written [*RecCall*]. This is taken to mean that the clause being activated within the goal is a recursive call, and can be passed the same method as the clause containing the goal (figure 5–17).

Whereas clause types in non-recursive declarations are labelled clause by clause, a recursive declaration causes all the clauses in the declaration to be labelled in one process, according to repeated head variables and (non-recursive) goal methods. The types of all clauses in the declaration are determined in one pass, and labels are attached to these types in another pass. Since recursive calls have the special method [*RecCall*], these do not contribute to the labelling process. In the recursive declaration of figure 5–21, both clauses have the same method (implied by the types derived for the clause). All mutually recursive calls between the clauses use method [*RecCall*].

Because of the fact that recursive clauses are dealt with in a particular way, it is not permissible to have the recursively defined clauses appearing in goals in the recursive declaration, unless such goals appear immediately in a clause body. A declaration of the form

```
let rec F() = query C()
        and clause C() :- ...
```

is therefore illegal. The reason for this restriction is that the use of [*RecCall*] is only valid for (mutually) recursive clauses which directly invoke each other, and not for occurrences of clauses within arbitrary expressions.

5.9 Typechecking Logical Variables

One or two points should be noted in passing with respect to logical variables. A logical variable is treated (for typechecking purposes) as a binding of an identifier to an object of a particular type. The fact that logical variables have a peculiar behaviour (viz. they unify) is, for the purposes of typechecking, irrelevant. If a logical variable is uninstantiated, then it is treated by logic programs as an object of the appropriate type which can be unified, and is treated by functional programs as a particular kind of fault-value.

From a functional point of view, if a query expression featuring logical variables fails to unify all of the variables, then the query expression returns a value containing fault-values. The fact that these fault-values may have polymorphic type is irrelevant. However, since a logical variable can be instantiated in a different deductive system to the one in which it is defined, a variable with polymorphic

```
::    let query (clause(_)) x with x;

>   x = _ :  α

::    query (clause 3) x;

>   true :  bool

::    x;

>   3 :  int
```

Figure 5–18: Type Instantiation for a Logical Variable

```
::    let clause Bad() :- eq(x, x) with x;

Type clash in:  (x, x)
Purely local labelled type within clause:
Clause type is:  triv
Local type is:  α
```

Figure 5–19: An Attempt to Label a Local Type

type might acquire a value of monomorphic type by means of a subsequent deduction, as in the example of figure 5–18. In this example, the type of the logical variable becomes monomorphic to reflect the fact that its value has been instantiated by a second query. In general, a deduction will alter (instance) the types of any global logical variables it references.

This behaviour of logical variables corresponds to the behaviour of objects with reference-types in ML. The typechecking of such objects has been dealt with elsewhere [Damas 85], and is considered beyond the scope of this thesis.

```
::    let clause Foo(x, y) :-
         eq(in_lft x, in_rht x),
         eq([y], [y]);

   ▷     clause Foo (x, y) :-
            eq (in_lft x, in_rht x),                [1 = (α₁ + α₁)]
            eq ([y], [y])                           [1 = α₂ List]

>   Foo = ψ :  (α₁, β₂) Clause
```

Figure 5–20: A Derived Method

```
::    let rec clause  All_Same(x : x : x') :-
                         All_Same(x : x')    |
                      All_Same [_]    |
                      All_Same nil
                      and All_Lists_Same(x : x') :-
                            All_Same x,
                            All_Lists_Same x'    |
                         All_Lists_Same nil;

   ▷     clause All_Same (x :  x :  x') :-
            All_Same (x :  x')                       [RecCall]
       | All_Same [_]
       | All_Same nil

   ▷     clause All_Lists_Same (x :  x') :-
            All_Same x,                              [RecCall]
            All_Lists_Same x'                        [RecCall]
       | All_Lists_Same nil

>   All_Same = ψ :  α₁ List Clause
>+  All_Lists_Same = ψ :  α₁ List List Clause
```

Figure 5–21: A Mutually Recursive Declaration

Chapter 6

Concurrent Implementation of $\mathcal{L}$–HYBRID

6.1 Introduction

This chapter describes a compilation technique for $\mathcal{L}$–HYBRID. Programs written in $\mathcal{L}$–HYBRID are compiled into code for an abstract machine. The $\mathcal{L}$–HYBRID abstract machine directly supports the nondeterministic properties of the computational model, and contains the $\mathcal{F}$–HYBRID abstract machine. The $\mathcal{L}$–HYBRID machine can therefore execute $\mathcal{F}$–HYBRID programs.

Clauses and queries are compiled into sequences of simple instructions which implement unification and process reduction. The $\mathcal{L}$–HYBRID computational model is partially supported by sequences of instructions which implement computational primitives (for example, unification), and partially by means of a small number of extensions to the $\mathcal{F}$–HYBRID machine to support logical variables and the computational model. In addition, additional special-purpose registers, and a set of "high-level" instructions (representable in terms of lower-level instructions), implement the synchronisation primitives needed by the logic computational model. The extensions to the $\mathcal{F}$–HYBRID machine have been kept deliberately simple. There is no general unification instruction, but instead a small number of instructions to perform atomic unification actions. The model of nondeterministic process reduction is not supported directly by the abstract machine, but by means of code sequences which make use of synchronisation instructions.

6.2 Synchronisation Instructions

Processes in a logical system communicate in one of two ways. The first is explicit communication by means of shared logical variables; a unification performed on a variable by one process will be visible to any other process which has access to that variable. The second communication mechanism is implicit, and is concerned with the management of process reduction within a query.

Two forms of implicit communication may be identified. *And-communication* is defined as the passing of a *success* or *failure* condition to other processes in the same conjunctive (and-parallel) system, and *or-communication* as the passing of *success* or *failure* to processes in the same disjunctive (or-parallel) system.

Every $\mathcal{L}$–HYBRID process belongs to a conjunctive system, since a goal can only occur within the body of a clause, or within the outermost level of a query term or declaration. A process may, however, be the only member of a conjunctive system.

An $\mathcal{L}$–HYBRID process may also be a member of a disjunctive system. If a clause has several alternative branches, then the activation of that clause will result in the creation of a number of processes, one for each alternative branch. Each process is a member of the disjunctive system created for this clause, as well as being a member of the conjunctive context in which the clause was originally activated. A clause with only a single branch can be viewed as a disjunctive system with only one member, but in this case optimisation results in the disjunctive system being dispensed with.

An $\mathcal{L}$–HYBRID process may communicate *success* or *failure* to other processes in the same conjunctive system, by means of the instructions AndSuccess and

AndFailure. Both instructions cause termination of the process. Communication of *success* and *failure* with other disjunctive processes is done by means of the instructions OrSuccess and OrFailure. OrSuccess represents an attempt by a disjunctive process to *commit*; the process will terminate if any other process in the system has already committed. OrFailure represents the failure of a disjunctive process. If no other disjunctive processes exist (ie. have already failed), then failure must be communicated to the enclosing conjunctive system, since this must also fail.

In addition, there is an Otherwise instruction corresponding to the otherwise-form of a guard. A process which executes Otherwise will terminate if any other process in the same disjunctive system performs an OrSuccess (i.e. *commits*), and will continue execution if every other process in the system performs an OrFailure.

The instructions AndSuccess, AndFailure, OrSuccess, OrFailure, and Otherwise, although part of the $\mathcal{L}$–HYBRID machine instruction set, may be decomposed into sequences of other instructions. They are provided as instructions for the following reasons:

- The profiler (chapter 3) makes assumptions about the nature of the abstract code which are relaxed for terminating instructions (section 2.3.2). A single instruction such as AndSuccess is accepted by the profiler, whereas the equivalent sequence of simpler instructions would not be;

- Because abstract machine instructions are defined to be atomic (section D.2), direct implementation of the above instructions in the abstract machine allows critical regions within them to be implicitly respected.

Figure 6–1 presents the interpretation attached to the synchronisation instructions. Regions grouped by a vertical bar (" | ") are considered to be critical.

Associated with every executing conjunctive system is an *and*-cell, which is a block of storage allocated from the heap. The instruction

```
AndSuccess ≡ decrement And/0;
             if And/0 = 0 then And/1 := true;
             Stop;

AndFailure ≡ And/1 := false;
             Stop;

OrSuccess ≡  B := Or/1;
             Or/1 := true;
             Or/2 := false;
             if B then Stop;

OrFailure ≡  decrement Or/0;
             if Or/0 = 1 then Or/2 := true
             else if Or/0 = 0 then AndFailure;
             Stop;

Otherwise ≡  if not Or/2 then OrFailure;
```

Figure 6–1: "High-level" Synchronisation Instructions

`AndCell` *n*

allocates such a block from the heap, and places a pointer to it on the stack. The block has two elements, the first having integer value *n*, and the second being *empty*. Before a conjunctive system is created, the pointer to the *and*-cell is placed in the register R_And; the register value will then be propagated to all members of the conjunction.

The first element of the *and*-cell represents the number of processes initially in the conjunction; the second element is assigned the outcome of the entire conjunction; `true` if successful, `false` if unsuccessful.

The effect of `AndSuccess` is to remove the process from the conjunctive system. Decrementing `And/0` updates the number of processes active. If no more processes are active, then the conjunctive system is deemed to have succeeded, and `And/1` is assigned `true`. `AndFailure` assigns `And/1` to `false`, failing the entire conjunction, and terminates. Since `And/0` is not decremented, no other successful process will decrement it to zero and overwrite `And/1`.

Every set of disjunctive processes has an associated *or*-cell, which is a block of storage allocated in the same way as for the *and*-cell. The instruction

```
OrCell n
```

allocates a cell from the heap, placing a pointer on the stack. An *or*-cell comprises three elements. The first element is the number of disjunctive processes, *n*. The second element, initially `false`, is used for the *commit* operation. The third is used to implement `otherwise`. The instruction `OrCell` is generally followed by an assignment of the (address of the) block allocated to the register `R_Or`.

Of all the processes in a disjunctive system, only one is permitted to *commit*. The first process which attempts to commit will succeed, and will then reduce itself to a conjunctive system (the tail of the clause). Any other process in the same system which attempts to commit will fail, and terminate.

The meanings of `OrSuccess` and `OrFailure` are shown in figure 6–1. `OrSuccess` attempts to commit by assigning `Or/1` to be `true`; if the previous value was already `true`, then the process terminates. `OrFailure` decrements `Or/0`; if the value becomes zero, then the disjunctive system is empty, and the process performs an `AndFailure`, thus failing the containing *conjunctive* system.

The instruction `Otherwise` will allow a process to proceed if it is the last process in the disjunctive system, and will terminate the process otherwise. If `Or/2` becomes `true`, then all other disjunctive processes have failed, and the `Otherwise` succeeds. if `Or/2` becomes `false`, then another disjunctive process has succeeded, and the `Otherwise` must fail.

6.3 Compilation of Clauses and Queries

A clause is implemented as a special kind of function. When a clause is activated, it is passed a single argument, which may be a value corresponding to an $\mathcal{F}$–HYBRID expression, or a structure containing logical variables. The clause does not return a result, but either *succeeds* or *fails*, depending on the outcome of the deduction it denotes. Any values computed are communicated to the calling context by means of logical variables.

A clause is compiled into a closure identical in structure to that of an $\mathcal{F}$–HYBRID function. The closure contains the code of the clause, together with the size of stack required for its activation, and a set of values denoting the free variables of the clause. The clause is activated as a process, with argument passed in register R_Arg.

The outcome of the query is communicated to the calling context by means of success and failure instructions, rather than by use of the result register R_Result. The success and failure instructions are responsible for maintaining the state of the particular deductive system in the *and-* and *or*-cells allocated by the calling process; use of R_Result would imply the creation of a dedicated process (with dedicated code) to monitor each deductive system, and the use of a complex result type to communicate the actions necessary in the reduction model.

Figure 6–2 illustrates the code generated for two very simple clauses. The first always succeeds regardless of its argument, and the second always fails. The instructions AndSuccess and AndFailure are used to communicate *success*, or *failure*, of the clause to the calling context. Both instructions cause the termination of the process executing them. In general, a unit clause will attempt to unify its argument (passed in R_Arg) according to the formal parameter. If the unification succeeds, the process executing the clause will perform an AndSuccess; otherwise, control will pass to the AndFailure instruction.

```
::    let clause Always(_)
                 and Never(_) :- false;

L11:InflateEnv 2;
L5: Code 2 [                              Code for Always
       L4: AndSuccess;                    Succeeds unconditionally
       L3: AndFailure;                    Failure label (never reached)
    ];
    NullClosure Local/0;
    Move Result/1;
L10:Code 3 [                              Code for Never
       L9: AndFailure;                    Fails unconditionally
       L8: AndFailure;                    Failure label (never reached)
    ];
    NullClosure Local/0;
    Move Result/2;
    Stop;

>   Always = ψ :  α Clause
>+  Never = ψ :  β Clause
```

Figure 6–2: Use of AndSuccess

When they occur as clause bodies, the literals `true` and `false` are treated as special cases by the compiler, and cause generation of `AndSuccess` and `AndFailure` instructions as appropriate. A clause body which contains a number of goals is compiled into a series of calls to the appropriate clauses. A simple clause with a single alternative branch (and no guard) has the form

`clause` *formals* `:-` G_1`,` ... `,` G_n

This clause will immediately perform an `AndFailure` if it cannot unify its actual argument with its formal parameter. Otherwise, it will *reduce* to the goals $G_1, \ldots, G_n$ (recall from section 6.2 that every executing goal belongs to a conjunctive system). In order to perform reduction, a goal *replaces* itself in the conjunctive system by the goals comprising its body, as described below.

6.3.1 Entry Sequences

A clause activation is similar to a function activation (section 2.4.3). The closure representing the clause to be activated is placed in `R_Func`, and the argument to be passed is placed in `R_Arg`. The activation of the tail of the clause consists of a number of clause activations performed one after another; each activation but the last is performed by a `Process` instruction, and the last is done by `TailApply`.

By definition, a *read-only* logical variable cannot be instantiated by the goal in which it appears with the read-only annotation. This condition is implemented by requiring that the process accessing the variable be suspended until the variable is instantiated by some other process (section 6.3.4).

In practice, it is difficult to ensure this read-only condition. A clause may attempt to pass an uninstantiated variable to several processes; in this case, each process must somehow be forbidden to unify the variable. A simple, if slightly restrictive, technique is chosen to circumvent this problem. The compiler makes sure that the argument to a goal is assembled by the goal process itself, via an *entry sequence*; the entry sequence causes the goal process to suspend, if necessary, until all read-only variables have been instantiated. This sequence is executed directly by the goal process, and contains a tail-recursive call to the clause itself. The entry sequence is an anonymous procedure which contains:

- the code to evaluate the expression representing the clause;
- a check for the clause expression being a fault-value;
- the code to assemble an argument to the clause;
- assignment of clause to R_Func and argument to R_Arg;
- a tail-recursive call to activate the clause.

The process created for each goal begins by executing this entry sequence; if any read-only variables are uninstantiated, the process will suspend at this stage. The clause code is activated by the `TailApply` within the entry sequence.

The closure for the entry sequence contains the values of identifiers in the clause expression and in the argument. Uninstantiated logical variables may be compiled into the closure without regard for any read-only annotation attached to them; such annotations are respected by the goal process when the entry sequence is activated.

New processes for goals are created by the `Process` instruction. `Process` behaves like the instructions `PushProcess` and `VecProcess` used by the $\mathcal{F}$–HYBRID compiler (chapter 2), but is a degenerate form of these, in that the `R_Result` register is not assigned any particular value. However, propagation of the synchronisation registers `R_And` and `R_Or` is guaranteed. A parent can therefore pass a reference to an *and*-cell to its children in `R_And`, all of whom will inherit a pointer to the same cell.

Figure 6–3 illustrates the code generated for a clause body comprising two goals. Each goal is conceptually executed by a newly created process; however, optimisation results in the final goal in the tail of a clause being activated by tail-recursion. Because `TwoCalls` reduces to two goals, the size of the conjunctive system in which the call to `TwoCalls` occurs will increase by one when `TwoCalls` reduces. The correct size is maintained in `And/O` by means of the instruction

```
Increase dest, n
```

which is defined to atomically add the value *n* to the cell referenced by *dest*.

A *unit clause* (a clause with no body) will succeed immediately upon unification. A *literal goal* (a boolean expression as goal) will succeed if the expression evaluates to `true`, and fail otherwise. The example clause (figure 6–4) contains an entry sequence which examines the value of the expression `(0 > 1)`; the `TestFault` instruction causes failure if the literal goal evaluates to a fault-value. A literal goal must also be activated from an entry sequence, to deal with read-only logical variables appearing in the literal expression itself.

A disjunction is denoted by a clause with several alternative branches, of the form

```
clause formals1 :- ...   |
       formals2 :- ...   |
       ...
```

The clause will fail if no unification to the formal parameters succeeds (ie. if all alternatives fail). Otherwise, *one* of the successful alternatives is nondeterministically chosen to reduce to a conjunction (or to succeed, if it is a unit clause). Each alternative branch will attempt to unify the actual parameter against its formal parameter, but only one will be permitted to succeed and reduce to the tail system.

Figure 6–5 illustrates a clause with two alternate branches. The code generated for the clause is divided into two sections, one for each branch. A `Fork` instruction is used to create processes for all but one of the alternate branches (the remaining branch being executed by the calling process).

`Fork` creates a new process which begins execution at a specified label. The newly created process has a new area of stack referenced by `R_Local` (with size

```
::    let clause TwoCalls(_) :- Always 3, Never nil;

L12:InflateEnv 1;
L11:Code 6 [
       L10:Triv;
           FromReg R_Arg;                             Save old
           FromReg R_Method;                          register values
           Increase And/O, 1;                         Increase size of conjunction
       L9: Code 2 [                                   Entry sequence for Always
               L8: From Global/0;                     Clause passed here
                   TestFault 0, L7;                   Fail if it's a fault
                   Int 3;                             Argument
                   MoveReg R_Arg;
                   MoveReg R_Func;
                   TailApply;                         Actual call to clause
               L7: AndFailure;
           ];
           |
           |                                          Closure for entry sequence;
           |                                          Always is passed here
           |
           MoveReg R_Func;
           Process;                                   First Goal
       L6: Code 2 [                                   Entry sequence for Never
                  |
                  |
                  |
           ];
           |
           |                                          Closure for entry sequence,
           |                                          Incorporates Never
           |
           MoveReg R_Func;
           TailApply;                                 Activation of second goal
       L3: AndFailure;                                Failure label for TwoCalls
    ];
    |
    |                                                 Closure for TwoCalls
    |
    Move Result/1;
    Stop;

>   TwoCalls = ψ :  α Clause
```

Figure 6–3: Clause Body Comprising Two Goals

```
::    let clause GivesFalse(_) :- (0 > 1);

L9: InflateEnv 1;
L8: Code 5 [                                  Code for GivesFalse
       L7: Triv;
           FromReg R_Arg;
           FromReg R_Method;
       L6: Code 2 [                           Entry sequence for (0 > 1)
              L5: Int 0;
                  Int 1;
                  GT;                         0 > 1?
                  TestFault 0, L4;            Fail if result is a fault
                  FalseJump L4;               Fail if result is false
                  AndSuccess;
              L4: AndFailure;                 Failure label
           ];
           NullClosure Local/0;
           MoveReg R_Func;
           TailApply;
       L3: AndFailure;
    ];
    NullClosure Local/0;
    Move Result/1;
    Stop;

>   GivesFalse = ψ :  α Clause
```

Figure 6–4: A Literal Goal

```
::    let clause AlwaysOneOrTwo(_) :- Always 1    |
                 AlwaysOneOrTwo(_) :- Always 2;

L12:InflateEnv 1;
L11:Code 6 [
      L10:OrCell 2;                                Two disjunctive processes
          MoveReg R_Or;
          Fork 6, L5;                              Process for second branch
          Triv;
          FromReg R_Arg;
          OrSuccess;                               Commit for first branch
          FromReg R_Method;
      L9: Code 2 [
                 |
                 |                                 Entry sequence for Always 1
                 |
          ];
          |
          |                                        Closure for entry sequence
          |
          MoveReg R_Func;
          TailApply;                               Call of Always 1
      L6: OrFailure;                               First branch failure label
      L5: Triv;                                    Second process starts here
          FromReg R_Arg;
          OrSuccess;                               Commit for second branch
          FromReg R_Method;
      L4: Code 2 [
                 |
                 |                                 Entry sequence for Always 2
                 |
          ];
          |
          |                                        Closure for entry sequence
          |
          MoveReg R_Func;
          TailApply;                               Call of Always 2
      L1: OrFailure;                               Second branch failure label
   ];
   |
   |                                               Closure
   |
   Move Result/1;
   Stop;

>  AlwaysOneOrTwo = ψ :  α Clause
```

Figure 6–5: Simple Disjunction

as specified in the Fork instruction), but all other registers (except the program counter) are inherited.

The most general form of a clause body consists of two sets of goals, separated by the commit symbol, '\':

```
clause formals :- G1, ..., Gn
                  \ T1, ..., Tm
```

If there is no explicit commit symbol ('\'), then the guard is assumed to be null. In this case, a process attempts to commit when it has completed its head unification.

The goals G_1 to G_n constitute the *guard*, and the goals T_1 to T_m the *tail*, of the clause body. Given a set of disjunctive clause alternatives, each will attempt head unification. If this is successful, the guard will be activated as a separate conjunctive system. Only if the guard succeeds will the process attempt to *commit* (exclude other disjunctive processes from reduction), and then reduce to the tail. Only one process will be permitted to commit successfully; any other process attempting the commit will immediately terminate. Recall from section 4.2 that unification is not guarded; redundant processes may still perform unifications that are visible to the query as a whole.

Figure 6–6 illustrates the code generated for a guarded clause. The guard is executed as a separate conjunctive system, with a newly allocated *and*-cell. The old value of R_And is saved on the stack, and restored after the guard has completed, so that the execution of the clause can reduce to a conjunctive system executing the clause tail (should the clause commit successfully).

6.3.2 Storage of Logical Variables

The local logical variables of a clause are allocated within a block of store claimed from the heap by a Block instruction. A separate block is allocated for each alternative branch of a clause, since the local logical variables are distinct.

```
::    let clause Guarded(_) :- Always 4 \ true    |
                 Guarded(_) :- Always 3 \ true;

L14:InflateEnv 1;
L13:Code 7 [
       L12:OrCell 2;                                   Two branches
           MoveReg R_Or;
           Fork 7, L7;                                 Process for second branch
           Triv;
           FromReg R_Arg;                              Save argument
           FromReg R_And;                              Save old and-cell
           AndCell 1;                                  New and-cell
                                                       for guard conjunctive system
           MoveReg R_And;
           FromReg R_Method;
       L11:Code 2 [
                  |
                  |                                    Entry sequence for Always 4
                  |
           ];
           |
           |                                           Closure for entry sequence
           |
           MoveReg R_Func;
           Process;
           From And/1;                                 Await result of guard
           From Local/2;                               Restore old and-cell
           MoveReg R_And;
           Deflate 2, 1;
           FalseJump L8;                               Fail if the guard failed
           OrSuccess;                                  Commit
           AndSuccess;                                 Clause tail: true
       L8: OrFailure;                                  Failure label
       L7: |
           |
           |                                           Second branch of clause
           |
           |
    ];
    |
    |                                                  Closure
    |
    Move Result/1;
    Stop;

>   Guarded = ψ :  α Clause
```

Figure 6–6: A Guarded Clause

The elements of a piece of store allocated by `Block` are initially *empty* (section 2.4.2). The `Uninstance` instruction is used to assign the elements a special value associated with an uninstantiated logical variable (section 6.3.4).

Uninstantiated logical variables are treated distinctly from store locations which are *empty*. An attempt by a process to copy an *empty* store location will immediately suspend the process; by contrast, an attempt to copy an uninstantiated logical variable will cause a *reference* (invisible pointer) to be created to the variable (section 6.3.4).

A parameter to a clause is decomposed in a similar manner to the decomposition of a parameter to a function, but the decomposition is done by unification. Argument decomposition results in unification to the logical variables referenced within the head of the clause.

Figure 6–8 illustrates the allocation of local logical variables. The logical variable x is declared implicitly in the head of the clause, and the variable y is declared explicitly by means of `with`. The `Block` instruction allocates space for both x and y.

The code for `XAndY` illustrates an important optimisation. Since the formal parameter variables in the head of a clause are initially undefined, the first unification to such a variable can be replaced by a simple assignment, with the same effect. If a variable only occurs once in the head of a clause, it is not necessary to unify to it at all.

The same local variable space is used for goals whose actual parameter is a single anonymous logical variable. Since the *uninstantiated* value cannot be passed in a register, and cannot reside on the caller's stack (because of the behaviour of logical variables (section 6.3.4)), the argument must be a reference to a variable allocated in a block on the heap. Figure 6–9 illustrates the allocation of these variables. A block of three locations is allocated. The first of these is reserved for x, and the other two locations are used for the anonymous variables passed to each call of `Always`.

6.3.3 Query Expressions and Query Declarations

A query expression may have one of two forms. An expression of the form

query G_1, ..., G_n

creates a conjunctive system for the evaluation of the goals G_1 to G_n, and returns **true** for a successful outcome, and **false** otherwise. Such a result is directly available from the *and*-cell created for the system; And/1 will be assigned a boolean value corresponding to the outcome of the query. The query expression need only return the value of And/1, after having tidied up the stack. The query expression of figure 6–10 activates a process for each goal in the query. The sequence of instructions

```
From And/1;
Result;
Stop;
```

causes the outermost process to await, and return, the outcome of the query.

An expression of the form

query G_1, ..., G_n with id_1, ..., id_m

creates a conjunctive system for G_1 to G_n, using logical variables id_1 to id_m. If the query succeeds, then the value of the query expression is the tuple

(id_1, ..., id_m)

If the query fails, the value of the expression is the fault-value

(fault : {infer})

The query term as a whole is executed by a new process. This means that an expression of the form

((query $goals_1$), ..., (query $goals_n$))

will execute as a concurrent set of processes, one for each query term. Execution by one process would mean that the process would await the result of each $goals_i$ before creating the goal system $goals_{i+1}$, thus restricting parallelism.

Query declarations are dealt with in a similar manner. If the query system succeeds, then the values unified to the logical variables in the declaration are made available in the scope of the declaration. If the system fails, then fault-values are generated.

A local query declaration establishes bindings within the scope of an expression. Space is allocated to the logical variables using a `Block` instruction in the same way as for functional local declarations, but the variables are made *uninstantiated* before the query is initiated. If the query succeeds, then the variable bindings made are available in the enclosed expression. Within the expression, the values of the variables are accessed in the conventional manner using `From`, their logical nature being irrelevant. If the query fails, then a fault-value is generated. Failed query declarations generate the same fault as failed functional declarations; the fault-value

```
    (fault :  {decl})
```

is returned as the result of the expression. In figure 6–11 a process is created for each goal in the query. The instruction `From And/1` causes the outermost process to await the result of the query. If the query succeeds, control passes to label `L5`, to calculate `x + 1`. In the query fails, control passes to `L1`, causing the generation of a fault-value.

Top-level query declarations store the logical variables in the top-level environment. These locations are available to the outermost process as offsets from `R_Result`; the cells are made *uninstantiated* before the query begins execution. Other processes in the system are passed references to these cells within closures. If the query succeeds, then the values unified to the variables are treated subsequently as conventional values. If the query fails, then fault-values are generated for the variables. In the example of figure 6–12, success of the query causes control to pass to L4. Failure causes a fault-value to be generated and copied to each of the store locations allocated for x, y and z.

Top-level query declarations may cause overwriting of the values of memory cells with new values; in this respect, the assignment convention laid down for the $\mathcal{F}$–HYBRID interpreter (section 2.3.1) is violated. Overwriting occurs when some unification has been performed on a top-level variable, and the query system subsequently fails; all top-level variables must be overwritten with a fault-value.

The overwriting operation is considered safe since the only active processes are those within the query system, which has just indicated failure; the outcomes of their computations will be discarded. Also, none of the remaining processes can interfere with the fault-values assigned to the variables.

For local query declarations of the form

```
let query goal1, ..., goaln
    with id1, ..., idn
in expr
```

failure of the query causes a fault-value to be returned; evaluation of *expr* is not attempted. The block allocated for the logical variables may therefore be immediately discarded, without any overwriting taking place.

6.3.4 Logical Variables

Logical variables behave in much the same manner as their counterparts in PROLOG. A logical variable is initially undefined (or *uninstantiated*), and is manipulated by the process of unification, which causes it to be instantiated to some data value. This value may itself contain uninstantiated values corresponding to other logical variables. The declaration of a logical variable simply serves to associate a name with a data object which behaves in a particular manner.

A representation is adopted for logical variables which does not make use of structure sharing [Warren 77]. Since PROLOG programs manipulate data objects purely by unification, structure sharing need only be supported in the unification algorithm. However, HYBRID programs treat objects as values which are manipulated functionally, by a number of primitive operations. Structure sharing would therefore have to be supported by every primitive operation.

As a result of a non-structure sharing approach, structures in the head of a clause are built for each clause activation, since each activation will have a distinct set of logical variables (section 6.4).

Logical variables are memory locations in the global address space. The creation of a logical variable is viewed as the binding of a name to an object in the global address space. No portion of the address space is allocated specially for logical variables; any memory location may be used as a logical variable, and the unification operations described below (section 6.4) are also well defined if performed over simple data values.

A logical variable is created at a particular location by an `Uninstance` instruction. The cell referred to by the instruction is assigned a special *uninstantiated* value. This value is distinct from any data value, any fault-value, and the value used for *empty*.

When an attempt is made to copy the *uninstantiated* value from one location to another, the destination location is assigned a second kind of value. A *reference* value is an invisible pointer to another cell, created and manipulated implicitly by the abstract machine. Any attempt to copy an *uninstantiated* value to another cell results in the second cell being assigned a *reference* to the first cell. An attempt to access a *reference*-cell results in access to the cell it references. Since *reference*-cells are manipulated implicitly by every machine instruction, a reference to a data value always appears as the data value itself, the dereferencing operation being performed automatically.

It might be thought that an invisible pointer scheme could be used to prevent processing bottlenecks which occur when processes access the *empty* value. However, such a scheme is rejected for the following reasons:

1. The mechanics of *empty* are assumed to be provided by the abstract machine at a very low level, and are used extensively throughout the imple-

mentation of $\mathcal{F}$–HYBRID and $\mathcal{L}$–HYBRID. The use of an invisible pointer scheme for *empty* would impose considerable overheads in store management. By contrast, the use of *reference*-cells is restricted to the propagation of uninstantiated logical variables within $\mathcal{L}$–HYBRID;

2. The use of invisible pointers for *empty* would invalidate any kind of synchronisation and control of the processes in a computation, since any process accessing *empty* would be given an invisible pointer and immediately continue execution. This would invalidate all of the concurrency control techniques derived in chapter 3, and result in the execution of a program being swamped by processes creating long chains of references to *empty* cells.

It is possible, during unification, to generate long chains of *reference*-cells. A long chain of reference cells terminating in a data value is equivalent to several copies of the data value. A chain of reference cells terminating in an uninstantiated value is equivalent to several references to that value. Instructions such as `IfInstanced` (section 6.4.2) assume that such chains will be shortened whenever a reference cell in the chain is accessed. If the end of a reference chain is a data value, then the reference cell is directly overwritten with the data value itself. If the end of a reference chain is an uninstantiated cell, then the reference is transformed to be a direct reference to the uninstantiated cell.

The behaviour of a logical variable cannot be explained in purely functional terms, since the unification process has no functional counterpart. However, logical variables (or the values contained within them) may be accessible to a functional part of a program. In this case, a logical variable is treated as a binding of a name to an object with a rather strange behaviour.

In functional terms, a free variable is treated as the fault-value

```
(fault : {uninst})
```

```
::    let query (clause (_)) x with x;

>   x = _ :  α

::    x + 1;

>   (fault :  {prim "+"}, {uninst}) :  int

::    x;

>   _ :  int
```

Figure 6–7: Uninstantiated Variables in a Functional Context

where {`uninst`} represents *uninstantiated*. As with other fault-values, it may be passed unaltered as argument and embedded in data structures, and will cause fault propagation if an attempt is made to access its value (figure 6–7).

Within queries, it is often necessary to mark logical variables as read-only. In operational terms, this means that the process making a read-only access to the variable is suspended until the variable is instantiated by another process. Since goals may contain functional expressions (either as parts of arguments, or by virtue of being literal goals), functional parts of programs may be invoked with logical variables which have still to be instantiated by another process in the query. As mentioned above, the values of such variables would just be treated as fault-values.

In addition, any identifier within a functional expression may be marked as read-only. Any process which attempts to evaluate this expression as part of a goal will be suspended until the variable has been instantiated by another process.

A read-only variable is retrieved by the instruction `Read`. `Read` behaves exactly as `From` if the cell is a (reference to a) data or fault-value; if the cell is *empty*, then the accessing process is suspended. However, if the cell is *uninstan-*

```
::    let clause XAndY(x) :- Always(x, y) with y;

L7: InflateEnv 1;
L6: Code 5 [
       L5: Block 2;
           Uninstance Local/0/0;                   Allocated for y
           Uninstance Local/0/1;                   Allocated for x
           FromReg R_Arg;                          Retrieve argument
           Move Local/1/0;                         Store to x
           FromReg R_Method;
       L4: Code 3 [                                Entry sequence
                  |
                  |
                  |
           ];
            |
            |                                      Closure for entry sequence
            |
           MoveReg R_Func;
           TailApply;
       L1: AndFailure;
    ];
     |
     |                                             Closure for XAndY
     |
    Move Result/1;
    Stop;

>   XAndY = ψ :   α Clause
```

Figure 6–8: Allocation of Local Logical Variables

tiated, the process also goes into a wait-state, to be resumed when the cell is instantiated by another process.

```
::    let clause Anons x :- Always(_), Always(_);

L13:InflateEnv 1;
L12:Code 5 [
      L11:Block 3;
          Uninstance Local/0/0;                    Allocated for x
          Uninstance Local/0/1;                    Two anonymous
          Uninstance Local/0/2;                    variables
          FromReg R_Arg;
          Move Local/1/0;                          Store x
          FromReg R_Method;
          Increase And/0, 2;                       Increase size of conjunction
      L10:Code 2 [                                 First entry sequence
                 |
                 |
                 |
          ];
          |
          |                                        Closure picks up 1st
          |                                        anonymous variable
          |
          MoveReg R_Func;
          Process;
          |
          |                                        Second entry sequence...
          |
          MoveReg R_Func;
          TailApply;
      L1: AndFailure;
   ];
   |
   |                                               Closure
   |
   Move Result/1;
   Stop;

>  Anons = ψ :  α Clause
```

Figure 6–9: Allocation of Anonymous Variables

```
::    query Always 1, Always false, Always nil;

L11:Code 4 [                                        The query term...
       L10:AndCell 3;                               Three goals
           MoveReg R_And;
           Triv;
       L3: Code 2 [                                 First goal
              L2: From Global/0;
                  TestFault 0, L1;
                  Int 1;
                  MoveReg R_Arg;
                  MoveReg R_Func;
                  TailApply;
              L1: AndFailure;
           ];
            |
            |
            |
           MoveReg R_Func;
           Process;
            |
            |                                       Second goal
            |
           MoveReg R_Func;
           Process;
            |
            |                                       Third goal
            |
           MoveReg R_Func;
           Process;
           Fail 1;
           From And/1;                              The outcome of the query
           Result;
           Stop;
    ];
     |
     |                                              Closure
     |
    TailApply;

>   true  :  bool
```

Figure 6–10: A Query Without Variables

```
::    let query Never x with x in x + 1;

L6: Block 1;
    FromReg R_And;
    AndCell 1;
    MoveReg R_And;
    Uninstance Local/1/0;                          Uninstantiate x
    Triv;
L4: Code 2 [
              |
              |                                    Entry sequence for Never
              |
    ];
    |
    |                                              Closure
    |
    MoveReg R_Func;
    Process;
    Fall 1;
    From And/1;
    TrueJump L5;                                   Query succeeds
    MoveReg R_And;
    Fall 1;
    Jump L1;
L5: MoveReg R_And;
    From Local/0/0;
    Int 1;
    Plus;                                          Calculate x + 1
    Result;
    Stop;
L1: Nil;
    Fault;
    Triv;
    ChainFault 3;                                  Construct (fault : {decl})
    Result;
    Stop;

>   (fault : {decl}) : int
```

Figure 6–11: Failure of a Local Query Declaration

```
::    let query Never(x, y, z) with x, y, z;

L5: InflateEnv 3;                                    Allocate x, y, z
    FromReg R_And;                                   Save and-cell
    AndCell 1;
    MoveReg R_And;                                   Set up new and-cell
    Uninstance Result/1;
    Uninstance Result/2;
    Uninstance Result/3;
    Triv;
L3: Code 3 [
              |
              |                                      Entry sequence for Never
              |
    ];
     |
     |                                               Closure
     |
    MoveReg R_Func;
    Process;
    Fall 1;
    From And/1;
    TrueJump L4;                                     Query successful
    Nil;
    Fault;
    Triv;
    ChainFault 3;                                    Construct fault
    Copy Result/1;
    Copy Result/2;
    Move Result/3;                                   Overwrite x, y, z
L4: MoveReg R_And;                                   Restore and-cell
    Stop;

>    x = (fault :  {decl}) :  α
>+   y = (fault :  {decl}) :  β
>+   z = (fault :  {decl}) :  γ
```

Figure 6–12: Failure of Top-level Query Declarations

6.4 Unification

$\mathcal{L}$-HYBRID supports the notion of unification found in PROLOG. When an instance of a clause is activated, it attempts to unify the argument passed to it (the actual parameter) according to the structure and value of its argument. This process is different to that used for passing parameters to functions; within a function, the formal parameter serves merely to denote the structure of the argument, and specify which parts of that structure should be bound to identifiers (a process denoted by the term *pattern matching*). Within a clause, the formal parameter is a structure which determines, amongst other things, the effect the unification process has upon logical variables passed within the argument. However, the formal parameter sections of functions and clauses have an identical syntax. Unification can be viewed as a generalisation of the parameter passing method used in function calls, since the unification process may instantiate one or more logical variables appearing in the argument, in addition to assigning variables appearing in the head (formal parameter section) of the clause. Variables may also be *repeated* in the head of a clause.

Due to the nature of the typechecking process, it has been assumed until now that the run-time representation of data objects contains no type information. In keeping with this philosophy, the unification process is strongly typed. At no stage during a unification process is it necessary to determine the type of object being unified; the static typechecking algorithm of chapter 5 performs all the necessary typechecking at compile-time.

6.4.1 Active and Passive Structures

A clause is a special kind of function. It consists of a number of branches, each of which has a formal parameter section and an optional body. The formal parameter of each branch is a structure of constants, expressions and logical variables to be unified to an actual parameter when the clause is activated.

A goal is an attempt to activate a clause. It consists of an expression representing a clause, and an actual parameter. The actual parameter is a structure of constants, expressions and logical variables to be unified to a formal parameter of the activated clause.

From this point of view, it would seem that there is no conceptual difference between a clause formal parameter and a goal actual parameter; both are structures which are manipulated by some unification algorithm when a clause is invoked. However, a distinction can be drawn between *active* and *passive* argument structures.

A formal parameter within a branch of a clause is considered to be active, since it does not directly represent a structure, but instead represents the unification operations which the clause will attempt to perform on an actual parameter. The object denoted by `clause(3)` can be thought of, not as a clause with formal parameter `3`, but as a clause which will attempt to unify an actual parameter with the value `3`. Similarly, `clause (_)` represents a clause which will not attempt to perform any unification whatsoever.

An actual parameter within a goal is considered to be passive, since it directly represents some structure to be unified by the clause being invoked. A term such as `query C(3)` is an attempt to activate the clause `C` with the value `3` as argument. The term `query C(_)` represents an attempt to activate `C` with a value `(_)`.

6.4.2 Structure and Value Unification

Each clause head is compiled with a unification algorithm determined by its formal parameter. This "skeletal" algorithm may require other information in the form of a method, which is passed when the clause is invoked.

A structure occurring within the head of a clause indicates an intent to unify the appropriate part of the argument to a structure of that kind; this will be termed *structural unification*. No method information is necessary in this case, since the structural operation does not depend on the type of argument.

A constant or expression occurring within the head of a clause indicates an intent to perform a unification against its value; this kind of operation (termed *value unification*) is performed according to type.

Corresponding to the indefinite number of unifiable types of object available in the language, there are an infinite number of unification algorithms, one for each type. In practice, a unification algorithm is constructed as a portion of code containing primitive unification instructions. Each such code portion assumes objects of the appropriate type, and unifies them accordingly. In the current implementation, a unification algorithm is constructed at each place where a value unification is required. Possible optimisations include having a predefined set of unification algorithms for the most commonly occurring types, and the construction of complex unification algorithms in terms of predefined simpler ones.

The unification instructions have the effect of transforming *uninstantiated* cells into data cells, or *references* to other cells. As a result of unification operations, long chains of reference cells may be built up.

A single unification operation (in PROLOG terms) is decomposed into a number of primitive unification instructions. The number of instructions executed depends on the complexity of the structures being unified, and on the complexity of their types. each unification instruction is atomic, although the entire unification operation may not be. Unification within a structure is performed from left to right; the unification order is important if a unification fails to complete, since no backtracking occurs.

The Instructions `IfInstanced` and `UnifyChain`

The instructions `IfInstanced` and `UnifyChain` are generated exclusively for head unifications. The instruction

    IfInstanced -> *label*

examines the top of the stack; if it encounters a data value (or a reference to it), then control is passed to *label*, chains of references being shortened in the process; otherwise, control passes to the next instruction. The instruction

```
    UnifyChain label
```

performs a primitive unification operation. It expects two arguments on the stack. The first argument (`Local/1`) is assumed to be a reference (or chain of references) to an undefined cell. The second argument (`Local/0`) is assumed to be a data value. `UnifyChain` transforms the undefined cell into a data cell, with the same value as `Local/0`. Both elements are removed from the stack, and control is unconditionally passed to *label.*

Figure 6–13 illustrates simple unification using `IsInstanced` and `UnifyChain`. The code generated assumes two execution paths for the clause. The first of these occurs if `IsThree` is called with an instantiated argument. In this case, control is passed immediately to label `L4`, which will perform an `AndSuccess` (at `L5`) if the value passed is identical to the formal parameter (viz. the integer `3`), and an `AndFailure` otherwise.

In general, there are a number of execution paths through the unification code of a clause. For each part of an argument, an instantiated cell causes control to pass to the label of the `IfInstanced` instruction; otherwise, control continues to the `UnifyChain` instruction. This allows unnecessary unification to be avoided.

The instructions `IfInstanced` and `UnifyChain` always appear in pairs. In addition, they form a critical region with respect to the logical variable being referenced. If the argument is already instantiated, then the unification at this level of the structure is considered complete. Otherwise, control will eventually pass to the `UnifyChain` instruction.

The reasoning behind this mechanism is twofold. Firstly, it removes the need for a number of primitive `Unify` instructions, one for each data type in the language. Secondly, it allows for optimisation of clauses which have complex formal parameter structures. If the clause of figure 6–14 is activated with an uninstantiated variable as argument, then only one primitive unification operation

```
::    let clause IsThree 3;

L8: InflateEnv 1;
L7: Code 3 [
       L6: Triv;
           FromReg R_Arg;                    Retrieve argument
           IfInstanced L4;                   Is the argument instantiated?
           Int 3;                            No: unconditionally
           UnifyChain L5;                    unify to 3
       L4: TestFault 0, L3;                  Argument is instantiated;
                                             ensure it isn't a fault
           Int 3;
           EqInt;
           FalseJump L3;                     Succeed if 3, fail otherwise
       L5: AndSuccess;
       L3: AndFailure;
    ];
    NullClosure Local/0;
    Move Result/1;
    Stop;

>   IsThree = ψ :  int Clause
```

Figure 6–13: Scalar Unification

is performed (the `UnifyChain`), and the clause immediately succeeds. The head structure is generated before unification commences, and then only the required unification operations need be performed, according to the actual parameter passed[1].

Since the $\mathcal{L}$–HYBRID abstract machine provides no facilities to explicitly declare critical regions, it is necessary to avoid interference between processes which have access to, and may attempt to unify, the same logical variable. The clause `UnifyOneOrTwo` of figure 6–15 will, upon execution, create a disjunctive system of two processes. The first of these will attempt to unify an argument against the data value `1`, and the second will attempt to unify an argument against `2`.

The critical region is enforced by allowing `IfInstanced` to block access to its argument from another process. If the argument to `IfInstanced` is already instantiated, then no interference can occur. However, if the argument is *uninstantiated*, then no other process must be permitted to unify it until the current process has performed a `UnifyChain`.

If the `IfInstanced` jump fails, then the argument can only be a reference cell pointing directly to an uninstantiated cell (from section 6.3.4). In this case, `IfInstanced` makes the uninstantiated cell *empty*. This blocks any other access to the cell by another process, since any attempt to access the *empty* cell will suspend the process.

`UnifyChain` assumes that its first argument (`Local/1`) is a reference cell pointing directly at an empty cell, and overwrites the empty cell without attempting to examine it. This operation serves to end the critical region.

The `Unify` Instruction

The instruction

[1]A further optimisation would be to create structures statically and build them into the closure of the clause.

```
    Unify -> label
```

is generated for unifications performed by type, rather than the structural unifications of section 6.4.2. It expects two arguments on the stack, which may be data values, or may be references to uninstantiated cells[2]. If either of the referenced cells is uninstantiated, then it is assigned a reference to the other cell, the cell values are removed from the stack, and control passes to *label*. Otherwise (if both are instantiated values) the values are left on the stack, and control passes to the next instruction. If *both* cells are uninstantiated, it is unimportant which cell is unified to the other.

Unification by type occurs within methods (section 6.5), but can also occur for the head of a clause when a head variable is repeated, or unification is performed with a constant expression. In figure 6–16, unification is performed on the two occurrences of the local logical variable `x`. However, since the type of `x` can be statically determined (from the goal `(x? > 0)`), the code generated for the clause `IntPair` can contain instructions to perform integer unification.

6.5 Method Construction

A method is a list of unification algorithms, each for a particular type of data object. Corresponding to every clause is a (possibly empty) method for each labelled type variable in its type. When the clause is compiled, it is assumed that the information necessary to unify these types will be passed as an implicit parameter. When the clause is activated, a method is constructed and passed to the clause.

The register `R_Method` is used to pass method information. When a clause is compiled, it is assumed that each labelled type will have a corresponding entry

[2] A stack location cannot be *uninstanced* itself, since the *uninstanced* value cannot be copied.

```
::    let clause Complex(_, _, _, x);

L8: InflateEnv 1;
L7: Code 4 [
       L6: Block 1;
           Uninstance Local/0/0;            Uninstantiate x
           TupleCell 4;                     Assemble (_, _, _, x)
           Uninstance Local/0/0;            First element
           Uninstance Local/0/1;            Second
           Uninstance Local/0/2;            Third
           From Local/1/0;
           Move Local/1/3;                  x is fourth
           FromReg R_Arg;                   Retrieve argument
           IfInstanced L4;
           From Local/1;                    Not instantiated:
           UnifyChain L5;                   unify and succeed
       L4: TestFault 0, L3;                 Fail if fault
           From Local/0/3;
           Move Local/3/0;                  Assign x
       L5: AndSuccess;
       L3: AndFailure;
    ];
    NullClosure Local/0;
    Move Result/1;
    Stop;

>   Complex = ψ : (α, β, γ, δ) Clause
```

Figure 6–14: A Complex Formal Parameter

```
::    let clause UnifyOneOrTwo 1    |
                 UnifyOneOrTwo 2;

L12:InflateEnv 1;
L11:Code 3 [
       L10:OrCell 2;
           MoveReg R_Or;
           Fork 3, L6;
           Triv;                          First Branch
           FromReg R_Arg;
           IfInstanced L8;              |
           Int 2;                       |  Critical Region
           UnifyChain L9;               |
       L8: TestFault 0, L7;               Argument instantiated;
           Int 2;                         equal to 2?
           EqInt;
           FalseJump L7;
       L9: OrSuccess;
           AndSuccess;
       L7: OrFailure;
       L6: Triv;                          Second Branch
           FromReg R_Arg;
           IfInstanced L4;              |
           Int 1;                       |  Critical Region
           UnifyChain L5;               |
       L4: TestFault 0, L3;               Argument instantiated;
           Int 1;                         equal to 1?
           EqInt;
           FalseJump L3;
       L5: OrSuccess;
           AndSuccess;
       L3: OrFailure;
    ];
    NullClosure Local/0;
    Move Result/1;
    Stop;

>   UnifyOneOrTwo = ψ :  int Clause
```

Figure 6–15: A Critical Region in a Disjunctive System

```
::    let clause IntPair(x, x) :- (x? > 0);

L13:InflateEnv 1;
L12:Code 6 [
           |
           |                                       Allocate x,
           |                                       assemble (x, x)
           |
           FromReg R_Arg;
           IfInstanced L4;
           From Local/1;                           Argument is uninstantiated
           UnifyChain L5;
       L4: TestFault 0, L3;
           From Local/0/0;                         x := first of tuple
           Move Local/3/0;
           From Local/0/1;                         Get second of tuple
           From Local/3/0;                         Get x
           Unify -> L6;                            |
           EqInt;                                  |  unification by type: int
           FalseJump L7;                           |
           Jump L6;
       L7: AndFailure;
       L6: |
           |                                       Body of clause
           |
    ];
    NullClosure Local/0;
    Move Result/1;
    Stop;

>   IntPair = ψ :  (int, int) Clause
```

Figure 6–16: A Repeated Monomorphic Variable

in the method passed with the argument. Thus, a labelled type α_1 will have a method `Method/0`, β_2 will have entry `Method/1` and so on.

Each method entry is a closure which may be activated by function call. A method entry takes no explicit argument, but assumes that the cells `Result/0` and `Result/1` contain the objects to be unified. A method entry returns `true` to its caller if it successfully unifies the objects, and `false` otherwise. Each method closure contains the code for the method entry, the size of stack needed to execute it, and the method of the calling environment.

The equality clause is illustrated in figures 6–17 and 6–18. Output preceded by "$\triangleright$" represents intermediate method information generated by the typechecker. In the definition of `eq` (figure 6–17), it is *assumed* that a method will be passed to each call of `eq`, using register `R_Method`. The method for `eq` contains one entry (corresponding to the single label in the type of `eq`). The method entry is activated by moving it from `Method/0` into the function register `R_Func`, and then using `Apply` to activ~~a~~ate it, returning `true` or `false` to the stack to indicate success, or failure, of the unification.

In the call to `eq` (figure 6–18), the code for the method entry corresponding to the type `int List` is compiled into a closure (`L7` onwards). The closure of the method entry contains the *old* method; this is necessary for method types which themselves contain labels. The method is placed in register `R_Method` before the goal is activated.

If a labelled type variable (such as α_1) appears in a method, it represents a call within the method to the method of the containing clause. If a labelled instantiated type (such as $\mathtt{int}_1$) appears in a method then the label is discarded.

The typechecker ensures that clauses within the same recursive declaration use the same method. The special method written [*RecCall*] represents a method identical to that of the caller; such instances are identified statically by the typechecker (section 5.3.2). The clause `Pairs` of figure 6–19 contains a single recursive call to itself, and so the value of register `R_Method` can be left undisturbed throughout an invocation of `Pairs` (although the first method entry must still

```
::    let clause eq(x, x);

L10:InflateEnv 1;
L9: Code 6 [
           |
           |                                   Allocation for x,
           |                                   Assembly of (x, x)
           |
           FromReg R_Arg;                      Retrieve argument
           IfInstanced L4;
           From Local/1;                       Argument uninstantiated:
           UnifyChain L6;                      unify and exit
       L4: TestFault 0, L3;                    Argument instantiated
           From Local/0/0;                     First of tuple
           Move Local/3/0;                     Store x
           From Local/0/1;                     Second of tuple
           From Local/3/0;                     Get x
           Unify -> L6;                        If both are instantiated, then:
           From Method/0;                      Retrieve the method,
           MoveReg R_Func;                         and
           Apply;                              call the method
           Deflate 1, 2;
           FalseJump L7;                       Jump if method failed
           Jump L6;                            Success
       L7: AndFailure;
       L6: AndSuccess;
       L3: AndFailure;
    ];
    NullClosure Local/0;
    Move Result/1;
    Stop;
```

`>    eq =` ψ `:   `(α_1, α_1) `Clause`

Figure 6–17: Definition of Equality Clause

be placed in R_Func in order to activate it). The instructions `FromReg R_Method` and `CopyReg R_Method` are necessary in the case where a recursive clause makes calls to some other clause with a different method structure, to avoid the register value being corrupted; in the case of `Pairs`, they are unnecessary.

```
::    let query eq([1], x : y) with x, y;

   ▷     query
            eq ([1], x :  y)                        [1 = int List]
         with x, y

L11: |
     |                                              Allocate x, y
     |
    Block 1;
L7: Code 5 [                                        Method for int List
           |
           |
           |
    ];
    From Local/2;                                   Get old method list
    Closure Local/1;                                Put into closure of new
    Move Local/1/0;
    MoveReg R_Method;                               Pass method list
L9: Code 4 [
           |
           |                                        Entry sequence:
           |                                        Assemble ([1], x :  y),
           |                                        call eq
           |
    ];
     |
     |                                              Closure
     |
    MoveReg R_Func;
    Process;                                        Activate goal
    Fall 1;
    From And/1;                                     Outcome of goal
    TrueJump L10;
     |
     |                                              Generate fault-value
     |
    Copy Result/1;                                  Overwrite x and y
    Move Result/2;                                  with fault-value
L10:MoveReg R_And;                                  Query succeeds
    Stop;

>   x = 1 :  int
>+  y = [] :  int List
```

Figure 6–18: Activation of Equality Clause

```
::    let rec clause Pairs [(x, x) | x'] :- Pairs x'    |
                     Pairs nil;

   ▷    clause Pairs [(x, x) | x'] :-
           Pairs x'                              [RecCall]
      | Pairs nil


L17:InflateEnv 1;
L16:Code 3 [
      L15:OrCell 2;                              Two branches
          MoveReg R_Or;
          Fork 8, L11;                           Process for Pairs [...]
           |
           |                                     Code for Pairs nil
           |
      L11: |
           |                                     Construct [(x, x) | x'],
           |                                     preliminary unification
           |
          From Method/0;
          MoveReg R_Func;
          Apply;                                 Unification by method
          Deflate 1, 2;
          FalseJump L7;
          Jump L6;
      L7: OrFailure;                             Method unification fails
      L6: Fall 1;                                Method unification succeeds
      L5: From Local/0/1;
          Move Local/4/0;
          Fall 1;
      L3: OrSuccess;
          FromReg R_Method;                      Save old method...
          CopyReg R_Method;                      but use for recursive call
           |
           |                                     Recursive call
           |
      L1: OrFailure;
   ];
   |
   |                                             Closure
   |
   Stop;
```

> Pairs = ψ : (α_1, α_1) List Clause

Figure 6–19: A Method for Recursive Call

Chapter 7

Conclusions and Future Work

This thesis has attempted to show that, with a suitable concurrent architecture, high levels of abstraction can be supported in programming languages without sacrificing control over the concurrent behaviour of executing programs. However, as is typical in research projects, many other questions have been raised, and not all of them can be answered in the scope of this work.

It has been demonstrated that, with a suitable choice of computational model (chapter 2), and using suitable optimisation techniques (chapter 3), it is possible to implement an applicative language so that the style of programming in the language directly influences the amount of concurrency utilised in its execution. Although the merits of such an approach are fairly self-evident, it is difficult to accurately judge the relative gains of different programming styles without accurate measures of the cost of concurrent operations in a specific architecture.

A number of important applicative programming paradigms lend themselves easily to the optimisation techniques presented in chapter 3. The use of parallel structure assignment (section 3.2) makes the concept of a *place-holder*, such as the *future* of Multilisp [Halstead 85], unnecessary; place-holding is performed automatically by *empty* values residing in heap structures. The techniques derived in section 3.5 allow fine control of the concurrency utilised in such cases. More particularly, the identification of structural tail-recursion as an optimisation technique allows major gains to be made for a large class of applicative programs, in particular those using recursive list traversal functions. Such gains are twofold. There is a saving in space (and number of idle processes) since such a function is executed by a single process; however, such savings may be made in a conventional applicative system by the use of accumulating parameters [Henderson 80]. The major gain of structural tail-recursion is that a function may return a partial result value to be consumed by another function, whilst completing evaluation of the result. Nested systems of such functions exhibit systolic properties [Kung 78], with the additional advantage that the data values being passed may be of arbitrary structure. Such behaviour compares favourably with the co-routine interpretation of lazy evaluation, but has the advantage that all the constituent processes are active at the same time.

The concept of using locally bound variables to introduce concurrency follows naturally from traditional notions of abstraction, where identifiers are used to denote values calculated in some separate part of a program. The notion of "distance" between the declaration of a variable and its use lends itself conveniently to a concurrent interpretation: expressions which are some distance apart in a program are more likely to be executed by different processes.

The concept of using "bottleneck" functions like FLATTEN and WAIT (chapter 3) comes dangerously close to violating the abstraction of applicative programming, since it relies on some understanding of the implementation of partial application, and how it affects concurrency. In addition, some insight is required into the representation of data objects in order to produce, on demand, a FLATTEN function for some arbitrary type. This drawback could be overcome by providing a facility to generate correct FLATTEN functions from context, employing some preprocessing or macro expansion technique. It should be noted, however, that the WAIT and FLATTEN functions are derived as a natural consequence of the optimisation techniques derived at the start of chapter 3, rather than being facilities developed in their own right, and it would be hoped that other coding practices could be developed to exploit the optimisation techniques in a more elegant manner. Such exploitation might take the form of *pragmas* to "customise" the optimisation and concurrency extraction as required. As mentioned in chapter 1, such a scheme would result in the programmer losing fine control over the concurrency, but this might be a small price to pay for separating the task of writing programs from that of "tuning" their execution.

If the above techniques for control of concurrency are totally disregarded, the concurrency utilisation in $\mathcal{F}$–HYBRID programs is reasonably encouraging. Writing a program applicatively in $\mathcal{F}$–HYBRID without regard for performance generally results in roughly 50% of the generated processes performing useful work at any one time, although if some consideration is given to visualising the underlying algorithm and identifying the flow of data values through the system, this figure can be improved considerably. Clearly, new paradigms for writing efficient programs would have to be identified.

The concept of faults-as-values (chapter 2, appendix A), although aesthetically pleasing, does not lend itself to simple implementation. Even given hardware assistance for generating and detecting fault-values, the code generated by the compiler must perform numerous checks for fault-values occurring in circumstances where the underlying machine could not be expected to perform the correct behaviour. A simple expression such as

if *cond* **then** e_1 **else** e_2

requires careful consideration should *cond* evaluate to a fault-value; neither e_1 nor e_2 can be regarded as the value of the expression, and control must pass to a sequence of instructions generated to deal with this eventuality. In addition, the operations which must be performed to propagate a fault-value presented to a primitive operator are overly complex. The support for logical variables within the abstract machine also proved to be more complex than at first imagined. The technique of implementing "invisible pointers" in hardware (or, at least, microcode) is already established [Weinreb 84], but extensions of this technique would be necessary to treat logical variables in the manner required by section D.1.2. A future research topic would be the presentation of fault-values to the programmer in a manner which more closely reflected the nature and occurrence of the original error; contemporary pretty-printing techniques could be brought to bear to this end [Oppen 80].

The implementation of $\mathcal{L}$–HYBRID has been simplified greatly by the assumption of nondeterminism in the underlying hardware. However, the compilation of $\mathcal{L}$–HYBRID programs to abstract machine code makes critical sections (such as those required during unification) difficult to implement above the lowest level, "per-instruction" basis. It is unfortunate that this shortcoming allows interference between *or*-parallel clauses, and makes the order of unification steps apparent. The proper implementation of guarded clauses requires that a guard system, and all systems within it, may perform unifications without the variable bindings being propagated beyond the guard. This is no mean feat, given that variables within a guard may already be bound to other variables in some enclosing system. Treating each entire guard as a critical section would solve this problem, but only at a great cost to concurrency. Use of shared association lists for variable bindings would be another, rather formidable, solution.

The lack of any kind of remote process termination has been found to pose no great threat to the performance of a deductive system. The only *or*-parallelism in a system is due to alternative guards (since a clause tail becomes part of the enclosing *and*-system), and guards seldom contain long, complex calculations. However, a facility to implicitly terminate processes would improve the $\mathcal{F}$–HYBRID implementation. Implicit process termination would allow infinite computations to be dealt with safely, as well as giving other performance improvements.

The advantages of cross-calling applicative and logic programs are documented elsewhere [Robinson 80, Sato , Barbuti]. However, new issues are brought to light by the combination of the deterministic language $\mathcal{F}$–HYBRID with the nondeterministic language $\mathcal{L}$–HYBRID. It appears initially that $\mathcal{L}$–HYBRID, by its use of unguarded logical variables, quickly infests any associated $\mathcal{F}$–HYBRID functions with their strange behaviour. In addition, the facility to pass functions and clauses as arguments to each other allows rather strange, "tightly-coupled" hybrid systems to be built rather easily. Certainly, the use of local assertions and clauses-as-arguments should provide insights into solutions of contemporary issues such as the framing problem [McCarthy 83]. The proper exploitation of a hybrid declarative programming system requires careful consideration.

The polymorphic typechecking algorithm presented in chapter 5, whilst allowing functional objects to be passed as arguments to clauses, is somewhat lacking in elegance. The restrictions on the form and content of recursive clause declarations, and lambda-bound clauses, should be investigated further, and the general concept of labelled types requires some serious theoretical investigation. The main strength of the algorithm presented is its ability to generate and propagate distinct unification methods for distinct types, regardless of their structure. This technique is ideally suited to the problem of implementing equality over abstract data types. Ironically, this fact was realised too late; the implementation of an abstract type mechanism for HYBRID had already been considered irrelevant to the project. The results of this thesis have already inspired simpler and more elegant schemes for performing polymorphic comparison of non-abstract data types [Milner 86].

A number of comments need to be made about the underlying concurrent architecture. The concept of having a global address space, and allowing any process to access any part of it, is eminently desirable from the point of view of a compiler writer; it provides an elegant abstract level upon which to build a variety of concurrent systems. However, more consideration needs to be given to an architecture which seems to promote undesirable global communication. An implementation of the architecture would probably consist of a number of processing elements, each with a portion of local store; these portions of store would comprise the whole of memory available to the system. Under such a scheme, non-local memory accesses would cause communication between the processor requiring the memory access, and the processor to which the store would be local. In this way, the global address space could be supported by a segmented memory model with processors communicating by means of messages. A prototype of such an architecture is discussed elsewhere [Treleaven b, Foti]; alternatively, such a scheme might be implemented on a number of processors such as transputers [INMOS 84a], if non-local memory accesses can be made to generate read and write requests between processors.

Given a global address space, certain observations can be made which allow the number of non-local memory accesses to be reduced substantially. The first of these is that each HYBRID process is allocated a portion of stack space which is *almost* totally local to it. The only violation of this locality comes from the use of **PushProcess**, which results in one process having reference to the stack space of another process. One solution of this problem would be to allow every child process created by a **PushProcess** instruction to reside on the same processor as its parent, whilst allowing **VecProcess**, which is free of stack interference, to cause process migration. Investigation would have to be made of the effect on concurrency resulting from such an approach. Another solution would be to allocate a section of the heap for the result of a **PushProcess**, and use an invisible reference (as in section 6.3.4) to give the parent process access to it.

The assumption of a totally local stack could yield considerable performance benefits. Contemporary compilation techniques for procedural languages often make use of an abstract stack code which, rather then being executable in its own right, is considered as a stream of directives for an optimising code generator [Robertson 81]. A local stack might be largely dispensed with, in favour of a set of fast, general purpose registers (for example, a register file of the form found in the RISC I microprocessor [Patterson 81]).

Given that any architecture has a bounded number of processing elements, the ability of each of these to multitask is obviously highly desirable. However, it might still be infeasible to assume that every attempt to create a process will be successful. Given that some process creation requests may fail, a generalisation of the fault propagation scheme of section A.8 would allow a fault-value to be generated for each failed attempt. From the point of view of an applicative program, the result would be the generation of fault-values at the deepest levels of an excessively large application; the price to pay for this scheme would be a loss of determinism in such cases. However, without some way of reasoning about the failure of a deduction due to lack of resources, it would be difficult to provide meaningful feedback in a logic application.

It has been assumed throughout this thesis that garbage collection is feasible in a multiprocessing environment of the type described. Schemes have been documented which perform free list recombination in a concurrent environment [Rudalics 85]; a cyclic reference counting scheme [Brownbridge 85] would provide another approach. Concessions can be made due to the fact that local stack space does not need garbage collecting (section 2.2), although fragmentation must be avoided by some means. Since the size of cells claimed from the heap is determined at compile time, "chunking" of heap requests could be performed statically, resulting in less heap congestion and fragmentation.

Multiprocessing schemes give rise to another class of garbage collection problem, that of reclaiming processes whose computations are no longer required. One solution is to implement a garbage collector which can "out-perform" any process; if the garbage collector finds a process generating an unreferenced result, then the process can be terminated. An alternative scheme might involve some kind of resource allocation from parent process to child. Under such a scheme, every process would have some resource value which would be shared out to all of its children. Any process which ran out of resource would be terminated (or suspended until more resource became available). It would be interesting to see whether such a resource allocation scheme would provide the benefits of lazy evaluation (the ability to manipulate potentially infinite objects) as well as the benefits of concurrent execution.

Bibliography

[Ashcroft 77] E. A. Ashcroft and W. W. Wadge. LUCID—a non-procedural language with iteration. *Communications of the ACM*, 20(7), July 1977.

[Barbuti] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the integration of logic programming and functional programming. Dipartimento di Informatica, Universita di Pisa/CNUCE-C.N.R., Pisa.

[Brownbridge 82] D. R. Brownbridge, L. F. Marshall, and B. Randell. *The Newcastle Connection, or Unixes of the World Unite.* Technical Report, Computing Laboratory, University of Newcastle upon Tyne, July 1982.

[Brownbridge 85] D. R. Brownbridge. Cyclic reference counting in combinator machines. In *Proc. Conference on Functional Languages and Computer Architecture*, 1985.

[Browning 80] S. A. Browning. *The Tree Machine: A Highly Concurrent Computing Environment.* Technical Report, Department of Computer Science, California Institute of Technology, 1980.

[Cardelli 81] Luca Cardelli. *Sticks & Stones—An Applicative VLSI Design Language.* Internal Report CSR–85–81, University of Edinburgh, Department of Computer Science, July 1981.

[Cardelli 83a] Luca Cardelli. The functional abstract machine. *Polymorphism—The ML/LCF/Hope Newsletter*, 1(1), January 1983.

[Cardelli 83b] Luca Cardelli. ML under unix. *Polymorphism—The ML/LCF/Hope Newsletter*, 1(3), December 1983.

[Chen 86] Marina C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *13th ACM Proceedings on Principles of Programming Languages and Systems*, 1986.

[Clocksin 81] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Dahl 72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[Damas 85] L. M. M. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, Department of Computer Science, 1985.

[Dennis 79] J. B. Dennis. The varieties of data-flow computers. In *Proceedings of 1st International Conference on Distributed Computer Systems*, 1979.

[Foti] L. Foti, D. English, R. Hopkins, P. Treleaven, and L. Wang. A reduced instruction set multi-microcomputer system (RIMMS). Computing Laboratory, University of Newcastle upon Tyne.

[Fuchi 83] Kazuhiro Fuchi. The direction the FGCS project will take. *New Generation Computing*, 1(1), 1983.

[Halstead 85] Jr. Robert H. Halstead. Multilisp—a language for concurrent symbolic communication. *ACM Transactions on Programming Languages and Systems*, 7(4), October 1985.

[Harper 85] Robert W. Harper. *Introduction to Standard ML.* Technical Report, University of Edinburgh, Department of Computer Science, October 1985.

[Henderson 80] Peter Henderson. *Functional Programming—Application and Implementation. International Series in Computer Science*, Prentice-Hall, 1980.

[Hoare 78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978.

[Hudak 86] Paul Hudak and Lauren Smith. Para-functional programming—a paradigm for programming multiprocessor systems. In *13th ACM Proceedings on Principles of Programming Languages and Systems*, 1986.

[INMOS 84a] INMOS Ltd. *IMS T424 Transputer Reference Manual.* November 1984.

[INMOS 84b] INMOS Ltd. *OCCAM Programming Manual.* Prentice-Hall International, 1984.

[Kowalski 82] Robert Kowalski. Logic programming. December 1982. An Invited Paper to be Presented at IFIP83.

[Kung 78] H. T. Kung and Charles E. Leiserson. *Systolic Arrays for VLSI.* Technical Report, Carnegie-Mellon University, April 1978.

[Landin 64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, 1963–64.

[Mago 80] G. A. Mago. A cellular computer architecture for functional programming. In *Proceedings CompCon*, 1980.

[McCarthy 83] J. McCarthy. Some expert systems need common sense. In *Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar—Man-Machine Interaction*, September 1983.

[Mead 80] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[Milner 77] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, (17), 1977.

[Milner 80] Robin Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[Milner 85] Robin Milner. The standard ML core language. *Polymorphism—The ML/LCF/Hope Newsletter*, 2(2), October 1985.

[Milner 86] Robin Milner. Polymorphic equality for ML. February 1986. Informal Presentation, LFCS, University of Edinburgh.

[Monteiro 84] L. Monteiro. A proposal for distributed programming in logic. In J. A. Campbell, editor, *Implementations of PROLOG*, Ellis Horwood, 1984.

[Moto-Oka 83] Tohru Moto-Oka. On the founding of this journal. *New Generation Computing*, 1(1), 1983.

[Mycroft 81] Alan Mycroft. *The Theory and Practice of Transforming Call-by-Need into Call-by-Value.* Internal Report CSR–88–81, University of Edinburgh, Department of Computer Science, July 1981.

[Mycroft 83] Alan Mycroft and Richard O'Keefe. A polymorphic typechecker for PROLOG. In *Proceedings of Logic Programming Workshop, Portugal*, 1983.

[Oppen 80] D. C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4), October 1980.

[Patterson 81] D. A. Patterson and C. H. Sequin. RISC I—a reduced instruction set VLSI computer. In *Proc. Eighth International Symposium on Computer Architecture*, May 1981.

[Robertson 81] Peter S. Robertson. *The Production of Optimised Machine-Code for High-Level Languages using Machine-Independent Intermediate Codes.* PhD thesis, University of Edinburgh, Department of Computer Science, November 1981.

[Robinson 80] J. A. Robinson. *LOGLISP—An Alternative to PROLOG.* Technical Report, Syracuse University, December 1980.

[Rudalics 85] M. Rudalics. Parallel memory management on a multiprocessor system. *Microcomputers, Usage and Design*, 1985.

[Sato] Masahiko Sato and Takafumi Sakurai. Qute users' manual. Department of Information Science, University of Tokyo.

[Shapiro 83a] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in concurrent prolog. *New Generation Computing*, 1(1), 1983.

[Shapiro 83b] Ehud Y. Shapiro. *A Subset of Concurrent Prolog and Its Interpreter.* Technical Report TR-003, Institute for New Generation Computer Technology, February 1983.

[Sussman 72] Gerald Jay Sussman and Drew Vincent McDermott. *Why Conniving is Better than Planning.* Technical Report Memo No. 255a, Massachusetts Institute of Technology A. I. Laboratory, April 1972.

[Treleaven a] Philip C. Treleaven. Decentralised control flow programming. Computing Laboratory, University of Newcastle upon Tyne.

[Treleaven b] Philip C. Treleaven, Richard Hopkins, David E. English, and Lewis Foti. Design 1 of RIMMS (a reduced instruction set microprocessor system). Computing Laboratory, University of Newcastle upon Tyne.

[Treleaven c] Philip C. Treleaven and Isabel Gouveia Lima. Future computers—logic, data flow, ..., control flow? Computing Laboratory, University of Newcastle upon Tyne.

[Treleaven 82] Philip C. Treleaven, Richard P. Hopkins, and Paul W. Rautenbach. Combining data flow and control flow computing. *The Computer Journal*, 25(2), 1982.

[Turner 81] David A. Turner. The semantic elegance of applicative languages. In *Proceedings of Functional Programming Language and Computer Architecture Conference*, 1981.

[Warren 77] David H. D. Warren. *Implementing PROLOG—Compiling Predicate Logic Programs.* Technical Report 39 & 40, Department of Artificial Intelligence, University of Edinburgh, May 1977.

[Weinreb 84] D. Weinreb and D. Moon. *Lisp Machine Manual.* Technical Report, Symbolics Corp., 1984.

[Welsh 79] J. Welsh and D. W. Bustard. Pascal-plus—another language for modular multiprogramming. *Software—Practice and Experience*, 9, 1979.

[Wilner 80] Wayne T. Wilner. *Recursive Machines.* Technical Report, Xerox Palo Alto Research Center, 1980.

Appendix A

A Tutorial for $\mathcal{F}$–HYBRID

A.1 Introduction

This is an introduction to the experimental programming language $\mathcal{F}$–HYBRID, a purely applicative subset of the declarative language HYBRID. HYBRID is a combined applicative and logic language which allows functional program segments to activate program segments written in a logical form, and vice versa. In this appendix, the applicative language will be considered in isolation. This appendix is based largely on an introductory document for Standard ML [Harper 85], but has been altered to highlight features peculiar to $\mathcal{F}$–HYBRID.

$\mathcal{F}$–HYBRID qualifies to be called applicative for two reasons:

- Functions are first class objects. They may be passed as arguments, returned as results, and embedded in data structures. Functions may be denoted by anonymous expressions, and not just by declarations.

- $\mathcal{F}$–HYBRID is free from any imperative constructs. There are no variables in the conventional sense of the term, and no form of assignment statement. The principal control mechanism in $\mathcal{F}$–HYBRID is recursive function application; there is no notion of sequential flow-of-control.

𝓕-HYBRID is statically scoped. All identifier references are resolved at compile-time, rather than during the execution of a program. This leads to more efficient programs, and avoids the class of errors introduced by identifiers being accidentally redefined in the environment of an existing program.

𝓕-HYBRID is strongly typed; the type of any legal expression is determined statically by the compiler. Strong typing ensures that no type errors can occur at run-time.

A.2 Using the System

The HYBRID compiler is interactive. Expressions entered at the terminal are immediately typechecked, compiled and executed, yielding a result. Declarations entered at the terminal are established for use by all subsequent expressions (and further declarations).

In this document, a block of text ~~outlined by a box~~ preceeded by a '::' represents some interaction with the compiler. The prompt '::' is issued by the system when it is ready to accept an expression or declaration. Lines beginning with '>' denote feedback from the compiler after the execution of a program fragment.

A.3 Lexical Conventions

The lexical conventions for 𝓕-HYBRID resemble those of a conventional programming language. Statements may be spread over several lines, since spaces and newlines are considered equivalent (except within strings), and serve merely to separate tokens. A statement is terminated by a semicolon, ';'. Comments are introduced by the token '--', and are terminated by the end of the line.

A number of characters are considered to be *isolated symbols*; they may occur without leading or trailing spaces without ambiguity. Brackets and punctuation symbols are considered to be isolated, as are certain symbols such as ‘&’, ‘@’, ‘^’ and ‘~’.

There are a number of keywords which are reserved, and therefore may not be used as identifiers. In addition, some identifiers have a special meaning to the system, and may not be re-defined.

A.4 Simple Expressions

One of the simplest forms of statement in $\mathcal{F}$–HYBRID is a simple expression involving constants:

```
        ::      2 + 2;

        >   4 :  int
```

When an expression is presented to the system, it responds with the expression's value and type. The type of an expression is determined totally by the compiler; at no stage does any explicit type information need to be specified.

The type of an object represents its structure, and the kind of operations which can be performed on it. All objects of type `int` represent values ranging over the (positive and negative) integers, and are therefore amenable to the basic arithmetic operations. The set of prime numbers does not form a distinct type since they are structurally identical to (and a subset of) the set of integers. Also, the set of primes could not constitute a type because it is not closed under the basic arithmetic operations.

There are a number of basic, primitive types. For each primitive type, there are primitive constants which represent values of that type (for example, 2 is a constant of type `int`). For most primitive types, there are built-in operators which act on values of these types.

Integers

Constants of type integer (denoted by int) are written as a sequence of digits. There are a number of built-in operators which range over objects of type int, and denote the arithmetic functions:

```
::     2 - 3 * 4;

>   ~10 :  int

::     10 / ~5 + 3 mod 2;

>   ~1 :  int
```

Note that integer negation is represented by '~'; the operator '-' is used solely for subtraction. Operators are assigned a conventional precedence, which may be overridden by brackets.

Booleans

There is a built-in boolean type (denoted by bool) with the two constants true and false. The boolean operators are '&' (logical *and*), or, and not:

```
::     true or true & false;

>   true :  bool

::     (true or true) & false;

>   false :  bool

::     not false & false;

>   false :  bool

::     not (false & false);

>   true :  bool
```

The operators '<', '>', '<=' and '>=' may be applied to integers to yield boolean results. In addition, the generic operators '=' and '<>' may be applied to objects of integer, or boolean, type:

```
::    (3 >= 4) = (true = false);

>   true :  bool
```

Strings

There is a basic string type (written `str`) whose constants are written as character sequences between double quotes ('"'). Strings may be compared using '=' or '<>'.

The basic operations over strings are as follows: `length` returns the length of a string. The infix operator '^' concatenates two strings. `ord` takes two arguments; the first is a string, and the second is a number between one and the length of the string. The result is the ASCII value of the specified character of the string. chr takes a numeric argument between 1 and 127, and returns a string whose only character is the ASCII character corresponding to that argument. ord and chr will generate *fault-values* (section A.8) if their arguments are outside the required range.

```
::    ("abc" ^ "def", length("abc" ^ "def"));

>   ("abcdef", 6) :  (str, int)

::    (chr 65, chr 200);

>   ("A", (fault :  {prim "chr"})) :  (str, str)

::    (ord("abcd", 1), ord("abcd", 4), ord("abcd", 5));

>   (97, 100, (fault :  {prim "ord"})) :  (int, int, int)
```

Triv

There is a data type called `triv`, with a single value written '`()`'. This is generally used within variants, or to model the calling of functions without arguments (section A.7).

```
::     ();

>   () :  triv
```

Conditional Expressions

There is a conditional expression which yields one of two values depending of the value of a boolean argument:

```
::     if 3 > 4 then "Wrong"
                else "Right";

>   "Right" :  str
```

The expression after the **then** must have the same type as the expression following the **else**. The **else**-branch may not be omitted. Conditional expressions may be nested to an arbitrary depth:

```
::     if 3 > 4 then
          if true then "Foo1" else "Foo2"
       else
          if (if true then false else 1 > 0)
          then "Foo3"
          else "Foo4";

>   "Foo4" :  str
```

The keyword **else** may be omitted within several cascaded conditional expressions (although the meaning remains unchanged):

```
::     if 2 = 0 then "Zero"
       if 2 = 1 then "One"
       if 2 = 2 then "Two"
                 else "Many";

>    "Two" :  str
```

It is worth noting at this stage that the primitive boolean operators '&' and or evaluate both their arguments. However, the conditional construct only evaluates either its then-branch or its else-branch. This point will become important when considering recursive functions and nondeterministic programs (appendix B).

A.5 Structured Types

There are a number of *structured* types in the language, corresponding to data structures in a conventional programming language. Objects of a structured type are constructed from other objects which may be primitive (for example, integers), or themselves structured.

Corresponding to each structured type is a particular syntax for creating objects of that type. Corresponding to the *constants* of a primitive type are *constructors* of a structured type. A constructor is a special identifier used to create a structured data object.

Tuples

A tuple of values may be written with the values separated by commas and enclosed in brackets:

```
::     (1, false, "A");

>    (1, false, "A") :  (int, bool, str)
```

Note that the values within a tuple need not be of the same type. The comma is not regarded as an infix operator; the expression above has a different value (and type) to the expression `(1, (false, "A"))`.

Tuples may be compared using '`=`' and '`<>`':

```
::    (1, false, "A") = (1, true, "A");

>   false :  bool
```

Two tuples are considered equal if all elements of the tuple are equal. It is illegal to compare tuples of different length, since they do not have the same type.

Lists

A list is an ordered sequence of values of the same type. The empty list is written `nil`, and lists are constructed using the *cons* operator, written '`:`':

```
::    1 : 2 : 3 : nil;

>   [1; 2; 3] :  int List

::    [false | nil] : nil : [true; true] : nil;

>   [[false]; []; [true; true]] :  bool List List

::    nil;

>   [] :  α List
```

The system prints out lists in an alternative notation, which may also be used for input. The elements are printed between square brackets, and separated by semicolons. The tail of the list may be written between the symbol '`|`' and the closing bracket; by default, the tail is assumed to be `nil`.

Two lists are considered to be equal if they are the same length, and their corresponding elements are equal:

```
::    [1; 2; 3] = [1; 2];

>   false :  bool

::    [1; 2; 3] = [2; 1; 3];

>   false :  bool

::    [1; 2] = 1 : 2 : nil;

>   true :  bool
```

Disjoint Sums

Objects of two distinct types can be grouped to form objects of one single type known as a *disjoint sum.* A disjoint sum is a structure containing a data value, together with an implicit tag denoting which of the two distinct types it conforms to. The sum may be tagged as a *left sum*, represented syntactically by the constructor in_lft, or a *right sum*, represented by in_rht:

```
::    in_lft 3;

>   (in_lft 3) :  (int + α)

::    in_rht (in_lft [1; 2; 3]);

>   (in_rht (in_lft [1; 2; 3])) :  (α + (int List + β))

::    [in_lft 2; in_rht false; in_lft ~3; in_rht true];

>   [(in_lft 2); (in_rht false); (in_lft ~3); (in_rht true)]
:  (int + bool) List
```

The component object of a disjoint sum may be extracted by means of the predefined functions out_lft and out_rht. The predefined functions is_lft and is_rht allow the tag part of a disjoint sum to be tested; is_lft is true of a sum created using in_lft, and conversely for is_rht.

The comparison operators are defined over disjoint sums; the tags, and the contained values, must be the same:

```
::    in_lft 2 = in_rht false;

>   false :  bool

::    [in_lft 2; in_rht false]
      = [in_lft 2; in_rht false];

>   true :  bool
```

Variant Types

A variant type can be viewed as a generalisation of a disjoint sum; a variant can be created over a data object whose type forms one of any number of types in the variant. The different tag values of the variant are distinguished by means of distinct names (or *labels*) within the type of the variant.

A variant structure is written between the braces '{' and '}'. Generally, the label is written before the value contained within the variant:

```
::    {Foo(3)};

>   {Foo 3} :  {Foo :  int}
```

The typechecker automatically determines the number of, and labels for, any particular variant, depending on the context:

```
::    [{First(3)}; {Second(false)}; {Third(nil)}];

>   [{First 3}; {Second false}; {Third []}] :  {First
:  int, Second :  bool, Third :  α List} List

::    {1 + 2} = {+(1, 2)};

>   true :  bool
```

Note that operators (but not constructors) can be used as labels, and in their correct fix position.

If necessary, it is possible to specify a variant together with a list of the possible labels such a variant may have; this can be thought of as a form of type coercion:

```
::    {A(3)};

>   {A 3} :  {A : int}

::    {A(3) | A, B, C};

>   {A 3} :  {A : int, B : α, C : β}
```

The labels are separated by commas, and separated from the body of the variant by a vertical bar, '|'.

Variant labels are intended to model *functors* as found in PROLOG. However, a label is not a string, and it is not possible to access the characters making up the label. A label without any variant value can be written purely as the label within braces; it is treated as a variant with '()' as value:

```
::    [{Nothing}; {Nothing()};
       {Something(1 + 2 + 3, false)}];

>   [{Nothing}; {Nothing}; {Something (6, false)}]
:  {Nothing, Something :  (int, bool)} List
```

A variant type with no values attached to any labels can be thought of as an enumerated type:

```
::    [{Red}; {Green}; {Blue}];

>   [{Red}; {Green}; {Blue}] :  {Blue, Green, Red} List
```

The keyword is may be used to test the tag of a variant object:

```
::    {Red} is Red;

>   true :  bool

::    {Red} is Blue;

>   false :  bool
```

The keyword **as** may be used to extract a data object from within a variant:

```
::     {Red(3)} as Red;

>   3 :  int

::     {Blue} as Blue;

>   () :  triv
```

There is a *case expression* associated with variant types; this may be viewed as a generalisation of the conditional expression described earlier. Within a case expression, each possible label of a variant must be catered for:

```
::     case {Red} of
          {Red}.   "It's red";
          {Green}. "It's green";
          {Blue}.  "It's blue"
       end;

>   "It's red" :  str
```

The case expression may also be used to extract a data object from within a variant (section A.6).

A.6 Declarations and Bindings

A *declaration* is a means of associating identifiers with values. A declaration serves to introduce an identifier to the system; an occurrence of that identifier within a program represents the value that the identifier was *bound* to in the declaration. Declarations may be *local*; a declaration made locally is valid only within a certain scope, and does not affect the state of declarations (the *environment*) outside that scope. Declarations which are global (ie. are made at *top-level*) last indefinitely, or until superceded.

Simple Declarations

The following is the simplest form of declaration, followed by an expression using the identifier just declared:

```
::    let x = 20;

>   x = 20 :  int

::    if false then x - 5 else x + 5;

>   25 :  int
```

The identifier x represents the value 20 in all subsequent expressions and declarations, unless a subsequent (or local) declaration redefines x. Any other declarations which make use of this value for x will be unaffected if x is subsequently rebound to a new value (possibly of another type):

```
::     let y = [x; x - 1; x * x];

>    y = [20; 19; 400] :  int List

::     let x = false;

>    x = false :  bool

::     y;

>    [20; 19; 400] :  int List
```

Compound Declarations

Several identifiers may be bound in parallel, using the keyword and to separate the declarations:

```
::     let y = "New 'y'"
           and z = y;

>    y = "New 'y'" :  str
>+   z = [20; 19; 400] :  int List
```

Note that z gets bound to the *old* value of y. In such parallel declarations, the right hand sides of the declaration are evaluated before the left hand bindings are performed.

It is possible to cascade declarations using the keyword enc (for *enclose*); this allows one declaration to be seen within another:

```
::     let Ten = 10
           enc Hundred = Ten * Ten;

>    Ten = 10 :  int
>+   Hundred = 100 :  int
```

The first declaration may be made *local* to the second declaration by means of the keyword ins (for *inside*):

```
::    let Six = 6
          ins Sixty = Six * Ten;

>   Sixty = 60 :  int

::    Six;

Unbound identifier:  Six
```

There is an additional declaration keyword which may be used in certain circumstances. The keyword **rec** prefixed to a declaration causes the declaration to be considered recursive; the names being declared are made visible within the declaration itself:

```
::    let L1 = nil;

>   L1 = [] :  α List

::    let L1 = 1 : L1;

>   L1 = [1] :  int List

::    let rec L2 = 1 : L2;

>   L2 = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
1; 1; 1; 1; 1; ...] :  int List
```

Within 𝓕-HYBRID, the use of **rec** is restricted to the declaration of lists and functions.

The keywords and, **enc** and **ins** may be thought of as infix operators over declarations; and has a higher precedence than **enc** or **ins**. In addition, and has a higher precedence than the prefix **rec**, so that mutually recursive declarations can be made in the following manner:

```
::    let rec List1 = 1 : List2
              and List2 = 2 : List1;

>   List1 = [1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2;
1; 2; 1; 2; 1; ...] :  int List
>+  List2 = [2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1; 2; 1;
2; 1; 2; 1; 2; ...] :  int List
```

Declarations may be of arbitrary complexity, and the precedence of the declaration operators may be overridden by means of the braces '{' and '}':

```
::    let {x = 3 enc rec y = x : y}
          and {z = false enc w = not z};

>   x = 3 :  int
>+  y = [3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
3; 3; 3; 3; 3; ...] :  int List
>+  z = false :  bool
>+  w = true :  bool
```

It is illegal to have occurrences of enc or ins within a rec.

Local Declarations

The declarations illustrated above have all been global or *top-level* declarations; the identifiers introduced by a declaration have been available for subsequent expressions also entered at top-level. It is useful to make *local* declarations, in order to introduce identifiers with a scope limited to a particular expression.

A declaration may be followed by the keyword in and an expression within which the declaration is to have affect:

```
::    let x = true in [x; x; x];

>   [true; true; true] :  bool List
```

It may be more convenient to follow an expression by the keyword where with the declaration after it:

```
::    [x; x; x] where x = true;

>   [true; true; true] :  bool List
```

Local declarations may feature the declaration operators as required. An expression with a local declaration is also an expression in its own right, and may thus be used in any context where an expression is permitted:

```
::    [2; x;
       n + m where n = 1 enc m = n + 2;
       let a = 2
       in  if true then let b = ~5 in ~b
                   else a + 3]
      where x = 3;

>   [2; 3; 4; 5] :  int List
```

Bindings and Pattern Matching

The left hand side of a declaration need not be restricted to being an identifier; it may instead be a *pattern* of identifiers with the form of a structured data type. This provides the facility to *decompose* structured data objects within declarations.

A *tuple* may be decomposed into its constituent elements by means of a tuple pattern in the left hand side of a declaration:

```
::    let Tup = (1, false, nil, (2, 3));

>   Tup = (1, false, [], (2, 3)) :  (int, bool, α List,
(int, int))

::    let (x, y, z, w) = Tup;

>   x = 1 :  int
>+  y = false :  bool
>+  z = [] :  α List
>+  w = (2, 3) :  (int, int)

::    let (_, _, _, (_, x)) = Tup;

>   x = 3 :  int
```

The reserved identifier '_' represents an *anonymous* identifier whose value is to be disregarded. The following declaration is therefore valid (although useless):

```
::    let _ = Tup;
```

A pattern may contain constants; in this case, the constant must have the same value as the appropriate part of the right hand side expression (otherwise the binding will fail):

```
::    let (1, a, b, c) = Tup;

>   a = false :  bool
>+  b = [] :  α List
>+  c = (2, 3) :  (int, int)
```

A pattern may contain constructors such as **true** and **false**; these must match the appropriate values in the right hand side. Lists may be decomposed by means of the constructors **nil** and '**:**'; the square bracket notation for lists is also permitted:

```
::    let x : y : nil = [1; 2];

>   x = 1 :  int
>+  y = 2 :  int

::    let x : y = [1; 2];

>   x = 1 :  int
>+  y = [2] :  int List

::    let [x; y; z] = 1 : 2 : nil;

>   x = (fault :  {decl}) :  int
>+  y = (fault :  {decl}) :  int
>+  z = (fault :  {decl}) :  int
```

Note that the last declaration fails because the pattern is attempting a decomposition on a list of incorrect length. When a top-level declaration fails, the identifiers introduced in the declaration are bound to fault-values (section A.8).

The constructors **in_lft** and **in_rht** may be used to decompose disjoint sums:

```
::     let in_lft x = in_lft(in_rht 4);

>    x = (in_rht 4) :  (α + int)

::     let in_lft(in_rht x) = in_lft(in_rht 4);

>    x = 4 :  int

::     let in_lft x = in_rht 5;

>    x = (fault :  {decl}) :  α
```

Variant types may be decomposed by means of a variant pattern:

```
::     let ({Red x}) = {Red 3};

>    x = 3 :  int
```

Note that a pattern beginning with a variant requires the variant to be bracketed. This is to avoid ambiguity with complex declarations which are bracketed using '{' and '}'.

Patterns are also allowed within case expressions, where they serve to bind the value part of a variant so that it can be referred to within the expression:

```
::     case {Red(3, 4)} of
          {Red(x, _)}.   ~x;
          {Blue x}.      if x then 1 else 0
       end;

>    ~3 :  int
```

A.7 Functions

A function is a special type of data object which may be *applied* to a value to return a result. In addition, functions are are first class values within ℱ-HYBRID; they may be passed as arguments, returned as results, or embedded in data structures.

Function application is represented by writing a function expression next to an expression representing the function's argument. The entire expression denotes the result of applying the function to the argument. A function need not be denoted merely by an identifier; any expression may be applied to an argument, as long as that expression yields a function value.

A function may only be applied to one argument. To get the effect of passing several arguments to a function, a tuple of values is passed; alternatively, partial application may be used (section A.7). A function may be called with (intuitively) no arguments by passing the trivial value '()' as a single argument.

There are a number of built-in, *primitive* functions available in the ℱ-HYBRID system. Many of these are infix or prefix operators; infix and prefix notation is just a special syntax for function application:

```
::    1 + 2 + 3;
>   6 :  int
::    +(+(1, 2), 3);
>   6 :  int
::    let x = (1, 2) in +x;
>   3 :  int
```

The addition function (denoted by '+') is a data value in its own right, and need not be directly applied to arguments:

```
::    +;

>   λ :  ((int, int) → int)

::    [+; +; +];

>   [λ; λ; λ] :  ((int, int) → int) List

::    let Plus = + in Plus(1, 2);

>   3 :  int
```

Similarly, constructors such as in_lft and in_rht may be regarded as data values:

```
::    in_lft;

>   λ :  (α → (α + β))

::    (in_lft, in_rht);

>   (λ, λ) :  ((α → (α + β)), (γ → (δ + γ)))

::    let F = in_lft;

>   F = λ :  (α → (α + β))

::    F 3;

>   (in_lft 3) :  (int + α)
```

Note that the F has been introduced by means of a standard declaration; the name F has been bound to the (built-in) function represented by the constructor in_lft.

Function Declaration

Functions are defined in a straightforward manner. A function declaration features the function identifier followed by its parameter, which takes the form of a pattern:

```
::    let Square x = x * x
          and Add(x, y) = x + y
          and Mul3 [x; y; z] = x * y * z
          and Nothing(_) = ();

>    Square = λ :  (int → int)
>+   Add = λ :  ((int, int) → int)
>+   Mul3 = λ :  (int List → int)
>+   Nothing = λ :  (α → triv)
```

Since the formal parameter of a function is a pattern, it is possible for the pattern to fail to match the actual parameter to the function:

```
::    Mul3 [1; 2];

>    (fault :  {bind}) :  int
```

When this happens, a fault-value is generated (section A.8). A function may be declared in a *clausal* form where several alternative patterns are provided as formal parameters. The alternative clauses of a function are separated by the symbol '|'. The first pattern to match the parameter causes the corresponding function body to be evaluated:

```
::    let First nil = 0    |
          First(x : _) = x;

>    First = λ :  (int List → int)
```

A pattern may contain simple constant (such as an integer constant). An expression is also permitted as part of a pattern, if proceeded by the keyword `lit` (for *literal*). In this case, the corresponding part of the argument must have the same value as the expression:

```
::     let Ten = 10;

>    Ten = 10 :  int

::     let F Ten = () in (F 0, F 10);

>    ((), ()) :  (triv, triv)

::     let F(lit Ten) = () in (F 0, F 10);

>    ((fault :  {bind}), ()) :  (triv, triv)
```

The keyword `lit` may be omitted if its argument cannot be confused with a binding:

```
::     let F(Ten + 0) = () in (F 0, F 10);

>    ((fault :  {bind}), ()) :  (triv, triv)
```

It is possible for several alternative clauses of a function to match the actual argument; in this case, the *first* pattern matched causes the corresponding function body to be activated:

```
::     let Pair(x : x') = in_lft(x, x')    |
           Pair(_) = in_rht()
       in  Pair [1; 2; 3];

>    (in_lft (1, [2; 3])) :  ((int, int List) + triv)
```

Recursive Functions and Partial Application

By default, the body of a function will not be able to refer to the function itself. The name of the function will be unavailable, so that an attempt to use that name will result in that name being sought in the previous environment:

```
::     let Fact n = if n = 0 then 1
                    else n * Fact(n - 1);

Unbound identifier:  Fact
```

Recursive functions are introduced using the keyword `rec`. Mutually recursive functions may be defined using `and`:

```
::    let rec Fact n = if n = 0 then 1
                       else n * Fact(n - 1);

>   Fact = λ :  (int → int)

::    let rec List1 0 = nil    |
              List1 n = 1 : List2(n - 1)
              and List2 0 = nil    |
                  List2 n = 2 : List1(n - 1);

>   List1 = λ :  (int → int List)
>+  List2 = λ :  (int → int List)

::    Fact 7;

>   5040 :  int

::    List1 7;

>   [1; 2; 1; 2; 1; 2; 1] :  int List
```

Higher Order Functions

An expression of arbitrary complexity may evaluate to a function; such an expression may be applied directly to an argument:

```
::    (if 3 > 2 then List1 else List2) 7;

>   [1; 2; 1; 2; 1; 2; 1] :  int List
```

Since functions are first class objects in the language, they may be passed as arguments to other functions, and returned as results from functions:

```
::    let ApplyTo0 F = F 0;

>   ApplyTo0 = λ :  ((int → α) → α)

::    ApplyTo0 F where F x = x + 1;

>   1 :  int
```

ApplyTo0 takes a function as argument, and returns the result of applying that function to the value 0.

A common example of a higher order function is Map, which takes a function and a list as arguments, and returns the result of applying that function to each element of the list:

```
::    let rec Map(_, nil) = nil    |
              Map(F, x : x') = F x : Map(F, x');

>   Map = λ :  (((α → β), α List) → β List)

::    Map(F, [1; 2; 3]) where F x = [x; x * x];

>   [[1; 1]; [2; 4]; [3; 9]] :  int List List


::    let IncrBy n =
             let F x = x + n
             in  F;

>   IncrBy = λ :  (int → (int → int))

::    let Add5 = IncrBy 5
          and Add10 = IncrBy 10
      in  (Add5 2, Add10 2);

>   (7, 12) :  (int, int)

::    IncrBy 5 2;

>   7 :  int
```

The function IncrBy takes an integer as argument, and returns as result the function which adds that value to numbers. In the last example above, the expression IncrBy 5 returns a function which is then applied to the value 2. Note that function application associates from left to right; the above expression is regarded as (IncrBy 5) 2, and not as IncrBy (5 2).

Functions need not refer only to local identifiers, but may refer to any identifier which is in scope. In the example above, the local function F refers to a global identifier n (which is local to the containing function IncrBy).

The expression IncrBy 5 is referred to as a *partial application*, since it results in an functional value which can be further applied to another argument. Functions can also be defined using partial application, by providing a number of formal parameters in the function definition:

```
::    let IncrBy n x = x + n;
>   IncrBy = λ :  (int → (int → int))
::    IncrBy 5 2;
>   7 :  int
```

Functions may be denoted by a special form of expression, without having to be bound to an identifier. Such an expression is called a *lambda-expression*, and has a specific syntax:

```
::    lambda x. x + 1;
>   λ :  (int → int)
```

The keyword lambda is followed by the formal parameter. The body of the function follows the '.'. The expression above represents the function which adds 1 to its argument. Lambda expressions may be bound to identifiers in declarations; the following two declarations are therefore equivalent:

```
::     let F x = x + 1;

>    F = λ :  (int → int)

::     let F = lambda x. x + 1;

>    F = λ :  (int → int)
```

An expression containing lambda-expressions may be applied directly to an argument:

```
::     (lambda x. x + 1) 10;

>    11 :  int

::     (lambda F. F 0) (lambda x. x + 1);

>    1 :  int
```

A lambda-expression may be made to represent a higher order function, if several parameters are provided for partial application:

```
::     (lambda n x. x + n) 5 2;

>    7 :  int

::     (lambda n. lambda x. x + n) 5 2;

>    7 :  int
```

The two expressions above are equivalent; the first form of lambda-expression is regarded as an abbreviation for the second, nested, expression.

Like function definitions, lambda-expressions can be clausal:

```
::     (lambda x : x'. in_lft(x, x')    |
                   _. in_rht()) [1; 2; 3];

>    (in_lft (1, [2; 3])) :  ((int, int List) + triv)
```

Note that a lambda-expression cannot represent a recursive function, unless the expression is bound to an identifier within a `let rec` declaration, since there is no name attached to the expression by which it can refer to itself.

A.8 Exceptions and Fault-Values

𝓕–HYBRID is statically typed, and a number of checks are performed on programs during the compilation phase. This means that many classes of program fault are detected before the program reaches the execution phase. However, there are certain classes of error which cannot be detected by static, compile-time analysis, and these errors (or *exceptions*) have to be dealt with during the execution of a program.

Most modern languages have some form of error signalling and recovery mechanism, so that such exceptions can propagated through a program and dealt with by a suitable *exception handler*. 𝓕–HYBRID adopts an alternative scheme for generating and dealing with exceptions.

When an exception is generated, the error is not propagated back through the program. Instead, a *fault-value* is created, and execution of the program continues as before. Fault-values are considered to be first class objects in the language: they may be passed as arguments, returned as results, and embedded in data structures.

Fault-values are created under exceptional circumstances by primitive operators, and may also be created explicitly. Fault-values are also *propagated* by primitive operators and language constructs; an attempt to perform a primitive operation on a fault-value results in a new fault-value which holds a record of the operation attempted. A fault-value thus contains a list of the operations attempted on it.

A fault-value has an *underlying representation*, which is an object of type

```
{apply, bind, cond, decl, infer,
 prim :  str, signal :  str, uninst} List
```

There is a constructor (denoted by `fault`) which allows objects of this type to be converted into fault-values and back again. This underlying representation is a list of values, each one of which corresponds to an attempted operation on the fault-value. The last element of the list corresponds to the operation which created the fault-value in the first place; elements in front of it correspond to operations which have been performed subsequently. Each element is a variant structure, with each variant label denoting a particular class of fault.

A simple fault-value can be generated by a failure from one of the built-in operators:

```
::    let Fault1 = 1 / 0;

>   Fault1 = (fault :  {prim "/"})  :  int

::    let Fault2 = head nil;

>   Fault2 = (fault :  {prim "head"})  :  α
```

The underlying representation of such a fault is a list whose single element is a variant. This representation is accessed by using the constructor `fault`:

```
::    let (fault x) = Fault1;

>   x = [{prim "/"}]  :  {apply, bind, cond, decl, infer,
prim :  str, signal :  str, uninst} List

::    head x;

>   {prim "/"}  :  {apply, bind, cond, decl, infer, prim
:  str, signal :  str, uninst}

::    let (fault(x : x')) = Fault2;

>   x = {prim "head"}  :  {apply, bind, cond, decl, infer,
prim :  str, signal :  str, uninst}
>+  x' = []  :  {apply, bind, cond, decl, infer, prim
:  str, signal :  str, uninst} List
```

The variant `{prim}` denotes the fact that the fault-value was generated by a primitive operation; the variant value is a string representing the name of the operation itself. Primitive operations will also *propagate* fault-values, by attaching such a variant to (the representation of) a fault-value presented as an argument:

```
::    Fault2 + 1;

>   (fault :  {prim "+"}, {prim "head"}) :  int

::    Fault2 + 1 > 2;

>   (fault :  {prim ">"}, {prim "+"}, {prim "head"})
:  bool

::    not(Fault2 + 1 > 2);

>   (fault :  {prim "not"}, {prim ">"}, {prim "+"}, {prim
"head"}) :  bool
```

The variant {apply} is generated when an attempt is made to apply a fault-value as a function to an argument:

```
::    Fault2 1;

>   (fault :  {apply}, {prim "head"}) :  α

::    Fault2 Fault2 1;

>   (fault :  {apply}, {apply}, {prim "head"}) :  α
```

The variant {bind} is generated by a function when the argument it is applied to fails to match any pattern specified as formal parameter:

```
::    (lambda x : x'. false) nil;

>   (fault :  {bind}) :  bool

::    (lambda x : x'. false | nil. true) Fault2;

>   (fault :  {bind}) :  bool
```

A function will not necessarily fail when passed a fault-value as an argument;

in particular, any a fault-value passed as argument will match against a single identifier as formal parameter:

```
::    (lambda x. [x]) Fault2;

>   [(fault :  {prim "head"})] :  α List

::    (lambda x. x + 1) Fault2;

>   (fault :  {prim "+"}, {prim "head"}) :  int
```

Note in the second case above that the fault-value is propagated by the '+' operation, since it is passed unaltered as argument x.

A fault-value can be decomposed by means of the constructor `fault` used in a formal parameter; any non-fault passed as argument will fail to match against `fault`:

```
::    let as_fault(fault x) = x;

>   as_fault = λ :  (α → {apply, bind, cond, decl, infer,
prim :  str, signal :  str, uninst} List)

::    as_fault Fault2;

>   [{prim "head"}] :  {apply, bind, cond, decl, infer,
prim :  str, signal :  str, uninst} List

::    as_fault 3;

>   (fault :  {bind}) :  {apply, bind, cond, decl, infer,
prim :  str, signal :  str, uninst} List
```

The variant {cond} is generated when an attempt is made to use a fault-value to determine the value of a conditional or case expression:

```
::    if Fault2 then 1 else 2;

>   (fault :  {cond}, {prim "head"}) :  int

::    case Fault2 of
         {Red}.   1;
         {Green}. 2;
         {Blue}.  3
      end;

>   (fault :  {cond}, {prim "head"}) :  int
```

The variant {decl} is generated when a pattern used within a local declaration fails to match the value being bound:

```
::    x where x : x' = nil;

>   (fault :  {decl}) :  α
```

Note that top-level declarations which fail in this way cause the identifiers in the declaration to be bound to fault-values:

```
::    let Bad1 : Bad2 = nil;

>   Bad1 = (fault :  {decl}) :  α
>+  Bad2 = (fault :  {decl}) :  α List
```

The variants {infer} and {uninst} are generated by logic programs (appendix B). The variant {signal} is not generated by any of the primitive operators, but is provided for user-defined functions. Functions defined in library files generally use {signal(x)} for inappropriate arguments (where x is a string denoting the name of the function).

Fault-values are amenable to comparison (if their type is a meaningful comparable type), *but* such comparison always yields false; this may be interpreted as meaning that no two fault-values are ever equal. This property holds for patterns in formal parameters and declaration bindings; *literal* fault-values (specified with lit) will *never* match any provided argument:

```
::    Fault1 = Fault1;

>   false :  bool

::    (lambda lit Fault1. Fault1) Fault1;

>   (fault :  {bind}) :  int

::    (lambda fault x. fault x) Fault1;

>   (fault :  {prim "/"}) :  α
```

Fault-values may, however, be compared by performing comparison over the underlying representations:

```
::    as_fault Fault1 = as_fault Fault1;

>   true :  bool
```

It is possible to generate a fault-value whose underlying representation is the empty list:

```
::    fault nil;

>   (fault) :  α
```

Note also that the underlying representation of a fault-value may itself contain fault-values:

```
::    fault [{signal "a"}; Fault2; {signal "b"}];

>   (fault :  {signal "a"}, (fault :  {prim "head"}),
{signal "b"}) :  α

::    fault({signal "a"} : Fault2);

>   (fault :  {signal "a"} | (fault :  {prim "head"}))
:  α
```

Appendix B

A Tutorial for $\mathcal{L}$–HYBRID

B.1 Introduction

This appendix is an introduction to the nondeterministic language $\mathcal{L}$–HYBRID. For detailed information on the execution of concurrent logic programs, the reader is referred to chapter 4, and to texts concerning Concurrent Prolog [Shapiro 83b, Shapiro 83a].

B.2 Notation

The syntax for clauses and deductions bears some similarity to the various dialects of PROLOG; some familiarity with PROLOG is therefore assumed. However, $\mathcal{L}$–HYBRID adopts lexical conventions appropriate to a functional language; this implies, in particular, that logical variables are introduced by declaration, and not determined by case distinction.

B.3 Simple Clause Definitions

Unit Clauses

A simple clause definition consists of the keywords `let clause` followed by the clause name and argument:

```
::    let clause Is_Colour "Red";

>   Is_Colour = ψ :  str Clause

::    let clause Is_Black "Black"
                 and Is_White "White";

>   Is_Black = ψ :  str Clause
>+  Is_White = ψ :  str Clause
```

The first declaration introduces a clause called `Is_Colour`. In logical terms, `Is_Colour` is a property which is asserted to be true of the value `"Red"`. In semantic terms, the name `Is_Colour` is bound to a clause object of type `str Clause`. In operational terms, `Is_Colour` is a clause which attempts to unify its argument against the string literal `"Red"`.

The second declaration introduces two clauses named `Is_Black` and `Is_White`, both of type `str Clause`. The keyword `clause` may be viewed as a prefix declarative operator, with lower precedence than the infix `and`. It is therefore possible to make mixed (clausal and non-clausal) declarations:

```
::    let {clause Is_White "White"}
          and Is_Blue x = (x = "Blue")
          and {clause Is_Black "Black"}
          and Grey = "Grey";

>   Is_White = ψ :  str Clause
>+  Is_Blue = λ :  (str → bool)
>+  Is_Black = ψ :  str Clause
>+  Grey = "Grey" :  str
```

A clause takes a single argument. The effect of passing several arguments to a clause is achieved by passing a tuple of arguments. The effect of calling a clause with no arguments may be achieved by passing the value '()'.

A clause may have several alternative branches. These are expressed in a form similar to that used for functions, with the alternatives separated by '|':

```
::     let clause Is_Monochrome "Black"    |
                  Is_Monochrome "White";

>    Is_Monochrome = ψ :  str Clause
```

The property Is_Monochrome is asserted to be true of the value "Black", and also true of the value "White". There is a similarity to be drawn between the above clause definition and the function definition

```
::     let Is_Monochrome' "Black" = true    |
           Is_Monochrome' "White" = true    |
           Is_Monochrome'(_) = false;

>    Is_Monochrome' = λ :  (str → bool)
```

The clause Is_Monochrome will succeed if passed either of the values "Black" or "White". Similarly, the function Is_Monochrome' will return true for "Black" or "White", and false otherwise.

Non-Unit Clauses

The examples presented so far are of *unit clauses* (ie. clauses with no body). A non-unit clause has a set of query terms to the right of the inference symbol, ':-':

```
::    let clause Is_Monochrome x :- Is_Black x    |
                 Is_Monochrome x :- Is_White x;

>   Is_Monochrome = ψ :  str Clause

::    let clause Black_and_White(x, x') :-
                    Is_Black x, Is_White x';

>   Black_and_White = ψ :  (str, str) Clause
```

The clause `Is_Monochrome` features *disjunctive* (*or*-parallel) calls to `Is_Black` and `Is_White`; it succeeds if either of these goals succeeds. The clause `Black_and_White` features *conjunctive* (*and*-parallel) calls to `Is_Black` and `Is_White`; it succeeds only if *both* of these goals succeed. The names `x` and `x'` represent logical variables, used in this case to denote the formal parameters passed to the top-level clauses. Such variables are declared implicitly by occurrence within the head of the clause, in the same way that lambda-bound variables are declared within the formal parameter part of a function definition.

The anonymous logical variable is represented by the symbol '_':

```
::    let clause One_Black("Black", _)    |
                 One_Black(_, "Black");

>   One_Black = ψ :  (str, str) Clause

::    let clause Anything(_);

>   Anything = ψ :  α Clause
```

Clause declarations are, by default, non-recursive. This means that a clause declaration can access a previous clause of the same name. This provides the facility to "add alternatives" to a clause:

```
::    let clause Is_Colour "Blue"    |
                 Is_Colour c :- Is_Colour c;

>   Is_Colour = ψ :  str Clause
```

Because of the static binding rules of the language, however, note that any definitions which made use of the first definition of `Is_Colour` would be impervious to the second definition.

In a similar manner, it is possible to make "local" additions to a clause (in other words, to locally redefine the clause in terms of its more global counterpart), by means of a local declaration. Again, the local context could not use values defined in terms of the global clause, since these would be impervious to the local definition; such problems would have to be overcome by passing the newly-declared clause as an argument.

Recursive (and mutually recursive) clauses are defined in the same manner as recursive and mutually recursive functions: the declarative operator `rec` is used to enclose a set of declarations composed using `and`:

```
::     let rec clause Useless x :- Useless x;

>    Useless = ψ :  α Clause

::     let clause Useless_1 x
                  and rec Useless_2 x :- Useless_3 x
                          and Useless_3 x :- Useless_2 x;

>    Useless_1 = ψ :  α Clause
>+   Useless_2 = ψ :  β Clause
>+   Useless_3 = ψ :  β Clause
```

The operator `rec` may appear within clause declarations following `clause`; it serves to make that part of the declaration recursive. The operator `clause` may appear within a `rec`: it introduces a particular part of the recursive declaration to denote clauses.

Local Logical Variables

Any identifiers which occur freely within the head of a clause are accessible to the body of the clause. However, other logical variables may be explicitly declared for use in the body of the clause; such variables may be used to hold temporary values being used by the goals within the clause body. Local variables are listed at the end of the clause body, introduced by the keyword with and separated by commas:

```
::    let clause Never() :-
                     Is_Black x, Is_White x with x;

>   Never = ψ :  triv Clause
```

Each alternative part of a clause has a different set of local variables, and hence a different with-declaration:

```
::    let clause Always() :- Is_Black x with x   |
                 Always() :- Is_White y with y;

>   Always = ψ :  triv Clause
```

Clause Expressions

Clauses may be denoted by a particular form of expression, without having to be bound to an identifier. Such expressions are analogous to the lambda-expressions used to denote functions. The keyword clause introduces such a *clause expression*. The following two declarations are therefore equivalent:

```
::    let clause Both_Black(x, x') :-
                     Is_Black x, Is_Black x'
                 and Is_Grey "Grey";

>   Both_Black = ψ :  (str, str) Clause
>+  Is_Grey = ψ :  str Clause

::    let Both_Black = clause x, x' :-
                          Is_Black x, Is_Black x'
            and Is_Grey = clause "Grey";

>   Both_Black = ψ :  (str, str) Clause
>+  Is_Grey = ψ :  str Clause
```

The second declaration is simply binding two names to two clauses, each denoted by a clause-expression. Clauses are first class objects which may be bound directly by name, rather than explicitly using a clause declaration:

```
::    let Very_Dark = Is_Black;

>   Very_Dark = ψ :  str Clause

::    let Three_Clauses = [Is_Black; Is_Grey; Is_White];

>   Three_Clauses = [ψ; ψ; ψ] :  str Clause List

::    let [c1; c2; _] = Three_Clauses
      in  clause x :- c1 x, c2 x;

>   ψ :  str Clause
```

B.4 Queries

Query Expressions

A query expression represents the outcome of a logical deduction. A query expression may merely return the outcome of a deduction (success or failure), or may return results computed in logical variables. In the latter case, an unsuccessful query will result in a fault-value being generated.

The simplest form of query consists of the keyword **query** followed by a list of goal terms, separated by commas:

```
::    query Is_Black "Black";

>   true :  bool

::    query Is_White "Black";

>   false :  bool

::    query Is_Black(_), Is_White(_);

>   true :  bool
```

The result of such a query is a boolean value: `true` if the query was successful, `false` otherwise. In functional terms, the query expression evaluates to a boolean value; therefore, such an expression may be nested within other expressions:

```
::    [query Is_Black "Black"];

>   [true] :  bool List
```

The second form of query expression contains local logical variables, introduced using `with`:

```
::    query Is_Black x with x;

>   "Black" :  str

::    query Is_Black x, Is_White x' with x, x';

>   ("Black", "White") :  (str, str)
```

If one logical variable is declared in the `with`-part of the query, then a successful query returns the value instantiated within the logical variable. If two or more variables are declared, then the query returns a tuple of values, where each element of the tuple represents the value instantiated in the corresponding variable. If the query fails, then the query expression returns a *fault-value* containing a single variant {`infer`}:

```
::    "Colour: " ^ (query Is_Black x, Is_White x
                    with x);

>   (fault :  {prim "^"}, {infer}) :  str
```

Queries returning tuples of values can usefully be passed immediately to functions which take a tuple as argument:

```
::    let f(x, x') = x ^ " " ^ x'
      in  f(query Is_Black x, Is_White x' with x, x');

>   "Black White" :  str
```

Isolated logical programs can be embedded within functional programs by using a local clause declaration just within the scope of a query expression:

```
::    (let clause Is_White "Snow"
       in  query Is_White x with x) ^ " White";

>    "Snow White" :  str

::    query Is_White "Snow";

>    false :  bool
```

Since a clause takes a single argument which may be a tuple, it is possible for a single logical variable occurring as formal parameter to match a tuple of values as actual parameter, or vice versa. Contrast this behaviour with PROLOG:

```
::    let clause Nothing(_)
                 and Three_Nothings(_, _, _);

>    Nothing = ψ :  α Clause
>+   Three_Nothings = ψ :  (β, γ, δ) Clause

::    query Nothing(1, 2, false, "Hello"),
            Three_Nothings(_);

>    true :  bool
```

Query Declarations

It is possible to make the results of a deduction available as named identifiers by means of a conventional (possibly tupled) declaration:

```
::    let (b, w) = query Is_Black x, Is_White x'
                   with x, x';

>    b = "Black" :  str
>+   w = "White" :  str
```

However, the *query declaration* is more convenient way of naming the outcome of a query:

```
::    let query Is_Black b, Is_White w with b, w;

>    b = "Black" :  str
>+   w = "White" :  str
```

The syntax of the query is the same as for query expressions, but the names of the logical variables introduced by means of `with` are made accessible for subsequent (or local) use. Note that there is no *implicit* declaration of logical variables; any identifier which does not occur in the `with`-list of a query declaration is treated as a standard value binding:

```
::    let query Is_Black x, Is_Black x' with x;

Unbound identifier:  x'

::    let x' = "Black"
          ins query Is_Black x, Is_Black x' with x;

>   x = "Black"  :  str
```

The keyword `query` is a prefix declarative operator which introduces a query declaration. It is therefore possible to mix query declarations with other declarations:

```
::    let clause Is_Blue "Blue"
          and Is_Green "Green"
          enc query Is_Blue b with b
              and query Is_Green g with g
      in  (Is_Blue, b, Is_Green, g);

>   (ψ, "Blue", ψ, "Green")  :  (str Clause, str, str
Clause, str)
```

If a local query declaration *fails*, then the entire expression encompassing the declaration fails, and a fault-value (with variant {`decl`}) is generated. If a top level query declaration fails, fault-values are used in the bindings. In both cases, the behaviour is the same as that of a standard (functional) declaration which fails to pattern-match.

B.5 Complex Clauses and Constructors

Clauses may be defined with arbitrarily complex formal parameters, and queries may pass arbitrarily complex arguments to clauses. So far, the only argument types considered have been constants (for example, quoted strings), logical variables, and tuples. A clause may take the same sort of pattern as formal parameter as a function (although there is no equivalent to partial application). There is one important difference, however: a clause may contain repeated variables in its head:

```
::    let clause eq(x, x);

>   eq = ψ :  (α, α) Clause
```

Logically, the clause eq is true of any tuple of length 2 whose elements are identical (unifiable). Operationally, eq will attempt to unify its argument to a tuple of length 2, and then unify the first element with the second. Repeated variables may be present in any number, and according to any pattern (with the proviso that the clause argument has a legal type):

```
::    let clause Extr_Sum(in_lft x, x)    |
                 Extr_Sum(in_rht x, x);

>   Extr_Sum = ψ :  ((α + α), α) Clause

::    let clause Head_of(x : _, x);

>   Head_of = ψ :  (α List, α) Clause

::    let clause Foo({A x} : _, (x, in_lft y),
                     [x; [y]; x]);

>   Foo = ψ :  ({A : α List} List, (α List, (α + β)),
  α List List) Clause
```

Operationally, a clause with a repeated variable in its head can be expected to unify the two corresponding parts of an actual parameter. This is permitted so long as the type of the argument (determined when the clause is *called*) is a comparable type:

```
::    query Extr_Sum(in_lft 3, x) with x;

>   3 :  int

::    let query Head_of([[1; 2; 3]; nil], x),
                Head_of(x, y) with x, y;

>   x = [1; 2; 3] :  int List
>+  y = 1 :  int

::    query Head_of([(1, false, {A("Hi")}, [[3]])], x)
      with x;

>   (1, false, {A "Hi"}, [[3]]) :  (int, bool, {A : str},
int List List)
```

The clauses above are *invertible*; each clause will attempt to perform a unification between its formal and actual parameter, regardless of the (correctly typed) structure of the latter:

```
::    let query Head_of(x, 1), Head_of(y, x) with x, y;

>   x = [1 | _] :  int List
>+  y = [[1 | _] | _] :  int List List

::    let query Extr_Sum(x, (1, 2, 3)) with x;

>   x = (in_lft (1, 2, 3)) :  ((int, int, int) + (int, int,
int))
```

It is permissible to use the `fault` constructor within the head of a clause. However, due to the nature of the unification process and the representation of fault-values, the pattern within the `fault` cannot be merely a logical variable, but must be some pattern which represents a list:

```
::    let clause Extr_Fault(fault(x : _), x);
```

```
>   Extr_Fault = ψ :  (α, {apply, bind, cond, decl, infer,
prim :  str, signal :  str, uninst}) Clause
```

```
::    query Extr_Fault(head nil, x) with x;
```

```
>   {prim "head"} :  {apply, bind, cond, decl, infer, prim
:  str, signal :  str, uninst}
```

```
::    query Extr_Fault((x where x : _ = nil), x) with x;
```

```
>   {decl} :  {apply, bind, cond, decl, infer, prim :  str,
signal :  str, uninst}
```

B.6 Goal Structure

A goal consists of two parts: a clause to be activated, and a pattern to be passed to it as argument. The syntax for a goal is identical to the syntax for function application; the clause is written beside its argument, with the assumption of an implicit infix "clause application" operator between the two. The clause may be denoted by an arbitrary expression (of appropriate type); such an expression may contain further applications, but these are assumed to denote conventional function calls:

```
::    query (if true then Is_Black
                      else Is_White) x with x;

>   "Black" :  str

::    let F true = Is_Black    |
          F false = Is_White
      in  query F true x with x;

>   "Black" :  str
```

A clause expression may evaluate to a fault-value; in this case, the goal is considered to immediately fail, regardless of the argument:

```
        ::    query (head nil) (_);

        >   false :  bool
```

Literals

Within a logical deduction, is it possible to evaluate function expressions, and use such values to direct the progress of the deduction. Expressions may appear within arguments to clauses, and may be introduced within the head of a clause, preceded by the keyword `lit`:

```
        ::    let clause Is_Six 6
              in  query Is_Six(Double 3 where Double x = x + x);

        >   true :  bool

        ::    let s = 6
                  ins clause Is_Six(lit s)
              in  query Is_Six x with x;

        >   6 :  int
```

Note that the logical variables within a clause are *not* visible within literals in the head of the clause. This is to avoid confusion between the occurrences of identifiers which serve to declare logical variables, and those which serve to access them.

```
        ::    let clause Incr(i, i + 1);

        Unbound identifier:  i

        ::    let i = 3
                  ins clause Incr(i, i + 1);

        >   Incr = ψ :  (α, int) Clause
```

The (implicit) literal `i + 1` is evaluated, outside the scope of the logical variables, to a value which is then unified against; it contains no structural information, and is not further decomposed for the purposes of unification.

It is also possible to treat functional expressions as goals. If an expression evaluates to a boolean, then a value of **true** can be treated as *success*, and **false** can be treated as *failure*. Syntactically, any goal expression which cannot be regarded as a clause activation (for example, a constant, or a single identifier) is treated as a boolean expression. Additionally, any goal term enclosed in brackets is treated as a boolean expression:

```
::   let F t_val = clause (_) :- t_val;
>   F = λ :  (bool → α Clause)
::   query (F false) (_);
>   false :  bool
::   query (F true) (_);
>   true :  bool
::   x where query Is_Black x, (1 > 0) with x;
>   "Black" :  str
::   x where query Is_Black x, (1 < 0) with x;
>   (fault :  {decl}) :  str
```

Such *goal literals* may access logical variables; in these cases, the variables should be marked with the *read-only* marker, '?', for reasons which will become apparent later (section B.7). Any logical variable which occurs within a function expression in a goal (for example, in an expression evaluating to a clause) should be marked in a similar manner:

```
::    let clause Greater_Than(x, y) :- (x? > y?);

>   Greater_Than = ψ :  (int, int) Clause

::    let clause Is_True x :- x?;

>   Is_True = ψ :  bool Clause

::    let clause Foo(x, y) :- (if x?
                                then Is_Black
                                else Is_White) y;

>   Foo = ψ :  (bool, str) Clause

::    let rec Fact 0 = 1   |
              Fact n = n * Fact(n - 1)
          ins clause Fact(x, y) :- eq(Fact x?, y);

>   Fact = ψ :  (int, int) Clause
```

If a goal literal evaluates to a fault-value, the goal fails:

```
::    let clause Try_Head x :- (head x?);

>   Try_Head = ψ :  bool List Clause

::    query Try_Head(true : _);

>   true :  bool

::    query Try_Head nil;

>   false :  bool
```

B.7 Higher Order Functions and Higher Order Clauses

Operationally, a clause is treated as a special kind of function. It takes a special kind of argument (which may be a value, or may be some pattern of logical variables), and returns a special kind of result (*success* or *failure*). $\mathcal{L}$–HYBRID allows clauses to be treated as first-class objects; they may be passed as arguments, returned as results, and embedded in data structures. Functions may be higher-order over clauses, in the sense that they may takes clauses as arguments, or return clauses as results. Clauses themselves may be higher-order, since they may be passed argument patterns containing functions or clauses.

A clause may be passed as argument directly if it is denoted by a clause expression:

```
::    let Triple x = [x; x; x]
      in  Triple (clause a :- (a? > 0));

>   [ψ; ψ; ψ]  :  int Clause List

::    query (head it) 1;

>   true :  bool
```

A clause passed as a parameter to a function may be activated by means of a query expression or declaration within that function:

```
::    let Try C = query C [x; x; x] with x
      in  Try(clause 1 : _);

>   1 :  int
```

A function may also return a clause as result:

```
::    let Map_Clause C =
         let rec clause Map_C nil    |
                          Map_C(x : x') :- C x, Map_C x'
         in Map_C;

>   Map_Clause = λ :  (α Clause → α List Clause)

::    query (Map_Clause Fact) [(5, a); (4, b); (3, 6)]
      with a, b;

>   (120, 24) :  (int, int)

::    query (Map_Clause(Map_Clause Fact))
            [[(3, x)]; [(x, y)]]
      with x, y;

>   (6, 720) :  (int, int)
```

A clause may be passed a function as (part of) an argument; the function may be called within a literal goal, or in a clause argument within a goal:

```
::    let clause Same(F, x) :- eq(F? x?, x);

>   Same = ψ :  ((α → α), α) Clause

::    query Same((lambda x. x * x), 1);

>   true :  bool
```

A clause may be passed a clause as argument; the clause may then be invoked as one of the goals within the clause:

```
::    let clause ApplyToZero c :- c? 0;

>   ApplyToZero = ψ :  int Clause Clause

::    query ApplyToZero(clause x :- (x? > 0));

>   false :  bool
```

The guard system may consist of the single keyword `otherwise`; this may only occur within a clause which has more than one alternative branch, and only occur once within this clause. Such a guard will succeed if and only if all other guard systems fail.

The *otherwise* notation allows negation to be expressed by failure without resorting to a higher-order or nested query:

```
::    let clause Not_eq(x, x) :- false    |
                 Not_eq(_) :- otherwise \ true;

>   Not_eq = ψ :  (α, α) Clause

::    query Not_eq(1, 2);

>   true :  bool

::    query Not_eq(1, x) with x;

>   (fault :  {infer}) :  int
```

The use of otherwise, together with the facility to "add" definitions to a clause (by defining a new instanee of it in terms of an old one), it is possible to define a clause in which alternatives are sought in a strict sequential order:

```
::    let clause {C "Match 3"
                  ins C "Match 2"    |
                      C x :- otherwise \ C x
                  ins C "Match 1"    |
                      C x :- otherwise \ C x};

>   C = ψ :  str Clause
```

Atomicity of Unification

It is possible for a number of processes to attempt to unify against the same term, as illustrated in the following definition and use of the clause Clash:

```
::    let clause Clash(1, 2)    |
                 Clash(2, 2)
      in  query Clash(x, x) with x;

>   (fault :  {infer}) :  int
```

It would be expected that the first alternative would fail, and the second alternative would succeed. However, it is possible that the first alternative can instantiate the variable x to the value 1 before failing; in this eventuality, the second alternative (and the query as a whole) will also fail. Since there is this unprotected sharing between logical variables, it is necessary to clarify the way in which any particular unification is performed, since a gradual unification by one process can affect the execution of another.

Unification order can be summed up in the following two rules:

- Unification is made against a structure before being performed for elements of that structure;
- At a particular level of a structure, unification is performed from left to right.

This means that, for example, tuples are unified from the first element to the last, and lists are unified from head to tail.

The procedure of unifying two terms consists of a number of primitive unification operations. The unification operation, which consists of verifying that a variable is free, and then instancing it to a term, is considered to be atomic. This means that a variable cannot be instantiated to one term by one process, and then instantiated to another term by another process, in quick succession. Any unification consists of one or more of these primitive operations. Each primitive unification is atomic, but the unification as a whole may not be.

Logical Variables

A logical variable is treated as a binding of a name to an object with a rather strange behaviour. In most cases, the behaviour of a logical variable is unimportant, and the variable can just be regarded as a data value:

```
::    let query (clause 3) x with x;
>   x = 3 :  int
```

The above declaration has the effect of binding the value 3 to the name x; in other words, it has the same effect as the declaration `let x = 3`. The fact that x is a logical variable which becomes unified to the value 3 is not important, since the unification is complete by the time the declaration is established.

It is possible for a query term or declaration to leave logical variables uninstantiated:

```
::    let query (clause (_)) x with x;

>   x = _ :  α
```

In functional terms, a free variable is treated as the fault-value

```
(fault : {uninst})
```

As with other fault-values, it may be passed unaltered as argument and embedded in data structures, and will cause fault propagation if an attempt is made to access its value:

```
::    if x then 1 else 2;

>   (fault :  {cond}, {uninst}) :  int
```

Within functional programs, a free logical variable will match the fault-value with variant {uninst} (section D.1.2). Functional access to a free variable does not make the variable into a fault-value; it is just treated as such.

```
::    (lambda (fault [a]). a) x;

>   {uninst} :  {apply, bind, cond, decl, infer, prim
:  str, signal :  str, uninst}
```

Read-Only Variables

Within queries, it is often necessary to mark logical variables as read-only. In operational terms, this means that the process making a read-only access to the variable is suspended until the variable is instantiated by another process. Since goals may contain functional expressions (for example, as parts of arguments, or by virtue of being literal goals), functional parts of programs can be invoked with logical variables which have still to be instantiated by another process in the query. As mentioned above, the values of such variables would just be seen as fault-values. For this reason, all logical variables occurring in a functional expression within a query are forced to be read-only:

```
::    clause x :- (x > 0);

?Assuming read-only annotation:  x
>   ψ :  int Clause

::    clause C :- C 0;

?Assuming read-only annotation:  C
>   ψ :  int Clause Clause
```

In addition, any identifier within a functional program may be marked as read-only. Any logical process which activates this program as part of a goal will be suspended until the variable has been instantiated by another process. Since logical variables occurring within expressions in goals are forced to be read-only anyway, this facility is rarely necessary, but deals with the case where a function is passed a structure which contains free variables.

$\mathcal{L}$–HYBRID features another "layering" facility (apart from guard systems). From within logical queries it is possible to activate functional program fragments. Each such fragment can contain other clause definitions and queries. It is, therefore, possible have a deductive system parts of which are comprised of other deductive systems:

```
::    let clause Both(true, true)
                 and Either(true, _)    |
                     Either(_, true);

>   Both = ψ :  (bool, bool) Clause
>+  Either = ψ :  (bool, bool) Clause

::    query Either((query Both(false, true)),
                   (query Either(false, true)));

>   true :  bool
```

The above example shows one way of achieving logical-*or* within a conjunctive system. A less obscure method might be to use the functional operator or within a goal literal:

```
::    query ((query Is_Black x) or (query Is_White x))
      with x;

>   "Black" :  str
```

Note that it is possible to perform non-local unifications within such layered queries; there is no safeguard against a goal instancing a variable which is declared in a more global deductive system. In the example above, both of the sub-goals are attempting to instantiate a global logical variable. If such an effect is undesirable, then the appropriate variables should be annotated as read-only.

Negation can be expressed by use of the functional operator not within a goal literal:

```
::    query (not(query Is_White x)) with x;

>   (fault :  {infer}) :  str
```

In this example, Is_White unifies x and the inner query returns true. The literal goal of the enclosing query evaluates to false, failing the enclosing query and giving rise to the fault-value.

Appendix C

HYBRID Syntax

The meta-syntax used in this appendix is based upon that used in [Cardelli 81]. Identifiers denote non-terminals. Character sequences between quotes ("...") represent terminals; words within quotes are reserved. '|' is syntactic alternative. Square brackets are used to group subphrases. '?' denotes optional items. '*' denotes items to be repeated 0 or more times. '+' denotes items to be repeated 1 or more times. '/' is used to represent the separation of a number of occurrences of one phrase by another; for example, the phrases '$[\alpha\ /\ \beta]^{+}$' and '$\alpha\ [\beta\ \alpha]^{*}$' are equivalent.

```
TopTerm       →  ["diagnose" [[Ident | "*"] ["on" | "off"]? / ","]+
                  | "include" Str
                  | "let" FullDecl ["in" InnerTerm]?
                  | FullTerm
                 ] ";"

FullDecl      →  [InnerDecl / ["ins" | "enc"]]+

InnerDecl     →  "rec" InnerDecl
                 | Def ["and" InnerDecl]*

Def           →  "clause" FullClauseDef
                 | "query" Goals ["with" IdentList]?
                 | "{" FullDecl "}"
                 | [FnBind "=" FullTerm / "|"]+
                 | FullBind "=" FullTerm
```

```
FnBind          →  Prefix FullBind+
                   | FullBind Infix FullBind+
                   | FullBind Postfix FullBind*
                   | Ident FullBind+


FullClauseDef   →  "rec" FullClauseDef
                   | InnerClauseDef ["and" FullClauseDef]*


InnerClauseDef  →  "{" [FullClauseDef / ["ins" | "enc"]]+ "}"
                   | [ClauseBind [":-" ClauseBody]? / "|"]+


ClauseBind      →  Prefix FullBind
                   | FullBind Infix FullBind
                   | FullBind Postfix
                   | Ident FullBind


ClauseBody      →  [Goals "\" | "otherwise" "\"]? Goals
                   ["with" IdentList]?


Goals           →  [ [Prefix InnerBind
                     | InnerBind Infix InnerBind
                     | InnerBind Postfix
                     | SimpleTerm InnerBind
                     | SimpleTerm
                    ] / ","
                   ]+


FullBind        →  [InnerBind / ","]+


VariBind        →  Prefix InnerBind
                   | InnerBind Infix InnerBind
                   | InnerBind PostFix
                   | Ident InnerBind?


InnerBind       →  ["(" FullBind? ")"
                    | "[" [[FullBind / ";"]+ "|" FullBind]?]? "]"
                    | "{" VariBind ["|" IdentList]? "}"
                    | SimpleTerm
                    | Ident
                    | "_"
                   ] [":" InnerBind]?
```

```
FullTerm      →  "let" FullDecl "in" InnerTerm
                 | InnerTerm ["where" FullDecl]?

InnerTerm     →  ["if" InnerTerm "then" FullTerm]+ "else" FullTerm
                 | "case" InnerTerm "of"
                  ["{" VariBind "}" "." FullTerm / ";"]+
                  "end"
                 | "lambda" [FullBind+ "." FullTerm / "|"]+
                 | "clause" [FullBind [":-" ClauseBody/]? / "|"]+
                 | "query" Goals ["with" IdentList]?
                 | [SimpleTerm / ","]+

SimpleTerm    →  Prefix SimpleTerm
                 | SimpleTerm Infix SimpleTerm
                 | SimpleTerm Postfix
                 | SimpleTerm SimpleTerm
                 | ScannerTerm

ScannerTerm   →  ["(" FullTerm? ")"
                  | "[" [[FullTerm / ";"]+ ["|" FullTerm]?]? "]"
                  | "{" SimpleTerm ["|" IdentList]? "}"
                  | AtomicTerm
                 ] [["is" | "as"] Ident]*

IdentList     →  [Ident / ","]+

AtomicTerm    →  Ident "?"? | Int | Str | "_"

Ident         →  Letter [Letter | Digit | "_" | "'"]*

Prefix        →  "not" | "in_lft" | "in_rht" | "~" | "fault"

Infix         →  "or" | "&" | "=" | "<>" | "^" | "::" | ":" | "<" | "<=" | ">="
                 | ">" | "-" | "+" | "/" | "*" | "mod" | "<<" | ">>" | "o" |

Postfix       →  "++"

Int           →  Digit+

Str           →  """ ...sequence of characters... """

Letter        →  "A" ... "Z" | "a" ... "z"

Digit         →  "0" ... "9"
```

Appendix D

The HYBRID Abstract Machine Code

The HYBRID abstract machine is a concurrent control flow architecture. There is a global address space of fixed size memory cells, and a number of processes execute code and access data in this common address space.

There is a solitary mechanism for communication and synchronisation between processes: a simple message passing procedure is supported on memory locations in the global address space. Memory cells can be assigned a unique value (*empty*); any subsequent attempt by a process to access such a location causes the process to be suspended until the location is assigned a non-*empty* value (by some other process). This mechanism allows processes to communicate with one another, and also acts as a synchronisation primitive: one process can suspend itself, and be resumed by another process performing an assignment to memory. With the exception of this communication mechanism, no guarantees are made against interference between processes accessing the same location in memory.

D.1 The Abstract Machine

D.1.1 Memory Structure

The abstract machine has a global, flat address space of fixed size memory cells. Each memory cell may be *empty*, or may contain a *simple value* (such as an integer), or a *pointer value* (an address of a store location). The machine believes in a variety of data types. Simple values can be of integer type, or of boolean type (with value `true` or `false`), or of a special void type (with value *void*). Pointer values may be references to static objects (portions of code, or strings of characters), or to dynamic objects (blocks of store). It is assumed that pointer values can be distinguished from *void.*

The abstract machine is not intended to be type-secure: the result of performing operations on objects of inappropriate type is undefined. It is assumed that typechecking will be performed statically, at compile-time.

Each process has a small set of private, special purpose registers, and also has a small area of stack space, used for storing temporary and intermediate values. In addition, a process can claim vectors of storage from a dynamic heap, which can then be used to build structured data types. When a process is created, it is allocated a section of store to use for stack space. Such space may be reclaimed when the process terminates.

Heap storage is claimed in blocks of predetermined size during the execution of a program. A heap allocation by a process results in that process being given a pointer to a block of store of fixed size, all elements of which are *empty.* These elements can be assigned simple or pointer values, and the address of the block can be passed between different processes. A block may be reclaimed when no process possesses a pointer to it.

A program is initially executed by a single process. However, this process may dynamically create other processes to assist in the computation, and each such process may create more processes, and so on. A child process is completely independent of its parent, and may continue to run when the parent terminates. The total number of processes running on the abstract machine at any one time is considered unlimited, and it is assumed that an attempt to create a new process will always succeed.

D.1.2 Fault-Values

Fault-values are reserved kinds of data object that are treated specially by the HYBRID abstract machine. Fault-values may be generated explicitly, and in addition are generated implicitly by certain built-in operations.

Every fault-value has as *underlying representation.* Although a fault-value may have any type, it may be converted to and from an object of type

```
{apply, bind, cond, decl, infer,
 prim : str, signal : str, uninst} List
```

The constructor `fault` takes an object of this type, and generates the corresponding fault-value. The built-in function `as_fault` converts a fault-value into its representation, as illustrated in the example below:

```
::    fault [{uninst}; {signal "S"}];

>   (fault :  {uninst}, {signal "S"}) :  α

::    as_fault(head nil);

>   [{prim "head"}] :  {apply, bind, cond, decl, infer,
prim :  str, signal :  str, uninst} List
```

Fault-values are propagated by many primitive operations. Such propagation (referred to as *chaining*) results in a new variant object being "consed" to the front of the list representing the fault-value. For example:

```
::    (head nil);

>   (fault :  {prim "head"}) :  α

::    (head nil) + 1;

>   (fault :  {prim "+"}, {prim "head"}) :  int

::    (head nil) + 1 > 0;

>   (fault :  {prim ">"}, {prim "+"}, {prim "head"}) :  bool
```

Primitive operators (such as '+' and '>') perform this chaining operation implicitly when presented with fault-values. In other cases, the instruction ChainFault is used to perform chaining (for example, in conditional and case expressions). The operation

ChainFault n

where n is an integer $\geq$ 0, takes two arguments on the stack. The first argument (Local/1) is assumed to be a fault-value. The second argument (Local/0) is expected to be a value to be built into a variant cell with a variant tag represented by n. For example, given a fault-value

(fault : {apply})

and an argument e, the result of ChainFault n will be a fault-value of the form

(fault : {tag_n e}, {apply})

with the tag label tag_n determined by the value of n.

In a functional context, uninstantiated logical variables are treated as fault-values with a variant {uninst}; functional programs are said to *manifest* uninstantiated variables into this fault-value, although the variable itself remains unchanged. For example, the expression

```
    ::    let query (clause (_)) x with x
          in  x + 3;

    >   (fault :  {prim "+"}, {uninst}) :  int
```

results in the logical variable x being *manifested* to a fault-value with variant {uninst}, and *chained* with the variant {prim "+"}.

D.1.3 Registers

Every executing process has a set of private, special purpose registers. Each is capable of holding a simple or pointer value. Registers cannot be made *empty*, and cannot, for obvious reasons, be assigned *empty* from a store location. Such registers are not accessible to any other process. However, when one process creates another, the child process may inherit the values of one or more of its parent's registers.

The registers are named, and used, as follows:

`R_Local` is used as a stack pointer, and always points to the top of a process's stack space;

`R_Global` is a pointer to values in the environment of a process (the *closure*);

`R_Func` is used to pass function and clause closures from a parent process to a child;

`R_Arg` is used to pass arguments to functions and clauses;

`R_Result` is used to pass results back from functions and method calls;

`R_And` is used to store *and*-cells between conjunctive systems;

`R_Or` is used to store *or*-cells between disjunctive systems;

`R_Method` is used to pass method information to a process created for a method unification;

R_Caller is used to store a process context when an Apply is executed. The context is restored on encountering a termination instruction such as Stop;

R_Program is the program counter.

D.2 Abstract Instruction Set

Each instruction is written as a single-word name (eg, Jump), possibly followed by one or more arguments. A majority of the abstract machine instructions take arguments from the local stack. Zero-address instructions (for example, the arithmetic and comparison operators), take arguments from the top of the stack. Some instructions take arguments from an arbitrary point in the stack, referred to by an integer offset (≥ 0); the top of the stack has offset 0. The most general addressing mode makes use of an index register (for example, the stack pointer, R_Local), and a number of offsets. The effective address is calculated by advancing the pointer value of the register by the first value, dereferencing, advancing this pointer value by the second offset, and so on. As an example, the address Local/1/2 is equivalent to the expression Local[1][2] in the language C. Such *address vectors* may be of arbitrary (fixed) length.

Destinations of jumps are written Ln, where n is an integer > 0. Instructions may be labelled (eg. L3: Move ...), although not all labelled instructions are required to be destinations of jumps.

Each abstract instruction is defined to be atomic. This avoids race conditions in the case of the logical unification and synchronisation instructions, although the assignment rules laid out in section 2.3.1 make the condition unnecessary in most cases where a stack access is not involved.

In the descriptions which follow, *Src* is used to represent the stack offset of an instruction argument. *Dest* is the stack offset of an assignable destination. *VSrc* and *VDest* are vector equivalents of *Src* and *Dest* respectively; each represents a named register together with a vector of offsets. *Reg* represents a named register. *Lab* represents an instruction label. *n* is used to represent a literal integer argument (≥ 0). *s* is used to represent a literal quoted string.

The terminology "containing a structure" means "containing a pointer to a structure." If an instruction is described as encountering a fault-value, this should be taken to mean that *uninstanced* values are manifested beforehand.

`Inflate` *n*: Inflate the local stack by *n* elements. The top element is preserved by copying. After the inflation, cells `Local/1` to `Local/`*n* are *empty*.

`Deflate` *Size*, *n*: Deflate the local stack by *n* elements. The top element is preserved by copying.

`InflateEnv` *n*: Inflate the environment, by treating `R_Result` as a stack pointer. The top element (`Result/0`) is preserved by copying. After the inflation, cells `Result/1` to `Result/`*n* are *empty*.

`Rise` *n*: Increase amount of local stack accessible by decreasing `R_Local`. The top of stack (`Local/0`) becomes addressable as `Local/`*n*. Cells `Local/0` to `Local/`$n-1$ are *empty* after the `Rise`.

`Fall` *n*: Decrease amount of local stack accessible by increasing `R_Local`. The cell addressable as `Local/`*n* becomes `Local/0`.

`TestTrue` *SrcOffset*, *Label*: Examine `Local/`*SrcOffset*. If `true`, control is transferred to *Label*. If `false`, or a fault-value, there is no effect.

`TestFalse` *SrcOffset*, *Label*: Examine `Local/`*SrcOffset*. If `false`, control is transferred to *Label*. If `true`, or a fault-value, there is no effect.

`TestVariant` *Tag*, *SrcOffset*, *Label*: Regards `Local/`*SrcOffset* as a variant structure. If `Local/`*SrcOffset*`/0` has integer value *Tag*, control is transferred to *Label*. If `Local/`*SrcOffset* is a fault-value, or `Local/`*SrcOffset*`/0` has any other integer value, there is no effect.

`TestNil` *SrcOffset*, *Label*: Transfers control to *Label* if `Local/`*SrcOffset* is *void*. Has no effect if `Local/`*SrcOffset* is a structure, or a fault-value.

`TestCons` *SrcOffset*, *Label*: Transfers control to *Label* if `Local/`*SrcOffset* is a structure. Has no effect if `Local/`*SrcOffset* is *void*.

`TestLft` *SrcOffset*, *Label*: Regards `Local/`*SrcOffset* as a sum structure. If `Local/`*SrcOffset*`/0` has boolean value `false`, control is passed to *Label*. If `Local/`*SrcOffset* is a fault-value, or `Local/`*SrcOffset*`/0` has boolean value `true`, there is no effect.

`TestRht` *SrcOffset*, *Label*: Regards `Local/`*SrcOffset* as a sum structure. If `Local/`*SrcOffset*`/0` has boolean value `true`, control is passed to *Label*. If `Local/`*SrcOffset* is a fault-value, or `Local/`*SrcOffset*`/0` has boolean value `false`, there is no effect.

`TestFault` *SrcOffset*, *Label*: *Manifests* `Local/`*SrcOffset*. If the result is a fault-value, control is transferred to *Label*. Otherwise, there is no effect.

`ExtractVariantTag` *SrcOffset*, *DestOffset*: Regards `Local/`*SrcOffset* as a variant structure. Copies `Local/`*SrcOffset*`/0` to `Local/`*DestOffset*.

`ExtractVariantValue` *SrcOffset*, *DestVec*: Regards `Local/`*SrcOffset* as a variant structure. Copies `Local/`*SrcOffset*`/1` to *DestVec*.

`ExtractTuple` *SrcOffset*, *Index*, *DestVec*: Regards `Local/`*SrcOffset* as a tuple structure. Copies `Local/`*SrcOffset*`/`*Index* to *DestVec*.

`ExtractHead` *SrcOffset*, *DestVec*: Regards `Local/`*SrcOffset* as a *cons*-cell. Copies `Local/`*SrcOffset*`/0` to *DestVec*.

`ExtractTail` *SrcOffset*, *DestVec*: Regards `Local/`*SrcOffset* as a *cons*-cell. Copies `Local/`*SrcOffset*`/1` to *DestVec*.

`ExtractSumTag` *SrcOffset*, *DestOffset*: Regards `Local/`*SrcOffset* as a sum structure. Copies `Local/`*SrcOffset*`/0` to `Local/`*DestOffset*.

`ExtractSumValue` *SrcOffset*, *DestVec*: Regards `Local/`*SrcOffset* as a sum structure. Copies `Local/`*SrcOffset*`/1` to *DestVec*.

`ExtractFault` *SrcOffset*, *DestVec*: *Manifests* `Local/`*SrcOffset*. If `Local/`*SrcOffset* is a fault-value, then assigns *DestVec* the underlying representation. Otherwise, the effect is undefined.

`Triv`: Pushes the *void* value representing '`()`' onto the stack.

`Int` *n*: Pushes integer value *n* onto the stack.

`Str` *s*: Pushes string value *s* onto the stack.

`Copy` *DestVec*: Copies top of stack to *DestVec*; the stack remains unchanged.

`CopyReg` *Reg*: Copies top of stack into register *Reg*; the stack remains unchanged.

`Move` *DestVec*: Copies top of stack to *DestVec*; stack falls by one.

`MoveReg` *Reg*: Copies top of stack into register *Reg*; stack falls by one.

`From` *SrcVec*: Pushes value of *SrcVec* onto stack.

`FromReg` *Reg*: Pushes contents of register *Reg* onto stack.

`Read` *SrcVec*: Pushes value of *SrcVec* onto stack. Process will suspend if *SrcVec* is *uninstantiated.*

`Process`: Creates a process to execute the closure contained in `R_Func`. The new process is allocated a stack of size `Func/2`, and executes the code in `Func/0`. The register `R_Global` is assigned from `Func/1`. All other registers are inherited from the parent.

`VecProcess -> `*DestVec*: Similar to `Process`, except that the new process has register `R_Result` referencing *DestVec*. Also accommodates the case where `R_Func` is a fault-value. In this case, fault tag {`apply`} is chained to the value of `R_Func`, and this value is assigned to *DestVec* without a process being created.

`PushProcess`: Similar to `Process`, except that the new process has register `R_Result` referencing an *empty* cell pushed onto the parent's stack. Accommodates a fault-value in `R_Func`, as above.

`Apply`: Similar to `PushProcess`. Instead of a new process being created, the current process saves its register values in a block claimed from the heap; a reference to this block is then placed in `R_Caller`. The current process then adopts the state of the new process. Accommodates a fault-value in `R_Func`, as above.

`VariantCell` *Tag*: Pushes a structure of two elements onto the stack. The first element has integer value *Tag*. The second is *empty*.

`TupleCell` *Size*: Pushes a structure of *Size* elements onto the stack. All elements are *empty*.

`ConsCell`: Pushes a structure of two elements onto the stack. Both elements are *empty*.

`LftCell`: Pushes a structure of two elements onto the stack. The first element has value `false`; the second is *empty*.

`RhtCell`: Pushes a structure of two elements onto the stack. The first element has value `true`; the second is *empty*.

`ReturnTrue`: Assigns `true` to the result cell `Result/0`.

`ReturnFalse`: Assigns `false` to the result cell `Result/0`.

`Result`: Copies the top of the stack to the result cell `Result/0`.

`Stop`: If `R_Caller` is *void*, then terminate the current process. Otherwise, restore the process context in `R_Caller`. In either case, the current stack is discarded.

`SetResult` *DestVec*: Assign `R_Result` the address of the cell referenced by *DestVec*.

`TailApply`: If `R_Func` is a fault-value, then chain fault tag {`apply`} to it, copy this value to `Result/0`, and `Stop`; otherwise, discard the stack, and begin execution of the closure in `R_Func`, as for `Process`.

`Code` *Size* `[`*Code*`]`: Push a closure onto the stack. A structure of three elements is claimed from the heap. The first element is assigned a reference into the code area, to the first instruction of `[`*Code*`]`. The second element is left *empty*. The third element is assigned integer value *Size*.

`Block` *Size*: Synonymous with `TupleCell` *Size*; push a structure of *Size* elements onto the stack. All elements are *empty*.

`Closure` *DestVec*: Assumes `Local/0` to be a block of values for the closure at *DestVec*, or a *void* value (if the closure accesses no free variables). Copies `Local/0` into *DestVec*`/1`, and drops the stack by one element.

`NullClosure` *DestVec*: Assumes *DestVec* to reference a closure accessing no free variables. Assigns *DestVec*`/1` to be *void*.

`True`: Pushes value `true` onto the stack.

`False`: Pushes value `false` onto the stack.

`Not`: If top of stack is a fault-value, then chains fault tag {`prim "not"`} to it; otherwise, performs boolean *not* on the top of the stack.

`Neg:` If top of stack is a fault-value, then chains fault tag {`prim "~"`} to it; otherwise, performs boolean *not* on the top of the stack.

`And:` Takes two boolean values. If either of `Local/1` and `Local/0` is a fault-value, then remove both and return {`prim "&"`} chained to the fault-value; if both are fault-values, then use the fault-value `Local/1`. Otherwise, remove both and push their logical *and*.

`Or:` Takes two boolean values. Replaces `Local/1` and `Local/0` by logical *or*; if either is a fault-value, chains {`prim "or"`}, as above.

`GT:` Takes two integer values. Replaces `Local/1` and `Local/0` by boolean value `true` (`Local/1` > `Local/0`), or `false` (`Local/1` ≤ `Local/0`); if either is a fault-value, chains {`prim ">"`}, as above.

`LT:` As for `GT`, but pushes `true` if `Local/1` < `Local/0`, `false` if `Local/1` ≥ `Local/0`. Chains {`prim "<"`} for fault-values.

`GE:` As for `GT`, but pushes `true` if `Local/1` ≥ `Local/0`, `false` if `Local/1` < `Local/0`. Chains {`prim ">="`} for fault-values.

`LE:` As for `GT`, but pushes `true` if `Local/1` ≤ `Local/0`, `false` if `Local/1` > `Local/0`. Chains {`prim "<="`} for fault-values.

`Increase` *Src Vec*`,` *n*`:` Adds *n* to integer value of *Src Vec*; operation is atomic.

`Plus:` Takes two integer values. Replaces `Local/1` and `Local/0` by their product. If either is a fault-value, chains {`prim "+"`} as above.

`Minus:` Calculates difference; for fault-values, chains {`prim "-"`}.

`Times:` Calculates product; for fault-values, chains {`prim "*"`}.

`Div:` Calculates integer quotient; for fault-values, chains {`prim "/"`}.

`Mod`: Calculates integer modulus; for fault-values, chains {`prim "mod"`}.

`Length`: If top of stack is a fault-value, then chains fault tag {`prim "length"`} to it; otherwise, removes string from top of stack and pushes its length.

`Concat`: Performs string concatenation; for fault-values, chains {`prim "^"`}.

`Ord`: Expects `Local/1` to be a string and `Local/0` to be an integer > 0 and less than the length of the string. Returns the ASCII value of the character at that location within the string. Generates a fault-value with variant {`prim "ord"`} if the integer is out of range, and chains {`prim "ord"`} for fault-values as arguments.

`Chr`: If top of stack is a fault-value, then chains fault tag {`prim "chr"`} to it; otherwise, remove an integer value n from the top of the stack, and pushes a single character string with ASCII value n (for $1 \leq n \leq 127$), or generates a fault-value corresponding to {`prim "chr"`}.

`InLft`: Generates a sum structure. Claims a structure of two elements from the heap. The first element is assigned `false`, and the second is assigned from `Local/0`, which is removed from the stack.

`InRht`: Generates a sum structure. Claims a structure of two elements from the heap. The first element is assigned `true`, and the second is assigned from `Local/0`, which is removed from the stack.

`Nil`: Pushes *void* onto the stack.

`Head`: Head of list. If `Local/0` is a fault-value, then chains {`prim "head"`} to it. If `Local/0` is *void*, then removes it and generates a fault-value with {`prim "head"`}; otherwise, replaces the structure `Local/0` with its first element.

`Tail`: Tail of list. If `Local/0` is a fault-value, then chains {`prim "tail"`} to it. If `Local/0` is *void*, then removes it and generates a fault-value with {`prim "tail"`}; otherwise, replaces the structure `Local/0` with its second element.

`Is` *Tag*: If `Local/0` is a fault-value, then chains {`prim "is"`} to it; otherwise, treats it as a variant structure, with an integer as first element. If `Local/0/0` is equal to *Tag*, then pushes `true`, otherwise pushes `false`. The structure is removed.

`As` *Tag*: If `Local/0` is a fault-value, then chains {`prim "as"`} to it; otherwise, treats it as a variant structure, with an integer as first element. If `Local/0/0` is equal to *Tag*, then pushes `Local/0/1`, otherwise generates a fault-value from {`prim "as"`}. The structure is removed.

`Null`: If `Local/0` is a fault-value, then chains {`prim "null"`} to it; otherwise, pushes `true` if `Local/0` is *void*, or `false` if `Local/0` is a structure.

`Fault`: If `Local/0` is a fault-value, then chains {`prim "fault"`} to it; otherwise, transforms it into a fault-value.

`ChainFault` *Tag*: Expects `Local/1` to be a fault-value. Generates a variant cell with tag *Tag* and value `Local/0`, and chains this `Local/1`.

`EqTriv`: Pushes `false` if `Local/1` or `Local/0` is a fault-value, and `false` otherwise.

`EqInt`: Regards `Local/1` and `Local/0` as integers. Pushes `true` if they are equal, and `false` if they are unequal, or either is a fault-value.

`EqStr`: Regards `Local/1` and `Local/0` as strings. Pushes `true` if they are equal, and `false` if they are unequal, or either is a fault-value.

`EqBool`: Regards `Local/1` and `Local/0` as booleans. Pushes `true` if they are equal, and `false` if they are unequal, or either is a fault-value.

`Jump` *Label*: Performs unconditional jump to *Label*.

`TrueJump` *Label*: Pops boolean value `Local/0`; jumps to *Label* if `true`. Undefined for a fault-value.

`FalseJump` *Label*: Pops boolean value `Local/0`; jumps to *Label* if `false`. Undefined for a fault-value.

`Switch` $label_0$, ..., $label_n$: Pops integer value `Local/0`; jumps to $label_0$ for value 0, $label_1$ for value 1, and so on. Undefined for integer value < 0 or $> n$, or for a fault-value.

`Fork` *Size*, *Label*: Create a process to start execution of the current code portion at *Label*. The new process is allocated a stack of size *Size*; all registers except `R_Local` and `R_Program` are inherited from the parent.

`AndCell` *Size*: Pushes a structure of two elements. The first is assigned integer value *Size*, the second is left *empty*.

`OrCell` *Size*: Pushes a structure of three elements. The first is assigned integer value *Size*, the second is assigned boolean value `false`, and the third is left *empty*.

`AndSuccess`: Decrements `And/0`; if zero then assigns `And/1` to be `true`. Then performs a `Stop`.

`AndFailure`: Assigns `And/1` to be `false`. Then performs a `Stop`.

`OrSuccess`: Assigns `Or/1` to be `true` and `Or/2` to be `false`. If `Or/1` was `true` then performs a `Stop`.

`OrFailure`: Decrements `Or/0`. If it is now one then assigns `Or/2` to be `true`; if zero then performs `AndFailure`. Then performs a `Stop`.

`Otherwise`: If `Or/2` is `false` then performs `OrFailure`.

`IfInstanced` *Label*: If `Local/0` is *instantiated*, then transfers control to *Label*; otherwise, prunes the reference chain, and assigns the last cell of the reference chain to be *empty*.

`UnifyChain` *Label*: Assumes that `Local/1` is a single reference to an *empty* cell. Assigns the empty cell with `Local/0`, and removes both from the stack.

`Uninstance` *DestVec*: Uninstantiates the cell referenced by *DestVec*.

`Unify ->` *Label*: Unifies two arguments. If `Local/1` references an *uninstantiated* cell, then unifies it to `Local/0`, and removes both from the stack, and jumps to *Label*. If `Local/0` references an *uninstantiated* cell, then unifies it to `Local/1`, and removes both from the stack, and jumps to *Label*. Otherwise (both are instantiated), performs a `Skip`.

`Skip`: No operation. Continues to next instruction.