The Implementation of Practical Functional Programming Languages

by

Nigel Perry, BSc(Hons), CEng, MBCS

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

Imperial College of Science,Technology and Medicine,
University of London

BRITISH LIBRARY

IMAGING SERVICES NORTH

Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

PAGINATED BLANK PAGES ARE SCANNED AS FOUND IN ORIGINAL THESIS

NO INFORMATION IS MISSING

BRITISH LIBRARY

IMAGING SERVICES NORTH

Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

PAGE NUMBERING AS ORIGINAL

BRITISH LIBRARY

IMAGING SERVICES NORTH

Boston Spa, Wetherby

West Yorkshire, LS23 7BQ

Some creased pages

Best copy available

Abstract

This thesis is about the design and implementation of practical functional languages. We begin with a discussion of functional language I/O, and survey existing I/O schemes highlighting their drawbacks. We then present a new I/O scheme, based on what we term *result continuations*, which overcomes these drawbacks. Result continuations solve the I/O problem by arranging for all I/O to be performed on behalf of the program by the O/S *outside* the functional domain, thus maintaining referential transparency and providing I/O sequence control. We later extend our I/O system to provide an inter-language interface between functional and imperative programs.

We describe a new implementation method for functional languages, based upon a conventional imperative model modified to support and optimize the execution of functional languages. This provides comparable performance to imperative languages, and can be extended to coarse-grained implementations for current MIMD machines.

The ability to overload identifiers with multiple definitions is a convenient programming feature, allowing for example the same symbol to be used for integer and matrix multiplication. Our inter-language calling mechanism together with certain useful abstractions, such as abstract data types, require existential quantification of type variables in type definitions. We show how these two can be supported in a language with polymorphism and type inference, and detail both a type system and type-checking algorithm for handling the combination of all four. We demonstrate that despite the fundamental impact existential quantification has on the type system its implementation in a type-checker can share the same mechanism as polymorphism. We further demonstrate that overloading resolution can be handled without introducing an unacceptable performance penalty.

Our research was undertaken as part of the Hope+ project, part of Alvey Flagship, the aim of which was to construct a practical pure functional language system running on conventional sequential hardware.

Contents

1. Introduction 9
    1.1. Chapter Overview 10
    1.2. Acknowledgments 11

2. Overview of Hope+ 13
    2.1. Hope+ 13
        2.1.1. Types and Type Declarations 14
        2.1.2. Function Declaration and Definition 14
        2.1.3. Expressions 15
        2.1.4. Module System 16
    2.2. Problems with Hope+ 16
        2.2.1. The Type System and Overloading 17
    2.3. Summary 18

3. I/O for Functional Languages 19
    3.1. The definition of "I/O" 19
    3.2. I/O in Functional Languages 21
        3.2.1. "Accidental" breakages 22
        3.2.2. A Survey of Current I/O Proposals 22
    3.3. Environment Passing 23
        3.3.1. Explicit Environment Passing 23
        3.3.2. Implicit Environment Passing 24
        3.3.3. Summary of the Drawbacks with Environment Passing 25
    3.4. Streams 25
        3.4.1. Basic Streams 25
        3.4.2. Token Streams – Stoye 26
        3.4.3. Message Streams 28
        3.4.4. Interactions and Combinators – Thompson 29
        3.4.5. Combinators II – Dwelly 32
        3.4.6. Continuation Streams – Karlsson 34
        3.4.7. Haskell I/O 36
        3.4.8. Summary of Streams 37
    3.5. Result Continuations 37
    3.6. A Language with Result Continuations 40
        3.6.1. Hope+C Continuations 40
        3.6.2. An Example Hope+C Program 43
    3.7. Implementation Strategies for Result Continuations 43
        3.7.1. User Accessible Functions or Constructors 44
        3.7.2. Continuation Function Type 44
        3.7.3. Separate Success and Failure Continuation Functions 46
        3.7.4. Low or High Level Operations 46
    3.8. Higher-Order Programming with Result Continuations 46
        3.8.1. CPS Higher-Order Functions 47
    3.9. Conclusions and Related Work 48

4. The Efficient Sequential Implementation of Functional Languages 51
    4.1. Overview of Fpm2 52
        4.1.1. Developments from Fpm 53
        4.1.2. Implementation Model 54
        4.1.3. Summary 54
    4.2. The Language - Fpm+ 55
        4.2.1. The Syntax 55
        4.2.2. Program & Module Structure 56

4.2.3. Expressions 58
4.3. The Abstract Machine – Fpm/AM 59
4.3.1. The Fpm/AM Architecture 60
4.3.2. The Instruction Set 61
4.4. The Concrete Machine – Fpm/CM 62
4.4.1. Fpm/CM Implementation 62
4.4.2. Fpm/CM Architecture 62
4.4.3. Register Usage 63
4.4.4. Fpm/AM Operand Representation 64
4.4.5. Representation of Values 64
4.5. Comparative Performance 67
4.5.1. The Programs 72
4.5.2. Summary 78
4.6. Conclusions 78

5. Inter-language Working 79
5.1. Motivation 80
5.2. Providing Inter-language Working at the Function Level 81
5.2.1. Referential Transparency 81
5.2.2. Value Transfer 81
5.2.3. Memory Management 81
5.3. Language Level – Keeping Referential Transparency 82
5.3.1. Example: Bit-wise Boolean-and 84
5.3.2. Example: FFT with in-place vector update 85
5.4. Implementation Level 88
5.4.1. Value Transfer 88
5.4.2. Memory Management 89
5.5. Conclusion 90

6. Towards a Functional Operating System 91
6.1. Background 91
6.2. System Structure 92
6.3. Realization 93
6.4. Explicit Versus Implicit Multiple Processes 96
6.5. Conclusion 97

7. An Extended Type System 99
7.1. User-defined Overloading 100
7.1.1. The Interaction of Overloading and Type Inference 101
7.1.2. "Best Fit" Overloading Resolution 102
7.2. Existential Quantification 102
7.2.1. Overview of Universal Quantification for Function Types 103
7.2.2. Existential Quantification for Data Types 103
7.2.3. Type-checking Existential Quantification 105
7.3. Type Inference Rules 107
7.3.1. Set-Based Inference Rules for Overloading 108
7.3.2. Non Set-Based Rules for Overloading 109
7.3.3. Inference Rules for Existential Quantification 110
7.4. The Extended Type System in Hope+C 112
7.4.1. Base Types and Basic Type Expressions 112
7.4.2. Function Declarations 112
7.4.3. Type Declarations 113
7.4.4. Type Variables 113
7.4.7. Type Synonyms 114
7.4.8. Overloading 115
7.4.9. Type Inference in Qualified Expressions 115

8. Efficient Type-checking 117
8.1. A Simple Language 120
8.2. Type Expressions 121
8.3. Overview of the Type-checking Algorithm 123
8.3.1. On the Time/Complexity Performance of the Algorithm 124
8.4. The Type Substitution 127
8.4.1. Ground Type Substitution 127
8.4.2. Non-ground Type Substitution 127
8.4.3. Set Valued Type Substitutions 128
8.4.4. Dependent Type Substitutions 129
8.4.5. Operations on the Type Substitution 132
8.5. The Unification Algorithm 134
8.6. Conversion of Type Declarations 135
8.6.1. Type Visibility 135
8.6.2. Type Variables 136
8.6.3. Type Synonyms 136
8.6.4. Data Declarations 136
8.7. Type-checking Expressions 137
8.7.1. Identifier Lookup 137
8.7.2. Typing the Expression 139
8.7.3. The Upscan Function 140
8.7.4. The Downscan Function 142
8.8. Conclusion 142

9. Conclusions and Further Work 145
9.1. Overview 145
9.2. Future Work 146
9.3. Current Research 147
9.4. Conclusion 147

10. References 149

Appendix 1: Hope+C Continuations 153

Appendix 2: Example Hope+C Programs 159
A2.1. Example Programs: "wc" 159
A2.1.1. Version 1: "Stream/List" 159
A2.1.2. Version 2: "Character" 161
A2.2. Higher-Order Examples 162
A2.2.1. Higher-Order Continuations Module 162
A2.2.2. Version 3: "CPS" 163
A2.2.3. Version 4: "non–CPS wc/format" 164

Appendix 3: Fpm2 Details 167
A3.1. The Fpm/AM Opcodes 167
A3.2. Fpm/AM Built-in Functions 171
A3.3. The Imperative Interface 173

Appendix 4: The Comparison Programs 179
A4.1. Utility Functions 179
A4.2. The Programs 182
A4.2.1. Nfib 182
A4.2.2. Queens – Version 1 183
A4.2.3. Queens – Version 2 184
A4.2.4. HigherOrder 187
A4.2.5. Tak 188
A4.2.6. Takl 189
A4.2.7. Triangle 191
A4.2.8. Triangle – Vector Version 193
A4.2.9 Cat 196
A4.2.10 FFT 198

Appendix 5: Fpm+ to Fpm/AM Translation 209

1. Introduction

This thesis is concerned with design of practical functional languages and their practical implementation. Though functional languages have been a topic of research in computer science for a number of years, true *practical* implementations do not exist. We define a practical functional language system as one which supports at least the following:

- A full set of data types to support symbolic, character and numeric computation.
- An I/O system which provides for all types of interaction with the "outside world", for example; terminal I/O, device I/O and interrupt handling.
- A practical type system. It is no good providing "convenience" features such as overloading if they cause unacceptable compiler performance.
- The ability to interface to other languages in particular imperative ones. There are many standard library packages in existence, for example X Windows, which could usefully be employed by the functional programmer and to which access should be provided without compromising the purity of the functional environment.
- Efficient execution on current hardware. If functional languages are not comparable to existing languages, taking into account both the writing and execution costs, then they simply will not be used.

The first four points above make the language itself practical – in the sense that it can usefully be used to write programs, the last implies that the implementation is practical – in the sense that it is usable. In this thesis we describe research undertaken as part of the Hope+C system project, the aim of which was to construct a practical pure functional language system running on conventional sequential hardware. Though motivated by this project our results are not limited to the Hope+C language, and in particular are likely to form the basis of future language research work at Imperial. We cover three major topics; I/O, sequential implementation, and type checking; and two subsidiary ones; inter-language calling, and operating system design.

Note:

In this thesis all examples which relate to a particular programming language are written in the syntax of that language. This was thought preferable to devising our own standard syntax.

1.1 Chapter Overview

Hope+ is the basis upon which the various developments which form the research content of this thesis were built, we therefore start in chapter 2 with an overview of this language.

In chapter 3 we discuss the provision of I/O operations, the lack of which we believe is the primary reason why functional languages have not gained wide acceptance. However, the reason for this lack of I/O support is that by nature it involves *side effects* which are not expressible in a functional language. In an attempt to overcome this difficulty various proposals have been made, most being some variation of *streams* where I/O is modeled as a combination of data driven input and demand driven output over lists; that is the program output is a list, each element of which is successively demanded by the system, and the input to the program is a list which is constructed non-strictly and only evaluated when the program accesses an element. Despite the proliferation of proposals, no complete – in the sense that all types of I/O can be handled, and practically usable – in the sense that the synchronisation of external effects can easily be correctly controlled, has until now been presented.

In this chapter we start with a survey of existing I/O schemes and explain why they are inadequate. We then present a new scheme, which we have termed *result continuations*, which is both correct and capable of providing all I/O operations. We define an extension to Hope+ which uses result continuations to provide I/O. Finally we show that the style of programming created by using result continuations and the typical higher-order style of functional programming can easily be combined.

Chapter 4 describes an efficient implementation scheme for functional languages for current sequential machines. Most of the previous effort in implementing functional languages has been directed at designing new methods designed to exploit their particular properties and has resulted, for example, in the design and/or use of the; data flow, super-combinator, categorical combinator and graph reduction models. Though all these models fit the functional computational model well, they are all substantially different from that supported by conventional sequential hardware, and despite much implementation effort, to our knowledge, have so far failed to provide functional language implementations which are comparable in performance to those of imperative languages. Our approach, developed from that used in the original IC Hope

compiler system, is to use the conventional stack/heap model used for imperative languages combined with some extensions and modifications to support and optimize the execution of functional languages. This approach is surprisingly easy, provides comparable performance to imperative languages, and has the potential to be extended to existing parallel MIMD computers using coarse-grain language implementations – as these machines are constructed from multiple sequential CPU's.

Chapters 5 and 6 cover the two subsidiary topics of this thesis; inter-language calling and operating system design. Both of these are based on the result continuation I/O system described in chapter 3. The inter-language calling system provides functional level calling between functional and imperative languages while preserving the referentially transparency of the functional one. The operating system design is a simple message passing system which demonstrates that functional languages can be used to implement complex "system" programs.

In chapters 7 and 8, we discuss type systems and type checking algorithms. Hope+C supports polymorphism, overloading and type inference. This combination is not provided by the standard Milner system and, though there has been other research in this area, to our knowledge no type system has been proposed which is efficiently type checkable on a sequential machine. In chapter 7 we describe a new type system, and its use in Hope+C, which supports the above three features and also existential quantification which is required to support the inter-language calling mechanism of chapter 5. In chapter 8 we present an efficient type checking algorithm for our type system and describe its implementation in and for Hope+C.

The conclusions of the thesis are laid out in chapter 9.

1.2. Acknowledgments

Special thanks to Keith Sephton who; co-wrote the Hope+C compiler, helped design the user level syntax of the inter-language calling system, wrote some the the standard Hope+C library modules, helped debug the runtime system (with great patience!), and with whom I had many interesting and fruitful discussions.

Thanks to Tony Field, for his careful proof reading and comments on this thesis, and to the rest of my colleagues in the Functional Programming Research Group at Imperial College for using Hope+C, and generally putting up with me!

This research was part funded under the Alvey Flagship project.

2. Overview of Hope+

In this chapter we present an overview of the Hope+ functional language. Later in this thesis we describe the use of the Hope+ language and its implementation as a practical application of our research results. The I/O system developed in chapter 3 was incorporated into Hope+ to produce the language Hope+C, without this extension the writing of the Hope+ compiler in Hope+ itself would not have been possible. The implementation technique described in chapter 4 was applied to Hope+C, producing a high performance system. The type system of chapter 7 was added to Hope+C; and the type checking algorithm developed in chapter 8 used in the Hope+C compiler, effectively demonstrating its good time/complexity performance.

Hope+ has its problems, some uncovered by our research; which are mentioned in this and later chapters; but our obligations under Flagship did not allow for the altering of the language definition and implementation to address them[‡]. However, planned future language developments outside of Flagship will take into account the results of our research.

2.1. Hope+

The Hope+ language was designed to be used as the "systems implementation language" for the Alvey Flagship project, and is an extension of Hope [Burstall]. The requirements for Hope+ were drawn up by a Flagship committee, these were then discussed by a group at Imperial and a language design produced [Perry 88a]. Hope+ is a pure higher-order polymorphic functional language. It is not curried and has call-by-WHNF (weak head normal form) semantics[†]. It provides primitive stream-based I/O.

2.1.1. Types and Type Declarations

Hope+ has four basic types: integers (`num`), floating point (`real`), character (`char`), and the unit type (`void`). There is no automatic type coercion, for example between `num` and `real` values.

[‡] Adding in the I/O and type systems are orthogonal extensions which do not invalidate any of the Hope+ definition, and were hence allowable.

[†] Call-by-WHNF is defined as: value construction is non-strict but all functions arguments are evaluated to WHNF *before* the function is evaluated.

Type operators for product/tuple (#) and function (->) are built-in. There are three basic forms of type declaration (items in []'s are optional):

2.1.1.1. Type Variable Declarations

```
typevar <var1>, ..., <varn>
```

This statement declares *type variables* which can later be used to define polymorphic types. The type variables `alpha` and `beta` are pre-defined.

2.1.1.2. Sum Type Declarations

```
data <name>[(<arg1>, ..., <argn>)] == <const1> ++ ... ++ <constm>
     where: <const> = <identifier>[(<type>)]
```

This declares a *sum* type. Declarations may be parameterised, constructors may be nullary or have an associated type. There is provision to declare self and mutually recursive types. The types `truval` and `list` are pre-defined as:

```
data truval == false ++ true;
data list(arb) == nil ++ cons(arb # list(arb));
```

2.1.1.3. Type Synonym Declarations

```
type <name>[(<arg1>, ..., <argn>)] == <type>
```

This declares a type *synonym* – the `<name>` on the left hand side is effectively replaced using textual substitution by the `<type>` on the right hand side wherever it occurs.

2.1.2. Function Declaration and Definition

Functions are declared and defined *separately*. A function declaration is of the form:

```
dec <function name> : <type>
```

For example:

```
dec map : (alpha -> beta) # list(alpha) -> list(beta)
```

which is the declaration for the polymorphic map function. All functions must be defined before use, this restriction being imposed to allow easy interpretation.

Functions are defined by one or more equations, the appropriate equation being selected using *best fit* pattern matching. An equation is of the form:

```
--- <function name>[<pattern>] <= <expr>
```

Best fit pattern matching selects the equation whose pattern is "closest" to the argument values, informally it is defined as:

- Order of equations in program text is unimportant
- Constants are more *specific* than variables, i.e. they "match better"
- The equation with the most specific pattern matching the argument values is chosen. It is a runtime error if two patterns are found to be equally specific, the possibility of this happening is checked at compile time and a warning issued.
- If two or more pattern elements can be checked at any stage to determine the most specific match, then the leftmost element is checked first.

For example, given the function:

```
--- f(1, y) <= y;          (a)
--- f(x, 2) <= x;          (b)
--- f(a, b) <= a * b;      (c)
```

then the applications:

```
f(1,3)      selects equation (a)
f(5,2)      selects equation (b)
f(3,4)      selects equation (c)
f(1,2)      causes a runtime "ambiguous match" error,
            as both (a) and (b) are "best"
```

2.1.3. Expressions

Hope+ has the basic expression types common to most functional languages. Expressions in Hope+ are evaluated to WHNF, that is until the outermost constructor is found. When an expression is *suspended* no evaluation takes place, instead a *suspension* is constructed which contains the expression and the values of any variables from the current environment which may be required to evaluate it. Subsequent use of a suspended expression in a context which requires evaluation to WHNF causes the expression within the suspension to be evaluated. The Hope+ expression types are:

`<constructor> <expr>`

Construction. The argument expression is suspended.

`(<expr1>, ..., <exprn>)`

Tuple expression. Tupling is defined as a *construction* operation therefore the sub-expressions are suspended.

`if <expr> then <expr> else <expr>`

Conditional.

`let <pat> == <expr> in <expr>` and `<expr> where <pat> == <expr> end`

Qualified expressions. Both `let` and `where` forms are semantically equivalent and allow recursive definitions.

`<expr> <expr>`

Function application. The argument expression is evaluated to WHNF *before* the function is applied. If the argument expression is a tuple, then *each* element of the tuple is evaluated to WHNF before the application. (There is syntactic overloading involved here. Functions in Hope+ are *n*-ary, that is they take one or more arguments, and a function argument list is represented syntactically as a tuple; differentiation being by context. However an *n* argument function may also be applied to a single value which is an *n*-tuple, in this case the tuple is broken up and each element evaluated to WHNF. This behaviour is a carry over from IC Hope.)

`lambda <pat> => <expr> [| <pat> => expr>]* end`

Anonymous function expression.

2.1.4. Module System

Hope+ has a simple module system which allows functions and types to be exported/imported. There is no separate interface and implementation provision; a module is simply defined in one file, exported items being declared public.

2.2. Problems with Hope+

There are two practical problems in implementing and using Hope+; firstly it uses a type system for which no efficient type checker exists. Secondly in common with other functional languages

it lacks proper I/O facilities. Here we give a brief description of the problems posed by the type system, later chapters will describe solutions to both of the problems.

2.2.1. The Type System and Overloading

Apart from stating that it is polymorphic, the Hope+ definition does not specify the type system but, accounting for its Hope parentage, the Milner system is generally assumed. However the definition also states that functions and pattern variables may be overloaded, to any degree, by the programmer and the Milner system does not support this.

Chapter 7 of this thesis describes a new type system, developed to enable the Hope+ compiler to be written, which does support polymorphism and overloading. We also present an extension to this type system which supports existential quantification of type variables in data constructors. For example, in:

```
data printable == integer(num)
                ++ float(real)
                ++ other(alpha # (alpha -> list(char)));
```

the type variable `alpha` is existentially quantified: that is, when `other` is used to *construct* a value the type variable `alpha` may be instantiated to any type; when the argument value to `other` is *extracted* by pattern matching the only type assertion is that `alpha` is some type, no further information is available. In this example existential quantification implies that when a value/function pair is extracted from an `other` constructor the only permissible operation upon the extracted value is to apply the extracted function to it.

2.2.1.1. Declaration and Definition Pairing

In Hope+ functions are declared and defined separately and functions can also be overloaded, this leads to an interesting problem which is best described by example, given:

```
dec f : num # num -> num;                    (1)
dec f : real # real -> real;                 (2)

--- f(2, x) <= 3 * x;                        (a)
--- f(a, 3.0) <= a * 9;                      (b)
--- f(y, z) <= y - z;                        (c)
```

then definition (a) is clearly associated with declaration (1), similarly (b) goes with (2). But to which definition does (c) belong? The built-in subtraction operator is overloaded with the following types:

i) `num # num -> num`
ii) `num # real -> real`
iii) `real # num -> real`
iv) `real # real -> real`

therefore equation (c) could belong to either function (1) or (2). This leaves us with the decision whether to; associate the equation with one or both of the functions, or to produce a compile time error.

How to handle this situation it not specified by the Hope+ definition. Our solution is to generate *multiple* equations internally and associate one to each of the appropriate functions, this is explained in greater detail in chapter 7.

2.3. Summary

Compared to Hope, its predecessor, Hope+ adds: real numbers; a standard vector package; an uncommon evaluation strategy, namely call-by-WHNF; and user-defined overloading. The later extension poses some problems in the implementation of Hope+, which we address later in this thesis. Hope+, like its predecessor and other functional languages, lacks good I/O facilities, this problem is addressed in the next chapter.

Although we were involved in the definition of Hope+ it is not presented as part of the research undertaken, and any problems or peculiarities that exist in its definition are not directly addressed in this thesis, though some observations will be made. Our research was undertaken as a result of our obligation under Flagship to implement Hope+ and changes to the basic language were not possible in this context. The I/O system described in the next chapter was allowable as it is an orthogonal extension.

3. I/O for Functional Languages

In this chapter we discuss the provision of I/O facilities in functional languages. In sections 3.1 and 3.2 we define I/O and the synchronisation requirements which need to be met to enable the correct behaviour of an I/O system. In sections 3.3 and 3.4 we present a survey of the existing I/O systems/proposals for functional languages and explain where they are inadequate or fail to provide for the synchronisation requirements of I/O. In section 3.5 we introduce a new I/O system based upon the idea of *result continuations*, and establish that it provides the synchronisation to enable correct I/O behaviour to be guaranteed. In section 3.6 the practical use of our I/O system is demonstrated by describing its incorporation into the Hope+ language producing the language Hope+C, a development which enables the Hope+ compiler to be written in Hope+ itself. In section 3.7 we discuss alternative methods to those adopted in section 3.6 which may be used to incorporate result continuations into a functional language. We conclude, in section 3.8, with a discussion on the impact of result continuation based I/O on programming style.

3.1. The definition of "I/O"

In this chapter we shall use "I/O" as a generic term to cover all tasks which are related to the communication between a program and its environment, and the effect such communication may have. The tasks covered by this definition may be broken into the following broad categories: terminal I/O, device I/O, process creation, inter-process communication, signal handling and environment interaction. The definition and requirements of each category are as follows:

a) **Terminal I/O**

Terminal I/O covers the reading of input from a keyboard and the display of output on a screen. During terminal I/O there must be some control over the relative ordering of input and output operations so that, for example, input is not read in before a request for it has

been displayed. Graphical I/O is an extension of simple terminal I/O; allowing other input devices, such as a mouse, and the display of objects other than characters on output.

b) **Device I/O**

Device I/O covers communication with filesystems, discs, tape drives etc. The requirements for device I/O are similar to those of terminal I/O, but no synchronisation is normally required between input and output operations. Functional language proponents propose the replacement of filesystems by "persistent storage", which would remove them from the I/O arena, however even when these systems become available device I/O will still be required to raw discs, tape drives etc.

c) **Process Creation**

The ability for a process to initiate other separate processes. This is required, for example, if functional languages are to be used for programming tasks which require *explicit* use of parallel processes.

d) **Inter-process communication**

Inter-process communication covers I/O between two separate processes. The two processes involved may have a parent/child relationship, i.e. one was created by the other as in (c), or be unrelated. Once established, inter-process communication has similar requirements to terminal I/O (a) or device I/O (b) and can often be handled in a analogous way.

e) **Signal Handling**

A *signal* is an event to which a process must respond which originates from outside the process. Examples are; a human hitting an *interrupt key* on a keyboard, or a device notifying some urgent condition. Signals may be either *polled*, in which case the process must check periodically whether any have occurred, or *asynchronous*, in which case the process is automatically switched to respond to the event. Processes must be able to send, as well as receive, signals. For functional languages to be usable for the writing of operating systems, or any other program which may require asynchronous intervention by the user, for example a text editor, signal handling provision is essential.

f) **Environment Interaction**

A process must be able to obtain information from, and provide information to, the system upon which it is running. An example is obtaining the current date and time.

In the remainder of this chapter we shall use the above six categories as the yardstick against which a particular I/O system or proposal is evaluated.

3.2. I/O in Functional Languages

To maintain correct I/O behaviour the relative temporal ordering of individual I/O operations must be controlled. In an imperative language this is not difficult, as the temporal ordering of the evaluation is explicit in the program. However, in pure functional languages, as they are referentially transparent, there is usually no mechanism to control the temporal ordering of the evaluation, hence maintaining correct I/O behaviour is non-trivial[†]. There are two types of constraint needed to enable the temporal ordering of I/O operations; which we shall call *inter-request synchronisation* and *intra-request synchronisation*:

i) **Inter-request Synchronisation.**

This is the control over the order of *separate* I/O requests, e.g. the ability to ensure that a request prompt is printed before input is read. This type of synchronisation poses no major problem to imperative languages. However, functional languages are referentially transparent and due to this do not usually provide a programmer controlled method of evaluation order, which is what is required for inter-request synchronisation. Lazy evaluation strategies can complicate the problem as very careful thought has to be given as to *when* an operation will be performed, something which is usually more apparent when strict evaluation is employed.

ii) **Intra-request Synchronisation.**

This is the relationship between the issuing of a request for an action and the receipt of the response to that action. This is usually not an issue in imperative languages; as the

---

[†]It is non-trival as a programmer needs always to be considering *how* something will be executed – which is only possible with a thorough understanding of the evaluation strategy/denotational semantics. It is also a task from which functional programming is usually claimed to relieve the programmer.

request/response are bound together by the procedure call/return mechanism. However it frequently becomes an issue with functional languages due to the division by most stream-based systems of an I/O operation into the two distinct parts; issuing of the request; and receiving the response to operation. This then introduces all the problems associated with inter-request synchronisation.

3.2.1. "Accidental" breakages

In the descriptions below we shall sometimes refer to a system as being easy to break "accidentally". By this we mean that to ensure its correct behaviour the system relies either upon a behaviour which is not guaranteed by the language, though most implementations may provide it; or upon the programmer ensuring correct temporal ordering, which is non-trival.

For example, let the result of a program be a list of characters to be displayed on the terminal screen and the argument to the program be a list formed from the characters typed at the terminal keyboard. Under this scheme output will be produced by *construction* operations forming the result list and input will be obtained by *destruction* operations on the argument list.

For a single request/response operation the request must first be displayed by appending it to the result list, a construction operation, and then the response must be removed from the front of the argument list, a destruction operation. For this request/response operation to have the correct I/O behaviour the temporal ordering of these construction/destruction operations must be guaranteed, *without* the aid of any explicit sequencing construct. For the whole program to have the correct I/O behaviour the temporal ordering of *every* destruction operation on the argument list and construction operation on the result list must be ensured. This is a non-trival task for the programmer and an incorrect ordering is easily introduced.

Many of the systems we describe in the survey below attempt to either solve or circumvent this problem of accidental misbehaviour by using *programmed* solutions. We argue forthwith that none succeed completely in overcoming the problem.

3.2.2. A Survey of Current I/O Proposals

Existing proposals to provide functional language I/O fall into two main groups, *environment passing* and *streams*. In environment passing every process is passed a data structure

representing the environment; i.e. the file system, devices etc.; which it may access freely and must return a *new* value for this structure when it completes. In a stream system a process is passed a stream of information from the environment and produces another stream as its result. Both methods have their problems and limitations, in the next two sections we describe some of the major systems/proposals in each group and show when they are suitable and where they fail to either provide the I/O categories, or meet their requirements. Finally we describe a new system which provides for all I/O categories.

3.3. Environment Passing

The fundamental idea behind *environment passing* is to remove file systems, terminals etc. from the realm of *state* and instead consider them as *data*. A pure functional program can easily accept as input some structured value representing the environment and produce as output a new structured value. By employing lazy semantics, the input value need not yet be fully evaluated, thus allowing devices such as terminals to be represented whose input values are not yet fully known.

There are two approaches to providing the environment value, one as an *explicit* argument to the program, the other by passing it *implicitly*.

3.3.1 Explicit Environment Passing

Under this system a program is a function:

program: input value × initial environment → output value × new environment

Every function which either manipulates the environment or calls another function which does, *must* be passed it as an argument. The obvious objection is that it clutters up the program text and obscures the structure of the algorithm. Problems also arise if the program is executing as part of a larger system – how are environment modifications requested by different programs combined, for example. A simple solution is to allow only one program access to the environment (or part of it) at any time, but this has a rather undesirable effect on any potential parallelism.

3.3.2. Implicit Environment Passing

This system is an attempt to avoid the problem of environment passing cluttering up the program by making it an implicit argument to *every* function. An example of this approach is the language FL [Backus]. In FL the environment is termed the *history* and every function is of type:

```
f: <val, history> → <val', history'>
```

The history argument is always implicit and is only used by such functions as require it. For example, the function:

$tl: \langle x_1, \ldots, x_n \rangle \rightarrow \langle x_2, \ldots, x_n \rangle$

which returns the tail of a list ignores the history argument and returns it unaltered. In contrast the function:

```
in: <dev> →<val>
```

which returns the next input value from some device, returns a new value for the history.

Passing the environment implicitly avoids the main problem of explicit passing, namely obscuring the program, however it introduces a new one. In the FL program:

```
f: <in:`keyboard, in:`keyboard>
```

if the user types "Ok" is the function called with the argument <"O", "k"> or the argument <"k", "O">? As the history has not been explicitly passed between functions the order in which calls to in are made is not evident from the syntax. To resolve this problem FL has very explicit rewrite rules which state precisely the order of evaluation of function arguments, but this shifts a burden onto the programmer who always has to know which way the hidden mechanism is going to behave. It also forces an unnecessary serialization of function application order, even on those functions which don't access the history, and thus voids the normally useful feature of functional programs namely that they can transparently exploit parallel hardware. To counteract this serialization property, the authors of FL propose to produce a compiler which calculates which functions do not alter the history and to use this information to allow for parallel, or more efficient, evaluation strategies were possible. Though this mitigates this particular property, FL still suffers from the common drawbacks of environment passing.

3.3.3. Summary of the Drawbacks with Environment Passing

Both forms of environment passing suffer from a number of drawbacks:

- They only (easily) handle terminal and device I/O. The FL proposal for example only addresses these two and does not discuss signal handling etc.
- As programs are passed the file system as an argument and return a new one as a result, either no other program may run in parallel, or no program can access a file produced by another program which is running concurrently. Also, if programs do run in parallel and manipulate the file system in incompatible ways it is not specified what their combined affect should be. This is a major problem and would prevent useful programs such as command shells or editors like "Emacs" being written.
- They can be confusing, either by obscuring the program, or by relying on a hidden mechanism.

3.4. Streams

All the stream-based proposals are based on the same underlying mechanism. A program in a stream-based system is of type:

```
prog: list(input) → list(output)
```

that is, a program takes a sequence of input values, processes them, and produces a list of output values. Streams systems are *demand driven*, that is, evaluation of a program is caused by each element of its output list being demanded in sequence. The difference between the different proposals is the types and interpretation of *input* and *output*. The first and most common stream system is *Basic Streams*, which provide character only terminal I/O. The other systems described below have been developed to address some of the obvious limitations of basic streams, while (it would seem) retaining what some would call their simple elegance.

3.4.1. Basic Streams

Many functional language systems provide only basic streams of characters, the input list coming from a keyboard and the output going to a screen. To ensure synchronization between the screen output and terminal input the input stream is constructed lazily and the execution of the

program is controlled by demanding the output list one element at a time. This mechanism, in association with a non-strict language or call-by-WHNF language like Hope+, is designed to ensure that provided output values are placed on the output list before values are taken from the input list, then the output will be displayed before input is requested.

The advantage of basic streams is their simplicity. However, there are two significant disadvantages; firstly a generalized I/O system usually extends beyond that of simple character I/O,and secondly the synchronisation can be easily broken accidentally [3.2.1.]. Despite this many functional languages only provide I/O this way, among which are Hope+, LML [Augustsson 89] and Miranda™ [Turner 85].

3.4.2. Token Streams – Stoye

William Stoye [Stoye] has proposed a system that we shall call *token streams* (Stoye himself called them *synchronized streams*, but as we are discussing synchronisation we use the term token stream to avoid confusion). Using token streams a program is of type:

```
prog: list(In) → list(Out)
```

where:

```
Out ::=
      Screen String |                ; write string to screen
      KbdChar |                      ; request input character
      ReadFile String |              ; request file content
      WriteFileString String |       ; create file
      Delay Int |                    ; do nothing for short time
      ...

In ::=
      FromKbdChar Char |             ; response to KbdChar
      FileContent String |           ; response to ReadFile
      DelayOver |                    ; response to Delay
      ...
```

Under this scheme a program places a token on the output stream requesting some service and a response token is returned by the system on the input stream. The use of tokens instead of characters enables many more tasks to be accomplished than is possible using basic streams, the requirement of *always* outputting a request token before receiving a response token is designed to ensure synchronization.

This scheme does provide for a number of the I/O categories but as Stoye himself states, has limitations, in that it cannot handle signals or processes.

There is however a fundamental problem, the synchronization given by the request/response token ordering is provided in software; if the program attempts to input a response token *before* outputting a request token, which can easily occur accidentally [3.2.1.], then a problem arises. For example the following program to input to characters and output them in reverse order will fail:

```
get2 : list(In) → list(Out) × Char × Char × list(In)
get2(FromKbdChar(c1) :: FromKbdChar(c2) :: remainder)
   = ([KbdChar, KbdChar], c1, c2, remainder)

prog : list(In) → list(Out)
prog(input)
   = let (output, c1, c2, rest_of_input) = get2(input)
     in output ++ [Screen [c1,c2]]
```

The problem in this program lies in the pattern matching of the argument to `get2`; to check that an application of `get2` is valid its argument must be examined to ascertain that it contains at least two elements. However, until the body of `get2` has been evaluated the `KbdChar` tokens will not be output and therefore the input `FromKbdChar` tokens will not exist so the pattern match cannot succeed. Rewriting the function `get2` as:

```
get2(input)
   = ([KbdChar, KbdChar], hd input, hd (tl input), tl (tl input))
```

would in this case solve the problem; there is no pattern matching to force early evaluation and non-strict evaluation will ensure that the expressions "`hd input`" and "`hd (tl input)`" are not evaluated until after the `KbdChar` tokens have been output.

The above example clearly demonstrates that token streams fail to guarantee *intra-request* synchronisation. The actual manifestation of this problem would depend on the implementation used:-

- The system could return a null or error token if input is requested before a token is output. This would require the program to check *every* input token and take appropriate action if an error occurs, the only probable action being program termination.
- The system could terminate the program.
- The system could lock up awaiting an output token that will never appear.

The preferred approach would be to detect the error at compile rather than execution time, but this would require a full flow analysis of the program, which is too costly to implement.

3.4.3. Message Streams

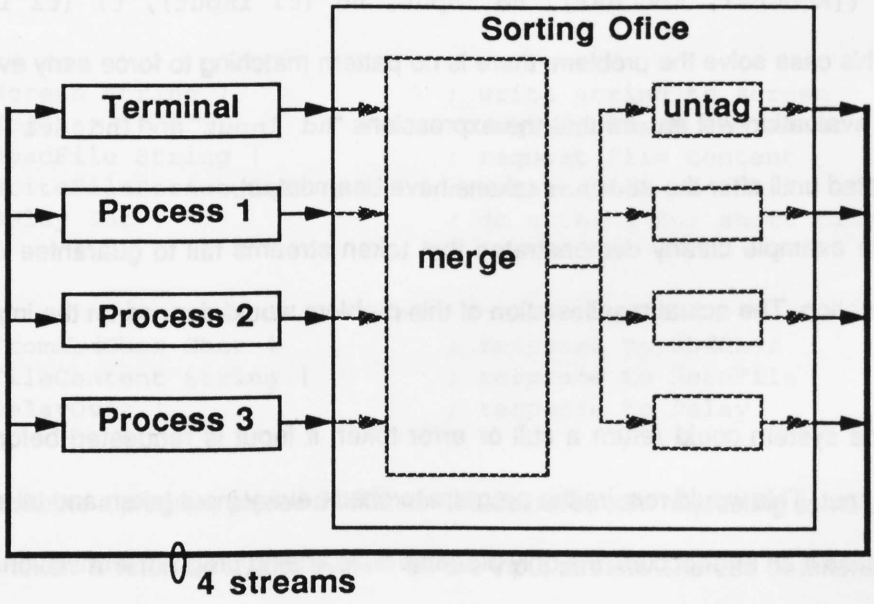
Having proposed token streams for *programs*, Stoye also proposes *message streams* for writing operating systems. A system consists of a number of separate processes along with a *sorting office*. Each process has an *address* which uniquely identifies it and produces a list of *(Address, Data_item)* pairs to communicate with other processes. The sorting office takes each process output and routes each value to the input list of the correct process (after removing its address part). In this scheme each process is of type:

```
process: list(Data_item) → list(Message)
```

where:

```
Message == (Address, Data_item)
```

Separate processes are used to represent the user program, the terminal keyboard, screen, any other devices, and a process creator – which creates new processes. The sorting office function relies on the existence of a *non-deterministic* merge – this non-determinism is constrained to occur at the "top-level" only. A simple example, taken from Stoye's paper, to output two characters with a non-deterministic order is constructed as follows:



a) The functions which make up the sorting office:

`Screen` is the input list to the process which handles the terminal, `untag` maps a list of (address, message) pairs to a list of messages given an address value, `merge` is the non-deterministic merge operation:

```
screen = untag 0 messages
input_to_1 = untag 1 messages
input_to_2 = untag 2 messages
input_to_3 = untag 3 messages
messages = merge (process_1 input_to_1)
                 (process_2 input_to_2)
                 (process_3 input_to_3)
                 (terminal screen)
```

b) The user processes:

```
process_1 in = [(0, hd in), (0, hd(tl in))]
process_2 in = [(1, 'a')]
process_3 in = [(1, 'b')]
```

This program will print either "ab" or "ba" on the screen.

Message streams extend the capabilities provided by token streams to cover process handling. The only I/O category which is not addressed in some way is (e) – signal handling. However, the synchronization problems inherent in the token stream method are even worse in this case. Not only may a program attempt to read an input message before sending a request, but as *all* inter-process messages arrive multiplexed onto a single stream, messages can arrive either out of sequence with their requested order or for which no request was made. This means that the simple protocol of token streams, namely output request token followed by input result token, is broken[†].

3.4.4. Interactions and Combinators – Thompson

In [Thompson 86] a *software* methodology to control I/O over character streams is proposed, the semantics of which are given in [Thompson 87]. Thompson first defines a basic unit of I/O, which he terms an *interaction*, and then defines functions to combine basic interactions into larger ones; these he terms *combinators*. By careful definition of these interactions and combinators he is able to build a synchronised interactive character-based I/O system.

[†]This is the same problem encountered when using communication networks such as X25; it can be solved but the protocol overhead would be high for an application as simple as program to screen communication.

3.4.4.1. Interactions

The system is developed for the language Miranda and relies fundamentally on the languages laziness and pattern matching semantics, as will be shown below. Miranda itself provides simple stream I/O and a program is of type:

```
program :: [char] → [char]
```

Instead of working directly with the raw character streams Thompson argues that interactive input is usually done in terms of lines and output is done in blocks (which need not be complete lines), so he defines his interactive programs over the types `input` and `output`, defined as:

```
input == [string]
output == [string]
  where
     string == [char]
```

An interaction is a function on the input stream, preprocessed into lines, and an initial state of some type. The result of an interaction is the remainder of the input stream, an output stream (which may be empty), and a new state which may be of different type to the input state. The type of an interaction is encapsulated in the parametric type `interact`:

```
interact * ** == (input, *) → (input, **, output)
```

Here is a very simple example of an interaction (Note: `uni` is a type containing the single value `Uni`, and is used when no meaningful value is required.):

```
hello :: interact uni uni

hello (x, Uni) = (tl x, Uni, toscreen)
where
     toscreen = "Enter your name:" : follow
     a        = hd x
     follow   = ["Invalid, try again\n"], a = "\n"
              = "Hello there" : [a], otherwise
```

This function consumes a single line of input and returns the remainder of the input and an output which is either a greeting or an error message. This function may appear obvious, but it only works because it has been carefully written. There are two points to note:

- The first line of the input is extracted by a local definition (`a = hd x`) and not by pattern matching. If pattern matching had been used then the input would have been read *before* the prompt had been output, which of course is not the desired effect!
- If the local definitions are rewritten in an apparently *functionally* equivalent way, as in:

```
toscreen = prompt : ["Hello there"], a <> "\n"
         = prompt : ["Invalid, try again\n"]
           where
              prompt = "Enter your name:"
a        = hd x
```

then the prompt would also be delayed until after the input had been read.

3.4.4.2. Combinators

To solve the above problems *combinators* are introduced. A combinator takes one or more interactions and possibly some other values and combines them in a well defined and well ordered manner. The goal is that these combinators may then themselves be combined to produce complex interactions which maintain the proper I/O semantics. We shall just give one example here, taken from Thompson's first paper, to give the flavour of the method. The example is rather long, but this does show the basic principle of programming with interactions; that of constructing larger interactions from combinations of smaller interactions

First some simple interactions for basic line I/O:

a) Output a string:

```
write :: string → interact * *
write outstring (in,st) = (in, st, [outstring])
```

b) Input a line:

```
readin :: interact uni string
readin (in, Uni) = (tl in, hd in, [])
```

Now a couple of combinators to show the programming style:

```
makeoutput :: output → (input,*,output) → (input,*,output)
makeoutput piece (in,st,out) = (in,st,piece++out)
```

This combinator simply creates some output. Note: due to the lazy semantics the value of `piece` can be output before any of the other values are known.

```
seq :: interact * ** → interact ** *** → interact * ***
seq inter1 inter2 x
   = make_output out1 (inter2 (rest,st))
     where (rest,st,out1) = inter1 x
```

`Seq` takes two interactions and performs them sequentially, note that `out1` is output before the second interaction is invoked. A combinator, `seq3`, to sequence three interactions is also defined in terms of `seq`.

For the rest of the combinators that are used in the example we give shall just their type and function:

```
change :: * → interact ** *
alt    :: condition * → interact * ** → interact * **
          → interact * **
apply  :: (* → **) → interact * **
```

With condition defined as: `condition * = * → bool`.

`Change` simply replaces the state value. `Alt` selects one of the interactions based on the result of the condition. `Apply` applies the function argument to the input state to produce the output state. The input and output values of the interaction are not effected.

And finally the example:

```
getposint = seq3 (write "Please enter an integer: ")
                 readin
                 (alt numeric_string
                      (seq (apply string_int)
                           (add_val_right True)
                      )
                      (seq (write "Try again.\n")
                           (change (0, False))
                      )
                 )
where
     add_val_right = apply (add_r x)
                           where add_r x v = (v, x)
     numeric_string = predicate to determine if number
     string_int = function to convert string to number
```

This example repeatedly prompts for input until a positive integer is typed, it then converts the input string to a numeric value and returns it. As can be clearly seen a lot of combinators and interactions are used even for this simple program.

This proposal can be thought of as "clean streams". The software methodology of interactions and combinators provides what Thompson calls *"functional structure rather than functional spaghetti"*. However, the fact that they are just simple character streams means they only satisfy I/O category (a), and their synchronisation relies totally on *a software discipline* and can easily be (accidentally) broken as the underlying streams are accessible [3.2.1.].

3.4.5. Combinators II – Dwelly

Independently from Thompson, Andrew Dwelly has made a similar proposal [Dwelly 87,88a], in which he describes a system for graphics. The main difference is that the input and output streams, over which the combinators and interactions operate, are *tuples* of values and a program is of type:

```
program: list(keyboard # mouse # ...) → list(screen # cursor # ...)
```

the ellipsis denoting that there may be other elements in the tuple, for example an indication of an interrupt or other event.

In [Dwelly 88b], the problem of synchronising the input and output of a functional program is tackled. This is based on the method, commonly used in imperative programs such as screen editors but not explicitly mentioned by other researchers into functional I/O, of turning off the operating system's "echoing" function. This is done in an attempt to enable a program to order the display of prompts and user response on the terminal. As a simple example of this method, the following function issues a request and inputs a number (without error checking):

```
getnum : input → output # integer # input
getnum in =
     ("Enter a number:" <> intToStr(num), num, remainder)
where
     (num, remainder) = parse_integer in
```

As already stated, Dwelly's work has many similarities to that of Thompson, and in general the same disadvantages hold. The use of tupled input/output streams is an attempt to solve the problem of handling multiple streams, but there are a number of disadvantages to the method:

- Any input device which does not have anything meaningful to return, e.g. no key has been hit on the keyboard, must supply a hiaton, i.e. a "nothing to report" value. The program must on each input examine each element of the tuple and either process or discard it as appropriate. This clearly complicates the program.
- New devices cannot be added dynamically during execution; tuples are of fixed size.
- There is no obvious way to extend the method to handle files etc.. Though I/O categories (a, b) are in some way met, (c, d, f) are not addressed at all and (e) can only be handled in a polled manner.

The control of echoing by the program, though novel in the functional arena, offers no major advance towards solving the general I/O problem. Dwelly himself states that not all problems are amenable to the method and that some programs *"should be carefully constructed"*. In particular, input could still be read from a terminal *before* a prompt has been printed, though it will not be echoed until *after* the prompt.

3.4.6. Continuation Streams – Karlsson

In his work on Nebula, a functional operating system, Karlsson proposes the use of *continuations* for I/O [Karlsson]. The idea of continuations comes from the tail functions of [Mazurkiewicz]. Previously continuations have been used to model state in denotational semantics [Strachey] and to provide a method of flexible program flow control in the imperative language Scheme [Steele].

Under Nebula a program uses a library of functions which provide a set of system calls. These functions operate on an input stream of operating system replies and an output stream of operating system requests. Each of these functions takes a number of arguments appropriate to the operation, the reply stream, and a continuation function. The result of the function is the system request stream, the head of which is a value representing the operation request, and the tail of which is the application of the continuation function to the head of the reply stream, which will be the system's response to the operation. This system relies on laziness, as the function which creates the request is also passed a stream whose head is the response to that request.

As examples we shall show two system calls, taken from the paper. First the *read* call:

```
[fileid → int → (string → interaction) → interaction]
read file length cont sys
     = (READ, (file, length)),
       cont ([string](head sys)) (tail sys)
```

where:

```
interaction == replies → calls
```

As a short note on Nebula syntax: [...] is used to contain type declarations, the `[string]` above is to type-cast the head of the system reply list; as presented Nebula is not typeable and requires these casts, however, Karlsson claims that Nebula could be modified to be typeable. This function constructs the system call stream whose head is a value requesting a file read operation and whose tail is the continuation function applied to the head of the reply stream, which is the result of the read, and the tail of the reply stream, which will (later!) contain the result of future requests. Similarly the *write* call:

```
[fileid → string → interaction → interaction]
write file str cont sys
     = (WRITE, (file, str)),
       cont sys
```

Note here that the system returns no response to a write request so the system reply stream is simply passed unaltered to the continuation function. Finally the *exit* call:

```
[programstatus → interaction]
exit status sys = status
```

On reading this request the system terminates the program.

Using the above, and others, programs can be easily constructed which perform any I/O related tasks. For example, a simple program taken from the paper, which repeatedly reads a character and outputs a response:

```
main where

      [string_array → interaction]
      main args = hello

      [interaction]
      hello = read input 1
                  [string → interaction]
                  {λs. case first s in
                        'h': 'H':
                              write output "Hello!\n" hello
                        'b': 'B':
                              write output "Bye!\n" (exit OK)
                        out
                              exit OK
                  }
```

Obviously the computation between issuing continuations can be much more than in this example, but it does show the structure of programs written using this method. All the I/O categories (a → f) can be met under Nebula by providing suitable system requests. As a final example we shall show how Nebula can cope with category (e), that of handling signals – which from the other proposals seems to be one of the least tractable problems for functional I/O systems. Nebula provides the following system call function:

```
[signal_type → interaction → interaction → interaction]
catch sig sigcont cont sys
     = (CATCH, (sig, sigcont)),
       cont sys
```

This function passes a continuation function, `sigcont`, to the system and then continues by applying the other continuation function, `cont`, to the system reply stream. On the occurrence of a signal, `sig`, the system discards the current system request stream and obtains a new one by applying the previously supplied continuation function, `sigcont`. Thus a new thread of computation is started when a signal is received and provided that a program issues new catch

requests when it wishes to return to a different state, not much computation need be lost when a signal occurs. Here is an example:

```
main where
     [string_array → interaction]
     main args = catch BREAK stop print

     [interaction]
     print = write output "Please hit interrupt key!\n" print

     [interaction]
     stop = write output "Thanks!\n" (exit OK)
```

This program simply loops printing a message until the interrupt key is hit – a rather pointless program, but the method works for larger examples.

From the above, it initially appears that continuation streams are the solution to the I/O problem. Others have shown interest in the idea, in particular, Haskell provides a similar system (see below). Unfortunately there are two problems, the first philosophical the other fundamental:

- Many people have rejected continuations on the basis that they feel that program structure is distorted by their use. Also the fact that a program has supplied as its input the responses to all the requests it will ever ask has been thought of as a "complicated" notion. These objections are however (only) philosophical, they do not in themselves invalidate the proposal.
- The fundamental problem, however, is that they are based on streams with synchronisation only guaranteed if both: the library functions are correct, which is again relying on software [3.2.1]; and used as the sole mechanism for operating on the request and result streams. This however is *not* guaranteed, so intra-request synchronisation is not either.

3.4.7. Haskell I/O

Haskell [Hudak 88a] is a recently proposed functional language. The Haskell report claims to unify the stream and continuation styles of I/O, providing an interface in each style. However, the underlying I/O model is stream I/O, and is a minor variation of Stoye's work, the other interface is simply a continuation *style* interface to the underlying streams, not unlike Nebula. Haskell I/O therefore has the same advantages and disadvantages as Stoye's streams and Nebula's continuations without providing any advance on either.

3.4.8. Summary of Streams

For the previous sections it can be seen that stream-based solutions for I/O have been very popular, presumably due to their perceived advantages:

- Basic streams are simple and can often easily be handled using a common functional programming paradigm – that of list processing.
- Streams can be extended in various ways to handle most I/O related tasks, though this may make them less amenable to straightforward list processing type manipulation.

Although some of the stream-based systems provide, or are capable of providing, all of the I/O categories they all suffer from the same fundamental problem: streams have an inherent synchronisation problem which manifests itself in different ways in the various systems and no proposal has succeeded completely in overcoming this problem. Thompson has progressed a long way by encapsulating all stream accesses to keep them safe, and has provided a formal semantics to prove his systems' correctness. Karlsson has also carefully encapsulated the stream accesses to ensure synchronisation. However, both these systems can be broken accidentally [3.2.1.].

3.5. Result Continuations

We now describe a new I/O system which provides for all of the I/O categories and which does not suffer from the problems of stream-based systems. The proposal is the convergence of two fundamentally different approaches to the provision of I/O:

a) **Functional/Imperative Separation.** It can been argued that given functional languages are referentially transparent and I/O is not, then the two simply *cannot* be combined. Therefore the only way to bring the benefits of functional programming to solving real problems is to somehow embed a functional core, to be used for computation, into an imperative outer shell, to be used for I/O operations. This would work, though the effect on program structure would be dramatic.

b) **Stream-based systems**. Though suffering from an underlying flaw, a great deal can be achieved using a stream-based system – as shown in particular by [Karlsson, Stoye and Thompson 86].

Our new proposal, which we call *result continuations,* combines ideas from both the above approaches. Under this system a program is of type:

$$\alpha \rightarrow \texttt{Result}$$

where

`Result : <operation request> # (β → Result)`

That is, a *program* is a function from some initial state, of type α, to a value, of type `Result`. A `Result` is a pair consisting of an *operation request* and a *continuation function*, of type β → `Result`. The evaluation of a program proceeds as follows

i) Apply the `program` function to an initial value of type α. This results in a value of type `Result`.

ii) From the result value determine which operation is requested and perform it. This will result in a value of type β.

iii) If the operation was not a termination request, apply the continuation function to β to obtain a new value of type `Result`. Goto ii)

iv) If the operation was a termination request then the sequence of computation initiated by the program has terminated. Return control to the system.

The program and each continuation function are written in a functional language and therefore referentially transparent. By providing a set of operation requests to cover all the I/O categories, we have a complete I/O system which is referentially transparent and provides for the relative temporal ordering of requests. It is instructive to see how this system compares to a) and b) above:

a) **Functional/Imperative Separation.** In this system we have divided the problem into its two parts, referentially transparent computation and imperative I/O, but we have achieved the separation without recourse to a user-supplied imperative shell. All the I/O is handled by the system, through the set of operation requests; this is analogous to a conventional operating system and its system calls.

b) **Stream-based systems.** The system can provide all the facilities of stream-based systems, while also ensuring correct I/O behaviour.

As shown in section 3.4, the fundamental disadvantage with all the stream-based systems was that synchronisation could be broken. Using result continuations referential transparency is maintained and both inter and intra-request synchronisation is guaranteed. This is due to the following three properties:

- Firstly, a program performs no I/O operations but merely terminates with a request for an operation to be performed. Thus referential transparency is unaffected.
- Secondly, since a program is supplied the result of any *previous* I/O operation as input (i.e. an operation which was requested *before* the program was started), then the result of an operation cannot be requested before that operation itself has been carried out; this guarantees intra-request synchronisation.
- Finally, since a program is supplied the result of any *previous* I/O operation and evaluates to a request to perform the *next* I/O operation, then the sequence of operations is strictly ordered. Thus inter-request synchronisation is also guaranteed.

For convenience, from now on we shall normally refer to "result continuations" as "continuations" and use the term "continuation for" to refer to "result value with an operation request for".

Result continuations follow on from previous work, in particular that of [Stoye, Karlsson]. However, unlike previous proposals result continuations are not based on streams, which are the root cause of those proposals problems. Stream-based systems, including the Nebula (and hence Haskell) continuation style variant, provide an I/O interface which is inherently insecure and rely on careful programming to provide synchronisation. The result continuation interface is inherently secure in that programs which would be incorrect due to the insecurity of stream I/O simply cannot be written using result continuations. We believe that relying on the language instead of a programmed solution for I/O synchronisation is far better.

Result continuations effect the way in which programs may be reasoned about as we must now consider a *sequence* of programs rather than just one, and standard functional language techniques do not apply in this situation. However, Milner's π-calculus [Milner 89], with its support for dynamic communication unlike its predecessor CCS, may be used. We have not applied the π-calculus to the system described here, but we are using it for our current work on concurrent functional programming based on result continuations.

3.6. A Language with Result Continuations

Hope+C is a research language, based on Hope+, designed to demonstrate the result continuation system. (Hope+ itself provides only simple stream-based I/O.) In this section we will first describe the types of continuations provided by Hope+C, illustrated by some simple program fragments. In Appendix 1 we list the full set of continuations currently provided by Hope+C. We then give a short example Hope+C program, some longer examples are provided in Appendix 2.

3.6.1. Hope+C Continuations

Continuations in Hope+C are represented by the predefined `Result` data type which has a separate constructor for each operation provided. In addition to the result of the previous operation, continuation functions are passed a value of type `Status`, which indicates whether the previous operation was completed successfully. These types are defined in Hope+C as follows:

```
data Status == Ok ++ Failed(num);
data Result ==
        GetChar(Status # char -> Result)
    ++ one constructor for each operation
    ;
```

3.6.1.1. Terminal I/O

All I/O in Hope+C is performed over *channels*. The are two pre-declared channels which are connected to the user's keyboard and screen. Continuations are provided to perform both character and line based I/O over these channels. For example, the following function displays a prompt and reads a line of input. It uses two continuations; `PutStr` to output a string, and `GetLine` to read a line of input:

```
dec ask : list(char) # (Status # list(char) -> Result)
        -> Result;
--- ask(question, myContinuation)
 <= PutStr(question,
           lambda _ => GetLine(myContinuation) end
          );
```

3.6.1.2. Device I/O.

Continuations are provided to open channels for reading and writing files; Hope+C runs over a host operating system so these are host files. The same I/O continuations used for terminal I/O

are available for these channels. For example, the following function attempts to open a file and terminates the program if it cannot do so. It uses three continuations: `OpenChannel` to open the file, `PutStr` to display an error message, and `Exit` to terminate the program:

```
dec mustopen : list(char) # (channel -> Result) -> Result;
--- mustopen(filename, myContinuation)
 <= OpenChannel(filename, read,
                lambda
                  (Ok, chan) => myContinuation(chan)
                | _ => PutStr("Could not open " <> filename,
                              lambda _ => Exit(0) end)
                end
               );
```

3.6.1.3. Processes.

There are two process models supported; the *parent/child* model which may have character based communication over a bidirectional channel, and the *server* model where processes communicate with an independent server process over a globally named channel. Continuations are provided to create child processes, start up a server process and run system commands. The process model is similar to that of BSD Unix™, but is simple enough to be supported on other systems. For example, the following function runs the system command "who" to obtain a list of users on the system. The continuation `RunCommand` allows the running of system commands as child processes, it takes four arguments: the location of the command, the command arguments, the command input, and a continuation function which is applied to the command output:

```
dec getwho : (list(char) -> Result) -> Result;

--- getwho(myContinuation)
 <= RunCommand("/bin/who", ["who"], "", myContinuation);
```

3.6.1.4. Inter-process Communication.

When a child process is created a channel to be used for communication between the parent and child is returned. The standard I/O continuations may be used on this channel. Continuations are also provided to create globally named channels and to create a local channel connected to a globally named channel. Again the standard I/O continuations can be used over these channels.

3.6.1.5. Signal Handling.

Continuations are provided to set up handlers for asynchronous signals, the evaluation of which *replaces* the evaluation of the current program when a signal occurs†. Continuations are also provided to send asynchronous signals to the current or any child process. For example, the following function, `Loop`, implements the inner loop of a simple command interpreter. By repeatedly setting the interrupt handler to be itself applied to the current state, the function provides a simple rollback facility; if an interrupt occurs while the `obey` function is calculating the next state then the interpreter is restarted with the previous state and a new prompt issued; as soon as a new state is calculated it is saved as the new rollback state. Two continuations are used: `OnSignal` which installs a signal handler, no evaluation of this handler will take place unless a signal occurs; and `Exit` which terminates the program. The function `ask` defined above is also used.

```
dec obey : list(char) # state -> state;
dec Loop : state -> Result;

--- Loop(currentState)
 <= OnSignal(InterruptSignal,
             Loop(currentState),
             ask("Command?",
                 lambda
                   (Ok, in) => Loop(obey(in, currentState))
                 | _ => Exit(0)
                 )
            );
```

3.6.1.6. Environment Interaction.

Continuations are provided to obtain the current date and time from the host operating system etc. For example, the following function returns the current time as a string. The continuation `GetTime` returns the time in seconds since midnight, `printf` is a library function for formatted conversion, it returns a string.

```
dec getTime : (list(char) -> Result) -> Result;
--- getTime(Cont)
 <= GetTime(lambda total
              => let (total2, s) == total divmod 60 in
                 let (h, m) == total2 divmod 60 in
                   Cont(printf("%d:%d:%d",[Unum(h),
                                           Unum(m),Unum(s)]
                               )));
```

---

†Hope+C does not provide for *polled* signals, though the result continuation mechanism can support them.

3.6.2. An Example Hope+C Program

The following is a simple program to evaluate the factorial of a number. Continuations, which are printed in bold, are used to; print a prompt (`PutStr`), read the users response (`GetLine`), format and print the result (`Printf`), and finally to terminate the program (`TheEnd`). `Printf` and `TheEnd` are not primitive continuations but functions defined in terms of primitive operations such as `PutChar` and are provided by a standard library [A2.1.1]. For simplicity none of the status returns from the system are checked in this program; the Hope+ "don't care" pattern ('_') is used to ignore them.

```
uses StandardXfer, CxXfer;   ! import conversion functions and Printf !

dec fact : num -> num;       ! simple recursive factorial !
--- fact(1) <= 1;
--- fact(n) <= n * fact(n-1);

dec printit: Status # list(char) -> Result;

--- main                     ! program execution starts here !
 <= PutStr("Input your number: ",
          lambda _ => GetLine(printit) end);

--- printit(_, inp)          ! function to print result !
 <= let n == strtonum(inp)   ! strtonum comes from StandardXfer !
    in Printf("The factorial of %d is %d\n",
              [Unum(n), Unum(fact(n))], TheEnd);
```

In Appendix 2 two more examples are given, both of which emulate the Unix word count program ('wc'), one is written using character I/O operations and the other using "stream" operations. These show that different programming styles can easily be accommodated under the result continuation system.

3.7. Implementation Strategies for Result Continuations

In Hope+C, result continuations have been implemented using a public sum type and each continuation function has a different range type which matches the result type of the operation being performed. This is not the only style in which result continuations may be introduced into a language, in this section we cover some of the other possibilities to demonstrate the adaptability of the result continuation system to different programming styles.

3.7.1. User Accessible Functions or Constructors

A result continuation is a *value* which is returned from a program, in Hope+C this is clear from the representation of the type `Result` as a public sum type. Alternatively, the continuations could be implemented as an abstract type with the user only having access to functions which return continuation values. A possible advantage of the latter approach is that it enables the continuations to be easily split up into functional subsets, for example; character I/O and process operations. Each subset can be provided by a different module and programmers need only know about those modules which they use. Using the former approach all the constructors of the data type must come from the same module, so all the continuations are visible at the same time. Another possible advantage of the latter approach is that it blurs the distinct between those operations which are primitive continuations and those which are implemented by function libraries.

In Hope+C the continuations were represented by a public sum type due to the different semantics of function application and value construction, the former being call-by-WHNF and the latter non-strict. For example, if the `OnSignal` continuation were provided to the programmer as a function then the signal handler argument would be evaluated to WHNF even if no signal ever occurred, as a constructor argument the signal handler remains unevaluated until, and if, a signal occurs. The choice between the constructor and function variations in Hope+C effects the amount of computation done, but not the result or effect of the program. In a non-strict language either choice would have the same semantics.

3.7.2. Continuation Function Type

There are two basic possibilities for the style of the continuation function. The first, as used in Hope+C, is for the continuation function to have a different type for each continuation, for example:

```
data Result ==
    Getchar(channel # (status # char -> Result))
 ++ Putchar(channel # char # (status -> Result))
 ++ ...
```

The second possibility is for each continuation function to have the same type, with the different return possibilities represented by a sum type, for example:

```
type Result == Request # (Return -> Result);
data Request ==
    Getchar(channel)
 ++ Putchar(channel # char)
 ++ ...
data Return ==
    Failure(reason)
 ++ Getchar(char)
 ++ Putchar
 ++ ...
```

The former does not require pattern matching of the result and the return values are directly accessible. It also allows a failure result to be easily ignored, where this is appropriate. For the latter method pattern matching must always be used to access the return values, but higher-order control constructs can sometimes be easier to write. The difference is largely cosmetic as one method may easily be mapped into the other. As an example, the following is a function to implement a "while" style construct, the latter style being used. The function `while` repeatedly executes a body function (parameter 4) as long as the return from a condition function (parameter 2) passes a predicate test (parameter 3)†:

```
dec while : Return                ! initial value !
          # (Return -> Request)   ! condition request !
          # (Return -> truval)    ! test on cond. return !
          # (Return -> Result)    ! body function !
          # (Return -> Result)    ! continuation !
          -> Result;

dec wtest : Return                ! condition return !
          # (Return -> Request)   ! condition request !
          # (Return -> truval)    ! test on cond. return !
          # (Return -> Result)    ! body function !
          # (Return -> Result)    ! continuation !
          -> Result;

--- while(init, cond, test, body, finish)
 <= (cond(init), wtest(?, cond, test, body, finish));
--- wtest(ret, cond, test, body, finish)
 <= if test(ret)
    then (body(ret), while(?, cond, test, body, finish))
    else finish(ret);
```

†The "?" as a function argument is Hope+C partial application syntax. I.e. the value of f(?, x) is a one argument function g(y) = f(y, x).

3.7.3. Separate Success and Failure Continuation Functions

As so far described, a result continuation possesses only one continuation function, to which both successful and failure returns are passed. Alternatively each continuation may have two continuation functions, one each for the success and failure cases. For example:

```
data Continuation ==
    Getchar(channel # (char -> Continuation)
                    # (status -> Continuation))
 ++ Putchar(channel # char # Continuation
                           # (status -> Continuation))
 ++ ...
```

This approach saves checking for failure, but does make it harder for execution to continue along the *same* path regardless of success or failure.

3.7.4. Low or High Level Operations

In the above examples the continuations have provided *low-level* operations, such as single character I/O. *High-level* operations, such as formatted output, can either be provided as basic continuations or as library functions which themselves use low-level continuations. We feel that only low-level operations should be provided with high-level ones being written in the host functional language, but the alternative choice may be made.

3.8. Higher-Order Programming with Result Continuations

At first it may seem that using result continuations will have a major impact on programming style, in particular the use of higher-order functions. Most of the examples given in the previous sections are first-order and their structure is very similar to that of an imperative language solution to the same problem. The "while" example, however, was a higher-order function written using result continuations, we refer to this as *continuation passing style* (CPS), which we define more precisely below. To demonstrate that the typical functional language higher-order programming style may continue to be used when using result continuations in this section we present two common higher-order functions written in CPS. In Appendix 2 we include a module of CPS higher-order functions and two more versions of our example program, wc, written this time using higher-order functions.

Definition of CPS functions

A "normal" function has a type of the form: $\alpha \rightarrow \beta$, and a definition of the form: f x $\leftarrow$ E, where α, β are arbitrary types and *E* is an expression in *x*. A CPS function has a type of the form: $\alpha \# (\beta \rightarrow \gamma) \rightarrow \gamma$, where γ is the continuation type, and a definition of the form: f x g $\leftarrow$ g E, where *g* is the continuation function; that is, instead of returning its result value a CPS function applies a functional argument to it.

3.8.1. CPS Higher-Order Functions

The first function we define is `compose` which forms the composition of two functions. The usual definition of this function would be:

```
dec compose : (beta -> gamma) # (alpha -> beta)
            -> (alpha -> gamma);
--- compose(f, g) <= lambda x => f(g x) end;
```

The CPS version of compose takes two CPS functions and forms a CPS composition function, its definition is:

```
dec compose : (beta # (gamma -> delta) -> delta)
             # (alpha # (beta -> delta) -> delta)
            -> (alpha # (gamma -> delta) -> delta)
--- compose(f, g) <= lambda (x, h) <= g(x, f(?, h)) end;
```

Next, two variations of the higher-order list reduction function, first a typical (non-curried) definition:

```
dec reduce : (alpha # beta -> beta) # beta # list(alpha)
           -> beta;
--- reduce(f, acc, nil) <= acc;
--- reduce(f, acc, head::tail) <= reduce(f, f(head, acc), tail);
```

Now a CPS version:

```
dec reduce : (alpha # beta # (beta -> delta) -> delta)
             # beta # list(alpha) # (beta -> delta)
           -> delta;
--- reduce(f, acc, nil, cont) <= cont(acc);
--- reduce(f, acc, head::tail, cont)
 <= f(head, acc, reduce(f, ?, tail, cont));
```

As a simple example of how CPS higher-order functions can be used, the following function averages a list of numbers, printing each number as it is processed and then the average before passing it on to its continuation:

```
dec average : list(real) # (real -> Result) -> Result;
--- average(l, after)
 <= let SumAndPrint
        == lambda (elem, (count,sum), cont)
           => Printf("%e\n", [Ureal(elem)],
                        lambda _ => cont(count+1, sum+elem) end)
           end
    in let AverageAndPrint
           == lambda (count,sum)
              => let aver == sum / count in
                    Printf("Average = %e\n", Ureal(aver),
                           lambda _ => after(aver) end)
              end
       in reduce(SumAndPrint, (0, 0.0), l, AverageAndPrint);
```

As can be seen from these examples, the writing of CPS functions is not difficult. The examples given in Appendix 2 show that a problem which would be solved in a higher-order style, if designed to operate on program resident data, can easily be written in a CPS higher-order style, if operating over data obtained via result continuation I/O.

3.9. Conclusions and Related Work

Result continuations are a powerful and flexible I/O system for functional languages, which do not suffer from any of the synchronisation problems of existing proposals. They have been shown to be practical though their inclusion in Hope+C and their subsequent use in a number of programming projects, both large and small. The Hope+C compiler itself is written in Hope+C and uses result continuations for all its I/O. We are currently undertaking research into the combining of the functional and object/process-oriented programming paradigms to produce a concurrent functional language. The use of result continuations to support message passing is fundamental to this development.

Much of the related work has already been mentioned, though our coverage inevitably is not comprehensive. Work recently reported by [Gordon 88] has both similarities and differences to our work. His system, as described in [Gordon 89], is also a continuation based system and though the syntax is very different has many similarities to result continuations. There are also some fundamental differences; in particular whereas we choose to implement list I/O operations as primitives, Gordon provides a lower-level and more flexible operation from which list I/O operations may be constructed. The disadvantage, in our view, of Gordon's approach is that it introduces non-determinism into the language, which we choose to avoid due to the problems

this causes – especially to some of the work on program transformation. In research already started, but outside the timescale of this thesis, we plan to address non-determinism in a functional language by introducing explicit concurrency, further details cannot be provided until this research has progressed further.

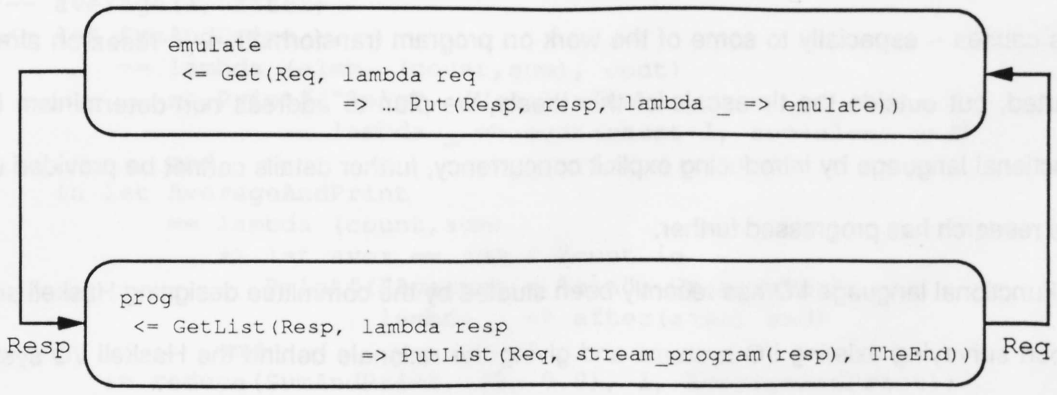
Functional language I/O has recently been studied by the committee designing Haskell and a report surveying existing I/O systems and giving the rationale behind the Haskell I/O system published by Hudak and Sundaresh [Hudak 88b]. In this report it is argued that stream and continuation I/O systems have the same expressive power; this is done by showing that programs may be written to emulate one over the other. It is argued that streams should be the *primitive* I/O mechanism with continuations being provided by software emulation, this choice being justified on the basis that emulating streams over continuations must incur a space leak. The following is their emulation algorithm implemented in Hope+C:

```
data Request == getChar(channel) ++ other requests;
data Response == fail(num) ++ retChar(char) ++ other responses;
type StreamProgram == list(Response) -> list(Request);

dec errorResp : list(Response);
--- errorResp <= fail(-1) :: errorResp;

dec emulate : StreamProgram -> Result;
--- emulate(sprog)
 <= case sprog(errorResp) in
      nil => Exit(0)
    | getChar(chan) :: rest
      => GetChar(chan,
                 lambda
                   (Ok, c)
                   => emulate(lambda resp
                                => tail sprog(retChar(c) :: resp)
                                end)
                 | (Fail(n), _)
                   => emulate(lambda resp
                                => tail sprog(fail(n) :: resp)
                                end)
                 end)
    | other cases
    end;
```

Briefly, the space leak occurs in this algorithm due to the repeated re-evaluation of the stream program `sprog` to obtain the next output token, see [Hudak 88b] for further details. Using the set of result continuation operations provided by Hope+C as described in this thesis this space leak cannot be avoided. However, in a recent preliminary report [Perry 89c] we show how the



result continuation mechanism can be used to support coroutines and remote function calls, both of which are difficult to implement using streams. Using the result continuation mechanism in this more general way it is possible to emulate streams over continuations without producing any space leaks by setting up two coroutines, one of which runs the stream program and performs list I/O, the other which performs item I/O and interprets the stream requests. A sketch of this system is given in the diagram above. We have produced an experimental version of Hope+C which includes these extensions and have used it in part to demonstrate the ability of result continuations to emulate stream-based programs in this way. This work is still in progress and further details are outside the scope of this thesis.

It is worth noting in passing that [Hudak 88b] considers only Nebula [Karlsson] stream continuations and therefore assumes that synchronisation is a software rather than a semantic issue. This is in contrast to result continuations where synchronisation is provided by the semantics [3.5].

Finally, in Cupitt's recent thesis [Cupitt] he discusses functional language I/O and operating system design [see Chapter 6]. The work stems in part of that of [Turner 87], but Cupitt found that streams I/O as originally envisaged by Turner was impractical and changed to continuation based I/O. Cupitt's continuations are implemented over Miranda streams [Turner 85], so are closer to Nebula [Karlsson] than to result continuations. However, his switch to continuations from streams clearly supports the thesis of this chapter, and his practical reasons for doing so contradicts the suggestions of [Hudak 88b].

4. The Efficient Sequential Implementation of Functional Languages

One of the requirements of a practical functional language is the provision of an efficient implementation. In this chapter we present a system, Fpm2, which has been designed to enable functional languages to be implemented on current sequential hardware with comparable performance to that imperative languages. Although this design is targeted at sequential machines it is also extendable to coarse-grained parallel implementations on current MIMD parallel machines.

Most functional language implementations are based upon new abstract machine architectures which have been specially designed to exploit the peculiar properties and requirements of these languages; for example the G-Machine [Augustsson 88], FAM [Cardelli 84], CAM [Cousineau], TIM [Wray] and SKI combinators [Turner 79]. Many of these implementation methods have been further refined so that they are a closer match to the model supported by current sequential hardware, or to enable non-strictness and higher-order functions to be more efficiently supported; for example the Spineless Tagless G-Machine [Peyton Jones 88] and super-combinators [Hughes].

A different approach was taken by Field [Field 85], and since elaborated in [Field 88], when implementing the FP/M abstract machine [Bailey] for a VAX compiler for IC Hope, an implementation of the strict functional language Hope [Burstall]. Here, FP/M was compiled using a fairly conventional stack and heap model similar to that used for Algol-60. *Thunks* were used to implement function closures, as is done in Algol-60, but unlike the conventional stack/heap model multiple stacks were used: one for function arguments and local values and the other for system information such as the function return addresses. Despite not being designed particularly for optimum speed, the FP/M implementation produced reasonably fast code[†] which was slower than comparable imperative languages but faster than existing functional language implementations.

[†] No benchmark figures survive for Field's VAX FP/M, but a straight transliteration into 68K for a Sun-3 produced an Nfib rating of just over 100K reductions/second, significantly slower than Fpm2 but faster than LML [see 4.7]

The implementation technique introduced in this chapter, which we have called Fpm2, builds upon the ideas of the original FP/M system. Fpm2 uses a version of FP/M, Fpm+ [Perry 87], which consists of the original FP/M extended to support the module structure of Hope+. The compilation of Fpm+ into native machine code is however completely different from that of the original FP/M system, although the model used is still the conventional stack and heap. The multiple stacks and fixed heap of the FP/M system are replaced by a single stack and dynamic heap, full support for suspensions is added so that Fpm2 can be used to implement Hope+C, and a number of optimizations are included which improve performance by targeting the differences between Algol-60 and Hope+C – most notably the higher usage of thunks in Hope+C. The performance of Fpm2 is roughly three times that of FP/M and is shown to be comparable to that of imperative languages in our benchmark tests [4.5].

In the remainder of this chapter we describe Fpm2 in detail, elaborate on the differences between FP/M and Fpm2, and include benchmark results which compare the Hope+C/Fpm2 combination to SML, LML and Pascal.

4.1. Overview of Fpm2

The Fpm2 system consists of three parts:

Fpm+ – The Compiler Target Language

The Fpm+ language is a simple S-expression language suitable for translating high level functional languages into as part of their compilation route. Fpm+ is itself currently a functional language but future developments, for example the support for in-place update, may not guarantee this.

Fpm/AM – The Abstract Machine

The Fpm/AM is an abstract machine definition designed for easy compilation from Fpm+, and to conventional architectures.

Fpm/CM – The Concrete Machine

The Fpm/CM is an implementation of the Fpm/AM. The Fpm/CM currently runs on Sun-3 workstations.

4.1.1. Developments from FP/M

As mentioned above Fpm2 is a development of FP/M, we present here an overview of the developments which form our research contribution, the rest of this chapter covers them in greater detail.

The Compiler Target Language

Fpm+ [Perry 87] is an extension of FP/M which was designed by [Bailey] and later described in [Field 88]. The extensions to FP/M are minor and support the module system of Hope+.

The Abstract and Concrete Machines

FP/M was first used as part of the IC Hope compiler for VAX machines [Field 85], and the basic compilation scheme from FP/M to machine code is described in [Field 88]. In this implementation the division between the FP/M abstract and concrete machines is less clear cut than in Fpm2; FP/M is compiled directly into VAX macro assembler using a large number of pre-defined macros, these macros form the equivalent of our Fpm/AM. VAX FP/M uses two stacks and one heap; the system stack is used only for function return addresses, and is the only dynamic memory structure. A separate fixed size stack is used to hold function arguments, the separation of function return addresses and arguments onto separate stacks being done to ease the task of garbage collection. The FP/M heap is allocated a fixed area of memory and can not grow dynamically during program execution. IC Hope supports very limited non-strictness in the form of a "lazy cons", all other data structures being strict. Thus support for suspensions is very limited with only the macro instructions M_HEAD, M_TAIL and M_LCONS dealing with them. Suspensions and closures are constructed on the heap and before evaluation are unpacked onto the stack.

For Fpm2 a clear distinction was first made between the abstract and concrete machines. The instructions of the Fpm/AM are based on the VAX macros of FP/M, but with many new additions. Hope+ has a much greater support for non-strictness, this requires the Fpm/AM instruction set to support suspensions for any data structure including tuples.

The Fpm/CM is a completely new implementation written in 68K assembler for Sun-3 machines. In the Fpm/CM only one stack is used, with both the function return address and arguments being stored on the system stack, which is of course dynamic. This removes the

limitation of the fixed sized argument stack of VAX FP/M. The Fpm/CM heap is dynamically allocated at the start of program execution and grows dynamically to meet the requirements of the program. This again removes the limitation of the fixed sized heap found in VAX FP/M. The garbage collector is based on the same two-space copying algorithm used by VAX FP/M, but enhanced to allow for the mixed address/value stack and to support the dynamic heap resizing. As in VAX FP/M, closures and suspensions are constructed as heap objects. However, when evaluated they are not unpacked onto the stack, instead the code for their functions bodies is compiled to directly access the heap structures to obtain the values of bound variables. This change saves a considerable amount of redundant data movement between the heap and the stack. There are many other differences between VAX FP/M and Fpm/CM, including support for non-Hope+ structures in the heap, the continuation I/O system [chapter 3], and the inter-language calling interface [chapter 5].

4.1.2. Implementation Model

Unlike many implementations of functional languages Fpm2 is compiled directly for a "conventional" stack/heap model rather than going via another one first, such as the graph reduction model used by the G-Machine [Augustsson 88] and its descendants. Although other models have been designed to suit functional languages, our stack/heap model is closer to the hardware and by choosing this approach higher performance is more immediately obtainable, as is demonstrated by the comparative timings below [4.5]. In particular, the current implementation of Fpm/AM and Fpm/CM for Sun-3s supports our assertion as it is significantly faster for strict problems and comparable for lazy ones when compared to the LML G-Machine implementation.

4.1.3. Summary

Fpm2 is a major development from the ideas of FP/M, which supports the non-strictness requirements of Hope+ while providing superior performance through a completely redesigned implementation. The performance of Fpm2, as judged from benchmark results, is comparable or better than that of other functional language abstract machines. The remainder of this chapter first describes the constituents of Fpm2; Fpm+, Fpm/AM and Fpm/CM; in detail.

4.2. The Language - Fpm+

Fpm+ is a simple S-expression language. The language has three forms of values: simple unstructured values – which are used to represent integers, reals, characters and booleans; lists and tuples.

The language is untyped, though the constant operator does distinguish the type of its argument, and therefore no type checking is done. It is assumed that the compiler from the original source language – in our case Hope+C – has already checked that the program is correctly typed. Not only is this assumption made, but built-in non-arithmetic functions assume that their operation is valid, for example, the built-in function to take the head of a list assumes that it is not given an empty list! These assumptions make Fpm+ only suitable for automatic generation by compiler as an intermediate code.

4.2.1. The Syntax

This is the complete BNF for Fpm+. Terminals are shown in **bold**.

```
<fpm program> ::= ( <module definition list> )
<module definition list> ::= <module definition>+
<module definition> ::= (<module name> <uses list> <export list>
                         <function list> <ignored> )
        | 0 <uses list> <ignored>
                         <function list> <expression> )
<uses list> ::= ( <module name>* )
<export list> ::= ( <function number>* )
<function list> ::= ( <function definition>* )
<function definition> ::= ( 1 <number of arguments> <ignored>
                         <function name> <module name> )
        | ( 0 <number of arguments> <expression>
                         <function name> <ignored> )

<expression> ::= <constant value>
        | <local value>
        | <conditional>
        | <construct tuple>
        | <function application>
        | <temporary value>

<constant value> ::= ( cv <value> )
<local value> ::= ( lv <local name> )
<conditional> ::= ( if <predicate expression> <expression>+ )
<construct tuple> ::= ( ct <expression> <expression>+ )
<function application> ::= ( af <function tag> <expression>* )
<temporary value> ::= ( tv <arity> <expression> <expression> )
```

<function tag> ::= (**ud** <function number>)
| (**bi** <function number>)
| (**ev** <function expression>)

<function expression> ::= <local value>
| <conditional>
| <construct tuple>
| <function application>
| <temporary value>
| (**cv** (<number of arguments> <ignored> <expression>))

<module name> ::= <quoted string>
<function name> ::= <quoted string>
<number of arguments> ::= <integer>
<local name> ::= <integer>
<value> ::= <integer> | <real> | '<character>' | <quoted string> | (<value>*)
<quoted string> ::= "<character>*"
<real> ::= <integer>.<unsigned integer>{**e**<integer>}
<integer> ::= <digit>+

<digit> ::= **0** | ... | **9**
<character> ::= any printable character | <character representation>
<character representation> ::= **\n** | **\t** | **\f** | **\'** | **\"**
<ignored> ::= **0** | <expr>

Notes:

- the <ignored>s are present only for backward compatibility with Fpm.
- In the current compiler all alphabetic terminals are represented by numbers: **cv**, **af**, **lv**, **if**, **ct** and **tv** by 1, 2, 4 → 7 respectively; and **ud**, **ev** and **bi** by 1 → 3 respectively.

Fpm+ does not specify whether strict of lazy evaluation semantics are used, it is left to the high level language and abstract machine to agree on this matter.

4.2.2. Program & Module Structure

An Fpm+ program consists of a list of modules. If none of the modules is a program module then the program is a set of separately compiled modules. A program containing a program module, along with any modules referenced but not in the current program, is an executable program. It is an error for a program to contain more than one program module.

4.2.2.1. "Ordinary" Modules

A module is represented by an s-expression consisting of:

<module name> The name of the module.

<uses list> The names of any modules that this module imports – may be empty.

<export list> A list of functions, represented by a number denoting their position in the <function list> below, that this modules exports.

<function list> A list of function definitions – see Function Definitions below.

4.2.2.2. Program Module

A program module is represented by an s-expression consisting of:

0 This distinguishes it from ordinary modules.

<uses list> As above.

<function list> As above.

<expression> The top-level expression, whose denotation is the value of the program.

4.2.2.3. Function Definitions

Functions may either be locally defined or imported from other modules. A local function definition consists of:

0 To indicate this is a locally defined function.

<number of arguments>
The arity of the function. (Note: In Hope+C functions take one argument, being either a single value or an *n*-tuple. Hope+C functions which take *n*-tuple arguments are translated into Fpm+ functions which take *n* discrete arguments.)

<expression> The expression defining the function.

<function name> The name of the function.

An imported function definition consists of:

1 To indicate this function is imported

<number of arguments>
The number of arguments the function takes.

<function name> The name of the function.

<module name> The module from which the function is imported.

4.2.3. Expressions

There are six expressions in Fpm+: *constant value*, *local value*, *conditional*, *construct tuple*, *apply function* and *temporary value*.

4.2.3.1. Constant Value – cv

<constant value> ::= (cv <value>)

The expression returns the value of its constant argument. The argument may be an integer, a real, a character, a string, or a constant list written in parentheses – no checking is done on the types of the list elements.

4.2.3.2. Local Value – lv

<local value> ::= (lv <local-name>)

The <local-name> is an integer, *p*, and the expression returns the value of the p^{th} local value. Arguments to a function of arity *n* are referred as local values *1* through *n*, and temporary values resulting from *tv* expressions (see below) are referred to as local values *n+1* through *m*.

4.2.3.3. Conditional – if

<conditional> ::= (if <predicate expression> <expression>+)

The conditional has multiple branches and is equivalent to a case statement. The <predicate expression> is evaluated to give an integer *n* in the range from zero upwards. The result of the expression is then the result of n^{th} expression after the predicate. Fpm+ represents the boolean values true and false by the integers 1 and 0 respectively; thus given three arguments with the first being a boolean-valued expression, this expression implements a standard conditional.

4.2.3.4. Construct Tuple – ct

<construct tuple> ::= (ct <expression> <expression>+)

This expression returns an *n*-tuple constructed from its *n* argument expressions.

4.2.3.5. Apply Function – af

<function application> ::= (af <function tag> <expression>*)
<function tag> ::= (ud <function number>)
| (bi <function number>)
| (ev <function expression>)

The function is applied to the list of expressions and the resultant value is returned. The function tag is one of:

ud n — The n^{th} user defined function is applied. Functions are numbered from 1 in the <function list>.

bi n — The built-in function designated by *n* is applied. The built-in functions are not part of Fpm+, but are defined by Fpm/AM.

ev <function expression> — If the <function expression> is not a *cv* then it must return a closure, which is applied to the arguments. For *cv* expressions the constant is a function body which is applied to the arguments.

4.2.3.6. Temporary Value – tv

<temporary value> ::= (tv <arity> <expression$_1$> <expression$_2$>)

This is the "local block" expression of Fpm+. If the <arity> is 1 then the result of expression$_1$ is assigned as new local value *n+1*, where *n* is the current number of local values. If the <arity> is greater than one then the result of expression$_1$ is assumed to be a *n*-tuple value, and elements 1 through <arity> of the tuple are assigned as new local values *n+1* through *n+<arity>*. The result of the whole expression is then the value of the expression$_2$; after evaluating expression$_2$ the new local values are discarded.

4.3. The Abstract Machine – Fpm/AM

Fpm/AM is an abstract machine for the implementation of Fpm+. It is designed to be simple and sufficiently minimal to be easily implemented on most current architectures. Fpm/AM defines an extra value, in addition to those supported by Fpm+ namely the *suspension* which represents an un-evaluated function application consisting of a function pointer and a set of argument values. Using suspensions, Fpm+ code representing a Hope+C program can be compiled, with the correct calling semantics, into Fpm/AM by applying the following rules:

- Compiling the *ct* (construct tuple) expression into Fpm/AM code which constructs a tuple with elements each of whose value is:
  - The value of the corresponding sub-expression of the *ct* expression; if it was a *cv* (constant value) or *lv* (local value) expression.

- A suspension consisting of: an automatically generated function, whose body is the original sub-expression of the *ct* expression; together with the set of local values referred to by the sub-expression.

- Defining the application of the built-in *cons* function, which is used to construct a list cell, to construct suspensions of its argument values.
- Defining the built-in functions which select elements of lists and tuples to evaluate any suspensions they extract. This includes the *unpack* function used to place all the elements of a tuple on the stack prior to a function application, and thus means that *at* (apply function) implements call-by-WHNF.

The first two items above require action by the Fpm⁺ → Fpm/AM compiler, the last is an implementation requirement on Fpm/CM. Together the above three implement the calling semantics required for Hope⁺C. The full translation rules between Fpm⁺ and Fpm/AM are given in Appendix 5. (Details of built-in functions, such as *cons* above, are given later [4.3.2.2.].)

4.3.1. The Fpm/AM Architecture

Fpm/AM is a simple machine consisting of:

- An (implicit) program counter.
- A single register capable of holding any Fpm⁺ value, i.e. integers, reals, characters, booleans and pointers to lists, tuples or closures. This is used for intermediate values and function results.
- A stack for the storage of local values, again of any type. The direction of growth is not specified, only that values can be pushed, popped and addressed relative to the top.
- A heap for the storage of lists, tuples, closures and suspensions. When constructing a new object space is first requested as a number of "cells", each capable of holding any value, and then the required values are pushed onto the heap, after which the final value of the heap pointer is taken as the new object pointer. When accessing heap objects, the individual elements are addressed relative to the object pointer. The direction of growth of the heap is not specified.
- A number of basic instructions, to handle for example; function call, return, heap allocation, basic arithmetic etc.

Fpm/AM also defines a set of built-in functions, which perform basic operations like arithmetic, list construction and list and tuple element selection. Many of these functions are also available as Fpm/AM basic instructions, the function form only being used when a closure is required. The full set of built-in functions is given in Appendix 3.

4.3.2. The Instruction Set

An instruction is of the form:

<opcode> <operand> {, <operand>}*

where an operand is one of:

lit(<value>)	A literal ("immediate") value. <value> may be an integer, character, real or label. Labels are used when constructing closures and suspensions to obtain the function address. (Note: The Hope+ → Fpm+ compiler converts booleans into the integers 0 and 1, so they are not present here.)
lab(<label>)	A label. Used to reference both code and data labels generated by the Fpm/AM compiler.
work	The work register. Used for intermediate values and function results.
push	Push a value onto the stack.
pop	Pop a value off the stack.
sp	The stack pointer. Used when doing direct stack manipulation such as discarding a number of stack values.
sp(<offset>)	Access the <offset>th value on the stack. <offset> may be zero, which refers to the top of the stack.
ap(<offset>)	Access the <offset>th value relative to the argument pointer, used only in functions which are evaluated as suspensions or closures.
pusheap	Push a value onto the heap, used when constructing tuples, closures and suspensions.

Operand types are grouped into those allowable as source operands and those as destination operands:

Source Operands: lit, pop, sp(<offset>), ap(<offset>), work, sp.

Destination Operands: lab, push, pusheap, sp(<offset>), ap(<offset>), work, sp.

There are a large number of Fpm/AM <opcode>s and these are listed in Appendix 3.

4.4. The Concrete Machine – Fpm/CM

The Fpm/CM described here is our implementation for the Sun-3, other implementations are possible.

4.4.1. Fpm/CM Implementation

The Sun-3 is based upon the Motorola 68020 cpu and runs the Unix™ operating system. Fpm/CM is implemented on the Sun-3 using:

- A set of macro definitions, written using the Unix™ m4 macro processor, which convert Fpm/AM instructions into 68020 assembler code.
- A runtime kernel, written in 68020 assembler, which contains the code for program initialisation, memory allocation, garbage collection, suspension evaluation and closure application.
- A collection of modules, written either in an Fpm/AM and assembler mix or in Hope+C, which implement the standard modules of Hope+, including the built-in functions. These make up the library "libfpm.a".
- A collection of modules which implement the continuation kernel and standard Hope+C continuation modules. These are also in the library "libfpm.a".

4.4.2. Fpm/CM Architecture

The Fpm/CM architecture consists of:

- **System Stack.**
  This is the standard stack addressed using register *a7* and grows downwards from top of memory. It is used for the Fpm/AM stack and to store the program counter during a function call.
- **Heap.**
  The heap is dynamically allocated above all other data and code and may grow during program execution. The heap is divided into two semi-spaces and garbage collected by a copying garbage collector which switches the current semi-space. Within a semi-space, heap space is allocated from the top down and register *a5* is used as the heap pointer.

Register *d5* is used as a free space counter and is decremented by the Fpm/AM *claim* instruction, if a claim makes *d5* negative then a garbage collect is triggered.

- **Work Register**.

  Register *d7* is used as the Fpm/AM work register.

- **Argument Pointer**.

  Register *a6* is used as the Fpm/AM argument pointer. It is set to address the heap structure when evaluating a suspension or closure.

- **C Heap**.

  The C heap is used for dynamic allocation during inter-language working [chapter 5], it is dynamically allocated at program start-up immediately above the program code and data, but is fixed in size.

- **C Stack**.

  The C stack is used as the system stack during inter-language working, it is also sometimes used by Fpm/CM to store non-heap pointer data. The C stack is dynamically allocated at program start-up and is placed above the C heap and below the standard heap. The stack grows downwards and register *a2* is used as stack pointer.

4.4.3. Register Usage

The following table details the use of each 68020 register by the Fpm/CM:

a0,a1	Used as temporary registers.
a2	C stack pointer.
a3,a4†	Used as temporary registers for tagged values.
a5	Heap pointer.
a6†	Argument pointer during suspension and closure application. This register is setup up by the kernel code (or macro) which is used to evaluate the suspension or closure. It is automatically saved by the kernel (or macro) if a function calls another function or evaluates a suspension.
a7	Stack pointer.
d0,d1,d2	Used as temporary registers.

d3,d4	Unused at present
d5	Heap free space counter.
d6	The value 4. This is used by Fpm/CM to represent, the integer 1, the boolean true and the nil list value.
d7†	The Fpm/AM work register, used for intermediate results and function return values.

† During garbage collection objects addressed by these registers are relocated and the registers updated accordingly.

4.4.4. Fpm/AM Operand Representation

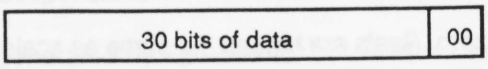
Each Fpm/AM operand type is mapped to an appropriate 68020 operand as follows.

lit(value)	#value
lab(name)	name
work	d7
sp	a7
push	a7@-
pop	a7@+
sp(offset)	a7@(offset:4) [each value is 4 bytes long]
ap(offset)	a6@(2,offset:4) [for suspensions]
	or a6@(6, offset:4) [for closures] [allows for tag value, see 4.4.5. below.]
pusheap	a5@-

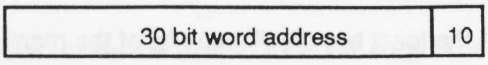
4.4.5. Representation of Values

Fpm/CM values are represented by words (4 bytes, 32bits) which all contain a 2-bit tag field, in contrast to the "boxed" implementation often used, e.g as in the G-Machine, in which all values are pointers to heap structures which contain an identifying tag word. Tags where chosen both to save the extra space and indirection required for boxed values, and because we concluded that they can be implemented with sufficient efficiency. The allocation of the tags has been chosen so that a large number of operations do not need to de-tag a value before using it.

Simple values: appear on stack and in heap objects

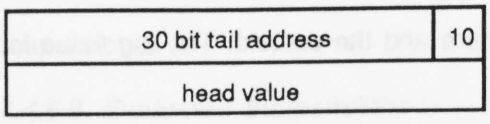


Integer, char and real

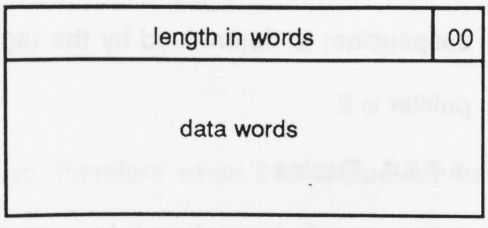


Pointer

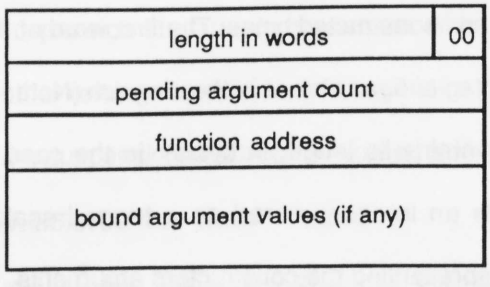
Heap objects



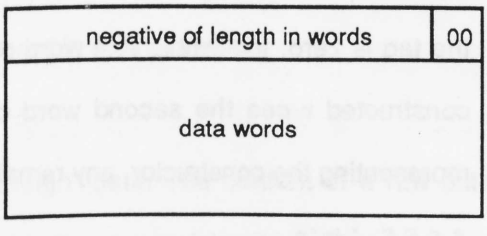
List cell



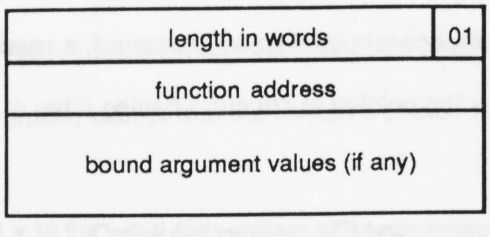
Tuple



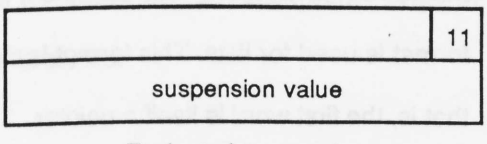
Functon closure



"Raw data"



Suspension



Evaluated suspension

Fpm/CM Data Formats

4.4.5.1. Scalar Values

No distinction is made between integers, booleans and characters. Integers are stored as 30-bit 2's complement values. A boolean is represented by 0 or 1, characters are represented by the numbers 0 through 255. The *nil* list value is also represented as a scalar with the value 1, see 4.4.5.5. below.

4.4.5.2. Real Values

Real values are represented by a 30-bit "IEEE" real, that is, a 32-bit IEEE real representation with the least two significant bits of the mantissa implicitly zero. Reals are tagged the same as scalars, differentiation is by context; as for integers, booleans and characters.

4.4.5.3. Pointer Values

All structured values are referenced via pointers. The type of the structure; tuple, list, closure or suspension; is determined by the tag of the structure and the context. The tag value for a pointer is 2.

4.4.5.4. Tuples

Tuples are used to represent Hope+C product types and constructed types. The first word of the tuple contains the length in words of the tuple, and a tag of 0, as shown in the diagram (Note: as the tag is zero, the whole first word of the tuple contains its length in bytes). In the case of constructed types the second word of the tuple is an integer (stored as a tagged scalar) representing the constructor, any remaining words representing the constructors arguments.

4.4.5.5. Lists

Instead of representing lists as tuples, which is how other constructed types are stored, a special format is used for lists. This format is distinguished by the pointed to structure having a tag of 2, that is, the first word is itself a pointer.

The nil constructor is represented by the scalar 1. Note that this means that if the tail of a list cell is nil, then the list cell looks like a single element tuple [4.4.5.4].

4.4.5.6. Closures

Closures consist of; a function pointer, a pending argument count, and possibly some bound values. The bound values, if any, are from the environment in which the closure was constructed. (These are only present if the closure was formed by compiling a Fpm+ *ev* function tag containing a *cv* expression, these expression result from the compilation of a Hope+C lambda expression.) The pending argument count is the number of arguments required by the function; as Fpm/AM does not support currying, the pending argument count must be satisfied when a closure is evaluated. A closure has a tag of 0 and is distinguished from a tuple by the context (this distinction is implicit, no runtime checks need be made).

4.4.5.7. Suspensions

Suspensions consist of; a function pointer, and possibly some bound values. Suspensions are generated by Fpm/AM for *ct* (construct tuple) expressions, or by the built-in function *cons*. The function address points to a function constructed by the compiler to evaluate the suspended expression. The bound values, if any, are values from the environment at the time of suspension construction which were referenced by the expression being suspended. The tag value for a suspension is 2.

4.4.5.8. Evaluated Suspensions

There may be more than one pointer to a suspension, therefore when it is evaluated, the suspension structure is overwritten with the result and the tag value changed. Therefore if any other references exist to the suspension it need not be evaluated again. A tag of 3 indicates that the suspension has been evaluated and the value is stored in the next word.

4.4.5.9. "Raw Data" Values

"Raw data" values are Fpm/CM objects used for "foreign" data. The content of a raw data structure is simply copied as-is by the garbage collector, without examining its contents – thus it may not contain any pointers to other Fpm/CM objects. Raw data structures are constructed like tuples, except that the negative of their length in words is stored.

4.5. Comparative Performance

The thesis behind Fpm2 is that by using an implementation model which is closely allied to that supported by the hardware, functional languages can be compiled more efficiently than if a special purpose functional language model is used which matches the language model but not the hardware one, as for example in the G-Machine. The thesis also claims that compiling functional languages in this way should produce implementations with comparable performance to those of imperative languages. To support our thesis we now present the results of some benchmark tests between our Hope+C/Fpm2 combination, two other major functional language implementations, and Pascal. The compiler versions used were; Hope+C version 3.2.2, LML version 0.95 [Augustsson 89], New Jersey SML version 0.39 [Appel], and standard Sun Pascal

---

† The compiler version numbers do not directly indicate age/maturity of the product, for example the first LML compiler was available before the Hope+C one...

as shipped with SunOS 3.2†. The benchmark programs have been selected to exercise all aspects of the implementations; data structuring, non-strictness, integer and real arithmetic and function call/return; and consist of some commonly used in the functional programming community and some from Gabriels Lisp benchmarking suite [Gabriel]. We make no detailed attempt to explain the individual results, which is always a difficult task when benchmarking, but rather intend the set as a whole to give an indication of the comparative performance of the four languages.

All the programs were run on a Sun 3/60 with 12Mb of memory. The execution time and memory usage of each program was measured using the standard C-shell `time` command, each program being run five times and the results averaged. In the tables below we give four statistics: *execution time*, which is measured in seconds; *average data size*, this is the average size of the programs data and stack segments during execution; *maximum real memory usage*, the programs were run on a virtual memory system, this figure is the maximum amount of real memory actually used; and *program size*, this is the size of the programs code segment during execution, rounded to the nearest 8Kb. Due to the relatively small size of these benchmarks, the program size statistic is more an indication of the size of the runtime support for each language, rather than a comparison of the code size produced by the compilers. Each table gives the results for the program written in Hope+C, LML, SML, Pascal, and Pascal compiler with the optimize flag on. The following paragraphs detail the individual programs, the source of which is included for reference in Appendix 4.

4.5.1. The Programs

Nfib

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	8.1	29.3	24.5	13.5	11.7
Av. Data Size (K)	64	240	448	16	24
Max. Real Mem (K)	8	32	58	8	16
Program Size (K)	32	24	8	3	3

Nfib is a the modified version of the fibonacci function which is a standard benchmark in the functional world. The difference between Nfib and the fibonacci function is that it calculates the number of function calls executed to produce the result rather than the fibonnaci number, thus if

the result is divided by the execution time an approximation to the number of function calls executed per second can be obtained. For the comparison we calculated *nfib(30)*, from the results the Nfib ratings for the four languages are; Hope+C 332K, LML 92K, SML 110K, and Pascal 199/230K. These figures suggest that for problems involving only simple arithmetic and function calls Hope+C is superior to either LML or SML; this is unsurprising due to the closeness of the Fpm2 model to the hardware model. The relative performance of Hope+C and Pascal is more surprising and is probably due to Fpm2 not needing to maintain a frame pointer register during and across function calls. Though we were unable to benchmark the IC Hope/Fpm implementation from which we developed the ideas underlying Fpm2, we do know that a port of the VAX FP/M onto Sun-3's had a Nfib rating of just over 100K. This gives an indication of the contribution made by Fpm2 compared to FP/M.

Queens

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	68.6	60	81.3	-	-
Av. Data Size (K)	576	640	1328	-	-
Max. Real Mem (K)	76	130	259	-	-
Program Size (K)	40	24	8	-	-

This is the problem of finding all the ways of placing *n* queens on an *n* square chess board such that no queen is threatened by another, and is often used as a benchmark. The problem involves little numerical computation but performs a large amount of list accessing. For the comparison tests we used a board size of 10. Two versions of this program were used; the first was based on a version shipped as an example program with the LML compiler. This program finds the solutions a column at a time and for each column generates a set of *n* possible solutions by placing a queen in each row; it then removes those which are invalid. The program also uses higher-order functions. The results show LML to be the fastest, by 14.3%, but Hope+C to have the better memory usage, by 10.3%/41.5% for virtual/real respectively. SML has the worst execution time and memory usage but the smallest code size!

Queens – Version 2

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	11.7	18.1	22.1	33.5	30
Av. Data Size (K)	456	248	456	600	600
Max. Real Mem (K)	70	32	59	146	146
Program Size (K)	40	24	8	8	8

This is our second version of Queens, the algorithm used produces a smaller set of possible solutions at each iteration, for a given column *m* it would only generate *(n - m + 1)* solutions as *(m - 1)* rows have already been occupied; this saves a large amount of computation, as shown by the results. The algorithm also does not use higher-order functions. For this version of Queens the performance of Hope+C and LML is reversed; Hope+C being faster, by 35.4%, and LML having better memory usage, by 45.6%/54.3% for virtual/real respectively. SML execution time is comparable to LMLs, memory usage to Hope+C. All three functional languages perform better than Pascal. (These two variations of Queens highlight the difficulty of effective comparisons; given the same problem different solutions give a different ranking of the languages.)

HigherOrder

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	15.6	18.1	8.3	-	-
Av. Data Size (K)	544	2536	1616	-	-
Max. Real Mem (K)	71	495	255	-	-
Program Size (K)	40	24	8	-	-

This program exercises higher-order function application. It was written in an attempt to determine if the relative performance between Hope+C and LML on the two versions of the Queens program was due to better support for higher-order functions by LML. For the tests we used an input value of 400 which causes 320800 *cons*es and higher-order applications to be performed. From the results SML was fastest, by 46.8%, while Hope+C had the better memory usage, by 66.3%/72.2% virtual/real respectively. The surprising performance of SML, compared to its ranking in all the other tests, is probably due to it being strict compared to the non-strict data construction of Hope+C/LML. The performance of LML compared to Hope+C, being 16% slower, does not however help to explain the difference between the two versions of the Queens program.

Tak

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	0.3	2.9	2.9	0.3	0.3
Av. Data Size (K)	56	232	408	16	16
Max. Real Mem (K)	9	32	55	3	3
Program Size (K)	32	24	8	8	8

This is the first of three benchmarks taken from the program suite used by [Gabriel] to evaluate various Lisp systems. Tak is a variant of the Takeuchi function and is dominated by function calls; 63609 recursive calls and comparisons, 47706 subtractions by 1. The results here are dramatic. Hope+C and Pascal are both 886% faster than LML and SML. Hope+C uses less memory than LML, by 75.9%/71.9% virtual/real respectively.

Takl

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	5.5	6.5	8.5	6.6	5.3
Av. Data Size (K)	64	240	448	24	24
Max. Real Mem (K)	9	33	59	3	3
Program Size (K)	32	24	8	8	8

Takl is a non-numeric version of the Tak program which uses lists of length *n* instead of the integer *n*. The program does little list construction; performing only 36 *cons* operations; but traverses the lists a great deal; performing 818900 tail selections, 425352 tests for empty lists, and 473059 calls to the list length comparison function *shorterp*. For this test SML was the slowest, contrary to expectations after its ranking in HigherOrder above. Hope+C was the fastest of the functional languages, by 18.2%, and used less memory, by 73.3%/72.7% virtual/real respectively. Hope+C had a comparable speed to Pascal but used 3 times as much memory, however the memory usage is small in both cases.

Triangle

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	1459.9	1327.9	1312	-	-
Av. Data Size (K)	560	256	864	-	-
Max. Real Mem (K)	70	33	114	-	-
Program Size (K)	40	24	8	-	-

This program finds all the solutions to a small puzzle; given a equilateral triangular board of 15 pegs, remove one of the centre pegs to start and then attempt to remove the remaining pegs using draught-style moves until only one is left. This program is a heavy user of list operations; the game board is represented by a 15 element list, 7820920 indexed selections to the list are made and it is "modified" 971616 times. To update a single element of a list of length *n* requires on average *n+2* list cell constructions, therefore in this program an estimate of the number of constructions used to modify the board is 7287120. For this program SML is the fastest; LML is 1.2% slower, Hope+C is 10.1% slower; however SML has the worst memory usage.

Triangle – Vector Version

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	302	-	-	233.6	189.2
Av. Data Size (K)	536	-	-	124	124
Max. Real Mem (K)	70	-	-	-	-
Program Size (K)	40	-	-	-	-

Hope+C supports functional vectors and in the original [Gabriel] Lisp program vectors were used. Therefore, as a comparison we re-coded the Hope+C version to use vectors and also added a Pascal version. As can be seen from the table the result for Hope+C is dramatic, the execution time dropping to 23% of the SML list-based version, 20.7% of the Hope+C list-based version. However, Pascal did better then Hope+C, executing 22.6%/37.4% faster and using 23.1% of the memory.

The difference between the list and vector-based versions clearly demonstrates that the choice of data structure is important; and adds weight to our claim that Hope+C is a *practical* language, as by providing a wider selection of data structures it enables more efficient programs.

FFT

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	2.0	-	3.4	1.6	1.6
Av. Data Size (K)	71	-	68	5	5
Max. Real Mem (K)	328	-	456	40	5
Program Size (K)	40	-	8	16	8

This program performs a 1024 point complex fast fourier transform. The Pascal version uses the standard iterative vector FFT algorithm and is based directly on the [Gabriel] Lisp version, the Hope+C and SML versions use a recursive list FFT algorithm. There is no LML version as LML does not support real numbers. Pascal is the clear winner here, running 20% faster than Hope+C and 53% faster than SML. The memory use of Pascal is also much smaller than the functional versions, though again SML uses the most. For further comments of FFT see [5.3.3] below.

Cat

	Hope+C	LML	SML	Pascal	Optimized Pascal
Execution Time (s)	8.7	53.1	26.7	33.8	33.7
Av. Data Size (K)	70	32	58	5	5
Max. Real Mem (K)	-	-	-	-	-
Program Size (K)	-	-	-	-	-

Cat is an emulation of the Unix™ "cat" command which concatenates a series of files, we chose this program to gauge the I/O throughput of the four languages. The I/O systems of our four languages are very different from each other; Hope+C uses result continuations [chapter 3], LML has simple character streams, SML has *side-effected* operations, and Pascal has inflexible procedural I/O. Due to these differences it is not possible to use a common algorithm, instead we have written the program in the most "natural" way for each language.

For the tests we concatenated three copies of the system dictionary ("/usr/lib/words", 201032 bytes) into the system bit-bucket ("/dev/null"). The results were very surprising with Hope+C executing fastest, being 206.9% faster than SML, but having the worst memory usage, 14 times that of Pascal.

4.5.2. Summary

These comparative tests show that Hope+C is significantly faster for integer numerical computation than either LML or SML; and comparable, or sometimes even a little faster, than Pascal. For problems involving large amounts of data structure processing it is comparable to LML and SML in execution time; however, Hope+C invariably has much better memory usage. For I/O, Hope+C would seem to have a much higher thoughput than the other languages.

4.6. Conclusions

The system described here has been implemented and forms part of the Hope+C compilation system. The compiler itself, being written in Hope+C, is compiled via Fpm2 and performs well. The comparative tests demonstrate that, for conventional sequential hardware, the Fpm2 model, though based on one developed for imperative languages, provides better performance and memory usage than other models developed specifically for functional languages.

The current code generator for Fpm/CM makes little attempt to produce high quality code and performs, for example, no register optimization and many redundant memory/register transfers. Also there is no work done to optimize the code which checks for suspensions and many redundant tests are currently executed. As part of ongoing research a non-strict version of Fpm2 is being constructed which is far more intelligent regarding suspensions and will also probably perform some lower-level code optimizations as well. We therefore expect future worthwhile performance improvements will be achieved.

5. Inter-language Working

In this chapter we discuss the provision of inter-language working at the function call level between functional and imperative languages. To provide inter-language working two areas have to be addressed: the syntax and semantics of the inter-language function call at the language level, and the combination of the runtime environments of the inter-working languages.

When providing inter-language calling between two imperative languages, the syntax and semantics of the inter-language function call can be the same as that of a normal function call, and this is the solution normally adopted. However, this solution is not possible between functional and imperative languages as it would destroy the referential transparency of the functional language. In this chapter we present a solution to this problem by building upon the result continuation I/O system of chapter 3.

The combination of the runtime environments of inter-working languages has no general solution as it depends on how the runtime environments of the languages being combined represent and manage their data values, stacks and heaps. As an example of how these problems can be solved for two languages which use very different runtime environments, we detail how the Fpm2 system of chapter 4 can be extended to support inter-language working with C.

In the remainder of this chapter we first discuss why inter-language working is desirable [5.1], and then elaborate on the areas which need to be addressed to provided inter-language working between functional and imperative languages [5.2]. In [5.3] we present an inter-language system and its realization in Hope*C.

5.1. Motivation

Inter-working between functional and imperative languages is desirable for two main reasons:

Code Reuse: Accessing Software Libraries

There are a large number of standard software libraries, for example the X Windows interface or the NAG numerical package, which have been written in standard imperative languages, for the above examples in C and Fortran respectively. The ability to use such libraries from functional languages greatly increases their flexibility, whilst also saving the resources which would otherwise have to be expended in re-implementing the libraries in the functional language.

Accessing facilities not provided by the language

Different languages are more suited to certain tasks; for example Snobol-4 for string processing, C for systems' programming, and Hope+C for list processing. Providing inter-language working facilities allows different parts of a problem to be solved in the most suitable language; and avoids having to design languages suitable for all tasks, which has proved both difficult to achieve and has produced unwieldy and unpopular languages, for example PL/1.

Furthermore, some optimization techniques which improve speed or space requirements, and which are not yet generally applicable to functional languages, can be manually applied to functional programs through the use of code written in an imperative language. For example, the in-place updating of data structures can often increase the performance of a program dramatically, however, techniques to detect when this is safe in a functional language are not yet mature enough for use in practical systems. However, a functional program could use code written in an imperative language to implement in-place updating and hence achieve performance improvements; we given an example of this technique in [5.3.1.2]. This use of inter-language working also introduces the possibility that future research developments will be able to use the facility to support the automatic transformation of inefficient functional programs into efficient mixed language programs.

5.2. Providing Inter-language Working at the Function Level

There are three main areas which much be addressed when providing inter-calling at the function level between functional and imperative languages: *referential transparency*, *value transfer* and *memory management*. The first is a requirement on the functional language level, the other two are requirements on the language implementation level.

5.2.1. Referential Transparency

Pure functional languages do not support side effects. Imperative languages, such as C and Pascal, not only allow side effects, but offer very little control over their use. This difference is one of the fundamental motivations behind the development of functional languages and any interface between them and imperative languages must protect this difference. The lack of inter-language calling in current functional languages can be attributed to the difficulty of achieving this requirement cleanly.

5.2.2. Value Transfer

Different language implementations represent their values (e.g. integers, reals, etc.) in different ways, and in particular functional and imperative language implementations often use totally different encodings. Any interface must provide a method of value translation between the two encodings used.

5.2.3. Memory Management

There are many different methods of memory management. Imperative languages are usually implemented using a non-relocatable heap system for dynamic memory allocation. Some imperative languages require programmer controlled allocation and de-allocation of dynamic memory, some are designed to provide automatic allocation and garbage collection of unused space, while some others are a hybrid of the two. In contrast all functional languages provide fully automatic allocation and garbage collection of their dynamic memory transparently as part of their normal operation. The issue of whether the heap is relocatable or not does not concern the programmer and an implementation could use either type. Combining the dynamic memory

strategies of any two inter-working languages is a basic design problem, regardless of whether they are functional or imperative.

In section 5.3 we present a general solution to the problem of maintaining referential transparency during inter-language calling between functional and imperative languages, based upon the result continuation mechanism of chapter 3. We then present an implementation of inter-language calling for Hope+C and give two examples of its use. In section 5.4 we address the implementation issues, and show that these do not have a general solution. We then present the particular solutions used by us for an interface between Hope+C and C, the design of which may be used as a guide to solutions for other language combinations. A complete description of all the programmer accessible functions provided to support this Hope+C/C interface is included in Appendix 3.

5.3. The Language Level – Keeping Referential Transparency

To maintain referential transparency we view imperative functions as external I/O "black box" devices which implement the required function; this approach is just semantic and no "real" I/O need take place, though the imperative function could be remote if desired. Under this view a function call becomes an output operation to pass the function argument values followed by an input operation to obtain the function result. To guarantee correct function call semantics the argument output and result input actions must be combined into a single I/O request. As this approach gives imperative function calls the semantics of I/O operations they can be handled using the result continuation I/O system of chapter 3 and referential transparency will be maintained.

To demonstrate the above approach to inter-language calling we show how it can be incorporated into the Hope+C language. To do this requires a method of declaring the type of the foreign language function, to satisfy Hope+C's strong typing, and a new basic continuation to support inter-calling.

The type of a foreign function is constructed using the special type constructor ':=>' defined as[†]:

```
infix :=> : 1;
data domain :=> range == internal_representation;
```

Using this constructor a foreign function can be declared using the standard `dec` statement. For example a function to perform the bit-wise boolean-and of two integers could be declared by:

```
dec bitand : (num # num) :=> num;
```

Functions declared with a type constructed using ':=>' may be defined by equations in the normal way, however if no equations are given then the compiler will take them to be references to foreign functions instead of reporting an undefined function error. For example, the following program fragment selects a foreign function based on a conditional test:

```
! Hope+C function !
dec Debugging : truval;

! External C functions for bit-wise boolean-and !
dec normal_and : (num # num) :=> num;
dec debug_and : (num # num) :=> num;

! Select appropriate C function !
dec bitand : (num # num) :=> num;
--- bitand <= if Debugging then debug_and else normal_and;
```

To call imperative functions the continuation `CallImp` is added to the `Result` type introduced in [3.6.1]:

```
data Result
  == existing constructors
  ++ CallImp((alpha :=> beta) # alpha # (beta -> Result));
```

The `CallImp` constructor has the *existentially* quantified type:

$$\exists\alpha, \beta.\ (\alpha :=> \beta) \times \alpha \times (\beta \rightarrow \text{Result}) \rightarrow \text{Result}$$

This type allows imperative functions of different types to be called by a Hope+C program. If existential quantification were not allowed then either `CallImp` must be declared with a constant type, allowing only imperative functions of that type to be called, or the `Result` type must be parameterised by the domain and range type of imperative functions, as in:

```
data Result(domain, range)
  == existing constructors
  ++ CallImp((domain :=> range) # domain # (range -> Result));
```

[†] Note that the constructor `internal_representation` in this definition is not exported from the standard environment module so it cannot be referenced by a user program.

A Hope+C program would now have to be declared with specific types for the domain and range, for example the `main` function [3.5] of a program which called the `bitand` function would have to be declared as:

```
dec main : Result(num # num, num);
```

This would again only allow imperative functions of a single type to be called from a single Hope+C program but separate programs could specify different types for the range and domain. Both of the above alternative options is overly restrictive, hence the advantage of existential quantification.

The `CallImp` continuation is used in the same manner as other result continuations; for example the following program fragment calls the imperative function `bitand`:

```
CallImp(bitand, (234, 123), program continuation function)
```

To demonstrate the use of `CallImp` we now give two example programs; the first uses C to perform bit-wise boolean-and, an operation not supported by Hope+C; the second is an example of how C can be used to incorporate in-place update into a functional program.

5.3.1. Example: Bit-wise Boolean-and

The bit-wise boolean-and function is not provided by the Hope+C standard libraries and is a typical, albeit trivial, example of a function for which C is well suited to writing. The following program shows how the function may be written in C and accessed from Hope+C using `CallImp`. First the Hope+C code, this declares the function `bitand` as imperative and prints the result of applying it to 23 and 43:

```
dec bitand : (num # num) :=> num;! external C function !

--- main                              ! call C external !
 <= CallImp(bitand, (23, 43), display);

--- display(x)                        ! display result !
 <= Printf("23 & 43 = %d\n", [Unum(x)], TheEnd]);
```

The C code for `bitand` is very simple, the type `Cell` and functions `MakeNum` and `IndexNum` are provided as part of the Hope+C/C library [Appendix 3]:

```
Cell bitand(args) Cell args;
{  int a, b;
   a = IndexNum(args, 1); /* get first arg */
   b = IndexNum(args, 2);
   return MakeNum(a & b); /* perform & and return */
}
```

This code extracts the arguments as C values from the Hope+C tuple using the library function `IndexNum`, performs the bit-wise boolean-and, and then constructs a Hope+C integer value using `MakeNum`.

Normally when an existing C function is to be called a wrapper function must be written in C which converts the Hope+C argument values, calls the target function, and then converts the result into a Hope+C value. No wrapper function is used here as the C function to be performed was the single expression '`a & b`'.

5.3.2. Example: FFT with in-place vector update

This example demonstrates how inter-language calling might be used to improve the efficiency of a functional program by using it to implement in-place update of structured values. Our example program is fast fourier transform (FFT) using the usual iterative vector algorithm, which is very inefficient if coded directly in a functional language. An efficient functional algorithm does exist for FFT, however this is not true in general for other problems, so this example gives an indication of what might be achieved in those cases.

In section 4.7. we used FFT as one of our benchmarks. The Pascal version was written using the usual iterative vector algorithm which updates the vectors in-place. The two functional versions, written in Hope+C and SML, use a recursive list-based algorithm to achieve a performance close to the Pascal version.

Here we first present a functional transliteration of the iterative vector FFT algorithm into Hope+C. This is inefficient due to the large number of vector updates, each of which requires the copying of the *complete* vector. We then convert this program to use inter-language working with C functions to achieve in-place vector updating and demonstrate that a significant performance improvement is achieved.

The full Hope+C code for the iterative vector algorithm is given in Appendix 4, where it can be compared to the Pascal version. For the purposes of this example we shall only concentrate on the code which performs the "butterflies" [Gabriel], in Pascal this is the following program fragment:

```
while i < n do
begin
   ip := i + le1;
   tr := ar[ip]*ur - ai[ip]*ui;
   ti := ar[ip]*ui + ai[ip]*ur;
   ar[ip] := ar[i] - tr;
   ai[ip] := ai[i] - ti;
   ar[i] := ar[i] + tr;
   ai[i] := ai[i] + ti;
   i := i + le
end;
```

This loop is imbedded within two others and is the heart of the FFT algorithm. On each iteration it performs four vector updates. When transliterated into Hope+C the `while` loop is replaced by tail recursion, the vector updates by calls to `perm_vector`, and all loop variables by function arguments; resulting in the following Hope+C code:

```
--- do1(ai, ar, le, le1, n, ui, ur, i)
 <= if i > n
    then (ai, ar)
    else
    (  let ip == i + le1 in
       let tr == (ar@ip * ur) - (ai@ip * ui) in
       let ti == (ar@ip * ui) + (ai@ip * ur) in
       let ar1 == perm_vector(ar, lambda (ix,v)
                                  => if ix = ip
                                     then (ix, ar@i - tr)
                                     else if ix = i
                                     then (ix, ar@i + tr)
                                     else (ix, v)
                                  end) in
       let ai1 == perm_vector(ai, lambda (ix,v)
                                  => if ix = ip
                                     then (ix, ai@i - ti)
                                     else if ix = i
                                     then (ix, ai@i + ti)
                                     else (ix, v)
                                  end) in
           do1(ai1, ar1, le, le1, n, ui, ur, i + le)
    );
```

Each call to `perm_vector` generates a *new* copy of the vector, which in our case has 1024 elements, which differs from the original by only two elements. This procedure is clearly inefficient compared to the four in-place updates performed by the Pascal version. Benchmarking the Hope+C program gives an execution time of 237.1s and memory usage of 600K, which does not compare favourably with the 1.6s/40K for the Pascal version.

To use in-place updating the program must be converted to continuation passing style to enable the `CallImp` continuation can be used. Then the two calls to `perm_vector` can be replaced by a call to a C function to perform the vector updates. The full source of the converted

program is included in Appendix 4, here again we only give the code to perform the "butterflies", which now consists of two functions, one written in Hope+C and the other in C. The Hope+C function is:

```
--- do1(ai, ar, le, le1, n, ui, ur, i, cf)
 <= if i > n
    then cf(ai, ar)
    else
    (  let ip == i + le1 in
       let tr == (ar@ip * ur) - (ai@ip * ui) in
       let ti == (ar@ip * ui) + (ai@ip * ur) in
          CallImp(butterfly, (ai, ar, i, ip, ti, tr),
                  do1(?, ?, le, le1, n, ui, ur, i + le, cf))
    );
```

In this version the tail-recursion of the first version has been replaced by passing a partially-applied function as the continuation to the `CallImp` result continuation constructor, and the two calls to `perm_vector` have been replaced by a single call to the C function `butterfly`. The code for the C function which simply in-place updates the vectors is:

```
Cell butterfly(args) Cell *args;
{  long i, ip; float tr, ti; Cell ai, ar;

   /* extract individual arguments from arg tuple */
   ai = Index(args, 1); ar = Index(args, 2);
   i  = IndexNum(args, 3); ip = IndexNum(args, 4);
   ti = IndexReal(args, 5); tr = IndexReal(args, 6);

   /* update in-place the two vectors */
   InjectReal(ar, ip, IndexReal(ar, i) - tr);
   InjectReal(ai, ip, IndexReal(ai, i) - ti);
   InjectReal(ar, i, IndexReal(ar, i) + tr);
   InjectReal(ai, i, IndexReal(ai, i) + ti);

   /* construct and return the result tuple */
   return Tuple(2, ai, ar);
}
```

(Note: in the Fpm2 [chapter 4] system Hope+C vectors are represented as tuples.) The functions `Index`, `IndexNum` and `IndexReal` are library functions to extract Hope+C, C integer and C real values respectively from Hope+C tuples. The function `InjectReal` updates a Hope+C tuple element with a C real value and the `Tuple` function constructs a Hope+C tuple. This combined Hope+C/C program performs the same four vector updates as the original Pascal version, improvement over the Hope+C version which generates a new copy of each vector. Benchmarking this version of the program gives an execution time of 3.0s and a memory usage of 248K. A summary of the benchmark results of all the versions is given in the following table:

	Hope+C	C/Hope+C	Pascal	Optimized Pascal
Execution Time (s)	237.1	3.0	1.6	1.6
Av. Data Size (K)	76	54	5	5
Max. Real Mem (K)	600	248	40	5
Program Size (K)	40	8	16	8

Vector-Based FFT Algorithms

The remaining difference in execution time between the Pascal and the Hope+C/C versions is due to the non-optimal nature of the code currently produced by the Fpm2 system for floating point operations.

5.4. The Implementation Level

There is no general solution to the problems of combining the runtime systems and structures of two languages so that they may coexist within the same memory space and pass values between themselves. Rather each pair of languages has to be considered separately, in our case Hope+C and C, and a specific solution produced. In this section we discuss the changes required to the Fpm2 system, introduced in chapter 4, to enable it to support the `CallImp` continuation described above.

5.4.1. Value Transfer

The manner in which values are stored in C and Fpm2 differ considerably. The C language, in common with most imperative languages, stores its data values using the various representations provided by the hardware, and these differ in both size and encoding, both of which are inferred/implied from context and are not themselves part of the encoding. Fpm2, to enable both polymorphism and garbage collection, stores all its values in only two types of objects:

- *fixed size objects* which reside on the stack or in heap objects, and store basic values (number, real etc.) and pointers to...

- *variable size objects* which are stored in the heap – these consist of a collection of fixed size objects.

Both types of objects contain a tag to indicate their contents, and in the case of heap objects also their size.

To pass values between Fpm2 and C they must be converted between the two formats. This conversion cannot be achieved in Hope+C due to its strong typing, however programming the conversion in C is trivial as it is weakly typed and provides bit-level operations. In the Fpm2/C interface the mechanics of this conversion are completely encapsulated in the C library [Appendix 3] provided and the Hope+C/C programmer need be unaware of the internal formats used by Fpm2.

5.4.2. Memory Management

C and Fpm2 provide completely different styles of dynamic memory allocation. C provides a non-relocatable, non-garbage collected heap. It is the programmers responsibility to allocate and free storage using a set of standard library functions and little error checking is performed by these functions. The library functions themselves assume they are in sole control of dynamic memory allocation and interact directly with the operating system services on this basis. Fpm2 has a relocatable, garbage collected heap. The allocation and reclamation of heap memory is automatic and does not involve any explicit programmer action. The Fpm2 system also interacts with the operating system services on the assumption that it is in sole control.

It is not possible to combine these two regimes into a scheme which uses a single pool of dynamic memory. Our solution to supporting both is to have the Fpm2 system first set aside an area of memory to be used by C as its heap. To force C to use this memory the Fpm2 system intercepts all calls from C code to the operating system for memory related services and then performs the operations itself using this area of memory. This arrangement provides a "virtual" interface to the memory services which the C code cannot distinguish from that of the operating system. The precise mechanics of how the Fpm2 system intercepts the C code operating system calls are not covered here, being very closely tied to the low-level operation of our particular host operating system. Another, more portable, method of combining the two heap schemes would be for the Fpm2 system to use the C library memory functions, this method was not adopted by us as it requires more complex (and probably slower) code in the Fpm2 system.

5.5. Conclusion

By classing inter-language calling as an I/O operation we have shown that a referentially transparent interface between a functional language and and imperative one can be defined.

The Hope+C/C interface described in this chapter has been integrated into the Fpm2 system and has been used to construct an interface between Hope+C and the C X-11 windowing library, as well as a number of smaller projects. The X-11 interface has been used to implement a powerful graphical interface to, among other things, the Flagship Program Transformation Environment [Sephton].

6. Towards a Functional Operating System

As an example of the power of the result continuation method of providing I/O in functional languages we have sketched the design of an operating system. This chapter describes that design and then makes some comparisons with other proposals for functional operating systems.

6.1. Background

The idea of writing operating systems using functional languages is not new; [Henderson], [Karlsson], [Stoye] and more recently [Jones] have all proposed systems, the work of Jones in particular being very comprehensive. All these designs have been based on some variation of streams, to route information between separate "processes", together with a non-deterministic merge operation, which is *non-functional*, to combine these streams fairly. All the designs have very similar system structures, for example an examination of the diagrams given by Karlsson and Stoye for their respective designs will show that they are topologically equivalent. However these existing designs, being based on streams, all inherit the fundamental problems of stream I/O systems [see chapter 3]. They also include a non-deterministic merge in their implementation languages, thus making them non-referentially transparent.

In this chapter we sketch an operating system design based around result continuations, thus avoiding the problems of stream I/O. To handle non-determinism a small pre-emptive process scheduler is defined which is written in assembler and *not* a functional language, thus avoiding introducing a non-referentially transparent construct into the language. This two tiered approach is not unusual for an operating system written in a high-level language, c.f. Unix™.

6.2. System Structure

The system we propose, see figure 6.1, is similar to a conventional message passing operating system and consists of four parts:

Firmware (+ Hardware)

The firmware provides three concepts:

- **Processes.** The firmware is able to create, run, suspend and delete processes. Processes are created, suspended and deleted under the control of the operating system process via a continuation-based interface. Processes may also be suspended pending the occurrence of some event, at which time the firmware will re-schedule them. There are (at least) three process priority levels, which are used to help determine which process to run. They are used, highest priority first, for; device processes, the operating system process, and user processes.
- **Events.** An event may be generated by either hardware, a process, or a message, and are addressed to a particular process. A hardware event is asynchronous and invariably causes a process switch.
- **Messages.** All processes communicate with each other using messages, these messages are sent over channels. The firmware provides for the opening, naming, communicating over, and closing of channels.

Device Processes

The device processes handle the low-level interface to the hardware devices. They handle all events from the hardware and communicate with the operating system process over a bi-directional message channel.

Operating System Process

This handles all requests from user processes and depending upon them may issue messages to the device processes or interact with the firmware using its continuation-based interface. The operating system also receives event notification from the firmware and device processes and determines whether to pass them on to the user processes.

User Processes

These correspond to the user programs.

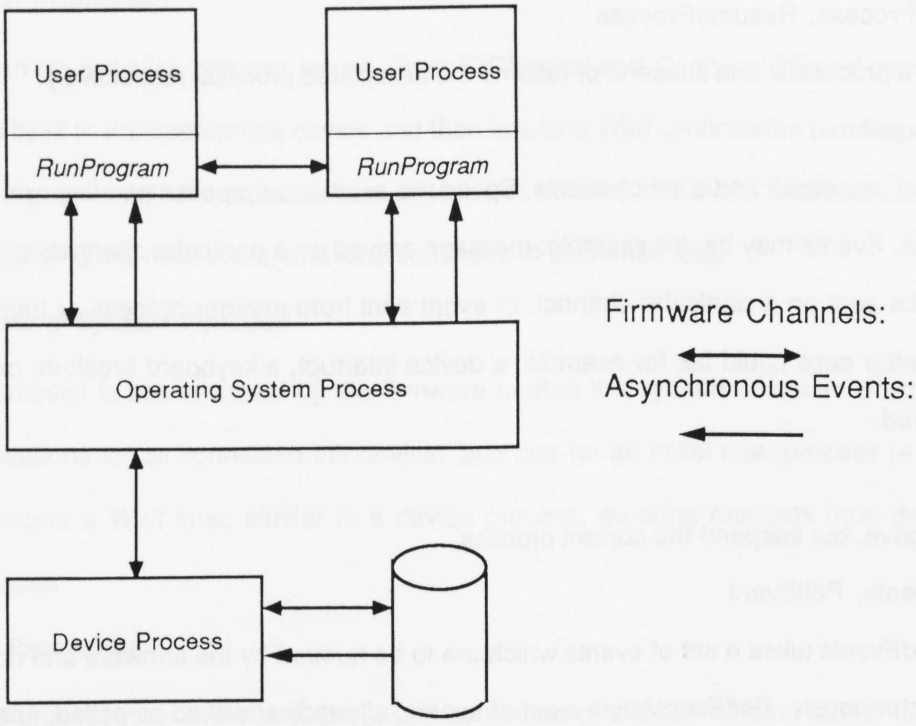


Figure 6.1

6.3. Realization

We look at implementing the above structure by considering how it might be programmed in Hope+C, and sketch what new continuations would need to be provided.

Firmware

The above specification for the firmware can be implemented on current "conventional" hardware, and is neither large nor difficult to write. All non-determinism, i.e. the arrival of events and dispatching of processes, is encapsulated in the firmware and is hidden behind its continuation-based interface. A basic set of continuations to interface to the firmware would be:

StartNewProcess

Take; a function, one or two channels, and process type (e.g. device or user); create a new process and return its *processID*. Connect the given channel(s) as the standard input and output of the new process.

TerminateProcess

Take a processID and terminate the associated process.

SuspendProcess, ResumeProcess

Take a processID and suspend or resume the associated process, respectively.

WaitProcess

Take a processID and a set of events. Spend the associated process pending one of the events. Events may be, for example: message arrived on a particular channel; message may be sent on a particular channel; or event sent from another process or hardware. The latter case could be, for example; a device interrupt, a keyboard break-in, or timer elapsed.

Wait

As above, but suspend the current process.

PolledEvents, PollEvent

PolledEvents takes a set of events which are to be queued by the firmware and not sent asynchronously. *PollEvent* takes a set of events, all ready specified as polled, and tests to see if there are any pending.

SignalProcess

Sends an event to a specified process.

NewChannel, DiscardChannel

Open a new channel and return its *channelID*, and close a channel respectively.

NameChannel, ConnectChannel

Given a channelID associate a global name with it, and given a name return its associated channelID respectively.

SendMessage, ReceiveMessage

Send and receive messages on given channels respectively.

OpenIOChannel, ConnectIOEvent

Used by device processes to respectively; open a channel connected to a given device, and request hardware events from a particular device to be routed to this process (either asynchronously or polled).

Most of the above continuations are only available to the device or operating system processes, user processes obtaining the services via requests to the operating system process. However, once a channel has been established user processes are allowed directly send and receive messages over it, to avoid the overhead of involving the operating system process.

Device Processes

On start-up a device process issues *OpenIOChannel* and *ConnectIOEvent* continuations to attach itself to the appropriate device and then issues a *Wait* continuation pending an event. On receiving an event, the process is re-scheduled, handles the event and then issues another *Wait* pending the next event, and thus continues in an infinite loop.

Operating System Process

This process is bootstrapped by the firmware to start the system. It issues *StartNewProcess* continuations for all connected I/O devices and one for an initial user process (e.g. "login"). It then enters a *Wait* loop, similar to a device process, awaiting requests from device or user processes.

User Processes

These processes execute user programs by means of a small runtime kernel function which evaluates the user function to obtain a continuation and then sends a message to the operating system representing that continuation. It then applies the continuation function to the operating systems response. A skeleton of this would be:

```
--- RunProgram(userfunction, lastanswer)
 <= let (op, cf) == userfunction(lastanswer)
    in SendMessage(op, RunProgram(cf, ?))
```

Overview

In the above, each process in the system is written in a pure functional manner using result continuations, and no side-effects or non-determinism is used. We have removed completely the non-deterministic merge used by previous proposals, not even the firmware needs such an operation.

The only non-functional part of the system is in the firmware. This responds to asynchronous hardware events and updates its process table and current process accordingly. This part of the system is small and easily written. Previous proposals, using non-deterministic merge, have not required such firmware process support, we shall justify our choice in the next section. Though the firmware would be written in assembler, this is not unusual for an operating system written in a high-level language, c.f Unix™.

6.4. Explicit Versus Implicit Multiple Processes

In the Nebula operating system design by [Karlsson] there is really only one process; user processes, device processes and the operating system are all just functions within one large program. This approach provides a clean functional solution, especially as a multiprocessor machine can evaluate different parts of the program in parallel due to the semantics of functional languages, but it does leave the problem of how to handle device interrupts.

In a conventional system, the occurrence of a device interrupt causes a context switch to be made, but under Nebula this is not possible. Karlsson does not detail how his device handlers actually communicate with the physical devices, just describing them as a kind of *"mini operating system"*, but the result stream from a device handler is periodically tested using an *"evaluated-test instruction"*. This is more primitive than, but exists for the same reason as, the non-deterministic merge function of other proposals. But what happens if one of the user functions is involved in a large computation? When will the device list be tested? The Nebula proposal does not detail how to solve this problem to provide realistic response times to device interrupts – which is vital in an operating system.

In [Stoye] the author first describes a system using a single Miranda program, but later gives details of the timeslicer implemented on the SKIM machine. SKIM does not have the same problems with device interrupts as Nebula as such tasks are off-loaded onto a *"host microprocessor"*. Stoye's operating system, being fundamentally similar to Nebula, would suffer from the same response time problems if expanded to handle real devices.

In this proposal we have taken the (conventional) approach of having many explicit separate processes, in particular device handlers are separate processes from the operating system, together with a firmware context switch. This approach provides both modularity and efficiency.

The multiple process approach also has another major advantage: using result continuations we can poll for events or have them delivered asynchronously; or alternatively, suspend awaiting an event. If a single process were used, each method would cause problems; if polling was used we could not guarantee an acceptable response time; if events are delivered asynchronously, to guarantee referential transparency, we must discard the current computation, losing work, and

start another; finally, the system could not suspend awaiting an event, as no computation would ever be performed.

This lead us to the design based on explicit multiple processes and context switching on an event, which provides the following two important properties: a context switch preserves the state of the current process, so no work is lost; and device processes can suspend awaiting an event, thus they don't waste processor time (i.e. we have *non-busy* wait).

6.5. Conclusion

As stated at the beginning, this chapter only presents a *sketch* of an operating system, though the above is sufficient to demonstrate that a real system could be constructed using a functional language with result continuations, before one could be constructed further work would have to be done. For example, we have not discussed what the type of a message passed over a (firmware) channel is. Clearly character only communication as is currently provided by Hope+C is inadequate. There are two possible solutions; define a message type into which everything must be converted, or provide a means to open a typed channel. A decision on this and other topics would require further work. In his recent thesis [Cupitt] describes the design and implementation of a small operating system written in Miranda. Cupitt's design is based upon the use of continuations, which he implements over Miranda streams thus making them closer to Nebula [Karlsson] than result continuations, and refers to our earlier work [Perry 88e]. Cupitt's work is very comprehensive and clearly demonstrates the case for using continuation based I/O for functional operating systems.

Our current research effort is towards the design of a concurrent functional programming language, with separate processes communicating over typed channels. This work is not targeted at writing an operating system, but to providing a concurrent functional process/object-oriented programming environment, but will be applicable to this task. However, we present no results of this research here, as it is currently in progress and is outside the timescale of this thesis.

7. An Extended Type System

In this chapter we introduce an extended type system which combines the features of the Hindley/Milner system [Milner 78] with user defined overloading and existential quantification. User defined overloading is a convenience feature, that is it does not add any expressive power to a type system but rather makes using it easier. Overloading has been incorporated into many monomorphic programming languages, for example Algol68 [van Wijngaarden]. However, the goal of combining it with a polymorphic type inference system such as Milners has proved elusive and only systems which are either inefficient for sequential implementations or provide a restricted form of overloading currently have been produced [Leviant, Kaes, Hudak]. Existential quantification is supported by a number of existing type systems [Cardelli, Mitchell]. It provides a powerful *abstract type* facility [Mitchell], and in a related manner also enables the inter-language working mechanism of chapter 5 to be typed. However, unrestricted existential quantification is difficult to type-check mechanically and thus has not been incorporated into many language implementations.

The contribution of our type system is twofold; firstly it supports both polymorphism and overloading and, as described in chapter 8, has a type-checking algorithm which supports type inference, is implementable in a pure functional language, and has a good time/complexity performance for overloading resolution. Secondly, we limit the scope of existential quantification so as to produce a parallel between the construction/destruction of existentially quantified data values and the application/definition of universally quantified functions. This enables type-checkers to be easily constructed which use the same basic method to handle both forms of quantification.

In this chapter we first specify informally the two extensions in sections 7.1 and 7.2, and then give the formal inference rules for the system in 7.3. We conclude, in 7.4, with a description of the use of the type system in Hope+C and detail some restrictions placed on its generality by the Hope+ definition [Perry 88a].

Syntax

The following syntax is used for types, expressions and function definitions in subsections 7.1 and 7.2 below.

```
<type> ::=  <primitive type>     e.g. num, real etc.
          | <type variable>      e.g. α, β etc.
          | <type> × <type>
          | <type> → <type>
          | ∀<type variable>.<type>
          | ∃<type variable>.<type>

<sum type definition> ::= <identifier> = <tag> <type> {<orbar> <tag> <type>}*

<expression> ::=
          <identifier>
          | (<expression> {, <expression>}* )     tuples
          | <expression> <expression>             application
          | <expression> <op> <expression>        infix application
          | if <expression> then <expression> else <expression>
          | let <expression> == <expression> in <expression>
          | λ <expression> = <expression>

<function definition> ::= <identifier> {: <type>} <expression> = <expression>
```

To simplify the discussion the above syntax differs from that of Hope+C. Our syntax for types explicitly indicates universal and existential quantifiers, whereas in Hope+C quantification is implicit. Our syntax for expressions is a subset of Hope+C, it is a trivial task to extend it to the full Hope+C syntax. Both syntaxes are also simplified by the use of typographical devices.

7.1 User-defined Overloading

Overloading does not add "power" to a type system but does make it more convenient to use. For example, in a language supporting both numbers and sets it is often convenient to use the same identifier to denote both numeric addition and set union, as in:

```
+ : number × number → number
+ : set α × set α → set α
```

In our system the same identifier may be used to denote any number of different functions, constructors or local values with the restriction that within a single module no identifier may be used to denote more than one global function or constructor of a given type, as this would introduce unresolvable ambiguities. That is, the following is illegal:

```
Union = aNum number | aChar character
aNum : number → Union
```

as the constructor `aNum` and the function `aNum` have identical types and if `aNum` were to be used in an expression, the problem of determining the instance intended is undecidable. However the following *is* legal:

```
Union = aNum number | aChar character
aNum : Union → boolean
```

as here the two `aNum`s have distinct types.

7.1.1. The Interaction of Overloading and Type Inference

Our type system supports both overloading and type inference, however this combination can give rise to ambiguities due to the lack of type declarations, in addition to those introduced by overloading itself. For example, consider the function:

```
min x y = if x < y then x else y
```

defined in a context where the identifier '<' is overloaded to represent both numeric less-than and set inclusion, i.e.:

```
< : number × number → boolean
< : set α × set α → boolean
```

Inferring the type of `min` now results in an ambiguity as two possible types can be attributed to the function:

```
min : number × number → number
```

and:

```
min : set α × set α → set α
```

We resolve such ambiguities by the following rule: If a function or local value is determined by type inference to have more than one possible type, then a separate function or local value is implicitly defined for each possible type. For example, the above definition of `min` is treated as though it were the two definitions:

```
min : number × number → number = if x < y then x else y
min : set α × set α → set α = if x < y then x else y
```

Other resolution rules could be chosen, the simplest being to class a definition such as `min` to be not well-typed. Another is to first transform such definitions automatically into equivalent ones which are parameterised by the identifiers causing the ambiguity, as in:

```
min : α × α × (α × α → boolean) → α
min x y < = if x < y then x else y
```

which gives `min` the effective type: $\alpha \times \alpha \times (\alpha \times \alpha \rightarrow \text{boolean}) \rightarrow \alpha$. This requires us to alter all calls to the transformed functions, automatically inserting the extra parameters, for example:

```
min(1, 2) ⇒ min(1, 2, <number × number → boolean)
```

7.1.2. "Best Fit" Overloading Resolution

For an expression to be well-typed the function, constructor or local value denoted by an identifier must be resolvable statically. As an option in our type system, we allow overloading resolution to be performed on a *best fit* basis if resolving on an exact match basis results in more than one possible choice for the value denoted by a particular identifier. An instance of an identifier is defined to be a better fit in a given context if using it instantiates fewer type variables than using any other possible instance. For example, given the two definitions:

```
= : α × α → boolean          (a)
= : Union × Union → boolean  (b)
```

then in the expression:

```
3 = 4
```

the identifier '=' can trivially be determined to be an occurrence of definition (a), however in the expression:

```
Anum 3 = Anum 4
```

the identifier '=' could represent either definition (a) by instantiating α to `Union`, or (b) by exact type match. In this case best fit resolution allows definition (b) to be chosen as it instantiates fewer type variables, zero compared to two, than using (a).

7.2. Existential Quantification

In this section we shall first give an overview of universal quantification as provided by Hindley/Milner [Milner 78], and then introduce existential quantification as supported by our system. As in Hindley/Milner the quantification of types is limited as the scope of the quantification is always implicit, in the last part of this section we show that this limited scope results in a type system which can be type-checked using fundamentally the same mechanism as that required for straight Hindley/Milner. For a more detailed discussion of universal and existential quantification the reader is referred to [Cardelli], on whom we base our notation.

7.2.1. Overview of Universal Quantification for Function Types

The Hindley/Milner type system provides *limited universal* quantification allowing *polymorphic* functions to be defined. For example, the list mapping function:

```
map : ∀α, β. (α → β) × list α → list β
```

The quantification is limited as the scope of the quantifier is implicitly global to the entire type expression and cannot be changed. This implies that lambda-bound variables are monomorphic which disallows polymorphic function arguments, for example, the following function cannot be defined in Hindley/Milner:

```
f : ∀β, γ. (∀α. α → list α) × β × γ → list β × list γ
f(g, x, y) = (g x, g y)
```

However, as with globally defined functions, let-bound variables may be polymorphic. For example, a function similar to the above could be defined:

```
f : ∀β, γ. β × γ → list β × list γ
f(x, y) = let listify == λ x = [x] in
                (listify x, listify y)
```

where `listify` is assigned the type:

```
listify : ∀α. α → list α
```

7.2.2. Existential Quantification for Data Types

In a parametric tagged sum type the constructors of the type are given universally quantified types, though the type itself is not quantified in any way. For example, for the parametric list type:

```
list val = nil | cons(val × list val)
```

the constructor `cons` is given the type:

```
cons : ∀α. α × list α → list α
```

Using parametric types homogeneous data structures can be constructed. To construct heterogeneous types requires that the type itself is quantified; this can be done using *existential* quantification. In our type system we provide limited existential quantification for types, limited in the same way as for universal quantification in that the scope of the quantification is implicit, which in this case is either the whole type expression, or in the case of sum types is a separate scope for each individual constructor. For example, a heterogeneous list type can be defined by:

```
Plist = Pnil | ∃α. Pcons α × Plist
```

Using such a type, heterogeneous list can be constructed, as in:

```
Pcons(3, Pcons("asda", Pnil)) : Plist
```

Not all existentially quantified types are useful, as is shown by the above example: though an object of type `Plist` can be constructed, once the first argument to `Pcons` is *packaged* inside a `Plist` nothing further can be done with it as its type is hidden. For example, in the function:

```
f : Plist → some type
f Pcons(x, y) = some expression
```

the types of `x` and `y` are:

```
x : ∃α. α
y : Plist
```

The type ∃α. α provides no usable information about `x`, simply stating that *"there exists a type t such that* `x` *is of type t"*. However, existentially quantified types provide a very useful facility, that of *abstract* types [Mitchell], where the implementation type is deliberately hidden. For example:

```
stack == ∃α. α × (int × α → α)×(α → int × α)
```

is the type of an abstract stack with elements of type int. One value of this type is:

```
(nil, cons, headtail) : stack
```

where `nil`, `cons` and `headtail` are; the empty list, the list constructor and a list destructor function, respectively. If a value of type `stack` were decomposed by pattern matching, as in:

```
f : stack → some type
f(val, push, pop) = some expression
```

the types of the bound variables would be:

```
∃α.(val : α, push : int × α → α, pop : α → int × α)
```

These types restrict the functions which can be applied to `val` to be only `push` and `pop`, and the implementation of type α is kept hidden.

In the context of Hope+C, the existential typing mechanism is used to support inter-language working. Recall, from chapter 3, that the constructor `CallImp` is used to call C functions from Hope+C and that it is defined as:

```
data Result ==
    CallImp(alpha # (alpha :=> beta) # (beta -> Result))
 ++ other constructors;
```

Existential quantification allows the type variables `alpha` and `beta` to appear on the right hand side of the definition of type `Result`, which in turn allows C functions of *different* types to be called using this interface. If existential quantification were not available, then the definition of `Result` would have to be something like:

```
data Result(Cdom, Crng) ==
    CallImp(Cdom # (Cdom :=> Crng) # (Crng -> Result))
 ++ other constructors;
```

and a Hope+C program would then be of type:

```
main : Result(t1, t2)
```

where `t1` and `t2` are some actual types. This would only allow a given Hope+C program to call C functions of *one* type, namely `t1` → `t2`, which would severly limit its flexibility.

7.2.3. Type-checking Existential Quantification

Existing type systems which use existential quantification, such as [Cardelli], do not limit the scope of quantification as we do and, as a consequence, require new type-checking mechanisms. Our type system has been specified so that typing existential quantification need require no new mechanism in the type-checker, but rather is possible using a variation of that used to type universal quantification. The limiting of the scope of universal quantification in the Hindley/Milner system has not proved a severe restriction in practice. We argue that our limiting the scope of existential quantification is also not a severe practical restriction, but rather its inclusion is a powerful extension realised at minimal cost.

The reason for not requiring any new mechanism to type-check our limited existential quantification stems from the similarity of polymorphic functions and existentially quantified constructors, as explained below. The ability to type-check both Hindley/Milner universal quantification and our existential quantification using the same mechanism *does not* imply that an arbitrary type-checker which implements Hindley/Milner can be modified to support existential quantification, but rather that a type-checker can be written which uses the same mechanism for both forms of quantification, as is shown later in 7.2.

Typing Function Application and Definition

When *applying* a polymorphic function the type variables in its type may be instantiated to any type. For example, consider the polymorphic list mapping function, defined above, in the following two expressions and the associated typing and instantiations (reading "⇐" as "instantiated to"):

```
map(float, [1, 2, 3]) : list int
     with α ⇐ int, β ⇐ real
map(real_to_string, [2.6, 4.7]) : list list character
     with α ⇐ real, β ⇐ list character
```

However, when *defining* a polymorphic function, due to the restriction imposed on the scope of universal quantification, the variables bound to the universally quantified arguments have *some* monomorphic type, i.e. they have an existentially quantified type. For example, consider the definition of the map function:

```
map f (cons h t) == cons (f h) (map f t)
```

here the types of the variables are given by:

```
∃α, β.(h : α, t : list α, f : α → β)
```

Thus to type polymorphic functions a type-checker is effectively supporting both universal and existential quantification.

Typing Data Construction and Extraction

When a value of an existentially quantified type is *constructed* the variables in the type may be instantiated to any type; this is analogous to the case of universally quantified function application. For example given the type:

```
pack == ∃α. α × (α → int)
```

then in typing the following expressions, in a context requiring a value of type `pack`, the variable α would be instantiated as follows:

```
("asda", length) : pack, with α ⇐ list character
(2, square) : pack, with α ⇐ int
```

The mechanism required by the type-checker to generate these typings is intrinsically the same as that for typing the arguments to a polymorphic function. When a value of existential type is *decomposed* by pattern matching, the types of the bound variables are *some* monomorphic type, this is analogous to the case of the definition of a universally quantified function. For example, in the expression:

```
let (value, fun) == x in fun(value)
```

where x is of type `pack`, the types of the bound variables are:

```
∃α.(value : α, fun : α → int)
```

Therefore, the same mechanism used by the type-checker for typing a function definition can be used when typing the decomposition of an existentially quantified type.

7.3. Type Inference Rules

Expressions

As the basis for our type inferences rules we shall use a simple language with the following syntax for expressions:

$$e = x \mid \lambda x.e \mid e_f\, e_a \mid \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$$

where: $x \in identifiers$

Type Expressions

Our syntax for types and the operations over them are conventional with the addition of sums. Types consist of type variables (α), products ($\times$), functions ($\rightarrow$) and tagged sum types ($\cup$). Type schemes are extended type expressions and include universal quantification ($\forall$). *Instantiation* converts a type scheme into a type by replacing all the universally quantified type variables with actual types, *generalisation* produces a type scheme from a type by universally quantifying the free type variables. In the following "$\equiv$" is read as "is defined as":

Types: $\tau ::= \alpha \mid \rho \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \bigcup T_i \tau_i$

where: $\alpha \in$ *type variables*, $\rho \in$ *primitive types*, $T \in$ *tags*

Type schemes: $\sigma = \tau \mid \forall \alpha.\sigma$

Instantiation:

Given $\sigma = \forall \alpha_1, \ldots, \alpha_n.\tau'$ then $\tau < \sigma \Leftarrow \tau = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau'$

Generalisation:

$\overline{A}\tau \equiv \forall \alpha_1, \ldots, \alpha_n.\tau$ where α_i is free in τ, but not in A

Type Inference Rules for Hindley/Milner

These follow [Clément] and differ slightly from those of [Damas], but were proved equivalent by Clément. We shall use these later as the basis for the inference rules for our system.

[VAR] $$A.\ \alpha{:}\ \sigma \vdash x : \tau < \sigma$$

[ABS] $$\frac{A.\ \alpha{:}\ \tau_d \vdash e : \tau_r}{A \vdash \lambda x.e : \tau_d \to \tau_r}$$

[APP] $$\frac{A \vdash e : \tau_d \to \tau_r,\ A \vdash e' : \tau_d}{A \vdash (e\ e') : \tau_r}$$

[LET] $$\frac{A \vdash e : \tau_1,\ A.\ \alpha{:}\ \overline{A\tau_1} \vdash e' : \tau_2}{A \vdash (\textbf{let}\ x = e\ \textbf{in}\ e') : \tau_2}$$

7.3.1. Set-Based Inference Rules for Overloading

To support overloading we can extended the [Clément] rules to work over *sets* of types, this is relatively straightforward. In these inference rules T and Σ represent sets of types and type schemes respectively:

[VAR] $$A.\alpha{:}\ \Sigma \vdash x : \{\tau \mid \sigma \in \Sigma, \tau < \sigma\}$$

[ABS] $$\frac{A.\alpha{:}\ \mathrm{T}_d \vdash e : \mathrm{T}_r}{A \vdash \lambda x.e : \{\tau_d \to \tau_r \mid \tau_d \in \mathrm{T}_d, \tau_r \in \mathrm{T}_r, A.\ \alpha{:}\ \tau_d \vdash e : \tau_r\}}$$

[APP] $$\frac{A \vdash e : \mathrm{T}_f,\ A \vdash e' : \mathrm{T}_d}{A \vdash (e\,e') : \{\tau_r \mid \tau_d \to \tau_r \in \mathrm{T}_f, \tau_d \in \mathrm{T}_d\}}$$

[LET] $$\frac{A \vdash e : \mathrm{T}_1,\ A.\alpha{:}\ \overline{A\mathrm{T}_1} \vdash e' : \mathrm{T}_2}{A \vdash (\textbf{let}\ x = e\ \textbf{in}\ e') : \mathrm{T}_2}$$

Definition: Well-Typed

A *program* is well-typed under these inference rules iff:

i) the type inferred for the top-level expression is a singleton-type, and

ii) within any singleton-typed expression there must be a unique assignment of a singleton-type to every sub-expression which without the surrounding context of the expression would be multi-typed.

Rule (i) simply requires that the top-level expression of a program be a single expression and not a set of overloaded expressions. Rule (ii) forbids programs where the overall type is deducible but where there is some unresolvable local ambiguity. For example, the following would not be well-typed:

```
f = 3 : number
f = "asda" : list character
g = 4 : number
g = "tesco" : list character
if f = g then "equal" else "not equal" : list character
```

Here each definition has a singleton-type, the identifiers `f` and `g` are overloaded, and the top-level expression is singleton-typed. However, the sub-expression `f = g` is multi-typed and there is no unique assignment of singleton-types decidable from the enclosing expression.

7.3.2. Non Set-Based Rules for Overloading

The advantage of the above inference rules for overloading is that the sets produced during the type-checking process are explicit and thereby provide a hint at how a type-checking algorithm may be constructed. However using the inference rules alone a type can be inferred for some ill-typed expressions and therefore a rather involved definition of well-typed is required in conjunction with them, though this definition does also hint at a suitable implementation algorithm. Our algorithm described later [Chapter 8] is based on the above rules and well-typing definition.

After discussion with [Patterson] we defined an alternative set of inferences rules, given below, which contain only one set and require a less complicated well-typing definition. The rules themselves are simpler, but they do not give any indication as to how they may be efficiently implemented. In these rules Σ is used to represent a set of type schemes:

[VAR] $$A.\ x\colon \Sigma \vdash x : \tau < \sigma \text{ where } \sigma \in \Sigma$$

[ABS] $$\frac{A.\ x\colon \tau_d \vdash e : \tau_r}{A \vdash \lambda x.e : \tau_d \to \tau_r}$$

[APP] $$\frac{A \vdash e : \tau_d \to \tau_r,\ A \vdash e' : \tau_d}{A \vdash (e\ e') : \tau_r}$$

[LET] $$\frac{A \vdash e : \tau_1,\ A.\ x\colon \overline{A\tau_1} \vdash e' : \tau_2}{A \vdash (\textbf{let } x = e \textbf{ in } e') : \tau_2}$$

These rules differ from those given above for Hindley/Milner only in that the [VAR] rule is changed to select a single type scheme (σ) from a set of possible type schemes (Σ).

Definition: Well-Typing

Under these inference rules an expression is well-typed iff there exists a *unique* choice of a type scheme σ for every variable in the expression, selected from Σ, which produces a successful type inference for the expression.

It is relatively straightforward to develop an implementation of these rules, albeit rather inefficient. Given an expression containing overloaded variables we first produce *n* new expressions which contain no overloading, one for each possible permutation of the overloaded variables, and then perform *n* simple type inferences. The expression is well-typed iff exactly *one* of these simple inferences succeeds.

7.3.3. Inference Rules for Existential Quantification

To support existential quantification we must first extend our definitions for types, type schemes, instantiation and generalisation to include existential quantification:

Types: $\tau = \alpha \mid \rho \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \bigcup T_i \varepsilon_i$

where: $\alpha \in$ *type variables*, $\rho \in$ *primitive types*, $T \in$ *tags*

Type schemes: $\sigma = \tau \mid \forall\alpha.\sigma$

$\varepsilon = \tau \mid \exists\alpha.\varepsilon$

Instantiation:

$\tau < \sigma \equiv \tau = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau'$ where $\sigma = \forall\alpha_1, \ldots, \alpha_n.\tau'$

$\tau < \varepsilon \equiv \tau = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau'$ where $\varepsilon = \exists\alpha_1, \ldots, \alpha_n.\tau'$

Generalisation:

$\overline{A}\tau \equiv \forall\alpha_1, \ldots, \alpha_n.\tau$ where α_i is free in τ, but not in A

$\bar{\varepsilon} \equiv [\rho_1/\alpha_1, \ldots, \rho_n/\alpha_n]\tau'$ where $\varepsilon = \exists\alpha_1, \ldots, \alpha_n.\tau'$, ρ_i new

With these definitions the inference rules given above for a system with overloading [7.3.2] are extended to include existential quantification by adding two rules [CON] and [DES] for tagged sum type construction and destruction respectively:

[VAR] $$A.\ \alpha\colon \Sigma \vdash x : \tau < \sigma \text{ where } \sigma \in \Sigma$$

[ABS] $$\frac{A.\ \alpha\colon \tau_d \vdash e : \tau_r}{A \vdash \lambda x.\ e\colon \tau_d \to \tau_r}$$

[APP] $$\frac{A \vdash e : \tau_d \to \tau_r,\ A \vdash e' : \tau_d}{A \vdash (e\ e') : \tau_r}$$

[CON] $$\frac{A \vdash T : \varepsilon_d \to \tau_r,\ A \vdash e : \tau_d < \varepsilon_d}{A \vdash (T\ e) : \tau_r}$$

[LET] $$\frac{A \vdash e : \tau_1,\ A.\ \alpha\colon \overline{A\tau_1} \vdash e' : \tau_2}{A \vdash (\textbf{let } x = e \textbf{ in } e') : \tau_2}$$

[DES] $$\frac{A \vdash T : \varepsilon_d \to \tau_r,\ A \vdash e : \tau_r,\ A.\ \alpha\colon \overline{\varepsilon_d} \vdash e' : \tau}{A \vdash (\textbf{let } T x = e \textbf{ in } e') : \tau}$$

These rules limit the generalisation of existential quantified types to let-bound variables in the same way that universally quantified types are restricted by in Hindley/Milner; this enables existential quantification to be easily supported as described earlier [7.2.3]. In [Mitchell] the equivalent of our rule [DES] has the additional restriction that none of the introduced ρ_i are free in the type τ, thus prohibiting the ρ_i from escaping from the scope of the let-binding.

This restriction is not imposed by our rules as the ρ_i are unique and allowing them to escape cannot therefore break encapsulation and give access to the original type hidden by the [CON] rule. For example the following Hope*C fragment is valid under our inference rules, but would be prohibited under those of [Mitchell]:

```
data pack == pack(alpha # (alpha -> num));

dec f : pack -> num;
--- f(x)
<= let (val, ordinal)
       == let pack(v, ord) == x
          in (v, ord)
   in ordinal(val);

let p == pack('c', ord) in f(p);
```

In this example the unique ρ_i introduced to type `v` and `ord` escape from the scope of the inner let-binding and are assigned to the types of `val` and `ordinal` in the outer let-binding. This does not break the encapsulation of `p` as the type of the values used to construct it are never available from within the function `f`, or more precisely anywhere outside the sub-expression `pack('c', ord)`.

7.4. The Extended Type System in Hope+C

We conclude this chapter with a description of the use of our type system in the Hope+C language. So as to conform to the Hope+ [Perry 88a] definition all global functions in Hope+C must have a type declaration‡. This means that although the type-checker infers type information, full inference is not easily available. However, a Hope+ program† consisting of a single module and defined wholly by nested qualified expressions does not posses any type declarations, so reliance on full type inference is possible, though not in a very convenient manner.

7.4.1. Base Types and Basic Type Expressions

Hope+C has five base types: integers (**num**), floating point (**real**), characters (**char**), boolean (**truval**) and the unit type (**void**). The basic type operators are product (**#**) and function (**->**). The syntax of basic type expressions is as follows:

```
<type>        ::=  num | real | char | truval | void
<type expr>   ::=  <type>
                   | <type expr> # <type expr>
                   | <type expr> -> <type expr>
                   | ( <type expr> )
```

Product (#) binds tighter than the function (->), so for example:

```
num # num -> real        is the type of a function from two integers to a real
num # (num -> real)      is the type of a pair consisting of an integer and a
                         function from integers to reals
```

7.4.2. Function Declarations

All global functions must have their type declared using a **dec** statement before they are used or defined:

```
<function decl>   ::=  dec <fun name> : <type expr> ;
```

As examples, the following function declarations are taken from the Hope+C standard library:

```
dec ord : char -> num;                          ! ordinal value !
dec divmod : num # num -> num # num;          ! divide & modulus !
```

---

‡It is outside the scope of this thesis, as Hope+ is a pre-existing invariant in our research, to discuss the relative merits or otherwise of global function declarations in Hope+.

†A Hope+C program (currently) always contains at least one type declaration, so this comment only applies to the Hope+ subset of Hope+C.

7.4.3. Type Declarations

New types are declared by the **data** statement and are defined by one or more *constructors*. (Note that only *disjoint* union types can be defined in Hope+C):

```
<type decl>     ::=  data <type name> { <parameters> }
                         == <const decl> { ++ <const decl> }* ;
<const decl>    ::=  <const name> { <type expr> }
<parameters>    ::=  <name>
                     | ( <name> { , <name> }* )
<type>          ::=  num | real | char | truval | void | <type name> { <parameters> }
```

For example, the Hope+C standard environment defines the following types:

```
data truval == false ++ true;                          ! the boolean type !
data list(val) == nil ++ cons(val # list val);              ! lists !
```

7.4.4. Type Variables

Both universally and existentially quantified types are defined using *type variables*. These are declared in Hope+C using the **typevar** statement:

```
<typevar decl>  ::=  typevar <tvar name> { , <tvar name> }* ;
<type>          ::=  num | real | char | truval | void | <type name> { <parameters> }
                     | <tvar name>
```

The type variables *alpha* and *beta* are declared in the Hope+C standard environment by:

```
typevar alpha, beta;
```

Type variables occurring in the declarations of *functions* are implicitly universally quantified, as specified by the type system above, allowing the definition of polymorphic functions. For example, the function:

```
dec map : (alpha -> beta) # list(alpha) -> list(beta);
--- map(_, nil) <= nil;
--- map(f, h::t) <= f(h) :: map(f, t);                              ! (a) !
```

is the polymorphic list map function, whose type is:

$$map : \forall \alpha, \beta \,.\, (\alpha \rightarrow \beta) \times list(\alpha) \rightarrow list(\beta)$$

Type variables occurring in *type* declarations are implicitly existentially quantified, as specified by the type system above. This allows us to implement abstract data types by encapsulating both a hidden concrete type and the associated access functions within a type declaration. For example, the ubiquitous "stack" type may be declared as follows:

```
data stack(val)
     == stack(alpha #                                    ! the stack !
              (val # alpha -> alpha) #                 ! push function !
              (alpha -> val # alpha) #                  ! pop function !
              (alpha -> truval)                       ! test for empty !
             );
```

This introduces the parameterised type:

$$\text{stack}(\tau) : \exists\alpha \,.\, \text{stack}\ \alpha \times (\tau \times \alpha \to \alpha) \times (\alpha \to \tau \times \alpha) \times (\alpha \to \text{truval})$$

```
dec liststack : stack(alpha)
--- liststack
 <= stack(nil,                                          ! empty stack !
          cons,                                       ! push function !
          lambda h::t => (h, t) end,                   ! pop function !
          lambda l => l = nil end                    ! test for empty !
         );
```

Such abstract data types can now be used without any knowledge of their implementation. As an example the following function to interchange the top two elements on a stack can only do so using the access functions provided:

```
dec swap : stack(alpha) -> stack(alpha);
--- swap(s & stack(v,push,pop,isempty))
 <= if isempty(v)
    then s
    else
       let (e1,rem1) == pop(v) in
       if isempty(rem1)
       then s
       else
          let (e2,rem2) == pop(rem1)
          in (push(e2,push(e1,rem2)),push,pop,isempty);
```

Note: abstract data types constructed using existentially quantified types would be more convenient if Hope+C provided labelled record types. The implementation of these is straightforward but in this discussion we restrict ourselves to the syntax of Hope+.

7.4.7. Type Synonyms

A type synonym is a shorthand for a type expression, and is declared by a **type** statement:

```
<synonym decl>   ::=   type <syn name> { <parameters> } == <type expr> ;
<type>           ::=   ... | <syn name> { <parameters> }
```

For example, a synonym for a pair of two values both the same type may be declared using:

```
type pair(typ) == typ # typ;
```

As type synonyms are not new types they are not existentially quantified. To avoid name clashes, any type variables appearing in the type expression of a synonym are "α-converted" when the synonym is included in another type expression. For example, given:

```
type pack == alpha # (alpha -> num);
dec cpack : pack # pack -> truval;
--- cpack((a,fa),(b,fb)) <= fa(a) = fb(b);
```

The type of the function *cpack* is:

cpack : $\forall\alpha,\beta\ .\ (\alpha \times (\alpha \rightarrow \text{num})) \times (\beta \times (\beta \rightarrow \text{num})) \rightarrow \text{truval}$

and not:

cpack : $\forall\alpha\ .\ (\alpha \times (\alpha \rightarrow \text{num})) \times (\alpha \times (\alpha \rightarrow \text{num})) \rightarrow \text{truval}$

7.4.8. Overloading

A single name may be *overloaded* with any number of different function, constructor and variable definitions; provided no two function or constructor definitions in the same module have the same type. For example, the Hope+C standard environment contains the following six declarations for '–':

```
dec - : num # num -> num;
dec - : num # real -> real;
dec - : real # num -> real;
dec - : real # real -> real;
dec - : num -> num;
dec - : real -> real;
```

7.4.9. Type Inference in Qualified Expressions

The types of variables defined by qualified expressions are not declared by the programmer and their type is inferred by the type-checker. For example, in:

```
let f == lambda (x,y) => x div y * y end in …
```

the type of `f` is inferred as:

f : num × num → num

As in Hindley/Milner, variables defined in qualified expressions where the defining expression is polymorphic are themselves polymorphic. For example, in:

```
let len == lambda
             nil => 0
           | cons(_,t) => 1 + len(t)
           end in …
```

the type of `len` is inferred as:

len : $\forall\delta$. list(δ) $\rightarrow$ num

Unlike Hindley/Milner, the defining expression of a variable may be multi-typed due to overloading, this is handled by implicitly defining multiple singleton-typed variables, one for each of the possible types. For example, given:

```
let max == lambda (x,y) => if x > y then x else y
in (max(2, 3), max("asda", tesco"))
```

Our type system would infer the following set of types for `max`:

max : {num $\times$ num $\rightarrow$ num, real $\times$ real $\rightarrow$ real, char $\times$ char $\rightarrow$ char,
list(char) $\times$ list(char) $\rightarrow$ list(char)}

With this set of types, the expression is well-typed as each application of `max` will assume exactly one of the types from this set.

8. Efficient Type-checking

In this chapter we present a type-checking algorithm designed to support our extended type system of the previous chapter. We start by listing the requirements of the type-checking algorithm, and then state why existing type-checking algorithms, or simple extensions of them, do not match these requirements. We then give an overview of our algorithm and discuss its time/complexity performance. Finally we describe the algorithm in detail and as a practical demonstration of the algorithm, intertwined with this are comments on its implementation in and for Hope+C.

The design criteria we set for the type-checking algorithm fall into three categories: the basic type features required to support the extended type system, extra features added to support type inference, and those features required to make the algorithm suitable for use in practical compilers. The basic type features required to be handled are:

a) Polymorphism.

b) Existential Quantification.

c) Overloading Resolution.

d) Type Inference.

Items (a-c) are to support the our type system; item (d) comes from our desire to provide the benefits of strong typing whilst allowing the option to omit explicit type declarations.

In order to make the algorithm more general and so applicable to a range of languages, we impose two further requirements on the algorithm:

e) Mixed Type Inference and Type Checking.

Some languages support a mixture of explicitly and implicitly typed definitions, for example the Hope+C language where functions are explicitly typed yet where qualified expressions are not.

f) Declaration/Definition Pairing.

In languages where function definitions are syntactically separate entities from function declarations, definitions often textually occur at an arbitrary point after the declarations, and need *not* be grouped together. A type-checker must determine to which declaration

a given definition belongs; this is unfortunately non-trivial in the presence of overloading as it is possible for one definition to belong to more than one declaration.

Finally, to make the algorithm suitable for use in practical compilers we include the requirement:

g) Efficiency.

This is non-trivial due to item (c) above. For the algorithm to be efficient it must perform with a good time/complexity performance in the presence of overloading. Also we wish the algorithm to be suitable for use in compilers written in functional languages, so it must have an efficient declarative formulation.

The primary difficultly in meeting the above criteria is achieving overloading (c) combined with polymorphism (a) and inference (d), while still maintaining a good complexity/performance ratio in a declarative algorithm (g). The other items are fairly trivial: (b) can be provided by a generalisation of the mechanism required to support (a), as shown in [7.2.3]. Item (e) can be supported by first inferring the type (d) and then unifying it with the declared type. Item (f) simply requires a strategy to handle the occasion of multiple declarations matching a definition, we choose to associate an instance of the definition to each declaration.

Unfortunately no existing type-checking algorithms meet all of the above criteria. Milner's algorithm [Milner 78], as used in ML [Milner 84], and the Ponder type system [Fairburn] do not support overloading resolution (c). Type-checking systems have been proposed which do support overloading, for example those of [Leviant][Kaes] and the system recently proposed for Haskell [Hudak][Walder], but these either are inefficient for sequential implementations or restrict overloading in some way. For example both *parametric overloading*, defined by Kaes, and *type classes*, proposed for Haskell, limit overloading to types of the same *shape*, by which we mean that any two types which are overloaded must both be instantiations of a common more general type. Thus they both allow the '-' symbol to be overloaded for integer and real subtraction:

```
- : num × num → num
- : real × real → real
```

as these two types have the same shape being instantiations of the type $\forall\alpha.\ \alpha \times \alpha \rightarrow \alpha$. But neither allow the same symbol to be used simultaneously for integer and real negation as well:

```
- : num → num
- : real → real
```

as these types are not instantiations of $\forall\alpha.\ \alpha \times \alpha \rightarrow \alpha$. (Haskell avoids this problem by including the '-' symbol in the BNF for the language, but this is the *only* operator dealt with in this way.)

Again, unfortunately, no existing algorithm is readily extendable to meet our requirements. For example, Milner presents two formulations of his type-checking algorithm [Milner 78]: algorithm W which is applicative but hardly an efficient algorithm, and a more efficient algorithm J, which is imperative. Neither of these algorithms would form suitable basis for the development of an algorithm to support the overloading or efficiency requirements.

We have therefore developed a new type-checking algorithm which is related to the Milner design, based on recursive walk of the parse tree, a type substitution list and a unification algorithm. However, unlike Milner's algorithm, the type substitution is not idempotent and includes set-valued and dependent substitutions, in addition to simple substitutions. The recursive tree walk and unification algorithm also differ so as to support existential quantification in addition to the extended type substitution.

The algorithm has three main components; the type substitution, the unification algorithm, and the recursive tree walk algorithm for checking expressions. In the remainder of this chapter we describe the algorithm and these components in detail. We start with the definition of a simple language which we use shall to describe the operation of our algorithm [8.1]; the definition of the type expressions which the algorithm will operate upon [8.2]; and then give an overview of the algorithms structure and discuss its time/complexity performance [defined in 8.3]. We then proceed to the three main components, starting with the definition of the type substitution a how its is manipulated [8.4]. The operation of the type substitution is central to the overall type-checking algorithms time/complexity performance and this section explains how this is achieved. This is followed by the definition of the unification algorithm [8.5] and finally a description of how declarations [8.6] and expressions [8.7] are type-checked. Intertwined with these descriptions we make comments on the implementation of the algorithm in Hope+C, these comments may be omitted on a first reading if desired. This description assumes the existence of a *symbol table manager* which provides functions to construct and maintain the *symbol table*. This manager may be constructed using standard techniques and its internal operation is not discussed here.

8.1. A Simple Language

We shall use the following language to describe the operation of our algorithm:

Expr ::=	
a, b, c, etc.	Constants
| A, B, C, etc.	Identifiers
| (<Expr>$_1$, ..., <Expr>$_n$)	Tuple expression
| <Expr>$_f$ <Expr>$_a$	Function/constructor application
| **let** <Expr>$_p$ == <Expr>$_l$ **in** <Expr>$_r$, **otherwise** <Expr>$_o$	Non-recursive qualified expression
| **letrec** <Expr>$_{p1}$ == <Expr>$_{l1}$ |...| <Expr>$_{pn}$ == <Expr>$_{ln}$ **in** <Expr>$_r$, **otherwise** <Expr>$_o$	Recursive qualified expression
| λ <Expr>$_{p1}$ ← <Expr>$_{r1}$ | ... | <Expr>$_{pn}$ ← <Expr>$_{rn}$	Lambda abstraction

In the above we allow *patterns* in both qualified expressions and lambda abstractions, so the language is a little more than a sugared lambda calculus – as is often used as a basis for typing algorithms [Peyton-Jones 87]. We argue that type-checking should be performed before pattern matching has been compiled out, as this makes it possible to inform the programmer of type errors in relation to the original program source. The **otherwise** clauses on the qualified expressions provide expressions whose value should be returned if the pattern in the qualified expression fails to match. We have omitted the **if** construct from the above language as its typing is identical to that of the application of an appropriately defined function with type:

boolean $\times \alpha \times \alpha \rightarrow \alpha$.

Implementation Note:

In a compiler expressions are represented by some data structure and must, during the process of type-checking, be annotated with types. Also there are usually more expression constructs than those shown above. In Hope*C the following two mutually recursive types are used to represent expressions, in the definition of *Etype* all e_i are instances of *Expr* (the syntax defined in 7.1 above and not Hope*C syntax is used here for clarity):

Expr == Etype × Texpr

Etype ==	
Constant c	constant
| Identifier i	identifier (untyped)
| Function f	function identifier
| Constructor f	constructor identifier
| Variable f	variable (pattern/qualified) identifier
| Tuple e_1 ... e_n	tuple (e_1 ... e_n)
| Apply e_f e_a	function/constructor application $e_f(e_a)$

|	If e_p e_t e_e	if expression
|	Let (e_{p1},e_{l1}) ... (e_{pn},e_{ln}) e_r e_o	non-recursive qualified expression let e_{p1} == e_{l1} | ... | e_{pn} == e_{ln} in e_r otherwise e_o, $n \geq 1$
|	Letrec (e_{p1},e_{l1}) ... (e_{pn},e_{ln}) e_r e_o	*possibly* recursive qualified expression
|	Fun (e_{p1},e_{r1}) ... (e_{pn},e_m)	function expression lambda $e_{p1} \Rightarrow e_{r1}$ | ... | $e_{pn} \Rightarrow e_m$, $n \geq 1$
|	Eqn f e_p e_r	equation definition, f $e_p \Leftarrow e_r$
|	Equiv e_p e_q	pattern equivalence, e_p & e_q
|	None	Filler for optional expression e.g. missing otherwise

There are few differences between the language represented by above types and our simple language, the major one being the provision for a type annotation, *Texpr*, on every expression. The *Function*, *Constructor* and *Variable* constructs are introduced as a result of type-checking and convey more information to the code generator than the generic *Identifier*. In Hope+C functions are defined by one or more equations, which need not be a contiguous block in the program, thus the *Eqn* construct is added to represent them. An *Eqn* construct can be typed in the same way as a lambda abstraction. However, if a function type declaration is present, as required by Hope+C but which our type-checker provides an option to omit, then an *Eqn* can be treated differently: first type the pattern and unify its type with the declared domain type of the function, then type the right hand side and unify its type with the declared range type. The advantage of handling *Eqn* constructs in this way is that the first unification may resolve type variables present in the pattern type, which in turn may simplify the typing of the right hand side. The *Equiv* construct is added to represent Hope+C pattern equivalence, this is typed by simply typing the two sub-expressions and then unifying the types inferred. Finally, the *None* construct is used when the optional *otherwise* clause of the *Let* construct is omitted.

8.2. Type Expressions

A type is represented by a type expression, defined as (using the syntax of 7.1 again):

<Type> ::=		
	<Type>$_1$ × ... × <Type>$_n$	product, $n \geq 2$
|	<Type> → <Type>	function
|	A, b, C, etc. (alphabetic Roman)	constant: integer, boolean etc.
|	A(<Type>$_1$, ..., <Type>$_n$)	parameterised type e.g. list(num), A ∈ constants
|	A, B, Γ, etc.	user type "variables" (polytype)
|	α, β, ψ, etc.	variable
|	{<Type>$_1$, ..., <Type>$_n$}	set of types
|	[<integer>]	parameter marker

We add three extensions to "conventional" type expressions: firstly the set construct which is to used represent the type of an expression which is multi-typed due to the presence of overloading. Secondly the user type "variables" which represent program defined type variables, for example those defined by the Hope+C `typevar` statement, we shall refer to these as *polytypes*. Finally the parameter marker is used within type expressions to mark where the parameters of a type should be inserted, parameters are numbered left to right starting the leftmost being number one. Parameter markers are not strictly type expressions and only occur in the symbol table definitions of user types (see *Identifier Lookup* [8.7.1] below).

Implementation Note:

In a type-checker types will be represented by a data structure, the following is the type used for Hope+C, all t_i are instances of *Texpr*:

Texpr ==		
	Empty	no type – attached to empty expressions
|	Prod $t_1 \ldots t_n$	product type, $t_1 \times \ldots \times t_n$, $n \geq 2$
|	Map t_d t_r	function type, $t_d \rightarrow t_r$
|	Tvar name	user "type variable"
|	Type name $t_1 \ldots t_n$	named type e.g. primitive or user-defined type. t_1 to t_n are the types parameters, if any.
|	Var i	variable, yet to be resolved
|	Sum $(n_1,t_1) \ldots (n_n,t_n)$	sum type, $(n_i,t_i)_{1 \leq i \leq n}$ are constructor/type pairs.
|	Tset $t_1 \ldots t_n$	set of types, $n \geq 2$; present when overloading remains unresolved.
|	Param i	an argument to a parameterised type
|	Terror *message*	type error.

There are a few additional constructs in this definition compared to that of <Type> above. These are used: to represent user-defined types in the symbol table, for 'space filling' the type annotations in the parse tree when no other value is suitable, and for representing type errors in the parse tree.

Representing type synonyms defined using the Hope+C `type` declaration requires no additional syntax. To represent a type defined using a Hope+C `data` declaration the construct *Sum* is added, this construct only used in type expressions stored in the symbol table and never in parse tree type annotations. The construct *Param* is used to mark the positions of the parameters of a parameterised type declaration. This construct also only occurs in type expressions stored in the symbol table; when the type expression for a parameterised type is extracted from the symbol table all *Param* constructors are replaced by the type expressions of

the actual parameters. The remaining additional constructs *Unknown*, *Empty* and *Terror* are used as parse tree type annotations representing; the initial type before type-checking has occurred, the type of a *None* construct, and errors discovered during type-checking – the argument to this construct being an explanatory error message.

8.3. Overview of the Type-checking Algorithm

The type-checker takes as input a parse tree and the current symbol table. The parse tree represents a *single* type declaration, function declaration or function equation and not the whole program. A simplified definition of the parse tree is:

<parse tree> ::= **type** <name> <type syntax>
| **fun dec** <name> <type syntax>
| **fun equ** <name> <pattern expression> <body expression>

The type-checker returns a new parse tree annotated with type information at every node (or annotated with error messages if type-checking fails), and if successful an updated symbol table with the declaration or definition included. A sketch of the algorithm for type-checking a program is as follows:

For **type** and **fun dec**:

Check and convert <type syntax> to an internal type expression, if successful enter into symbol table.

For **fun equ**:

a) The equation is type-checked in a bottom-up manner, checking the <pattern expression> first and then the <body expression>, and then annotating each node with its type and constructing a type substitution. Identifier nodes which are overloaded are annotated with the set of possible identifiers and their types, all other leaf nodes are annotated with either their singleton-type, or if their type is unknown with a type variable. The types of non-leaf nodes are formed by combining the types of their sub-nodes; if any sub-nodes are multi-typed all permutations for the node type are first constructed and then the invalid ones removed to produce the nodes singleton or multi-type annotation (construction/removal are combined, see 8.5).

b) If no type errors occurred during phase (a) then a top down pass replacing all type variables on nodes with their inferred types is performed. If possible any multi-typed nodes are resolved.

c) If no type errors occurred during phase (b) then success is reported and the equation entered in the symbol table. Otherwise it is reported that the overall type was deducible but that a local ambiguity could not be resolved.

It is obvious that type errors could occur during phase (a) above due to program errors, but the source of errors during phase (b) is more obscure. The following program segment shows how phase (b) might fail:

```
dec f : num;
dec f : real;
dec g : num -> real;
dec g : real -> real;

... cos(g f) ...
```

In this example the types of the sub-expressions `cos(g f)` and `g f` are both inferred to be `real`, however it is impossible to determine the type of `f`, and hence which definitions of `f` and `g` are being referred to; the expression is therefore ambiguous even though the overall type can be inferred.

8.3.1. On The Time/Complexity Performance of the Algorithm

In this section we discuss the performance of our algorithm relating it to that of the (non-overloading) Milner algorithm. For the purposes of this discussion we shall assume that the performance of the Milner algorithm is approximately linear in the size of the expression; for practical purposes this is true although recent theoretical work [Kanellakis] has shown that is can be exponential in some cases. Introducing overloading into a type system increases the time taken to perform type-checking, since for each use of an overloaded identifier the actual instance required needs to be determined, which in turn effects the types of the expressions containing the identifiers. We define the time/complexity performance of a type-checking algorithm as the ratio between time the algorithm takes to produce a type and the complexity of the overloading involved. Producing a precise formulation for the time/complexity ratio of our algorithm would be difficult, if not impossible, as it is complex function of the number of identifiers which are overloaded, the number of instances of each overloaded identifier, the

(di)similarity of the types of each instance, and the interaction between the resolution of the types of different nodes. However, the upper and lower time/complexity bounds related to the performance of the non-overloading Milner algorithm can be determined. To demonstrate the typical performance of our algorithm we present an example and the times taken to type-check it using Milner's algorithm without overloading, the naïve overloading algorithm of [7.3.2], and our improved algorithm.

First, let T_e be the time taken by the Milner algorithm to type-check an expression e, and t be the time taken to type-check an individual node of an expression. We shall assume that Milner's algorithm is approximately linear, so for an expression tree consisting of x nodes $T_e \approx xt$. Let C be a simple measure of the complexity of an expression defined as:

$$C = \prod n_i$$

where n_i is the number of possible instances for the i^{th} identifier in the expression.

When no overloading is present in an expression (i.e. $C = 1$) the time taken by our algorithm to type-check will be equal to that of the Milner algorithm as no multi-types need to be resolved. Thus the best case performance of our algorithm is T_e.

The worst case time/complexity performance is delivered by an algorithm which naïvely implements the inference rules of [7.3.2]. That is, an algorithm which first converts an expression containing overloading to a set of expressions, each of which contains a different instantiation of the overloaded identifiers, and then performs a simple Milner type-check on each element of the set. This results in a time/complexity of CT_e. Our algorithm's worst case performance is equal to that of this naïve algorithm, CT_e, but this occurs if and only if typing the top node of an expression tree succeeds in resolving multi-types and all permutations of overloaded identifiers are valid for all the sub-nodes of the top node. This condition is rare in practice, but not impossible. For example typing the expression:

```
float(x + y)
```

where:

```
x : real and int; y : real and int
+ : real × real → real and int × int → int
    and int × real → real and real × int → real
float : int → real
```

will take $16T_e$ in both the naïve algorithm and our improved algorithm, as only the typing of the `float` application node resolves which instances of `x`, `y` and + are required.

The time/complexity performance of our algorithm is typically much better than the worst case/naïve algorithm as instead of performing multiple type-checks of an expression only one check is made with the unification algorithm [8.5.] attempting to resolve overloading on a per node basis through the use of the set valued [8.4.3.] and dependent [8.4.4.] substitutions. This approach ensures that there is no overhead in typing nodes which involve no overloading or whose sub-nodes have already had their uses of overloading resolved.

As an example consider the fibonacci function:

```
fib n <= if n < 2 then 1 else fib(n-1) + fib(n-2);
```

The expression tree for this function has fifteen nodes (including the one in the pattern), so if no overloading is present the Milner algorithm would take $T_o \approx 15t$, this is the best case performance of our algorithm. However, in Hope+C the '<' operator is overloaded with four separate definitions, the '−' operator (which occurs twice) with six , and the '+' operator with four; which gives a value for C of 576! Therefore the naïve algorithm would take $576T_o \approx 8640t$ to type-check the function.

Our algorithm in typing the conditional expression will first type the predicate and then the sub-expressions. In typing the application node for '<' the unification will perform four separate checks, one for each instance of the function, only one of these will succeed uniquely identifying the instance of '<' and determining the type of `n`. The cost of these extra is $3t$, similarly in typing the branches multiple checks will be made giving a total extra cost of $16t$ and a total time for the whole expression of $T_o + 16t \approx 31t$.

To summarise; for typing fibonacci the best case (no overloading/Milner) is $15t$, our improved algorithm $31t$ and the worst case (naïve algorithm) $8640t$. As further examples, the times for the factorial function are; $12t$ Milner's algorithm, $23t$ our improved algorithm and $168t$ naïve algorithm; and for the integer average of an integer list; $19t$ Milner, $25t$ ours and $304t$ naïve.

The performance of type systems in general is something which warrants further research. The inherent difficulty of infering the average complexity of type-checking algorithms suggests that a detailed profiling analysis is required. Such an analysis would provide considerable insight into the behaviour of type systems, including our algorithm for overloading resolution, and could be used to guide the optimisation of the type-checkers.

8.4 The Type Substitution

The *type substitution* is at the centre of the type-checker's ability to handle type inference and resolve overloading with a good time/complexity performance. The type substitution is a mapping from type variables to their currently inferred types. The type substitution may map a type variable to one of: a ground type; a non-ground type; a set type; or a type dependency. In the examples of type substitutions '⇒' is read as "is mapped to".

Implementation Note:

In Hope+C the type substitution is maintained as a list of pairs of type expressions:

```
type subst == list(Texpr # Texpr);
```

In each pair the first element is a 'Var i' expression† and the second is the current inferred information about the type variable. If there is no current information for a type variable then there is no entry for it in the type substitution.

8.4.1. Ground Type Substitution

A ground type is one containing no variables or sets of types. For example, the following are valid ground type substitutions:

$\alpha \Rightarrow$ num, $\phi \Rightarrow$ (num $\times$ (real $\times$ char))

8.4.2. Non-ground Type Substitution

A non-ground type substitution is one which contains references to other type variables. For example:

$\alpha \Rightarrow$ num
$\tau \Rightarrow (\alpha \times$ char)

The presence of non-ground substitutions means that the type substitution is not idempotent, a property which is required by [Milner 78] but which prevents the correct resolution of overloading. A substitution is idempotent if applying the substitution to a type expression which has already had the substitution applied once does not alter the type expression, that is:

$$\forall e \in \textit{expressions}, S \in \textit{substitutions}.\ S(e) = S(S(e))$$

† The first element could alternatively be represented by the type variable number 'i' and not the complete expression 'Var i'. The choice between the two involves different trade-offs and neither is "better" in general.

Making a substitution idempotent removes dependencies between variables which are required to enable correct overload resolution. For example, the substitution (see [8.4.3] for the definition of set valued substitutions):

$\alpha \Rightarrow$ {num, real}
$\beta \Rightarrow \alpha$

defines α and β to have the same value, which is one of *real* and *num*. Making this substitution idempotent produces:

$\alpha \Rightarrow$ {num, real}
$\beta \Rightarrow$ {num, real}

which allows α and β to have *different* values, which is incorrect.

Implementation Note:

For efficiency chains of indirection through intermediate non-ground substitutions are kept short, thus improving the lookup time for substitutions [8.4.5]. For example, the substitution:

$\alpha \Rightarrow$ num, $\varepsilon \Rightarrow \alpha$, $\eta \Rightarrow \sigma$, $\varphi \Rightarrow \kappa$, $\sigma \Rightarrow \varphi$

is transformed into:

$\alpha \Rightarrow$ num, $\varepsilon \Rightarrow$ num, $\eta \Rightarrow \kappa$, $\varphi \Rightarrow \kappa$, $\sigma \Rightarrow \kappa$

8.4.3. Set Valued Type Substitutions

A set valued type substitution is one which maps a type variable to a set of types. These only occur during overloading resolution. The process of type-checking will (should) refine a set valued substitution by successively decreasing the number of elements in the set. Overloading resolution is complete when (if) a singleton set is produced, at which point the set is replaced by its only element. The type-checking process fails if does not refine all set valued substitutions to singletons and thus remove them. For example, the following is a set valued substitution:

$\eta \Rightarrow$ {num, δ}

Implementation Note:

For efficiency type expressions contain at most *one* set valued sub-expression, and then only as the outermost sub-expression. This restriction makes finding the set of current inferred values for a type variable possible *without* scanning the type substitution for imbedded type sets. For example, the type expression:

num × {real, char}

is transformed into:

{num × real, num × char}

However, this restriction *does not* mean that a type expression may not contain a type variable which currently has a set valued substitution, rather these variables must be left in to enable proper overloading resolution. For example, the following substitution:

$\alpha \Rightarrow$ {num, real}
$\varphi \Rightarrow \alpha \times$ char

must not be transformed into:

$\alpha \Rightarrow$ {num, real}
$\varphi \Rightarrow$ {num × char, real × char}

as this would remove the connection between α and φ [8.4.2].

8.4.4. Dependent Type Substitutions

A dependent type substitution has the form:

<type variable> ⇒ **match** <pattern>, <target>

where both <pattern> and <target> are type expressions and the <type variable> occurs in the <pattern> expression. The mapping of the <type variable> under this substitution is the value obtained for it when the <pattern> and <target> expressions are unified. For example, given the substitution:

$\alpha \Rightarrow$ {num → num, real → real}
$\zeta \Rightarrow$ **match** $\zeta \rightarrow \upsilon, \alpha$
$\upsilon \Rightarrow$ **match** $\zeta \rightarrow \upsilon, \alpha$

then the types of ζ and υ under the substitution, obtained by unifying the expressions $\zeta \rightarrow \upsilon$ and {num → num, real → real}, are either both num or both real.

Using dependent substitutions; when a new non-set substitution is determined for a type variable which appears as the target in one or more dependent substitutions, then those dependent substitutions may be immediately resolved. For example, if the mapping:

$\alpha \Rightarrow$ num → num

was added to the above substitution, then not only would the set valued mapping for α be resolved to a single type, but the mappings for ζ and υ could also be resolved as they are

dependent upon α. Similarly, when a new substitution is determined for a type variable which either has dependent substitution or which appears in the pattern of a dependent substitution then all related mappings can be updated. For example given the substitution:

$\alpha \Rightarrow \{num \times real, real \times char, num \times char\}$
$\beta \Rightarrow \{char \times char, num \times num\}$
$\zeta \Rightarrow$ **match** $\zeta \times \upsilon, \alpha$
$\upsilon \Rightarrow$ **match** $\zeta \times \upsilon, \alpha$
$\omega \Rightarrow$ **match** $\upsilon \times \omega, \beta$

then if the new mapping:

$\zeta \Rightarrow num$

is added the following sequence of updates occurs due to the dependencies:

$\alpha \Rightarrow \{num \times real, num \times char\}$;
$\upsilon \Rightarrow \{real, char\}$;
$\beta \Rightarrow char \times char$;
$\omega \Rightarrow char$;
$\upsilon \Rightarrow char$;
$\alpha \Rightarrow num \times char$

and the final substitution is:

$\alpha \Rightarrow num \times char$
$\beta \Rightarrow char \times char$
$\zeta \Rightarrow num$
$\upsilon \Rightarrow char$
$\omega \Rightarrow char$

It is important that if two or more type variables which are related by dependent substitutions are involved in the same type expression then they must be substituted consistently; failure to ensure this will result in incorrect typings. For example, under the substitution:

$\alpha \Rightarrow \{num \rightarrow num, real \rightarrow real\}$
$\zeta \Rightarrow$ **match** $\zeta \rightarrow \upsilon, \alpha$
$\upsilon \Rightarrow$ **match** $\zeta \rightarrow \upsilon, \alpha$

the type of the expression $\zeta \times \upsilon$ is:

$\{num \times num, real \times real\}$

and not:

$\{num \times num, real \times real, num \times real, real \times num\}$

which would be the result if υ and ζ were substituted independently of each other.

Implementation Note:

It is convenient to encode dependent type substitutions as a new form of type expression rather than modify the definition of the type substitution. This is achieved in Hope+C by adding a new construct to *Texpr*:

```
Texpr ==
        ... previous constructs ...
|       Match t_p t_t
```

Implementing the requirement for consistent substitutions of type variables related by dependent substitutions efficiently is essential if the type-checker is to perform with a good time/complexity ratio. In an imperative language it might be possible to mark (i.e. in-place modify) the substitution in some way so that future expansion of dependent substitutions could take account of earlier ones, however, this is not possible in a pure functional language. The algorithm we have adopted in the Hope+C implementation to provide consistent substitutions is to expand out the dependent mappings in a substitution to produce a set of substitutions immediately before applying the substitution, and then do multiple applications of the substitution forming a set type from the results. If the substitution is being applied as part of the unification of two type expressions then multiple unifications are performed and the resulting substitutions from each unify are then combined back into a single substitution by adding set valued and dependent substitutions as required. To ensure a good time/complexity performance for the type-checker, these expansion and re-combination operations are only performed when a substitution is being applied to a type expression which contains a type variable which has a dependent mapping under the substitution. This means that expressions which do not involve overloading do not pay any penalty for its support. As the re-combination operation is performed directly after two type expressions have been unified, if the unification has refined the substitution then this information is immediately available to subsequent substitution applications and type expression unifications, which reduces the number of type possibilities they have to operate with. The operations for expanding and re-combining substitutions are *expand_subst* and *merge_subst* respectively and are described in the next section [8.4.5]. The unification algorithm is described in [8.5] below.

8.4.5. Operations on the Type Substitution

This whole section is an "implementation note" and explains how the type substitution is implemented efficiently in Hope+C. Though describing the Hope+C implementation we use our notation for types for conciseness rather than the Hope+C type.

The following functions are the primary operations supplied to maintain and apply the type substitution. To the type-checker, the type substitution appears as an abstract data type and only those operations exported from the type substitution definition module may be used to operate upon it.

The definition of the type substitution enables the resolution of overloading, the efficient implementation of the operations upon the type substitution make it practical possibility. The following operations are implemented so that chains of non-ground substitutions are kept short. For example; if the new substitution $\alpha \Rightarrow \beta$ is added to the substitution $\beta \Rightarrow \phi$ then the result is $\{\alpha \Rightarrow \phi, \beta \Rightarrow \phi\}$ which minimises the levels of indirection, and hence improves performance. Similarly the effect of resolving a dependent substitution for one variable is propagated to other indirectly affected variables. For example; if the substitution:

$\varphi \Rightarrow$ real

is added to the substitution:

$\alpha \Rightarrow \{\text{num} \times \text{num}, \text{real} \times \text{real}\}$
$\omega \Rightarrow$ **match** $\omega \times \phi, \alpha$
$\phi \Rightarrow$ **match** $\omega \times \phi, \alpha$

then the resultant substitution is:

$\alpha \Rightarrow \text{real} \times \text{real}$
$\omega \Rightarrow$ real
$\phi \Rightarrow$ real

Thus any future references to ω, as well as to ϕ, do not involve the expansion of a *Match* expression, and are therefore quicker. The operations on the type substitution are: *apply_subst*, *extend_subst*, *update_subst*, *expand_subst* and *merge_subst*:

apply_subst *substitution type_expression*

All type variables in the *type_expression* are replaced by their value under the *substitution*. Set valued, non-ground and dependent types are correctly expanded. If there are any set valued types remaining in the *type_expression* after substitution they are "floated" to the outermost level so a single set of types is returned [8.4.3].

extend_subst *substitution type_expression$_1$ type_expression$_2$*

The *substitution* is extended with *type_expression$_1$* ⇒ *type_expression$_2$*.

update_subst *substitution type_expression$_1$ type_expression$_2$*

The existing substitution for *expression$_1$* is replaced by *expression$_2$*, if the existing substitution is set valued, non-ground or dependent, then any associated type substitutions are correctly updated.

expand_subst *substitution*

Given a *substitution* expand_subst converts it to a list of substitutions, each of which contains no set valued, non-ground or dependent types. This operation is required to ensure that all set valued and dependent substitutions are expanded consistently.

merge_subst *original_substitution list_of_substitutions*

This operation produces a new single substitution containing set valued, non-ground and dependent expressions. It is formed by combining the *list_of_substitutions*, which are the results of type-checking using a list of type substitutions generated by *expand_subst*, using information on dependent expressions gleaned from the *original_substitution*, which was the argument to the original *expand_subst* call.

The implementation of *apply_subst*, *extend_subst* and *update_subst* is trivial. The algorithm for *expand_subst* is:

a) Expanding all sets to produce a list of substitutions.

b) Removing all dependent types, what they depend upon will now be singleton-typed, so performing this operation will not introduce any new sets.

c) Removing all indirections.

The implementation of *merge_subst* is the most involved, a sketch of the algorithm is:

a) First merge the *list_of_substitutions* into a single substitution by introducing set valued expressions where appropriate. If one type substitution contains a value for a type variable and another does not then special care is needed to ensure that the resultant type substitution reflects this.

b) If any type variable which was substituted by a dependent expression in the *original_substitution* is now substituted by a set valued expression, then reinstate the dependent expression. This ensures that overloading dependencies are not lost.

Expand_subst and *merge_subst* form a pair which are used to ensure that dependent substitutions are expanded consistently. For example, applying *expand_subst* to:

$\alpha \Rightarrow \beta$, $\eta \Rightarrow$ Match($\eta \times \sigma$, γ), $\sigma \Rightarrow$ Match($\eta \times \sigma$, γ),
$\gamma \Rightarrow$ {num $\times$ char, char $\times$ real, real $\times$ num}

will result in the three substitutions:

i) $\alpha \Rightarrow \beta$, $\eta \Rightarrow$ num, $\sigma \Rightarrow$ char, $\gamma \Rightarrow$ num $\times$ char
ii) $\alpha \Rightarrow \beta$, $\eta \Rightarrow$ char, $\sigma \Rightarrow$ real, $\gamma \Rightarrow$ char $\times$ real
iii) $\alpha \Rightarrow \beta$, $\eta \Rightarrow$ real, $\sigma \Rightarrow$ num, $\gamma \Rightarrow$ real $\times$ num

Now assume that type-checking with the above three substitutions produces the following results:

i) $\alpha \Rightarrow \beta$, $\eta \Rightarrow$ num, $\sigma \Rightarrow$ char, $\gamma \Rightarrow$ num $\times$ char, $\varepsilon \Rightarrow$ bool
ii) *Failed*
iii) $\alpha \Rightarrow \beta$, $\eta \Rightarrow$ real, $\sigma \Rightarrow$ num, $\gamma \Rightarrow$ real $\times$ num, $\beta \Rightarrow$ list(num)

Applying *merge_subst* to obtain the final substitution, phase (a) above will produce:

$\alpha \Rightarrow \beta$, $\eta \Rightarrow$ {num, real}, $\sigma \Rightarrow$ {char, num}, $\gamma \Rightarrow$ {num $\times$ char, real $\times$ num},
$\varepsilon \Rightarrow$ bool, $\beta \Rightarrow$ {φ, list(num)}

Notice that a new variable φ has been introduced to preserve the information that substitution (i) did not have a mapping for β. Finally applying phase (b) above to reinstate any dependent substitutions will result in:

$\alpha \Rightarrow \beta$, $\eta \Rightarrow$ Match($\eta \times \sigma$, γ), $\sigma \Rightarrow$ Match($\eta \times \sigma$, γ), $\gamma \Rightarrow$ {num $\times$ char, real $\times$ num},
$\varepsilon \Rightarrow$ bool, $\beta \Rightarrow$ {φ, list(num)}

8.5. The Unification Algorithm

The unification algorithm is basically the same as that used by Milner [Milner 78, Robinson], but extended to handle set and dependent type expressions. The unification of two set valued type expressions is the set of the unifications of each possible pair of types formed by selecting one type from each set. To unify dependent type expressions the *expand_subst/ merge_subst* pair of operations defined above are used. A sketch of the unification algorithm is:

```
unify(t1, t2, subst)
<= if type variables of t1 and t2 DO NOT have dependent
      substitutions in subst
   then
      simple_unify(t1, t2, subst)
   else
      let subst_list == expand_subst(subst) in
      let (results, new_substs)
          == unzip map(simply_unify(t1,t2,?),subst_list)
      in (Tset(results), merge_subst(subst, new_substs))
```

where `simple_unify` is the Robinson unification algorithm extended to handle set type expressions as described above.

Implementation Notes:

These are a few notes on implementing the above algorithm, but do not cover all the nuances. If there are dependent substitutions it is better to first divide `subst` into two parts, one containing type variables used in `t1` and `t2` (directly or indirectly), the other containing the remainder; *expand_subst*/*merge_subst* can then be applied just to the first part and the second part added back onto the final substitution. This saves a lot of time and unnecessary copying. When performing the *map* operation, those *simply_unify*s which fail must be removed, and a check made that at least one succeeded. Any duplicates must be removed from *results* and if this produces a singleton set it must be replaced by its single member.

8.6. Conversion of Type Declarations

Having now covered type expressions, the substitution and the unification algorithm we come to the top-level type-checking algorithm. In this section we cover the checking and conversion to internal type expressions of program type and function declarations. In the next section we shall cover the checking of expressions and function equations.

8.6.1. Type Visibility

When converting type declarations, each identifier occurring in the definition is looked up in the symbol table to obtain its definition. No overloading of type names is allowed, a type defined in the current module obscures any definition with the same name occurring in an imported module, an imported definition overrides any previously imported definition (module imports are

ordered according to their textual occurrence). The same identifier cannot be used to define two or more types within the same module.

8.6.2. Type Variables

Type variables (as defined by `typevar` statements) have a new unique polytype assigned to them are then entered into the symbol table under the current module.

Implementation Note

Due to the lookup process [8.7.1], polytypes need only be unique *within* a single expression.

8.6.3. Type Synonyms

Type synonyms are parsed and converted into type expressions and then entered into the symbol table under the current module. For example:

```
type pack(val) == alpha # (alpha -> val);
```

will, assuming Γ is an unused polytype, be converted to:

pack (synonym, arity $\mathbf{1}$) : $\Gamma \times (\Gamma \rightarrow [1])$

8.6.4. Data Declarations

Data declarations are parsed and converted into type expressions, then for each constructor an entry is made in the symbol table and another entry is made for the type itself. For example, for:

```
data list(val) == nil ++ cons(val # list(val));
```

the following entries will be made:

list (type, arity 1) : Sum ("nil", list([1])) ("cons", [1] × list([1]) → list([1]))
nil (constructor) : list([1])
cons (constructor) : [1] × list([1]) → list([1])

Before constructors are entered into the symbol table a check is performed to ensure that there is no other constructor or function of the same name and type in the *same* module.

8.7. Type-checking Expressions

Typing an expression is basically a recursive tree walk in which lower nodes are first typed and the type of higher nodes is inferred from the types of their subordinate nodes. During this tree walk two main operations are used; identifier lookup to determine which instances of a given identifier are in scope, and type unification to determine the type of a node based on the type of its subordinates.

Implementation Note:

There are three entry points to the type-checker for typing expressions; the first takes a single Hope+C equation computes its type and enters the typed equation into the symbol table under any function which it matches. The second entry takes a single Hope+C expression and infers its type, no symbol table entries are made. The third entry takes a collection of Hope+C equations defining a single function and infers its type, no entries are made in the symbol table. The first entry is the one used by the compiler for Hope+C function bodies, the second for program module "top level expressions". The third exists for the use by the transformation environment and is not used directly by the compiler, supporting this pure type inference entry requires no extra functionality in the type-checker as it is equivalent to passing a Hope+C lambda expression to entry two. In the following description of the type-checking process, we only cover the first entry point as the other two are subsets of it.

8.7.1. Identifier Lookup

The function *lookup* returns the types of all instances of an identifier which are currently in scope. It converts parameter markers and polytypes in the type expressions according to the context of the lookup operation. A sketch of the algorithm is:

```
lookup(identifier, context, symbolTable)
=       in_scope_list = find(identifier, context, symbolTable)
        remove obscured identifiers in in_scope_list
        case context of
                Pattern:
                        if in_scope_list is empty
                        then generate a new type variable and return it
                        else replace all parameter markers in in_scope_list by new
                                type variables and all poltypes by new unique types
                Expression:
                        replace all polytypes and parameter markers in in_scope_list
                        by new type variables
                Equation:
                        replace all polytypes in in_scope_list by new unique types
        end case
        return in_scope_list
```

The symbol table manager must provide the function *find* which returns a list of the instances of an identifier currently in scope. This list must be ordered with the identifier in the nearest enclosing scope at the head of the list. The ordering of scopes is: variables defined in enclosing qualified expressions, if any, innermost qualified expression first; variables defined in the equations pattern, if any; functions defined in current module; and finally, functions defined in imported modules, last imported module first. As mentioned earlier; details of the symbol table manager are not given here, its internal structure not being relevant to the type-checking process.

After calling the symbol table manager to obtain the list of in scope instances the type-checker trims the result removing any identifiers which are obscured by another which has the same type. Polytypes and parameter markers occurring in the types of the identifiers are now converted. The conversion is dependent upon which of three possible contexts the identifier lookup is taking place, these contexts and the action taken are:

Checking a pattern, either in an equation or qualified expression

Identifiers in patterns can be only either constructors or new variables, the symbol table manager lookup process will only of course find the former! If no current instance of an identifier is found a new unique type variable is simply returned and no conversion is required. Otherwise the returned type(s) are converted and then returned. To support existential quantification, polytypes are replaced by new unique types, parameter markers are replaced by new unique type variables. For example the constructor `x` in the type:

```
data t(val) == x(alpha # (alpha -> val));
```

would have its type recorded in the symbol table as:

$$\Gamma \times (\Gamma \rightarrow [1]) \rightarrow t([1])$$

If `x` is looked up while checking a pattern then the following type would be returned:

$$G \times (G \rightarrow \phi) \rightarrow t(\phi)$$

where G is a new unique type name and ϕ a new unique type variable.

Checking an expression other than a pattern

Identifiers in expression can be any one of; functions, constructors or variables. Polytypes only occur if the identifier is polymorphic and replaced by new unique type variables. Parameter

markers only occur if the identifier is a constructor belonging to a parameterised type, these too are replaced by new unique type variables. For example, using the above declaration, if the type of x is looked up while checking an expression the type:

$$\alpha \times (\alpha \rightarrow \beta) \rightarrow t(\alpha)$$

would be returned, where α and β are new unique variables

Checking an equation

When checking an equation the type of the function it belongs to is looked up, this type must be converted if the function being defined is polymorphic. Polytypes in the type are converted to new unique types. Parameter markers do not occur. For example, when checking an equation for the function:

```
dec map : (α -> β) # list(α) -> list(β)
```

whose type is represented in the symbol table by:

$$(\Gamma \rightarrow \Omega) \times \text{list}(\Gamma) \rightarrow \text{list}(\Omega)$$

the lookup process would return the type:

$$(G \rightarrow W) \times \text{list}(G) \rightarrow \text{list}(W)$$

where G and W are new unique types.

8.7.2. Typing the Expression

As stated above in the outline sketch of the algorithm, typing an expression is accomplished by first performing a bottom up scan of the expression followed by a top down scan which replaces any type variables and resolves any set annotations left on identifier nodes. Both scans are defined by cases over our target language and their definitions follow directly from the inference rules given above [7.3]. The first entry point to the type-checker takes a Hope+C equation; represented by a function name, pattern, and right hand side expression triple; together with the current symbol table and type-checks it returning a new equation annotated with type information and, if successful, enters the equation into the symbol table. A sketch of the algorithm, omitting error checks is:

```
typeCheck(name, pattern, expression, symbolTable)
=    let    function_type = lookup(name, symbolTable)
            (type_pat, pat_2, sub_1, st_1) = upscan(pattern, empty substitution, symbolTable)
            (ignore, sub_2) = unify(sub_1, type_pat, domain(function_type))
            (type_rhs, rhs_2, sub_3, st_2) = upscan(expression, sub_2, st_1)
            (ignore, sub_4) = unify(sub_3, type_rhs, range(function_type))
            (pat_3, rhs_3) = downscan(sub_4, pat_2, rhs_2)
       in
            add_equation(name, pat_3, rhs_3, st_2)
```

Where *upscan* is the function which performs the bottom up typing pass, *downscan* is the function which performs the top down fixup pass, *unify* is the unification algorithm described above [8.5], *lookup* is defined in [8.7.1], and *add_equation* is the operation supplied to add a new equation to a function respectively.

The above sketch does not deal with the case of there being multiple definitions of the function *name* in the current module, this is handle by simply applying the above algorithm once for each definition and adding the equation to each function for which the type-check succeeds.

8.7.3. The Upscan Function

Upscan is a function from: an expression, the current substitution – which is initially empty, and the current symbol table; to: a type, an annotated expression, a new substitution and a new symbol table. A sketch of the algorithm is:

```
upscan(expr, substitution, symbolTable)
= case expr in
      a:
            (T, a.T, substitution, symbolTable) where T is the type of the constant

      A:
            let in scope list = lookup(A)
            in if in scope list is empty then
                  let α, st_2 = unique_var(symbolTable) in (α, A.α, substitution, st_2)
            else
                  let    T, st_2 = set of converted types of identifiers in scope
                               (may need unique_var hence st_2)
                         T_2 = T paired with full identifier ids
                         α, st3 = unique_var(st_2)
                         sub_2 = extend_subst substitution α T
                  in
                         (α, A.T_2, sub_2, st_3)

      (e_1, ..., e_n):
            let    t_1, e'_1, sub_1, st_1 = upscan(e_1, substitution, symbolTable)
                   ...
                   t_n, e'_n, sub_n, st_n = upscan(e_n, sub_n-1, st_n-1)
                   T = t_1 × ... × t_n
            in
                   (T, (e'_1, ..., e'_n).T , sub_n, st_n)
```

$e_f\ e_a$:

let $(t_f, e'_f, sub_1, st_1) = upscan(e_f, substitution, symbolTable)$
$(t_a, e'_a, sub_2, st_2) = upscan(e_a, sub_1, st_1)$
$T, sub_3 = unify_apply(sub_2, t_f, t_a)$
in
$(T, (e'_f\ e'_a).T, sub_3, st_2)$
where
unify_apply is a variant of the unification algorithm which unifies the domain of the function against the type of the argument expression and returns the range type.

let e_p == e_l **in** e_r **otherwise** e_o:

let $(t_o, e'_o, sub_1, st_1) = upscan(e_o, substitution, symbolTable)$
$(t_l, e'_l, sub_2, st_2) = upscan(e_l, sub_1, st_1)$
$(t_p, e'_p, sub_3, st_3) = upscan(e_p, sub_2, st_2)$
$(ignore, sub_4) = unify(sub_3, t_l, t_p)$
$(t_r\ e'_r, sub_5, st_4) = upscan(e_r, sub_4, st_3)$
$(T, sub_6) = unify(sub_5, t_o, t_r)$
in
$(T,$ **let** $e'_p == e'_l$ **in** e'_r **otherwise** $e'_o.T, sub_6, st_4)$

letrec $e_{p1} == e_{l1} \ldots e_{pn} == e_{ln}$ **in** e_r **otherwise** e_o:
similar to **let** *typing all the pattern expressions first so that all new pattern variables are entered into the symbol table before typing any of the left hand side expressions, thus enabling recursive references.*

$\lambda\ e_{p1} \leftarrow e_{p1} \mid \ldots \mid e_{pn} \leftarrow e_m$
let $(t_{p1}, e'_{p1}, sub_1, st_1) = upscan(e_{p1}, substitution, symbolTable)$
...
$(t_{pn}, e'_{pn}, sub_n, st_n) = upscan(e_{pn}, sub_{n-1}, st_{n-1})$
$(T_p, sub_p) = unify(sub_n, t_{p1}, \ldots, t_{pn})$
$(t_{r1}, e'_{r1}, sub_{n+1}, st_{n+1}) = upscan(e_{r1}, sub_p, st_n)$
...
$(t_m, e'_m, sub_{2n}, st_{2n}) = upscan(e_m, sub_{2n-1}, st_{2n-1})$
$(T_r, sub_r) = unify(sub_{2n}, t_{r1}, \ldots, t_m)$
$T = T_p \rightarrow T_r$
in
$(T, \lambda\ e'_{p1} \leftarrow e'_{r1} \mid \ldots \mid e'_{pn} \leftarrow e'_m.T, sub_r, st_{2n})$

In the above sketch details of how overloaded and polymorphic qualified expressions are handled are omitted, to handle these the following additional actions need to be taken:

Polymorphic let/letrec

After typing the pattern/left expression pair, if the type of any variable defined by the pattern contains a type variable which was introduced *during* typing the let/letrec then the pattern variable is polymorphic and all such type variables in its type must be converted to new unique polytypes. For example, typing the expression:

let q == nil **in** ...

where *nil* is the standard empty list constructor would result in *q* being assigned the type φ and the new substitution $\varphi \Rightarrow list(\kappa)$, where φ and κ are newly introduced type variables. To support

polymorphism κ must be converted to a polytype, producing the substitution $\varphi \Rightarrow \text{list}(K)$ where K is a new unique polytype.

Overloaded let/letrec

If after typing the pattern/left expression pair the type of any pattern variable is found to be a multi-type then multiple definitions of the pattern variable must be introduced, one of each type in the set. For example, typing:

let neg == - **in** ...

where '-' is in scope as both the integer and real negation functions, would produce a multi-typing for *neg*. Therefore, to support overloading the expression is transformed into:

let neg == $-_{int \to int}$ **in let** neg == $-_{real \to real}$ **in** ...

by the type-checker and two instances of *neg* are defined.

8.7.4. The Downscan Function

The function *downscan* simply takes the substitution and annotated pattern/expression pair resulting from *upscan/unify* and performs a top down scan replacing any type variables with their value under the substitution, resolves any remaining sets, and resolves any identifier nodes which have be annotated by *upscan* with a set of possible identifiers. Either of the resolutions can fail, if simple identifier resolution fails then best fit resolution [7.1.2] is tried before giving up with a "local ambiguity not resolvable" error. The *downscan* function is simply defined by cases over the target language.

8.8. Conclusion

The algorithm described above has been implemented in Hope+C and forms part of both the Hope+C compiler and the Flagship Programming Environment [Darlington 88, 89].

There are known cases where the overloading complexity is much higher than apparent from the program, and it is thus possible to write seemingly simply programs for which the type-checking time is unexpectedly long. Some, if not all, of the common cases could be handled by special code in the type-checker, but we are reluctant to do this until further research has been undertaken to attempt to discover a more general solution. Also, such programs appear to be rare, so finding a solution in the worst case is of questionable practical advantage.

Finally, the type-checker has been profiled and this has revealed that by far the greatest proportion of time is spent within the module which implements the type substitution. This seems to be due to the substitution being stored as a list which must be scanned linearly to find an entry. We have so far chosen not to pursue any modifications to the substitution data structure to provide improved lookup since by far the greatest reason for the degradation of compiler performance is the large memory usage for large input program sizes, this effects both garbage collection and swapping. Improving performance by tackling this problem would of course benefit all Hope+C programs and not just the compiler. Work is currently in progress, as part of a new research project outside the scope of this thesis, which addresses memory usage and hence the general performance of large Hope+C programs.

9. Conclusions and Further Work

9.1. Overview

In this thesis we have covered a number of topics related to the design and implementation of practical functional programming systems. We have demonstrated the practicality of the techniques and ideas by describing the Hope+C language system which is based upon them.

We presented a new I/O system, result continuations, for pure functional languages and shown that it is both sufficient, in that it can provide all types of I/O facility; correct, in that both inter and intra-request synchronisation is guaranteed; and easy to use. We then described an implementation of a language, Hope+C, based on our approach. This has been used for major programming projects, including its own compiler.

Next we presented an implementation system, Fpm2, based on an enhancement to an existing model. This technique is easy to implement on conventional architectures, upon which it provides comparable or superior performance to other purposely designed functional models.

We then discussed the design of an inter-language calling interface between functional and imperative languages based upon the result continuation mechanism, together with an implementation for Hope+C which provides an interface with C. This has been used extensively, most notably to implement an interface to the standard X-11 Windows package.

Finally we introduced a new type system and associated type-checking algorithm which supports polymorphism, existential quantification, full overloading and type inference. No existing algorithm combines these four properties and is also implementable in a pure functional language with a good time/complexity performance. This algorithm has been implemented as part of the Hope+C compiler.

9.2. Future Work

In the area of I/O further experience is required to determine the set of operations which should be supplied as basic continuations. We propose to further investigate the following topics:

- *General file and terminal operations.*

  The file and terminal operations of Hope+C are basic. For example, there is no provision for directory read and creation operations, or for control over terminal characteristics such as echoing.

- *Complete file system read and/or write operations.*

  This covers two separate possibilities; the provision of a "snapshot" of the file system as a single data structure, so that multiple file input operations need not issue multiple continuations; and the provision of "read file system" and "write file system" operations, probably with the file system being locked from other modification between the two.

- *Typed object I/O.*

  Hope+C will only read and write file of characters, which is rather limiting. The provision of typed I/O to files and possibly typed persistent storage is an area for further research.

In the domain of language implementation further work is possible in a number of areas. For example: the extension of Fpm2 to provide support for destructive update and loops, and work on coarse-grained parallel implementations on multiprocessor machines.

In the Hope+C/C inter-language calling system we have not integrated the I/O systems of the two languages or provided a method for C to call a Hope+C function unless it were passed to the C function as an argument. The integration of I/O is an involved issue with many problems since languages often handle files, signals etc. in radically different ways. The approach we have taken for the Hope+C/C interface is to let the host language, in our case Hope+C, be used for all these operations. Whether a suitable integration could be achieved would require further investigation. Calling general Hope+C functions directly from C is not supported yet but is easily incorporated.

Further theoretical work remains on the extended type system and the type-checking algorithm; we have not proved that a finite set of principal well-typings exist for an expression nor that the type-checking algorithm determines these well-typings.

9.3. Current Research

We are currently working on the development of a concurrent functional language with explicit process control. The aim of this work is to produce a functional language which supports the process/object-oriented paradigms, which have proved both popular and useful design methods, while still maintaining the benefits of functional programming. As part of this work a non-strict implementation of Hope+C is currently under way.

9.4. Conclusion

In this thesis we have shown that functional languages can be efficiently implemented so as to make them practical proposition for problem solving using sequential hardware. By extending them, without compromising their theoretical properties, to provide powerful I/O and inter-language support, we have widened their applicability. Finally, we have extended their already powerful typing systems to include support for overloading and existential quantification.

We believe the results presented in this thesis will contribute to the wider applicability of functional languages, thus bringing their benefits closer for everyday programmers.

10. References

[Appel] *Standard ML Reference Manual (PRELIMINARY)*
Appel, A., and MacQueen, D.B., New Jersey SML Package, Oct 89

[Augustsson 88] *Compiling Lazy Functional Languages* [Johnsson, T.],
Compiling Lazy Functional Languages, Part II [Augustsson, L.]
PhD Theses, Dept of Computer Science, Chalmers University, Göteborg, Sweden

[Augustsson 89] *Lazy ML Users Manual*
Augustsson, L., and Johnsson, T., PMG, Dept of Computer Science, Göteborg, July 89

[Bailey] *FP/M Abstract Syntax Description, Revision 6.0*
Bailey, R., Dept of Computing, Imperial College, Nov 85

[Backus] *FL Language Manual (Preliminary Version)*
Backus, J., Williams, J.H., & Wimmers, E.L., RJ 5339 (54809) 11/7/86 IBM Almaden Research Center

[Burstall] *Hope, An Experimental Applicative Language*
Burstall, R.M., MacQueen, D.B., & Sannella, D.T. , Edinburgh University CSR-62-80

[Cardelli 84] *The Functional Abstract Machine*
Cardelli, L., Bell Laboratories Computer Science Technical Report no. 107, 1984

[Cardelli 85] *On Understanding Types, Data Abstraction, and Polymorphism*
Cardelli, L., and Wegner, P., Computing Surveys, Vol 17, No. 4,pp. 471-522, Dec 85

[Clément] *A simple applicative language: Mini-ML*
Clément, D., Despeyroux, J., Despeyroux, T., and Khan, G., 1986 ACM Symposium on LISP and Functional Programming, p13-27.

[Cousineau] *The Categorical Abstract Machine*
Cousineau, G., Curien, P.L. and Mauny, M., Science of Computer Programming, 8, 1987

[Cupitt] *The Design and Implementation of an Operating System in a Functional Language*
Cupitt, J., PhD Thesis, University of Kent, Oct 1989

[Damas] *Principal Type Schemes for Functional Programs*
Damas, L., and Milner, R., p207, Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Jan 82.

[Darlington 88] *An introduction to the Flagship Programming Environment*
Darlington, J. et al, Proceedings of CONPAR 88, Cambridge University Press.

[Darlington 89] *A Functional Programming Environment Supporting Execution, Parti. Execution and Transformation*
Darlington, J. et al, Proceedings PARLE 89.

[Dwelly 87] *The Functional Specification of Interaction : an interim report*
Dwelly, A., ECRC Technical Report TR-MMI-08, April 87

[Dwelly 88a] *Marit : The Algebra of Dialogues*
Dwelly, A., ECRC Technical Report TR-MMI-11, Feb 88

[Dwelly 88b] *Synchronising the I/O Behaviour of Functional Programs with Feedback*
Dwelly, A., Information Processing Letters, May 88

[Field 85] *Fpm to Vax Vms Macros Compiler & Vax Vms Fpm Runtime System*
written by A. Field, Imperial College 1985

[Fairburn] Design and Implementation of a Simple Typed Language based on the Lambda Calculus
Fairburn, J., TR 75 Computer Laboratory, University of Cambridge May 85.

[Field 88] *Functional Programming*
Field, A.J. and Harrison, P.G., Addison-Wesley 1988, ISBN 0-201-19249-7

[Gabriel] *Performance and Evaluation of Lisp Systems*
Gabriel, R.P., MIT Press 1985, ISBN 0-262-07093-6

[Gordon 88] *Private Conversations*
Implementation of Lazy Functional Languages Workshop, Aspenäs, Sweden, Sept 88.

[Gordon 89] *PFL+: A Kernel Scheme for Functional I/O*
Gordon, A., University of Cambridge Computer Laboratory, March 89

[Henderson] *Purely Functional Operating Systems*
Henderson, P., in "Functional Programming and its applications", eds. Darlington, Henderson & Turner, Cambridge University Press, 1982

[Hudak 88a] *Report on the Functional Programming Language Haskell*
Hudak, P., Wadler, P., *et al.* Dec 88

[Hudak 88b] *On the expressiveness of purely functional I/O systems*
Hudak, P. and Sundaresh, R., Technical Report YALEU/DCS/RR665, Yale University, Department of Computer Science, Dec 88

[Hughes] *Graph Reduction with Supercombinators*
Hughes, J., Technical Monograph PRG-28, Oxford PRG, 1982

[Jones] *Functional Programming and Operating Systems*
Jones, S.B., and Sinclair, A.F., The Computer Journal Vol 32, No 2, April 89.

[Karlsson] *Nebula – A Functional Operating System*
Karlsson, K., Laboratory for Programming Methodology, Dept of Computer Science, Chalmers University, 1981

[Kaes] *Parametric Overloading in Polymorphic Programming Languages*
Kaes, S., pp 131 - 144, ESOP `88, LNCS 300, Springer Verlag, 1988

[Kanellakis] *Polymorphic Unification and ML Typing*
Kanellakis, P.C. and Mitchell, J.C., in Conference Record of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages, Jan 89.

[Leviant] *Polymorphic Type Inference*
Leviant, D., Computer Science Department, Carnigie-Mellon University, 1983

[Mazurkiewicz] *Proving Algorithms by Tail Functions*
Mazurkiewicz, A., Information and Control pp220-226, 18(1970)

[Milner 78] *A Theory of Type Polymorphism in Programming*
JCCS 17, 3, pp 348-375, 1978

[Milner 84] *A Proposal for Standard ML*
1984 ACM Symposium on Lisp and Functional Programming, pp 184-197, Austin, Aug 84

[Milner 89] *A Calculus of Mobile Processes, Parts 1 & 2*
Milner, R., ECS-LFCS-89-85/6, University of Edinburgh

[Mitchell] *Abstract types have existential type*
Mitchell, J.C., and Plotkin, G.D., p37-51, Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, Jan 85.

[Patterson] *Private Conversation*
Patterson, R., Imperial College, Jan 90.

[Perry 87] *Hope+C, A Continuation extension for Hope+*
Perry, N., Dept of Computing, Imperial College, IC/FPR/LANG/2.5.1/21, Nov 87

[Perry 88a] *Hope+*
Perry, N., Dept of Computing, Imperial College, IC/FPR/LANG/2.5.1/7, Issue 5, Feb 88

[Perry 88b] *Hope+ Compiler*
Perry, N. and Sephton, K.M., Dept of Computing, Imperial College, IC/FPR/LANG/2.5.1/14, May 88

[Perry 88c] *Fpm2, A Compilation System for Functional Languages*
Perry, N., Dept of Computing, Imperial College, IC/FPR/LANG/2.5.1/30, Aug 88
and in Proceedings of the Workshop on the Implementation of Lazy Functional Languages, Aspenäs, Sweden, Sept 88.

[Perry 88d] *Functional/Imperative Inter-language Working*
Perry, N., Dept of Computing, Imperial College, IC/FPR/LANG/2.5.1/31, Aug 88

[Perry 88e] *Short Note: Towards a Functional Operating System*
Perry, N., Dept of Computing, Imperial College, IC/FPR/LANG/2.5.1/32, Aug 88

[Perry 89a] *The Hope+C Compilation System*
Perry, N. and Sephton, K.M., Dept of Computing, Imperial College, Mar 89
and in Proceedings of the 1989 Beijing International Symposium for Young Computer Professionals

[Perry 89b] *I/O and Inter-language Calling For Functional Languages*
Perry, N., inProceedings IX International Conference of the Chilean Computer Society and XV Latin American Conference on Informatics, July 89

[Perry 89c] *Functional Coroutines and RPC*
Perry, N., Workshop on Complex Functional Systems: Proceedings, TR-HCI-15, European Computer-Industry Research Centre GmbH, Dec 89

[Perry 90] *A Functional Type Checker Supporting Overloading, Polymorphic Functions, Abstract Data Types and Type Inference*
Perry, N., Dept of Computing, Imperial College. *In preparation.*

[Peyton Jones 87] *The Implementation of Functional Programming Languages*
Peyton Jones, S.L., Prentice-Hall International, London, 87

[Peyton Jones 88] *The Spineless Tagless G-Machine*
Peyton Jones, S.L., in Proceedings of the Workshop on Implementation of Lazy Functional Languages, Aspenäs, Sept 88

[Robinson] *A machine-oriented logic based on the resolution principle*
Robinson, J.A., Journal of the ACM 12(1) pp. 23-49, Jan 1965

[Sephton] *Using "tuples", The Flagship Programming Environment*
Sephton, K.M, Chin, W.N., McLoughlin, L.M.J, Pull, H.M and While, R.L.
Department of Computing, Imperial College, Feb 90

[Steele] *The Revised Report on Scheme, a Dialect of Lisp*
Steele, G.L Jr. and Sussman, G.J., MIT AI Memo 452, Jan 78

[Stoye] *The Implementation of Functional Languages Using Custom Hardware*
Stoye, W., Technical Report 81, Computer Laboratory, University of Cambridge, Dec 85

[Strachey] *Continuations – a Mathematical Semantics for Handling Full Jumps*
Strachey, C. & Wadsworth, C.P., Technical Monograph PRG-11, University of Oxford, 74

[Thompson 86] *Writing Interactive Programs in Miranda*
Thompson, S., Computing Laboratory, University of Kent, Aug 86

[Thompson 87] *Interactive functional programs: a method and a formal semantics*
Thompson, S., Report No. 49, Computing Laboratory, University of Kent, Nov 87

[Turner 79] *A new implementation technique for applicative languages*
Turner, D., Software Practice and Experience, 9, pp31-49, 1979

[Turner 85] *Miranda - a non-strict functional language with polymorphic types*
Turner, D., in Proc. Conference of Functional Programming Languages and Computer Architecture, Nancy, Springer Verlag LNCS 201, 85

[Turner 87] *Functional Programming and Communicating Processes*
Turner, D.A., Conference proceedings for PARLE, June 87

[Van Wijngaarden] *Report on the algorithmic language Algol 68*
van Wijngaarden *et al*, Numer. Math. 14, 79-218

[Wadler] *How to make ad hoc polymorphism less ad hoc*
Wadler, P. and Blott, S., Proceedings of 16th ACM Symposium on Principles of Programming Languages, Jan 89

[Wray] *Non-Strict Languages – Programming and Implementation*
Wray, S.c. and Fairburn, J., The Computer Journal, Vol 32, No 2, 1989

Appendix 1: Hope+C Continuations

The following is a complete list of continuations currently defined for Hope+C. Currently, as of Hope+C release 3.1, not all these continuations are implemented – this is due to the allocation of time rather than any implementation problems. This appendix is abridged from [Perry 87].

Status Return

Most continuation functions are passed at least one value of type `Status`. This value is supplied by the system and reflects the success or failure of the (previous) requested operation. If the operation failed an integer error code indicating the reason for failure is returned as part of this value. A list of all error codes and their meanings is supplied along with the Hope+C compiler, in the descriptions below symbolic codes are used instead of integers,e.g. EOF for end of file encountered.

Program Termination

`Exit(num)`

The system returns to the system command level, passing back the given integer value as the exit status of the program. By convention a value of zero implies successful completion, any other value an error.

Input/Output Continuations

I/O is performed over channels, which are represented by the data type:

```
data channel == ...
```

The internal representation of a channel is hidden to the programmer. There are three pre–defined channels; `stdin`, `stdout` and `stderr`, the following functions (from module Cx) may be used to obtain their values:

```
dec stdin : channel;
dec stdout : channel;
dec stderr : channel;
```

Unless altered at the Unix™ command level (or in future by command to the programming system), `stdin` is connected to the user's keyboard, `stdout` to the user's screen and `stderr` also to the screen – this channel being intended for error messages. Other channels may be

opened to communicate with files, other terminals and other programs (see *Process Handling* below).

Only values may be output on channels, if a suspension is passed to any of these continuations it is first evaluated and the resultant value used. (Note: if a lazy object is supplied then each element is evaluated in turn, used, and then discarded.)

Accessing Files

To specify the type of channel required (`OpenChannel`) or the type of access mode allowed (`CheckFile`) the following data type is used:

```
data AccessType ==
        Read          ! input or read access !
    ++  Write         ! output or write access !
    ++  Append        ! write append access !
    ++  Bidirectional ! input/output or read/write !
    ++  Execute       ! file may 'executed' !
                      ! (see "Process Handling" below) !
```

The following three continuations allow files to be opened/closed and connected to channels, and to check the possibility of doing so:

`OpenChannel(list(char) # AccessType # (Status # channel -> Result))`
The system will attempt to open the file (or terminal) represented (as a Unix™ pathname) by the first argument. The channel is opened for input or output depending on the value of the second argument If the file does not exist and write or append access is requested then it is created.

`CloseChannel(channel # (Status -> Result))`
The given channel is closed, no further I/O may be accomplished on a closed channel.

`CheckFile(list(char) # AccessType # (Status -> Result))`
This system checks whether the access mode requested by the second argument would be possible on the file represented by the first.

Character I/O

`GetChar(Status # char -> Result)`
The system will read a single character from `stdin`. If an error occurs then `chr(0)` is returned as the character value.

`GetLine(Status # list(char) -> Result)`
The system will read a sequence of characters up to and including a newline from `stdin`. If an error occurs then nil is returned.

`PutChar(char # (Status -> Result))`
The system will write the given character to `stdout`.

`PutStr(list(char) # (Status -> Result))`
The system will write the given character list to `stdout`.

`PutJoin(list(list(char)) # (Status -> Result))`

The system will write the concatenation of the given character lists to `stdout`.

```
PutEChar(char # (Status -> Result))
PutEStr(list(char) # (Status -> Result))
PutEJoin(list(list(char)) # (Status -> Result))
```

These three continuations perform the same operations as the previous three, except that output is written to the `stderr`.

```
GetChar(channel # (Status # char -> Result))
GetLine(channel # (Status # list(char) -> Result))
PutChar(channel # char # (Status -> Result))
PutStr(channel # list(char) # (Status -> Result))
PutJoin(channel # list(list(char)) # (Status -> Continuation))
```

These five perform the same operations as their namesakes described above, except using the given channel. The channel must be of the appropriate type; read for get*X* and write for put*X* operations.

Object Level I/O

These continuations provide "object" level I/O, which is more akin to what might be provided by a persistent object store – though here only one data type is supported. They also provide a similar functionality to "streams". Once an object I/O continuation has been performed on a channel any subsequent I/O continuation will fail, as the channel will have been closed.

`GetObj(list(char) # (Status # list(char) -> Result))`

The system attempts to access for reading the file whose pathname is given by the first list. If successful the continuation function is passed a list representing the contents of the file. If unsuccessful for any reason, including a non–existent file, then the value `nil` is passed to the continuation.

`GetObj(channel # (Status # list(char) -> Result))`

Any remaining input available from the given channel is returned. The channel is then closed.

`PutObj(list(char) # list(char) # (Status -> Result))`

The system creates, or truncates, the file whose pathname is given by the first list. The value of the second list is then assigned to the file.

`PutObj(channel # list(char) # (Status -> Result))`

The second argument is written to the given channel and then the channel is closed. The same functionality is provided by a `PutStr` followed by a `CloseChannel`.

Process Handling

Processes created by these continuations are identified by a value of type `processID`:

```
data processID == …; ! internal structure hidden !
```

The `processID` may be used to send a signal to the created process (see *Signals* below). A `processID` is also returned by the `WaitProcess` continuation – see below. A service is identified by a value of type `serverID`:

```
data serverID == …; ! internal structure hidden !
```

Simple Filter Processes

```
RunCommand(list(char) # list(list(char)) # list(char)
                                    # (list(char) -> Result))
```

The first list specifies the pathname of the program to run. The list of lists is the arguments to be passed to the program, by convention the first should be the program name. The third argument is passed as the standard input to the initiated program. The system then evaluates the continuation function passing it a list formed from the standard output of the program. If any errors occur in invoking the program then the value nil is passed to the continuation function – note that this is indistinguishable from the program writing nothing to its standard output. If the program outputs anything to its standard error it will be sent to the current process's standard error – usually the terminal screen.

General Slave Processes

```
StartProcess(list(char) # list(list(char)) #
                              (Status # processID # channel -> Result))
```

The first list specifies the pathname of the program to run. The list of lists is the arguments to be passed to the process, by (Unix™) convention the first should be the process name. If the process is initiated successfully a bidirectional channel connected to the process's standard input and output and a `processID` are returned. The new process will inherit the current process's standard error channel.

```
StartFunProcess(Result # (Status # processID # channel -> Result))
```

The first argument specifies a continuation function to run as a separate process. If the process is initiated successfully a bidirectional channel connected to the new process's standard input and output and a `processID` are returned. The new process will inherit the current process's standard error channel. The new process's function is evaluated in the same environment as that of an initial main function – extended with any values supplied as part of a closure making up the first argument to `StartFunProcess`.

```
SpawnProcess(list(char) # list(list(char)) # channel # channel #
                              channel # (Status # processID -> Result))
```

Similar to `StartProcess`, except that no channel is opened to the new process which instead is given the three channel arguments as its standard input, output and error channels respectively.

```
SpawnFunProcess(Result # channel # channel # channel #
                                    (Status # processID -> Result))
```

Similar to `StartFunProcess`, except that no channel is opened to the new process which instead is given the three channel arguments as its standard input, output and error channels respectively.

```
WaitProcess(Status # processID # num -> Result)
```

The current process is halted until any one of its created process' terminates. When a process terminates its `processID` and exit status (supplied to that processes' `Exit` continuation if it was also a Hope+C process) are returned.

Server Processes

```
RequestService(list(char) # (Status # channel -> Result))
```

An attempt is made to establish communication with the server process specified by the first argument. If connection is successful a bidirectional channel connected to the server process is returned.

```
AdvertiseService(list(char) # (Status # serviceID -> Result))
```

A service with the name given by the first argument is established. If the service is successfully established a `serviceID` is returned.

`AcceptRequest(serviceID # (Status # channel -> Result))`
The program is halted until another process requests to communicate with the service identified by the first argument (if the other process is a Hope+C process it will do this with the `RequestService` continuation, described above). When a request is made a bidirectional channel to the requesting process is returned.

`RequestPending(serviceID # (Status # truval -> Result))`
Returns true if there are any outstanding requests for the given service, false otherwise. This allows a process to poll its pending requests without blocking as in `AcceptRequest`.

`WithdrawService(serviceID # (Status -> Result))`
The service identified by the first argument is withdrawn. Withdrawing a service does not affect any requests already granted and still in progress.

Signal Handling

Signals are represented by the following data type:

```
data SignalType == InterruptSignal
                ++ QuitSignal
                ++ KillSignal
                ++ UserSignal(num);
```

`OnSignal(SignalType # Result # Result)`
The system remembers the first continuation for later evaluation if a signal of the type given by the first argument occurs. The system then evaluates the second continuation.

`SendSignal(SignalType)`
The system generates the given signal and sends it to the current process. No continuation function is specified as either a continuation previously specified by `OnSignal` is evaluated, or a suitable message is issued and the process terminated as though an `Exit` continuation had been evaluated.

`SignalProcess(processID # SignalType # (Status -> Result))`
The system sends the given signal to the process identified by the first argument.

Miscellaneous

`MayRead(list(channel) # (list(truval) -> Result))`
In the case where a program has many channels open; either from files, devices or processes; it may be desirable to know which channels have data present and ready to be read – as the normal I/O continuations will block until there is data. This continuation takes a list of channels and for each one returns `true` if there is data present to be read; or `false` if the channel is not open for reading, is open for object I/O, or has no data waiting.

`MayWrite(list(channel) # (list(truval) -> Result))`
This is similar to `MayRead`, but checks that the given channel is ready to accept data.

`GetRandom(real -> Result)`
The system calculates a pseudo–random number in the range [0.0, 1.0) and passes it to the continuation function.

`SetRandomSeed(num # Result)`
The system uses the integer value to initialise its pseudo–random number generator. The system then evaluates the given continuation.

`GetDate(num -> Result)`
The SI day number of today is returned.

`GetTime(num -> Result)`
The number of seconds since midnight is returned.

Appendix 2: Example Hope+C Programs

A2.1 Example Programs: "wc"

These programs implement the "wc" (word count) utility of Unix. Two versions are first presented for comparison, the first treats a file as a stream/list and uses the *GetObj* continuation – this is similar to how a stream I/O program may be written. The second uses a character based view and the continuation *GetChar* – this is typical of how the problem would be solved in an imperative language. To highlight the continuations they have been printed in bold.

A2.1.1 Version 1: "Stream/List"

```
uses StandardEnviron, StandardLists, CxXfer, StandardXfer;

type string == list(char);
type num3 == num # num # num;

! count chars, words, lines in list !
dec wc : string -> num3;
dec wc2 : string # truval # num3 -> num3;

--- wc(file) <= wc2(file, false, (0,0,0));

--- wc2(nil, _, totals) <= totals;                          ! end of input!
--- wc2('\n'::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl+1));                       ! end of line !
--- wc2('\t'::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl));                             ! gap !
--- wc2(' '::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl));                             ! gap !
! other = word character, increment word count if not in word !
--- wc2(_::t, inword, (tc,tw,tl))
    <= wc2(t, true, (tc+1, (if inword then tw else tw+1), tl));

! now the control functions to process the input files one at a time !
dec scan_file : Status # string # string # list(string) # num # num3
          -> Result;
dec scan_files : Status # list(string) # num # num3 -> Result;

! process one file and print stats !
--- scan_file(Ok, file, name, rest, numdone, (tc,tw,tl))
    <= let (nc,nw,nl) == wc(file)
       in Printf("%d %d %d %s\n",
                 [Unum(nl),Unum(nw),Unum(nc), Ustr(name)],
                 scan_files(?, rest, numdone+1, (tc+nc,tw+nw,tl+nl)));
--- scan_file(_, _, name, rest, numdone, totals)
    <= Printf("Cannot open file %s\n", [Ustr(name)],
              scan_files(?, rest, numdone, totals));
```

```
! process list of files, empty name implies stdin !
--- scan_files(Ok, next::rest, numdone, totals)
    <= if next = nil ! read stdin !
       then GetObj(stdin,
                   scan_file(?,?,"<Standard Input>",
                             rest,numdone,totals))
       else GetObj(next, scan_file(?,?,next,rest,numdone,totals));
! all files processed, if more than one print totals !
--- scan_files(Ok, nil, numdone, (tc,tw,tl))
    <= if numdone > 1
       then Printf("%d %d %d total\n", [Unum(tl),Unum(tw),Unum(tc)],
                   TheEnd)
       else TheEnd(Ok);
! handle error !
--- scan_files(_, _, _, _)
    <= PutEStr("Error while writing to stdout, aborted\n", TheEnd);

! the main program, get arguments and call scan_files() !
--- main <= scan_files(Ok,              ! prime status !
                       if getargcount = 0
                       then [nil]     ! just stdin !
                       else getargl,  ! list of file names !
                       0,             ! = number of files processed !
                       (0,0,0)        ! zero totals !
                      );
```

Notes:

- *Printf* is not basic continuation but a function provided by the module CxXfer. It uses basic continuations such as *PutChar* to perform the actual I/O. This is an example of how easily "higher level" continuations may be built.
- *TheEnd* is a standard function whose definition is:

  ```
  dec TheEnd : alpha -> Result;
  --- TheEnd(_) <= Exit(0);
  ```

  and is a convenient way to terminate a program.
- *GetObj* returns the contents of a file (or stream, as for stdin) as a list of characters.
- Question marks, as in `scan_files(?, rest, numdone, totals)`, are Hope+C's partial application syntax.

A2.1.2 Version 2: "Character"

```
uses StandardEnviron, StandardLists, CxXfer, StandardXfer;

type string == list(char);
type num3 == num # num # num;

! count chars, words, lines in list !
dec wc : Status # char # channel # truval # num3 #
        (list(string) # num # num3) -> Result;

! now the control functions to process the input files one at a time !
dec scan_file : Status # channel # string #
                list(string) # num # num3 -> Result;
dec scan_files : Status # list(string) # num # num3 -> Result;

--- wc(Fail(_), _, chan, (c,w,l),
      (rest,done,(tc,tw,tl)))                          ! end of input !
    <= Printf("%d %d %d %s\n", [Unum(l),Unum(w),Unum(c),Ustr(name)],
                  scan_files(?, rest, done+1, (tc+c,tw+w,tl+l)));
--- wc(Ok, inp, chan, inword, (c,w,l), state)
    <= let tot == (lambda    ! "case" statement !
            '\n' =>   (c+1,w,l+1)                          ! end of line !
          | '\t' =>   (c+1,w,l)                                  ! gap !
          |  ' ' =>   (c+1,w,l)                                  ! gap !
          ! other = word char, increment word count if not in word !
          |   _ =>    (c+1,(if inword then w else w+1),l) )(inp)
       in GetChar(chan,
          wc(?, ?, chan, false, tot, state);

--- scan_file(Ok, chan, _, r, n, t)
    <= GetChar(chan, wc(?, ?, chan, false, (0,0,0), (r, n, t)));
--- scan_file(_, _, name, r, n, t)
    <= Printf("Cannot open file %s\n", [Ustr(name)],
                  scan_files(?, r, n, t));

! process list of files, empty name implies stdin !
--- scan_files(Ok, next::rest, numdone, totals)
    <= if next = nil ! read stdin !
       then wc(Ok,stdin,"<Standard Input>",rest,numdone,totals))
       else OpenChannel(next, Read,
                        wc(?,?,next,rest,numdone,totals));
! all files processed, if more than one print totals !
--- scan_files(Ok, nil, numdone, (tc,tw,tl))
    <= if numdone > 1
       then Printf("%d %d %d total\n", [Unum(tl),Unum(tw),Unum(tc)],
                   TheEnd)
        else TheEnd(Ok);
! handle error !
--- scan_files(_, _, _, _)
    <= PutEStr("Error while writing to stdout, aborted\n", TheEnd);

! the main program, get arguments and call scan_files() !
--- main <= scan_files(Ok,           ! prime status !
                   if getargcount = 0
                   then [nil]    ! just stdin !
                   else getargl, ! list of file names !
                   0,            ! = number of files processed !
                   (0,0,0)       ! zero totals !
                  );
```

Notes:

- *GetChar* inputs a single character from the given file.
- The "lambda" expression is used to emulate a "case" expression – which Hope+C does not provide directly.

A2.2. Higher Order Examples

A2.2.1 Higher Order Continuations Module

This module *CxHO* defines the common higher order functions *compose*, *reduce* and *map*. Also included is a simple utility function, %, which forms a CPS function from a non-CPS one. The definitions for the functions *compose* and *reduce* are very similar to their non-CPS counterparts. However non-CPS non-strict *map* would return having just constructed the first *cons* cell – without even evaluating the head. The CPS version presented differs as it processes the whole list before returning a value – this results in at least the spine of the result being constructed though the heads need not be evaluated.

```
module CxHO;
uses Cx;

pubfun compose, reduce, map, %;

typevar gamma;

! list reverse, used by map !
dec reverse : list(alpha) -> list(alpha);
dec rev2: list(alpha) # list(alpha) -> list(alpha);
--- reverse(nil) <= nil;
--- reverse(h::t) <= rev2(t, [h]);
--- rev2(nil, l) <= l;
--- rev2(h::t, l) <= rev2(t, h::l);

! compose: form the composition of two CPS functions !
infixrl compose : 1;
dec compose : (beta # (gamma -> Result) -> Result)
            # (alpha # (beta -> Result) -> Result)
           -> (alpha # (gamma -> Result) -> Result);
--- f compose g <= lambda (x, c) => g(x, lambda b => f(b, c) end) end;

! reduce: CPS left reduce/foldl !
dec reduce : (alpha # beta # (beta -> Result) -> Result)
           # beta # list(alpha) # (beta -> Result)
          -> Result;
--- reduce(_, r, nil, c) <= c(r);
--- reduce(f, r, h::t, c) <= f(h, r, reduce(f, ?, t, c));
```

```
! map: CPS apply function over list !
dec map : (alpha # (beta -> Result) -> Result)
        # list(alpha) # (list(beta) -> Result) -> Result;
! auxiliary function !
dec map_1 : (alpha # (beta -> Result) -> Result)
          # list(alpha) # list(beta) # (list(beta) -> Result)
          -> Result;
--- map(_, nil, c) <= c(nil);
--- map(f, h::t, c) <= f(h, lambda nh => map_1(f, t, [nh], c) end);
--- map_1(_, nil, r, c) <= c( reverse(r) );
--- map_1(f, h::t, r, c) <= f(h, lambda nh => map_1(f,t,nh::r,c) end);

! %: convert non-CPS function to CPS !
dec % : (alpha -> beta) -> (alpha # (beta -> Result)) -> Result;
--- %f <= lambda (x, c) => c(f(x)) end;

end;
```

Finally here are two more versions, the first is written using CPS for all major functions, the second only uses CPS for functions directly involved in I/O.

A2.2.2. Version 3: "CPS"

```
uses StandardEnviron, StandardLists, CxXfer, StandardXfer, CxHO;

type string == list(char);
type num3 == num # num # num;
! sum type used so we can handle errors easily !
data Result == NotFound(string) ++ Contents(string # string)
            ++ Count(string # num3);

! read a file !
dec rfile : string # (Result -> Result) -> Result;
--- rfile(name, c)
 <= ( if name = nil ! stdin !
      then GetObj(stdin, c_1)
      else GetObj(name, c_1)
    ) where c_1 == lambda
                     (Ok, val) => c( Contents(name, val) )
                   | (Failed(_), _) => c( NotFound(name) )
                   end;
! do the word count, note this is NOT a CPS function !
dec wc : Result -> Result;
! auxiliary function !
dec wc2 : string # truval # num3 -> num3;
--- wc(Contents(name,  le))
 <= Count(name, wc2( le, false, (0,0,0)));
--- wc(r & NotFound(_)) <= r;
--- wc2(nil, _, totals) <= totals;     ! end of input!
--- wc2('\n'::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl+1)); ! end of line !
--- wc2('\t'::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl));    ! gap !
--- wc2(' '::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl));    ! gap !
! other = word character, increment word count if not in word !
--- wc2(_::t, inword, (tc,tw,tl))
 <= wc2(t, true, (tc+1, (if inword then tw else tw+1), tl));
```

```
! print the individual counts and total them !
dec ptot : Result # (num # num3) # ((num # num3) -> Result)
        -> Result;
--- ptot(NotFound(f), totals, c)
 <= Printf("Could not open %s\n", [Ustr(f)],
          lambda _ => c(totals) end);
--- ptot(Count(f, (nc, nw, nl)), (nfile, (tc, tw, tl)), c)
 <= Printf("%d %d %d %s\n", [Unum(nl), Unum(nw), Unum(nc), Ustr(f)],
          lambda _ => c(nfile+1, (nc+tc, nw+tw, nl+tl)) end);
! print a total if required !
dec tprint : (num # num3) # (Status -> Result
        -> Result;
--- tprint((nfile, (tc, tw, tl)), c)
 <= if nfile > 1
    then Printf("%d %d %d Total\n", [Unum(tl), Unum(tw), Unum(tc)], c)
    else c(Ok);
! the main program !
--- main
 <= (tprint compose (reduce(ptot, (0,(0,0,0)), ?, ?)
                    compose map((%wc) compose rfile, ?, ?))
    )(if getargcount = 0 then [nil] else getargl, TheEnd)
```

A2.2.3 Version 4: "non–CPS wc/format"

In this version only the actual functions which do I/O are coded in CPS, the computational core is coded in normal higher order functional style.

This approach does suffer from a number of potential drawbacks; firstly depending on the strictness of *GetObj* either a large number of files may be open simultaneously or a large number of files may be "in core" at the same time. Secondly all of the output is constructed before any actual printing is done – this comes from the nature of *reduce_list*. These kind of issues are often ignored in functional programming, but their impact must be considered when writing practical programs. A "read file system" continuation could be used in this example to address the problem caused by *GetObj*, but whether this is the best solution is a topic for further research.

```
uses StandardEnviron, StandardLists, CxXfer, StandardXfer, CxHO;

type string == list(char);
type num3 == num # num # num;

! sum type used so we can handle errors easily !
data Result == NotFound(string) ++ Contents(string # string)
            ++ Count(string # num3);

! infix function compose !
typevar gamma;
dec o : (beta -> gamma) # (alpha -> beta) -> (alpha -> gamma);
infixrl o : 1;
--- f o g <= lambda x => f(g(x)) end;
```

```
! read a file !
dec rfile : string # (Result -> Result) -> Result;
--- rfile(name, c)
 <= ( if name = nil ! stdin !
      then GetObj(stdin, c_1)
      else GetObj(name, c_1)
    ) where c_1 == lambda
                     (Ok, val) => c( Contents(name, val) )
                   | (Failed(_), _) => c( NotFound(name) )
                   end

! do the word count !
dec wc : Result -> Result;
! auxiliary function !
dec wc2 : string # truval # num3 -> num3;
--- wc(Contents(name, file))
 <= Count(name, wc2(file, false, (0,0,0)));
--- wc(r & NotFound(_)) <= r;
--- wc2(nil, _, totals) <= totals;     ! end of input!
--- wc2('\n'::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl+1)); ! end of line !
--- wc2('\t'::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl));   ! gap !
--- wc2(' '::t, _, (tc,tw,tl))
    <= wc2(t, false, (tc+1,tw,tl));   ! gap !
! other = word character, increment word count if not in word !
--- wc2(_::t, inword, (tc,tw,tl))
    <= wc2(t, true, (tc+1, (if inword then tw else tw+1), tl));

! format the individual counts and total them !
dec ptot : Result # (string # num # num3) -> (string # num # num3);
--- ptot(NotFound(f), (out, nfile, totals))
 <= (out <> printf("Could not open %s\n", [Ustr(f)]), nfile, totals);
--- ptot(Count(f, (nc, nw, nl)), (out, nfile, (tc, tw, tl)))
 <= (out <> printf("%d %d %d %s\n",
                   [Unum(nl),Unum(nw),Unum(nc),Ustr(f)]),
     nfile+1, (nc+tc, nw+tw, nl+tl)
    )

! add a total if required !
dec addtotal : (string # num # num3) -> string;
--- addtotal((out, nfile, (tc, tw, tl)))
 <= if nfile > 1
    then out <> printf("%d %d %d Total\n",
                       [Unum(tl), Unum(tw), Unum(tc)])
    else out;

! the main program !
--- main
 <= (PutStr
     compose %(addtotal
               o reduce_list(ptot, (?, ("", 0, (0,0,0))))
               o map_list(wc, ?))
     compose map(rfile, ?, ?)
    )(if getargcount = 0 then [nil] else getargl, TheEnd);
```

Notes

- The functions *map_list* and *reduce_list* are from module StandardLists.
- The function *printf* is from module StandardXfer.

Appendix 3: Fpm2 Details

A3.1. The Fpm/Am Opcodes

The Fpm/AM opcodes are grouped as follows: heap allocation, data movement and construction, arithmetic/logical, control, function definition/call/return and miscellaneous. Each group will be covered in turn:

Heap Allocation

claim <number of cells>

Allocate space for a heap object. The heap pointer will be set so that the object may be constructed using pusheap addressing.

Data Movement and Construction

movehp <dest>

Move the current value of the heap pointer to <dest>. Used after a heap object has been created to obtain a pointer to it.

move <src>, <dest>

Move the source value to the destination.

cons <head src>, <tail src>, <dest>

Construct a list cell on the heap and place a pointer to it in <dest>.

cons <head src>, <tail src and dest>

Two operand version of *cons*.

index <src>, <offset>, <dest>

Extract the <offset>th element of <src> and place it in <dest>. If the element is a suspension then evaluate it first.

head <src>, <dest>

Extract the head of a list cell. If the head is a suspension evaluate it first.

tail <src>, <dest>

Extract the tail of a list cell. If the tail is a suspension then evaluate it first.

lindex <src>, <offset>, <dest>

lhead <src>, <dest>

ltail <src>, <dest>

As *index*, *head* and *tail* respectively, but do not evaluate any suspensions.

getvalue <src>, <dest>

As *move* above, but evaluate <src> if it is a suspension.

Arithmetic and Logical

A large number of basic arithmetic and logical instructions are defined. Many of these are used to implement built-in functions in one instruction, and the functional form is also available for use when constructing closures [A3.2].

<op> $<src_1>$, $<src_2>$, <dest>

Evaluate integer <dest> ← $<src_2>$ <op> $<src_1>$

where <op> is one of: add, sub, mul, div, and, or, divmod.

cmp<op> $<src_1>$, $<src_2>$, <dest>

cmpR<op> $<src_1>$, $<src_2>$, <dest>

Evaluate boolean <dest> ← $<src_2>$ <op> $<src_1>$

where <op> is one of: lt, le, eq, ne, ge, gt.

These instructions apply to integer and real values respectively.

<op>IR $<src_1>$, $<src_2>$, <dest> [int × real → real]

<op>RI $<src_1>$, $<src_2>$, <dest> [real × int → real]

<op>RR $<src_1>$, $<src_2>$, <dest> [real × real → real]

Evaluate <dest> ← $<src_2>$ <op> $<src_1>$

where <op> is one of: add, sub, mul, div

<op> <src>, <dest>

Evaluate <dest> ← <dest> <op> <src>

where <op> is any one of above.

not <src>, dest>

Logical inversion.

neg <src>, dest>

Integer negation.

negR <src>, dest>

Real negation.

chr <src>, <dest>

ord <src>, <dest>

Convert character to integer and vice-versa. (Note: these are equivalent to move in the current Fpm/CM.)

Control

This group covers labels and jumps, both unconditional and conditional.

label <name>

Define a label.

jump <label>

Unconditionally jump to label.

jump<op> <src_1>, <src_2>, <label>

Jump to label is <src1> <op> <src2> is true,

where <op> is one of: lt, le, eq, ne, ge, gt.

case <src>, <max>, <name>

addr <label>

The *case* instruction jumps to the n^{th} label from the table of labels starting at <name>. Entries in this table are generated using the *addr* instruction, *n* is calculated as follows:

$$n \leftarrow \begin{cases} \text{<src>}, & \text{if } 0 \le \text{<src>} \le \text{<max>} \\ \text{<max>} + 1, & \text{otherwise.} \end{cases}$$

jump<val> <src>, <label>

Jump to <label> if <src> = <val>,

where <val> is one of: true, false, empty (list).

fun <name>,<number of arguments>

Start definition of function <name> which takes <number of arguments>. When called the arguments will be on the stack, unless the function is a closure, in which case some arguments will be on the stack and those bound when the closure was constructed will be in the heap accessed using the argument pointer.

sfun <name>,<number of arguments>

Start definition of function <name> which takes <number of arguments>. When called the arguments will be in the heap and must be referenced using the argument pointer. This is the type of definition which is generated for functions which will form suspensions. When a suspension is evaluated the argument pointer contains the address the suspension structure in the heap so that arguments may be accessed relative to it.

ifun <name>, <number of arguments>

Identical to an *sfun*, except that this function evaluates directly to the call of another function. The Fpm/CM may take advantage of this and produce better code for this special case.

retn <nlv>, <narg>

sretn <nlv>, <narg>

Return from a *fun* and *sfun* respectively. The <nlv> is the number of local values left on the stack which must be discarded, and the <narg> is the number of arguments the function was called with, which in the case of *fun* defined functions, must also be discarded.

call <name>, <narg>

Call the function <name> passing it <narg> argument values on the stack. On return the arguments will have been removed and the result will be in the work register.

apply <src>, <narg>

Call the function represented by the closure <src>, passing it <narg> argument values on the stack. On return the arguments will have been removed and the result will be in the work register.

rcall <name>, <narg$_1$>, <nlv>, <narg$_2$>

This is equivalent to:

```
call <name>, <narg1>
retn <nlv>, <narg2>
```

except that the Fpm/CM may optimize stack usage.

srcall <name>, <narg$_1$>, <nlv>, <narg$_2$>

ircall <name>, <narg$_1$>, <nlv>, <narg$_2$>

To *sfun* and *ifun* what *rcall* is to *fun*.

Miscellaneous

The following instructions perform various "miscellaneous" tasks.

equate <name>, <external name>

Equate the local function <name> with the global <external name>. Used for all exported functions.

dataaddr <src>

dataval <src>

Generate a constant at the current location whose value is an object pointer or scalar (includes reals) respectively.

dataequ <label$_1$>, <label$_2$>

Set <label$_2$> to a value which when used as an object pointer addresses the data stored at <label$_1$>.

The previous three instructions, together with the label instruction are used for compiling constant list objects into the code.

A3.2. Fpm/AM Built-in Functions

This section lists all the Fpm/AM built-in functions. Most of these functions are also available as Fpm/AM instructions[A3.1] and the function form is only used when closures are required. Those functions flagged by † are only available as functions, those flagged by ‡ are only available as instructions; these latter ones are only used by the Hope+C to Fpm+ compiler and are

not accessible to the Hope+C programmer and their closures are never required. These functions are coded in Fpm+ by the use of small integers, these have been omitted from this section for clarity, but may be found in [Perry 88c] from which this list is abbreviated. The functions have been divided up into functional groups, only those whose names are unobvious or which do not appear as Fpm/AM instructions are explained.

Basic Arithmetic

plus, minus, times, div, mod, divmod, plusRR, plusIR, plusRI, minusRR, minusIR, minusRI, timesRR, timesIR, timesRI, overRR, overIR, overRI, overII,

powerRR†, powerIR†, powerRI†, powerII† (exponentiation),

float†, floor†, ceil†, round†, trunc†

Boolean

and, or, not

Comparison

equal, notequal, greater, less, grequal, lsequal, reallessthan, reallessequal, realgreaterthan, realgreaterequal, realunaryminus,

issame, isdiff (structural equivalence/non-equivalence)

List Operations

empty (return empty list), cons, head, tail, append†, select‡ (select n^{th} element), length†, isempty

Character Conversion

chr, ord

Miscellaneous

void (return void value), error‡ (abort program), nomatch‡ (report pattern matching failure)

Tuple Operations

index‡

Vector Operations

vectorsize†, vindex†, listtovector†

Apply Closure

apply‡

A3.3. The Imperative Interface

This appendix details all the types and functions provided as part of Fpm2 for high-level inter-language calling between Hope+C and C, or other "standard calling sequence" languages.

Cells

All values are stored in objects of type `Cell`:

```
typedef … Cell;
```

`Cell`s either contain a basic value, or a *reference* to a structured value. The system keeps track of all allocated Cells and if a garbage collection occurs maintains any references stored in `Cell`s appropriately. For more details see "Memory Management" below.

The actual representation of a `Cell` is unimportant to the high-level user, only that they may be passed as value arguments to functions and returned as function results (which is not true of every type in C).

Memory Management

Unlike the functional programmer, imperative programmers must handle the management of dynamic memory structures. This task is not trivial, but with a little care can be dealt with easily. The following rules must be remembered:

i) For every structured value returned of type Cell an internal object is constructed to keep track of that value, i.e. in case garbage collection should relocate it. Overwriting a Cell using the C assignment statement does not destroy the internal object, and the structured value will continue to occupy dynamic memory.

ii) To avoid the accumulation of a large number of unwanted objects, whenever an imperative function called from Hope+C returns, all the internal objects constructed during the evaluation of that function are reclaimed.

Given these two rules, imperative functions which only manipulate small data structures and which do not require to maintain global values between invocations, may be written directly without *any* explicit memory management. For other functions which either manipulate large amounts of data – which includes such common functions as "maplist" – or which require to store global values, the following facilities are provided:

i) The explicit marking of objects as "global". Values referenced by global objects are never reclaimed by garbage collection. Any object, local or global, may be freed by program action. Combining these two facilities allows objects to preserved between separate invocations of CallImp, and to be released when no longer required.

ii) All the basic selection functions (see below) have variations which overwrite the object identified by the Cell value instead of constructing a new object to hold the selection.

Object control

```
extern Cell MakeGlobal();        /* Cell MakeGlobal(c) Cell c; */
```
Mark the object referenced as global. Returns the Cell.

```
extern void FreeCell();          /* void FreeGlobal(c) Cell c; */
```
Mark the object referenced as free. Warning: further references to the object will not be caught as invalid.

Accessing Object Values

The following functions allow the extraction of a basic value out of a object, or of the selection of an element of a structured value referenced by a object. Those functions which return values of type `Cell` return references to *new* objects.

```
extern Cell Head();              /* Cell Head(cons) Cell cons; */
```
Return head of list cell.

```
extern Cell Tail();              /* Cell Tail(cons) Cell cons; */
```
Return tail of list cell.

```
extern Cell Index();             /* Cell Index(tuple, elem)
                                    Cell tuple; long elem */
```
Select element of tuple, tuple size cell is index 0.

```
extern long NumVal();            /* long NumVal(c); Cell c; */
```
Return integer value of object.

```
extern char CharVal();           /* char CharVal(c); Cell c; */
```
Return character value of object.

```
extern float RealVal();          /* float RealVal(c); Cell c; */
```
Return real value of object.

```
extern Cell Fpm_Nil;             /* the nil value */
```
The nil list value.

Constructing Objects

The following functions all construct *new* objects.

```
extern Cell Cons();          /* Cell Cons(head,tail) Cell head,tail; */
```
Construct a list cons structure.

```
extern Cell Tuple();         /* Cell Tuple(size, elem1, ..., elemsize)
                                long size; Cell elem1, ... elemsize; */
```
Construct a tuple, takes a variable number of arguments.

```
extern Cell MakeNum();       /* Cell MakeNum(i) long i; */
```
Construct an integer object.

```
extern Cell MakeChar();      /* Cell MakeChar(c); char c; */
```
Construct a character object.

```
extern Cell MakeReal();      /* Cell MakeReal(f); float f; */
```
Construct a real valued object.

In-place Update and Construction

These calls all cause an object to be *updated* to contain a new value. This minimises the number of intermediate objects constructed during a computation, and is essential when a large number of new objects would otherwise be generated, e.g. when traversing a large structure.

```
extern Cell RB_Head();       /* Cell RB_Head(cons) Cell cons; */
```
Update an object from referring to a list cons structure to refer to the head of the list cons structure. Return the head.

```
extern Cell RB_Tail();       /* Cell RB_Tail(cons) Cell cons; */
```
As above, but for tail.

```
extern Cell RB_Cons();       /*Cell RB_Cons(head,tail) Cell head,tail*/
```
Construct a new list cons structure containing the head and tail and then overwrite the tail object with a reference to it. Return a reference to the newly constructed object.

```
extern Cell RB_Index();      /* Cell RB_Index(tuple,elem)
                                Cell tuple; long elem; */
```
Update an object referring to a tuple to refer to an element of the same tuple. Return a reference to that element.

Lazy Selection and Forcing Evaluation

All the previous functions have "Hope*" semantics, i.e. any value they select from a structure is evaluated to WHNF. The following functions select values *without* evaluating any suspensions. The functions GetVal and EvalAll are provided to evaluate suspensions in the objects to which they are applied.

```
extern Cell L_Head();          /* Cell L_Head(cons) Cell cons; */
extern Cell L_Tail();          /* Cell L_Tail(cons) Cell cons; */
extern Cell L_Index();         /* Cell L_Index(tuple, elem)
                                  Cell tuple; long elem */
extern Cell LRB_Head();        /* Cell LRB_Head(cons) Cell cons; */
extern Cell LRB_Tail();        /* Cell LRB_Tail(cons) Cell cons; */
extern Cell LRB_Index();       /* Cell LRB_Index(tuple, elem)
                                  Cell cons; long elem; */
```

Lazy versions of Head, Tail, Index, RB_Head, RB_Tail and RB_Index.

```
extern Cell GetVal();          /* Cell GetVal(c) Cell c; */
```

Evaluate the suspension referred to by the object.

```
extern Cell EvalAll();         /* Cell EvalAll(c) Cell c; */
```

Evaluate all suspensions in the structure referred to by the object.

Comparisons

```
extern int IsNil();            /* int IsNil(a) Cell a; */
```

Return true (non-zero) if referenced object is the nil list.

```
extern int CellEq();           /* int CellEq(a, b) Cell a, b */
```

Return true if the two objects refer to an equivalent value.

```
extern int CellEqNum();        /* int CellEqNum(a, b) Cell a; long b */
extern int CellEqReal();       /* int CellEqReal(a,b) Cell a; float b; */
extern int CellEqChar();       /* int CellEqChar(a,b) Cell a; char b; */
```

Return true if the object refers to an integer, character or real of the same value.

Raw Data Tuples

The following functions allow the construction and accessing of Fpm2 "raw data" tuples.

```
extern long *RawVal();         /* long *RawVal(tuple) Cell tuple; */
```

Return a pointer to the data part of a raw data tuple.

```
extern long RawSize();         /* long  RawSize(tuple) Cell tuple; */
```

Return the size, in Fpm2 words, of a raw data tuple.

```
extern Cell  RawTuple();       /* Cell RawTuple(size, elemptr)
                                  long size; long *elemptr; */
```

Construct a raw data tuple of the specified size.

Calling Hope Functions

```
extern Cell CallHope();        /* Cell CallHope(closure, arg)
                                  Cell closure, arg; */
```

This function takes a Hope function closure and an argument to apply it to, which may be a basic or structured value. Return is the result of applying the closure.

Miscellaneous

The following are provided for convenience, and to minimise redundant object creation.

Selection of Basic Values

```
extern long HeadNum();        /* long HeadNum(cons) Cell cons; */
extern Cell ConsNum();        /* Cell ConsNum(head, tail)
                                 long head; Cell tail; */
extern long IndexNum();       /* long IndexNum(tuple, elem)
                                 Cell tuple; long elem */
```

Return the integer value of the head, tail or nth element respectively.

```
extern char HeadChar();       /* char HeadChar(cons) Cell cons; */
extern Cell ConsChar();       /* Cell ConsChar(head, tail)
                                 char head; Cell tail; */
extern char IndexChar();      /* char IndexChar(tuple, elem)
                                 Cell tuple; long elem */
extern float HeadReal();      /* float HeadReal(cons) Cell cons; */
extern Cell  ConsReal();      /* Cell ConsReal(head, tail)
                                 float head; Cell tail; */
extern float IndexReal();     /* float IndexReal(tuple, elem)
                                 Cell tuple; long elem */
```

As above but for character and real values respectively

Lazy Selection of Basic Values

```
extern long L_HeadNum();      /* long L_HeadNum(cons) Cell cons; */
extern long L_IndexNum();     /* long L_IndexNum(tuple, elem)
                                 Cell tuple; long elem */
extern char L_HeadChar();     /* char L_HeadChar(cons) Cell cons; */
extern char L_IndexChar();    /* char L_IndexChar(tuple, elem)
                                 Cell tuple; long elem */
extern float L_HeadReal();    /* float L_HeadReal(cons) Cell cons; */
extern float L_IndexReal();   /* float L_IndexReal(tuple, elem)
                                 Cell tuple; long elem */
```

Lazy versions of above.

In-place Update of Structures

```
extern Cell UpdHead();        /* Cell UpdHead(cons,val) Cell cons,val;*/
extern Cell UpdTail();        /* Cell UpdTail(cons,val) Cell cons,val;*/
extern Cell Inject();         /* Cell Inject(tuple, elem, val)
                                 Cell tuple, val; long elem; */
```

Replace the head, tail or n^{th} element of the given structure by a new value. This allows "in-place" updating of Fpm2 structures.

```
extern Cell UpdHeadNum();     /* Cell UpdHead(cons, val)
                                 Cell cons; long val; */
extern Cell InjectNum();      /* Cell Inject(tuple, elem, val)
                                 Cell tuple; long val; long elem; */
extern Cell UpdHeadReal();    /* Cell UpdHead(cons, val)
                                 Cell cons; float val; */
extern Cell InjectReal();     /* Cell Inject(tuple, elem, val)
                                 Cell tuple; float val; long elem; */
extern Cell UpdHeadChar();    /* Cell UpdHead(cons, val)
                                 Cell cons; char val; */
extern Cell InjectChar();     /* Cell Inject(tuple, elem, val)
                                 Cell tuple; char val; long elem; */
```

In-place update with integer, character and real values.

Appendix 4: The Comparison Programs

This appendix contains the source for the programs used in the comparison tests [4.7.].

A4.1. Utility Functions

The following programs use a number of utility functions to; obtain argument values, convert strings to numbers, generate sequences of integers, etc. Some of these functions are provided by the language implementations as part of their standard libraries, the source for those which are not is given here to save replication in the programs themselves.

Hope+C

All the programs import some additional library modules, the import statement is given once here. The infix function '--', called `count` in the other languages generates a sequence of integers in the given range; `concmap` produces the concatenation of the result of a `map` over a list; and `filter` forms a new list from those elements of another which pass a predicate test.

```
uses StandardLists, StandardXfer, CxXfer, StandardEnviron;

infix -- : 1;
dec -- : num # num -> list(num);
--- a -- b <= if a = b then [a] else a::(a+1 -- b);

dec concmap : (alpha -> list(beta)) # list(alpha) -> list(beta);
--- concmap(f, l) <= join map_list(f, l);

dec filter : (alpha -> truval) # list(alpha) -> list(alpha);
--- filter(_, nil) <= nil;
--- filter(ok, h::t)
 <= if ok(h) then h :: filter(ok, t) else filter(ok, t);
```

SML

The function `ston` converts a string to an integer. The functions `putint` and `putlist` output integers and lists respectively; SML I/O is side-effected.

```
fun dig x = ord x - ord "0";
local fun getn (h::t) n = getn t (n * 10 + dig(h))
        | getn nil n = n
in fun ston s = if s = "0" then 0
                           else (getn (explode s) 0)
end;
```

```
(* next two functions taken from SML book *)
local fun convert 0 = ""
        | convert n = convert(n div 10) ^ chr(ord "0" + n mod 10)
in fun putint s 0 = output s "0"
     | putint s n = if n < 0
                    then (output s "~"; putint s (~n))
                    else output s (convert n)
end;

local fun putlist' put s nil = ()
        | putlist' put s [x] = put s x
        | putlist' put s (x::xs) = (put s x;
                                    output s ", ";
                                    putlist' put s xs)
in fun putlist put s xs = (output s "[";
                           putlist' put s xs;
                           output s "]")
end;

fun count a b = a :: (if a = b then nil else count (a+1) b);

nonfix @;
fun concmap f l = fold @ (map f l) nil;

fun filter f nil = nil
  | filter f (h::t)
    = if (f h) then (h :: (filter f t))
               else (filter f t);
```

Pascal

Pascal lacks a function to convert strings to integers and has no built-in list type, the following functions implement these features.

```
function numarg(i : integer) : integer;
const
   LEN = 10;
var
   s : packed array [1..LEN] of char;
   n : integer;
   j : integer;
function
   isdigit(c : char) : boolean;
   begin
      isdigit := (c >= '0') and (c <= '9')
   end;
begin
   argv(i, s);
   n := 0; j := 1;
   while (j <= LEN) and isdigit(s[j]) do
   begin
      n := n * 10 + ord(s[j]) - ord('0');
      j := j + 1;
   end;
   numarg := n
end;
```

```
type
   List = ^Cons;
   Cons = record
             hd : integer;
             tl : List
          end;

function cdr(x : List) : List;
begin
   cdr := x^.tl
end;

function car(x : List) : integer;
begin
   car := x^.hd
end;

function cons(h : integer; t : List) : List;
var
   l : List;
begin
   new(l);
   l^.hd := h;
   l^.tl := t;
   cons := l
end;

function append(x, y : List) : List;
begin
   if x = nil
   then append := y
   else append := cons(car(x), append(cdr(x), y))
end;

function range(a, b : integer) : List;
begin
   if a = b
   then range := cons(a, nil)
   else range := cons(a, range(a+1, b))
end;
```

A4.2. The Programs

A4.2.1. Nfib

This program is the modified Fibonacci function beloved by functional programmers.

Hope+C Source

Note: This program produces more elaborate output than the others, which costs some minor extra execution time, but looks better!

```
dec nfib : num -> num;
--- nfib(n) <= if n < 2 then 1 else 1 + nfib(n-1) + nfib(n-2);

--- main
 <= let (_,Unum(n)) == strtonum(getarg(1)) in
       Printf("nfib(%d) = %d\n", [Unum(n), Unum(nfib(n))], TheEnd);
```

LML Source

```
let rec
   nfib n = if n < 2 then 1 else nfib(n-1) + nfib(n-2) + 1
in
   nfib (stoi (hd argv))
```

SML Source

```
fun nfib n = if n < 2 then 1 else nfib(n-1) + nfib(n-2) + 1;

fun driver ((ig1::(h::ig2)), ig3) = print (nfib (ston h));

exportFn("nfib_sml", driver);
```

Pascal Source

```
program nfib(input,output);

function nfib(n : integer) : integer;
begin
   if n < 2
   then nfib := 1
   else nfib := 1 + nfib(n-1) + nfib(n-2)
end;

begin
   writeln(nfib(numarg(1)))
end.
```

A4.2.2. Queens – Version 1

Queens calculates how many ways *n* queens can be placed on an *n* square chessboard so that no queen is under threat. This is the first version of the Queens program, it uses higher-order functions and try all *n* rows when attempting to find the valid positions in a column. A Pascal version of this program was not written.

Hope+C Source

```
dec safe : num # num # list(num) -> truval;
--- safe(x, d, nil) <= true;
--- safe(x, d, q::m)
 <= (x /= q) and (x /= q+d) and (x /= q-d) and safe(x, d+1, m);

dec ok : list(num) -> truval;
--- ok nil <= true;
--- ok(y::l) <= safe(y, 1, l);

dec gen : num # num -> list(list(num));
--- gen(_,0) <= [nil];
--- gen(nq,n)
 <= concmap(lambda b => filter(ok, map_list(?::b, 1 -- nq))
           end, gen(nq,n-1));

dec nsoln : num -> num;
--- nsoln(nq) <= length(gen(nq, nq));

--- main
 <= let (_,Unum(n)) == strtonum(getarg(1)) in
      PutStr(numtostr(nsoln n), TheEnd);
```

LML Source

```
let nsoln nq =
   let rec ok [] = true
        || ok (x.l)
           = let rec safe x d [] = true
                  || safe x d (q.l)
                     = x ~= q & x ~= q+d & x ~= q-d & safe x (d+1) l
             in safe x 1 l
   in
   let rec gen 0 = [[]]
        || gen n = concmap (\b.(filter ok (map (\q.q.b) (count 1 nq))))
                           (gen (n-1))
   in length (gen nq)
in
   nsoln (stoi (hd argv))
```

SML Source

```
fun nsoln nq =
   let fun ok [] = true
         | ok (y::m)
           = let fun safe x d nil = true
                   | safe x d (q::l)
                     = x <> q andalso x <> q+d
                       andalso x <> q-d andalso safe x (d+1) l
             in safe y 1 m
             end
   in let fun gen 0 = [[]]
            | gen n
              = concmap (fn b => (filter ok
                                    (map (fn q => q::b) (count 1 nq))))
                        (gen (n-1))
      in
         length (gen nq)
      end
   end;

fun driver ((ig1::(h::ig2)), ig3) = print (nsoln (ston h));

exportFn("queens_sml", driver);
```

A4.2.3. Queens – Version 2

This is the second version of the Queens program; it does not use any higher-order functions and only tries *(n - m + 1)* rows when finding the solutions for the m^{th} column.

Hope+C Source

```
dec safe : num # num # list(num) -> truval;
--- safe(x, y, nil) <= true;
--- safe(x, y, q::m) <= (x /= q) and (y /= q) and safe(x-1, y+1, m);

dec ok : num # list(num) -> truval;
--- ok(_,nil) <= true;
--- ok(n,l) <= safe(n-1, n+1, l);

dec place : list(num) # list(num) # list(num) -> num;
--- place(nil, nil, _) <= 1;
--- place(nil, _, _) <= 0;
--- place(h::t, l, b)
 <= if ok(h, b)
    then place(t <> l, nil, h::b) + place(t, h::l, b)
    else place(t, h::l, b);

dec nsoln : num -> num;
--- nsoln(nq) <= place(1 -- nq, nil, nil);

--- main
 <= let (_,Unum(n)) == strtonum(getarg(1)) in
       PutStr(numtostr(nsoln n), TheEnd);
```

LML Source

```
let nsoln nq
  = let rec ok n nil = true
          || ok n (h.t)
               = let rec safe x y nil = true
                       || safe x y (h.t)
                            = (x ~= h) & (y ~= h) & safe (x-1) (y+1) t
                 in
                    safe (n-1) (n+1) (h.t)
    in
       let rec place nil nil b = 1
            || place nil (h.t) b = 0
            || place (h.t) l b
                  = if ok h b
                    then (place (t @ l) nil (h.b)) + (place t (h.l) b)
                    else place t (h.l) b
       in
          place (count 1 nq) nil nil
    in
      nsoln (stoi (hd argv))
```

SML Source

```
fun nsoln nq
  = let fun ok n nil = true
        | ok n (h::t)
            = let fun safe x y nil = true
                    | safe x y (h::t)
                      = (x <> h) andalso (y <> h) andalso
                        safe (x-1) (y+1) t
              in
                 safe (n-1) (n+1) (h::t)
              end
    in let fun place nil nil b = 1
             | place nil (h::t) b = 0
             | place (h::t) l b
               = if ok h b
                 then (place (t @ l) nil (h::b)) + (place t (h::l) b)
                 else place t (h::l) b
       in
          place (count 1 nq) nil nil
       end
    end;

fun driver ((ig1::(h::ig2)), ig3) = print (nsoln (ston h));

exportFn("queens2_sml", driver);
```

Pascal Source

```
program queens2(input, output);

function safe(x,y : integer; b : List) : boolean;
var
   h : integer;
begin
   if b = nil
   then safe := true
   else
   begin
      h := car(b);
      safe := (x <> h) and (y <> h) and safe(x-1, y+1, cdr(b))
   end
end;

function ok(x : integer; b : List) : boolean;
begin
   if b = nil
   then ok := true
   else ok := safe(x-1, x+1, b)
end;

function place(x, y, b : List) : integer;
var
   h : integer;
   t : List;
begin
   if x = nil
   then
   begin
      if y = nil
      then place := 1
      else place := 0
   end
   else
   begin
      h := car(x); t := cdr(x);
      if ok(h, b)
      then place := place(append(t, y), nil, cons(h, b)) + place(t,
cons(h, y), b)
      else place := place(t, cons(h, y), b)
   end
end;

function nsoln(n : integer) : integer;
begin
   nsoln := place(range(1, n), nil, nil)
end;

begin
   write(nsoln(numarg(1)):1)
end.
```

A4.2.4. HigherOrder

This program exercises higher-order functions by constructing a large "matrix" with lists and then mapping and reducing over it to produce a single integer result. For an input of *n* this program performs *2n(n+1)* higher-order function applications and *cons*es.

Hope+C Source

```
dec gen : num # alpha -> list(alpha);
--- gen(0, _) <= nil;
--- gen(n, x) <= x :: gen(n-1,x);

dec mapred : num -> num;
--- mapred n
 <= reduce_list(lambda (y,tot) => tot + reduce_list(+, (y,0)) end,
                (map_list(lambda x => map_list(2 * ?, x) end,
                          gen(n, gen(n, 1))),
                 0)
               );

--- main
 <= let (_,Unum(n)) == strtonum(getarg(1)) in
       PutStr(numtostr(mapred n), TheEnd);
```

LML Source

```
let rec gen 0 x = []
     || gen n x = x.(gen (n-1) x)
in let mapred n = reduce (\y.\tot.tot + (reduce (+) 0 y)) 0
                         (map (\x.map (2*) x) (gen n (gen n 1)))
in mapred (stoi (hd argv))
```

SML Source

```
fun gen 0 x = []
  | gen n x = x::(gen (n-1) x);

local fun double x:int = x * 2 in
fun mapred n = fold (fn (y,tot) => tot + (fold (op +) y 0))
                    (map (fn x => map double x) (gen n (gen n 1))) 0
end;

fun driver ((ig1::(h::ig2)), ig3) = print (mapred (ston h));

exportFn("ho_sml", driver);
```

A4.2.5. Tak

This is the first of three programs taken from the [Gabriel] Lisp evaluation suite. The program computes a variation of the Takeuchi function and makes 63609 calls to the function `tak`.

Hope+C Source

```
dec tak : num # num # num -> num;
---tak(x,y,z)<= if not (y<x) then z
                else tak( tak(x-1,y,z),
                          tak(y-1,z,x),
                          tak(z-1,x,y));

--- main <= Printf("%d\n", [Unum(tak(18,12,6))], TheEnd);
```

LML Source

```
let rec
   tak x y z = if not (y<x) then z
               else tak (tak (x-1) y z)
                        (tak (y-1) z x)
                        (tak (z-1) x y)
in
   tak 18 12 6
```

SML Source

```
fun tak x y z = if not (y<x) then z
                else tak (tak (x-1) y z)
                         (tak (y-1) z x)
                         (tak (z-1) x y);

fun driver (ig1, ig2) = print (tak 18 12 6);

exportFn("tak_sml", driver);
```

Pascal Source

```
program tak(input,output);

function tak(x,y,z: integer) : integer;
begin
   if not (y < x)
   then tak := z
   else tak := tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y))
end;

begin
   writeln(tak(18,12,6))
end.
```

A4.2.6. Takl

Takl is a non-numeric version of the Takeuchi function which uses a list of length *n* to represent the integer *n*. This program performs 63609 calls to `tak`, 47706 calls to `cdr`, and 473059 calls to `shorterp`.

Hope+C Source

```
dec shorterp : list(num) # list(num) -> truval;
--- shorterp(_::l1t, _::l2t) <= shorterp(l1t, l2t);
--- shorterp(_, nil)<= false;
--- shorterp(nil, _::_) <= true;

dec cdr : list(num) -> list(num);
--- cdr(h::t)<=t;
--- cdr(nil)<=nil;

dec takl : list(num) # list(num) # list(num) -> list(num);
--- takl(x,y,z) <= if not (shorterp(y,x)) then z
                   else takl( takl(cdr(x),y,z),
                              takl(cdr(y),z,x),
                              takl(cdr(z),x,y));

dec downfrom : num -> list(num);
--- downfrom(0) <= nil;
--- downfrom(n) <= n :: downfrom(n-1);

--- main <= PutStr((listtostr(numtostr))
                   takl(downfrom(18), downfrom(12), downfrom(6)),
                   TheEnd);
```

LML Source

```
let rec
    putlist put xs
    = let rec
          putlist2 put nil = []
       || putlist2 put [x] = put x
       || putlist2 put (x.xs) = (put x) @ ", " @ (putlist2 put xs)
      in "[" @ (putlist2 put xs) @ "]"
and
    shorterp (l1h.l1t) (l2h.l2t) = (shorterp l1t l2t)
 || shorterp l1 [] = false
 || shorterp [] (l2h.l2t) = true
and
   takl x y z
   = if ~ (shorterp y x)
     then z
     else (takl (takl(tl x) y z)
                (takl(tl y) z x)
                (takl(tl z) x y))
and
   downfrom n  = if n = 1 then [1] else n.(downfrom (n-1))
in
   putlist itos (takl (downfrom 18) (downfrom 12) (downfrom 6))
```

SML Source

```
fun shorterp (l1h::l1t) (l2h::l2t) = (shorterp l1t l2t)
  | shorterp l1 [] = false
  | shorterp [] (l2h::l2t) = true;

fun takl x y z
  = if not (shorterp y x)
    then z
    else (takl (takl(tl x) y z) (takl(tl y) z x) (takl(tl z) x y));

fun downfrom(n) = if n = 1 then [1] else n::(downfrom (n-1));

fun driver (ig1, ig2)
     = putlist putint std_out
               (takl (downfrom 18) (downfrom 12) (downfrom 6));

exportFn("takl_sml", driver);
```

Pascal Source

```
program takl(input, output);

function shorterp(x, y : List) : boolean;
begin
   if x = nil
   then shorterp := y <> nil
   else if y = nil
   then shorterp := false
   else shorterp := shorterp(cdr(x), cdr(y))
end;

function downfrom(n : integer) : List;
begin
   if n = 0 then downfrom := nil
            else downfrom := cons(n, downfrom(n-1))
end;

function takl(x, y, z : List) : List;
begin
   if not shorterp(y, x)
   then takl := z
   else takl := takl(takl(cdr(x), y, z),
                     takl(cdr(y), z, x),
                     takl(cdr(z), x, y)
                    )
end;

procedure writeList(x : List);
begin
   if x <> nil then
   begin
      write(x^.hd:1); writeList(cdr(x))
   end
end;

begin
   writeList(takl(downfrom(18), downfrom(12), downfrom(6)))
end.
```

A4.2.7. Triangle

This program finds all the solutions to a small triangular puzzle. This program performs 5802572 calls to try, 7820920 references to the board, and 971616 updates to the board. Due to the heavy use of polymorphism and list manipulation no Pascal version of this program has been written.

Hope+C Source

```
type boardlist == list(truval);
type move      == (num # num # num);
type sequence  == list(move);
type solution  == list(move);

dec boardbegin : boardlist;
--- boardbegin <= [true,true,true,true,false,true,true,
                   true,true,true,true,true,true,true,true];

dec legalmoves : sequence;
--- legalmoves <= [( 1, 2, 4), ( 2, 4, 7), ( 4, 7,11), ( 3, 5, 8),
                   ( 5, 8,12), ( 6, 9,13), ( 1, 3, 6), ( 3, 6,10),
                   ( 6,10,15), ( 2, 5, 9), ( 5, 9,14), ( 4, 8,13),
                   (11,12,13), (12,13,14), (13,14,15), ( 7, 8, 9),
                   ( 8, 9,10), ( 4, 5, 6), ( 4, 2, 1), ( 7, 4, 2),
                   (11, 7, 4), ( 8, 5, 3), (12, 8, 5), (13, 9, 6),
                   ( 6, 3, 1), (10, 6, 3), (15,10, 6), ( 9, 5, 2),
                   (14, 9, 5), (13, 8, 4), (13,12,11), (14,13,12),
                   (15,14,13), ( 9, 8, 7), (10, 9, 8), ( 6, 5, 4)];

dec validmovep : move # boardlist -> truval;
--- validmovep((i,j,k), B) <= B@i and (B@j and not(B@k));

dec makemove : move # boardlist -> boardlist;
dec set : boardlist # num -> boardlist;
dec unset : boardlist # num -> boardlist;

--- makemove((i,j,k), B) <= set(unset(unset(B,i),j),k);
--- set(nil,x) <= nil;
--- set(h::t,x) <= if x=1 then true::t else h::set(t,x-1);
--- unset(nil,x) <= nil;
--- unset(h::t,x) <= if x=1 then false::t else h::unset(t,x-1);

dec try : num # boardlist # sequence # solution -> num;
--- try(depth,B,nil,seq)
 <= if depth = 14 then 1 else 0;
--- try(depth,B,h::t,seq)
 <= if depth = 14
    then 1
    else if validmovep(h,B)
    then try(depth + 1, makemove(h,B), legalmoves, h::seq)
         + try(depth, B, t, seq)
    else try(depth, B, t, seq);

---  main
 <= PutStr(numtostr(try(1,boardbegin,[(12,8,5)],nil)), TheEnd);
```

LML Source

```
let rec
   boardbegin = [true;true;true;true;false;true;true;
                 true;true;true;true;true;true;true;true]
and
   legalmoves = [( 1, 2, 4); ( 2, 4, 7); ( 4, 7,11); ( 3, 5, 8);
                 ( 5, 8,12); ( 6, 9,13); ( 1, 3, 6); ( 3, 6,10);
                 ( 6,10,15); ( 2, 5, 9); ( 5, 9,14); ( 4, 8,13);
                 (11,12,13); (12,13,14); (13,14,15); ( 7, 8, 9);
                 ( 8, 9,10); ( 4, 5, 6); ( 4, 2, 1); ( 7, 4, 2);
                 (11, 7, 4); ( 8, 5, 3); (12, 8, 5); (13, 9, 6);
                 ( 6, 3, 1); (10, 6, 3); (15,10, 6); ( 9, 5, 2);
                 (14, 9, 5); (13, 8, 4); (13,12,11); (14,13,12);
                 (15,14,13); ( 9, 8, 7); (10, 9, 8); ( 6, 5, 4)]
and
   validmovep (i,j,k) B = (select B i)
                          & ((select B j)& (~(select B k)))
and
   set [] x = []
|| set (h.t) x = if x=1 then (true.t) else (h.(set t (x-1)))
and
   unset [] x = []
|| unset (h.t) x = if x=1 then (false.t) else (h.(unset t (x-1)))
and
   makemove (i,j,k) B = (set (unset (unset B i) j) k)
and
   try depth B [] seq  = if depth=14 then 1 else 0
|| try depth B (h.t) seq
 = if depth = 14
   then 1
   else if (validmovep h B)
   then (try (depth+1) (makemove h B) legalmoves (h.seq))
        + (try depth B t seq)
   else (try depth B t seq)
in
   (try 1 boardbegin [(12,8,5)] [])
```

SML Source

```
val boardbegin = [true,true,true,true,false,true,true,
                  true,true,true,true,true,true,true,true];

val legalmoves = [( 1, 2, 4), ( 2, 4, 7), ( 4, 7,11), ( 3, 5, 8),
                  ( 5, 8,12), ( 6, 9,13), ( 1, 3, 6), ( 3, 6,10),
                  ( 6,10,15), ( 2, 5, 9), ( 5, 9,14), ( 4, 8,13),
                  (11,12,13), (12,13,14), (13,14,15), ( 7, 8, 9),
                  ( 8, 9,10), ( 4, 5, 6), ( 4, 2, 1), ( 7, 4, 2),
                  (11, 7, 4), ( 8, 5, 3), (12, 8, 5), (13, 9, 6),
                  ( 6, 3, 1), (10, 6, 3), (15,10, 6), ( 9, 5, 2),
                  (14, 9, 5), (13, 8, 4), (13,12,11), (14,13,12),
                  (15,14,13), ( 9, 8, 7), (10, 9, 8), ( 6, 5, 4)];

exception Nth;
fun nth (e::r) 1 = e
  | nth (e::r) n = nth r (n-1)
  | nth _ _ = raise Nth;
```

```
fun validmovep (i,j,k) B
  = (nth B i) andalso ((nth B j) andalso not(nth B k));
fun set [] x = []
  | set (h::t) x = if x=1 then (true::t) else (h::(set t (x-1)));

fun unset [] x = []
  | unset (h::t) x = if x=1 then (false::t) else (h::(unset t (x-1)));

fun makemove (i,j,k) B = (set (unset (unset B i) j) k);

fun try depth B [] seq  = if depth=14 then 1 else 0
  | try depth B (h::t) seq
    = if depth = 14
      then 1
      else if (validmovep h B)
      then (try (depth+1) (makemove h B) legalmoves (h::seq))
           + (try depth B t seq)
      else (try depth B t seq);

fun driver(ig1, ig2) = print (try 1 boardbegin [(12,8,5)] []);

exportFn("tria_sml", driver);
```

A4.2.8. Triangle – Vector Version

This version uses a vector to represent the board rather than a list, the original Lisp program found in [Gabriel] used vectors. Neither LML nor SML support vectors so this version has only been written in Hope+C and Pascal.

Hope+C Source

The program is based upon the list version given above with the functions `set` & `unset` removed and following changes:

```
type boardstate   == vector(truval);

dec boardbegin : boardstate;
--- boardbegin <= listtovector([true,true,true,true,false,true,true,
                                true,true,true,true,true,true,true,true]);

dec makemove : move # boardstate -> boardstate;
--- makemove((i,j,k), B)
 <= perm_vector(B, lambda (ix,e)
                   => (ix, if (ix = i) or (ix = j)
                           then false
                           else if ix = k then true
                           else e
                       )
                   end
              );
```

Pascal Source

This program differs from the Hope+C program by using a global variable for the list `legalmoves`, this was done as Pascal does not support "constant functions" and to leave `legalmoves` as a function would cause large amounts of redundant repeated computation.

```
program tria(input, output);

type
   boardstate = array [1..15] of boolean;
   triple = record x,y,z : integer end;
   Move = ^triple;
   List = ^Cons;
   Cons = record
            hd : Move;
            tl : List
          end;
   sequence = List;
   solution = List;

var
   m : Move;
   start : sequence;
   legalmoves : sequence;

function boardbegin : boardstate;
var
   i : integer;
   B : boardstate;
begin
   for i := 1 to 15 do B[i] := true;
   B[5] := false;
   boardbegin := B
end;

procedure build_legalmoves;
var
   t : Move;
begin
   new(t); t^.x :=  6; t^.y :=  5; t^.z :=  4; legalmoves := cons(t, nil);
   new(t); t^.x := 10; t^.y :=  9; t^.z :=  8; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  9; t^.y :=  8; t^.z :=  7; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 15; t^.y := 14; t^.z := 13; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 14; t^.y := 13; t^.z := 12; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 13; t^.y := 12; t^.z := 11; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 13; t^.y :=  8; t^.z :=  4; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 14; t^.y :=  9; t^.z :=  5; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  9; t^.y :=  5; t^.z :=  2; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 15; t^.y := 10; t^.z :=  6; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 10; t^.y :=  6; t^.z :=  3; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  6; t^.y :=  3; t^.z :=  1; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 13; t^.y :=  9; t^.z :=  6; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 12; t^.y :=  8; t^.z :=  5; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  8; t^.y :=  5; t^.z :=  3; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 11; t^.y :=  7; t^.z :=  4; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  7; t^.y :=  4; t^.z :=  2; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  4; t^.y :=  2; t^.z :=  1; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  4; t^.y :=  5; t^.z :=  6; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  8; t^.y :=  9; t^.z := 10; legalmoves := cons(t, legalmoves);
```

```
   new(t); t^.x :=  7; t^.y :=  8; t^.z :=  9; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 13; t^.y := 14; t^.z := 15; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 12; t^.y := 13; t^.z := 14; legalmoves := cons(t, legalmoves);
   new(t); t^.x := 11; t^.y := 12; t^.z := 13; legalmoves := cons(t, legalmoves)
   new(t); t^.x :=  4; t^.y :=  8; t^.z := 13; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  5; t^.y :=  9; t^.z := 14; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  2; t^.y :=  5; t^.z :=  9; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  6; t^.y := 10; t^.z := 15; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  3; t^.y :=  6; t^.z := 10; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  1; t^.y :=  3; t^.z :=  6; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  6; t^.y :=  9; t^.z := 13; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  5; t^.y :=  8; t^.z := 12; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  3; t^.y :=  5; t^.z :=  8; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  4; t^.y :=  7; t^.z := 11; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  2; t^.y :=  4; t^.z :=  7; legalmoves := cons(t, legalmoves);
   new(t); t^.x :=  1; t^.y :=  2; t^.z :=  4; legalmoves := cons(t, legalmoves)
end;

function validmovep(m : Move; var B : boardstate) : boolean;
begin
   validmovep := B[m^.x] and (B[m^.y] and not(B[m^.z]))
end;

function makemove(m : Move; B : boardstate) : boardstate;
begin
   B[m^.x] := false;
   B[m^.y] := false;
   B[m^.z] := true;
   makemove := B
end;

function try(depth : integer; B : boardstate;
             moves : sequence; seq : solution) : integer;
var
   h : Move;
begin
   if moves = nil
   then
   begin
      if depth = 14 then try := 1 else try := 0
   end
   else if depth = 14
   then try := 1
   else
   begin
      h := car(moves);
      if validmovep(h, B)
      then try := try(depth+1, makemove(h, B),
                      legalmoves, cons(h, seq))
                + try(depth, B, cdr(moves), seq)
      else try := try(depth, B, cdr(moves), seq)
   end
end;

begin
   build_legalmoves;
   new(m); m^.x := 12; m^.y := 8; m^.z := 5; start := cons(m, nil);
   write(try(1, boardbegin, start, nil):1)
end.
```

A4.2.9 Cat

This program emulates the Unix™ "cat" command, which either copies standard input to standard output or copies a set of files, given as arguments, to the standard output. The I/O systems of our four languages are very different from each other, so these programs are written to solve the problem in the most natural way for each language.

Hope+C Source

```
type string == list(char);

dec infile : Status # list(string) -> Result;
dec outfile : Status # string # string # list(string) -> Result;

--- infile(_, nil) <= Exit(0);
--- infile(_, h::t) <= GetObj(h, outfile(?, ?, h, t));

--- outfile(Ok, contents, _, files)
 <= PutStr(contents, infile(?, files));
--- outfile(Failed(_), _, name, _)
 <= Printf(stderr, "Failed to open: %s\n", [Ustr(name)], TheEnd);

--- main
 <= if getargcount = 0
    then GetObj(stdin, outfile(?, ?, "Standard Input", nil))
    else infile(Ok, getargl);
```

LML Source

```
#include <OK>

let rec
   infile nil = nil
|| infile (h.t) = case openfile h in
                     Yes file : file @ (infile t)
                  || No emesg : emesg @ " " @ h @ "\n"
                  end
in
   if length argv = 0 then input else infile argv
```

SML Source

```
fun copyfile s = if end_of_stream s
                 then {}
                 else (output std_out (input s 512); copyfile s);

fun infile nil = {}
  | infile (h::t) = (copyfile (open_in h); infile t);

fun driver (ig1::t, ig2) = if t = nil
                           then copyfile std_in
                           else infile t;

exportFn("cat_sml", driver);
```

Pascal Source

```
program cat(input, output);

const
   MAXLINE = 512;
type
   string = packed array [1..MAXLINE] of char;
var
   i : integer;
   fname : string;

procedure copyfile(var name : string);
var
   buf : char;
   infile : text
begin
   reset(infile, name);
   while not eof(infile) do
   begin
      while not eoln(infile) do
      begin
         read(infile, buf);
         write(buf)
      end;
      readln(infile); writeln
   end
end;

procedure copystdin;
var
   buf : char;
begin
   while not eof do
   begin
      while not eoln do
      begin
         read(buf);
         write(buf)
      end;
      readln; writeln
   end
end;

begin
   if argc = 1
   then copystdin
   else
      for i := 1 to argc-1 do
      begin
         argv(i, fname);
         copyfile(fname)
      end
end.
```

A4.2.10 FFT

Vector-Based Pascal Version

```
program fft(input, output);
const
   pi = 3.14152;
type
   vec = array [1..1025] of real;
var
   aimag : vec;
   areal : vec;

procedure fft(var ai, ar : vec);
var
   m, i, j, k, l, le, le1, n, nv2, nm1, ip : integer;
   ti, tr, ui, ur, wi, wr : real;
begin
   n := 1024;
   nv2 := n div 2;
   nm1 := n - 1;
   m := 0;
   i := 1;
   while i < n do
   begin
      m := m + 1;
      i := i + i
   end;
   if n <> exp(ln(2) * m) then halt;

   j := 1; i := 1;
   repeat
      if i < j then
      begin
      ti := ai[i]; ai[i] := ai[j]; ai[j] := ti;
      tr := ar[i]; ar[i] := ar[j]; ar[j] := tr
      end;
      k := nv2;
      while k < j do
      begin
      j := j - k;
      k := k div 2
      end;
      j := j + k;
      i := i + 1;
   until not(i < n);

   for l := 1 to m do
   begin
      le := trunc(exp(ln(2) * l));
      le1 := le div 2;
      ur := 1.0;
      ui := 0.0;
      wr := cos(pi / le1);
      wi := sin(pi / le1);
      for j := 1 to le1 do
      begin
         i := j;
```

```
            while i < n do
            begin
               ip := i + le1;
               tr := ar[ip]*ur - ai[ip]*ui;
               ti := ar[ip]*ui + ai[ip]*ur;
               ar[ip] := ar[i] - tr;
               ai[ip] := ai[i] - ti;
               ar[i] := ar[i] + tr;
               ai[i] := ai[i] + ti;
               i := i + le
            end;
         tr := ur*wr - ui*wi;
         ti := ur*wi + ui*wr;
         ur := tr;
         ui := ti
         end
      end
end;

procedure zeroit(var v : vec);
var
   i : integer;
begin
   for i := 1 to 1025 do v[i] := 0.0
end;

begin
   zeroit(aimag);
   zeroit(areal);
   fft(aimag, areal)
end.
```

List-Based Hope+C Version

```
uses StandardXfer, StandardLists, StandardReals;

type complex == real # real;
type Clist == list(complex);

dec add_and_sub : Clist # Clist -> Clist # Clist;
--- add_and_sub([], [])<=([], []);
--- add_and_sub((xr1, xi1)::l1, (xr2, xi2)::l2)
 <= let (Arest, Srest) == add_and_sub(l1, l2) in
       ((xr1+xr2,xi1+xi2) :: Arest, (xr1-xr2,xi1-xi2) :: Srest);

dec w2px : num # num # complex -> complex;
--- w2px(p, n, (xr,xi))
 <= let r == 2*Pi*p/n in
    let c == cos(r) in
    let s == sin(r) in
       (c*xr-s*xi, s*xr+c*xi);

dec mult_by_wp:Clist # num # num -> Clist;
--- mult_by_wp([], _, _) <= [];
--- mult_by_wp(x::l, p, n)
 <= w2px(p, n, x) :: mult_by_wp(l, p+1, n);
```

```
dec even_odd : list(alpha)->list(alpha) # list(alpha);
--- even_odd([]) <= ([], []);
--- even_odd(x::(y::l))
 <= let (evenrest, oddrest) == even_odd(l) in
       (x::evenrest, y::oddrest);

dec fft : Clist # num -> Clist ;
--- fft(g, n)
 <= if n = 1
    then g
    else
       let (even, odd) == even_odd(g)  in
       let fft1 == fft(even, (n div 2)) in
       let fft2 == fft(odd, (n div 2)) in
       let wpfft2 == mult_by_wp(fft2, 0, n) in
       let (added, subtracted) == add_and_sub(fft1, wpfft2)
       in
       added <> subtracted;

dec makedata : num -> Clist;
--- makedata 0 <= nil;
--- makedata n <= (0.0,0.0) :: makedata(n-1);

--- main
 <= let (a,_)::_ == fft(makedata(1024), 1024)
    in Exit(floor(a));
```

List-Based SML Version

```
fun add_and_sub nil nil = (nil, nil)
  | add_and_sub ((xr1 : real,xi1 : real)::l1) ((xr2,xi2)::l2)
    = let val (Arest, Srest) = add_and_sub l1 l2
      in
      ((xr1+xr2,xi1+xi2)::Arest, (xr1-xr2, xi1-xi2)::Srest)
      end;

val Pi = 3.14159265358979323844;

fun w2px p n (xr,xi)
  = let val r = 2.0*Pi*p/real(n) in
    let val c = cos(r) in
    let val s = sin(r) in
       (c*xr-s*xi, s*xr+c*xi)
    end end end;

fun mult_by_wp nil x y = nil
  | mult_by_wp (x::l) p n
    = (w2px p n x) :: (mult_by_wp l (p+1.0) n);

fun even_odd nil = (nil, nil)
  | even_odd (x::(y::l))
    = let val (evenrest, oddrest) = even_odd(l) in
       (x::evenrest, y::oddrest)
    end;

fun makedata 0 = nil
  | makedata n = (0.0,0.0) :: (makedata (n-1));
```

```
fun fft g n
  = if n = 1
    then g
    else
       let val (even, odd) = even_odd g  in
       let val fft1 = (fft even  (n div 2)) in
       let val fft2 = (fft odd (n div 2)) in
       let val wpfft2 = (mult_by_wp fft2 0.0 n) in
       let val (added, subtracted) = (add_and_sub fft1 wpfft2)
       in
       added @ subtracted
       end end end end end;

fun driver (ig1, ig2) = ((fft (makedata 1024) 1024); {});

exportFn("Rfft_sml", driver);
```

List-Based Pascal Version

```
program Rfft(input, output);

type
   complex = record re, im : real end;

   { Note: List type different than in A4.1,
     but same list functions used }
   List = ^Cons;
   Cons = record
          hd : complex;
          tl : List
       end;

var
   ig : List;

procedure add_and_sub(a, b : List; var Va, Vb : List);
var
   t : complex;
   ah, bh : complex;
begin
   if (a = nil) and (b = nil)
   then
   begin
      Va := nil; Vb := nil
   end
   else
   begin
      add_and_sub(cdr(a), cdr(b), Va, Vb);
      ah := car(a);
      bh := car(b);
      t.re := ah.re + bh.re;
      t.im := ah.im + bh.im;
      Va := cons(t, Va);
      t.re := ah.re - bh.re;
      t.im := ah.im - bh.im;
      Vb := cons(t, Vb)
   end
end;
```

```
procedure w2px(p, n : integer; x : complex; var R : complex);
const
   Pi = 3.14159265358979323844;
var
   r, c, s : real;
begin
   r := 2 * Pi * p / n;
   c := cos(r);
   s := sin(r);
   R.re := c*x.re - s*x.im;
   R.im := s*x.re + c*x.im
end;

function mult_by_wp(l : List; p, n : integer) : List;
var
   t : complex;
begin
   if l = nil
   then mult_by_wp := nil
   else
   begin
      w2px(p, n, car(l), t);
      mult_by_wp := cons(t, mult_by_wp(cdr(l), p+1, n))
   end
end;

procedure even_odd(l : List; var Even, Odd : List);
begin
   if l = nil
   then
   begin
      Even := nil;
      Odd := nil
   end
   else
   begin
      even_odd(cdr(cdr(l)), Even, Odd);
      Even := cons(car(l), Even);
      Odd := cons(car(cdr(l)), Odd)
   end
end;

function fft(g : List; n : integer) : List;
var
   even, odd, fft1, fft2, wpfft2, add, sub : List;
begin
   if n = 1
   then fft := g
   else
   begin
      even_odd(g, even, odd);
      fft1 := fft(even, n div 2);
      fft2 := fft(odd, n div 2);
      wpfft2 := mult_by_wp(fft2, 0, n);
      add_and_sub(fft1, wpfft2, add, sub);
      fft := append(add, sub)
   end
end;
```

```
function makedata(n : integer) : List;
var
   t : complex;
begin
   if n = 0
   then makedata := nil
   else
   begin
      t.re := 0;
      t.im := 0;
      makedata := cons(t, makedata(n-1))
   end
end;

begin
   ig := fft(makedata(1024), 1024)
end.
```

Vector-Based Hope+C Version

This is a transliteration of the [Gabriel] Lisp version, with tail-recursive functions replacing the iterative loops of the original.

```
uses StandardVectors, StandardReals;
uses StandardXfer, CxXfer;

dec l1 : num # num # num -> num;
--- l1(i, n, m) <= if i < n then l1(i+i, n, m+1) else m;

dec l6 : num # num -> num # num;
--- l6(k, j) <= if k < j then l6(k div 2, j - k) else (k, j);

dec l3 : vector(real) # vector(real) # num # num # num # num
     -> vector(real) # vector(real);
--- l3(ai, ar, i, j, nv2, n)
 <= let (ai1, ar1) ==
       if i < j
       then
       (  let swap == lambda (ix,v) => if ix = i then (j,v)
                                       else if ix = j then (i,v)
                                       else (ix,v) end in
             (perm_vector(ai, swap), perm_vector(ar, swap))
       )
       else
       (ai, ar) in
    let (k1, j1) == l6(nv2, j) in
    let j2 == j1 + k1 in
    let i1 == i + 1 in
       if i1 < n
       then l3(ai1, ar1, i1, j2, nv2, n)
       else (ai1, ar1);

dec do1 : vector(real) # vector(real)
        # num # num # num # real # real # num
       -> vector(real) # vector(real);
```

```
--- do1(ai, ar, le, le1, n, ui, ur, i)
 <= if i > n
    then (ai, ar)
    else
    (  let ip == i + le1 in
       let tr == (ar@ip * ur) - (ai@ip * ui) in
       let ti == (ar@ip * ui) + (ai@ip * ur) in
       let ar1 == perm_vector(ar,
                      lambda (ix,v)
                      => if ix = ip then (ix, ar@i - tr)
                         else if ix = i then (ix, ar@i + tr)
                         else (ix, v)
                      end) in
       let ai1 == perm_vector(ai,
                      lambda (ix,v)
                      => if ix = ip then (ix, ai@i - ti)
                         else if ix = i then (ix, ai@i + ti)
                         else (ix, v)
                      end) in
          do1(ai1, ar1, le, le1, n, ui, ur, i + le)
    );

dec do2 : vector(real) # vector(real)
       # num # num # num # real # real # real # real # num
      -> vector(real) # vector(real);
--- do2(ai, ar, le, le1, n, ui, ur, wi, wr, j)
 <= if j > le1
    then (ai, ar)
    else
    (  let (ai1, ar1) == do1(ai, ar, le, le1, n, ui, ur, j) in
       let ur1 == ur*wr - ui*wi in
       let ui1 == ur*wi + ui*wr in
          do2(ai1, ar1, le, le1, n, ui1, ur1, wi, wr, j+1)
    );

dec do3 : vector(real) # vector(real) # num # num # num
       -> vector(real) # vector(real);
--- do3(ai, ar, n, m, l)
 <= if l > m
    then (ai, ar)
    else
    ( let le == 2 ^ l in
      let le1 == le div 2 in
      let wr == cos(Pi / le1) in
      let wi == sin(Pi / le1) in
      let (ai1, ar1) == do2(ai, ar, le, le1, n, 0.0, 1.0, wi, wr, 1) in
         do3(ai1, ar1, n, m, l+1)
    );

dec fft : vector(real) # vector(real) -> vector(real) # vector(real);
--- fft(ar, ai)
 <= let n == vectorsize(ar) - 1 in
    let m == l1(1, n, 0) in
       if n /= 2^m
       then error([(999, "array size not a power of two")])
       else
       (  let (ai1, ar1) == l3(ai, ar, 1, 1, n div 2, n) in
             do3(ai1, ar1, n, m, 1)
       );
```

```
dec aimag : vector(real);
--- aimag <= defvector(1025, lambda _ => 0.0 end);
dec areal : vector(real);
--- areal <= aimag;

dec loop : num # num -> num;
--- loop(0, x) <= x;
--- loop(n, _)
 <= let (v, _) == fft(aimag, areal) in loop(n-1, vectorsize(v));

--- main <= Exit(loop(1, 1));
```

CPS Hope+C Version with inter-language calls to C

First the Hope+C code, this is a translation of the above program into CPS together with the replacement of the vector modifying calls by `CallImp` continuations to access C functions.

```
uses StandardVectors, StandardReals, StandardXfer, CxXfer;;

dec l1 : num # num # num -> num;
--- l1(i, n, m) <= if i < n then l1(i+i, n, m+1) else m;

dec l6 : num # num -> num # num;
--- l6(k, j) <= if k < j then l6(k div 2, j - k) else (k, j);

dec interchange : (vector(real) # vector(real) # num # num)
             :=> (vector(real) # vector(real));

dec l3 : vector(real) # vector(real) # num # num # num # num
       # (vector(real) # vector(real) -> Result) -> Result;
--- l3(ai, ar, i, j, nv2, n, cf)
 <= let l3_2 == lambda (ai1, ar1)
                => let (k1, j1) == l6(nv2, j) in
                   let j2 == j1 + k1 in
                   let i1 == i + 1 in
                      if i1 < n
                      then l3(ai1, ar1, i1, j2, nv2, n, cf)
                      else cf(ai1, ar1)
                end
    in if i < j
       then CallImp(interchange, (ai, ar, i, j), l3_2)
       else l3_2(ai, ar);

dec butterfly : (vector(real) # vector(real) # num # num # real # real)
           :=> (vector(real) # vector(real));

dec do1 : vector(real) # vector(real)
        # num # num # num # real # real # num
        # (vector(real) # vector(real) -> Result) -> Result;
--- do1(ai, ar, le, le1, n, ui, ur, i, cf)
 <= if i > n
    then cf(ai, ar)
    else
    (  let ip == i + le1 in
       let tr == (ar@ip * ur) - (ai@ip * ui) in
       let ti == (ar@ip * ui) + (ai@ip * ur) in
          CallImp(butterfly, (ai, ar, i, ip, ti, tr),
                  do1(?, ?, le, le1, n, ui, ur, i + le, cf))
    );
```

```
dec do2 : vector(real) # vector(real)
        # num # num # num # real # real # real # real # num
        # ( vector(real) # vector(real) -> Result) -> Result;
--- do2(ai, ar, le, le1, n, ui, ur, wi, wr, j, cf)
 <= if j > le1
    then cf(ai, ar)
    else
    ( let ur1 == ur*wr - ui*wi in
      let ui1 == ur*wi + ui*wr in
         do1(ai, ar, le, le1, n, ui, ur, j,
             do2(?, ?, le, le1, n, ui1, ur1, wi, wr, j+1, cf))
    );

dec do3 : vector(real) # vector(real)
        # num # num # num # (vector(real) # vector(real) -> Result)
        -> Result;
--- do3(ai, ar, n, m, l, cf)
 <= if l > m
    then cf(ai, ar)
    else
    ( let le == 2 ^ l in
      let le1 == le div 2 in
      let wr == cos(Pi / le1) in
      let wi == sin(Pi / le1) in
         do2(ai, ar, le, le1, n, 0.0, 1.0, wi, wr, 1,
             do3(?, ?, n, m, l+1, cf))
    );

dec fft : vector(real) # vector(real)
        # (vector(real) # vector(real) -> Result) -> Result;
--- fft(ar, ai, cf)
 <= let n == vectorsize(ar) - 1 in
    let m == l1(1, n, 0) in
       if n /= 2^m
       then error([(999, "array size not a power of two")])
       else
          l3(ai, ar, 1, 1, n div 2, n, do3(?, ?, n, m, 1, cf));

dec aimag : vector(real);
--- aimag <= defvector(1025, lambda _ => 0.0 end);
dec areal : vector(real);
--- areal <= defvector(1025, lambda _ => 0.0 end);

dec loop : num -> Result;
--- loop(0) <= Exit(0);
--- loop(n) <= fft(aimag, areal, lambda _ => loop(n-1) end);

--- main <= loop(1);
```

Now the C code for the two functions `interchange` and `butterfly`:

```
#include <hope.h>

Cell interchange(args) Cell *args;
{  Cell t;
   long i, j;
   Cell ai, ar;

   ai = Index(args, 1L);
   ar = Index(args, 2L);
   i = IndexNum(args, 3L);
   j = IndexNum(args, 4L);

   t = Index(ai, i);
   Inject(ai, i, Index(ai, j));
   Inject(ai, j, t);

   t = Index(ar, i);
   Inject(ar, i, Index(ar, j));
   Inject(ar, j, t);

   return Tuple(2L, ai, ar);
}

Cell butterfly(args) Cell *args;
{  long i, ip;
   float tr, ti;
   Cell ai, ar;

   ai = Index(args, 1L);
   ar = Index(args, 2L);
   i =  IndexNum(args, 3L);
   ip = IndexNum(args, 4L);
   ti = IndexReal(args, 5L);
   tr = IndexReal(args, 6L);

   InjectReal(ar, ip, IndexReal(ar, i) - tr);
   InjectReal(ai, ip, IndexReal(ai, i) - ti);
   InjectReal(ar, i, IndexReal(ar, i) + tr);
   InjectReal(ai, i, IndexReal(ai, i) + ti);

   return Tuple(2L, ai, ar);
}
```

Appendix 5: Fpm+ to Fpm/AM Translation

In this appendix we discuss the compilation of Fpm+ to Fpm/AM and give the translation rules. Compilation is comparable to the code generation phase of standard compilers; Fpm+ being quite close to a parse tree and Fpm/AM being at the level of an assembly language. The translation rules we give are at an intermediate level [compare with Augusston 88, Peyton Jones 87 & 88]; they include some implementation detail, but omit what can easily be filled in by a compiler writer.

A5.1 Phase 1 – "Lambda Lifting"

The first phase performs three actions, all of which involve *lambda lifting* [Peyton Jones 87]; closure generation, local to global function conversion, and suspension generation. This phase introduces two new Fpm+ expressions [4.2.3], *cc* and *cs*, which are variations of the *ct* (construct tuple) expression and build closure and suspension objects respectively.

A5.1.1. Closure Generation

Partial applications in the Fpm+ code are made explicit by constructing a new global *closure* function [A3.1] and replacing the application expression by an expression to construct a closure [4.4.5.6]. A closure function is coded to expect some of its arguments in the *heap* instead of the stack [4.3.1], these arguments being the ones present in the partial application and placed in the closure object. This method of closure generation ensures that every function knows exactly where its arguments are located, either on the stack or in the heap, and no arguments need to be moved on closure application; this differs from other implementations such as the G-machine [Augustsson 88] and is one reason for the performance of Fpm2.

The only expression which can generate a closure is an *af* (apply function). If the function tag is either *ev* (evaluate) with a constant function definition, *ud* (user defined), or *bi* (built-in); then a closure is generated if there are insufficient argument expressions. The closure generation algorithm is sketched in figure A5.1

Suspension and Closure Transformations

$$(\text{ct } e_1 \ \ldots \ e_n) \Rightarrow (\text{ct } (\text{cs } s_1 \ l_{1,1} \ldots \ l_{1,m}) \ldots \ (\text{cs } s_n \ l_{n,1} \ldots \ l_{n,m}))$$

where

$$\begin{aligned}
\{l_{i,1} \ \ldots \ l_{i,m}\} &= \text{LV}(e_i) \\
ac_i &= \text{number of } l_i \\
body_i &= \text{Renumber}(e_i, \ [l_{i,1} \Rightarrow 1, \ \ldots, \ l_{i,m} \Rightarrow ac_i, \\
&\qquad\qquad x > m_i \ \Rightarrow x - ac_i]) \\
s_i &= \text{NewFunction}(ac_i, \ ac_i, \ body_i)
\end{aligned}$$

$$(\text{af tag } e_1 \ldots e_n) \Rightarrow (\text{cc } c_i \ l_1 \ldots \ l_n), \text{ if } n < \text{arity(tag)}$$

where

$$\begin{aligned}
\{l_1 \ldots \ l_{heap}\} &= \bigcup_{i=1}^{n} \text{LV}(e_i) \\
ac &= \text{number of } l_i \\
a_i &= \text{Renumber}(e_i, \ [l_1 \Rightarrow 1, \ \ldots, \ l_{heap} \Rightarrow ac, \\
&\qquad\qquad x > heap \ \Rightarrow x - ac]) \\
total &= heap + \text{arity(tag)} - n \\
body &= (\text{af tag } a_i \ldots a_n \ (\text{lv } heap+1) \ldots (\text{lv } total - heap)) \\
c_i &= \text{NewFunction}(total, \ heap, \ body)
\end{aligned}$$

Auxiliary Functions

$\text{LV}(e, \rho)$	≡	set of local values in ρ used in e
$\text{Renumber}(e, \text{map})$	≡	expression e with all local value references renumbered according to map. This is required as Fpm2 uses numbers and not symbolic names.
$\text{NewFunction}(na, nh, e)$	≡	Define new global function with na arguments, of which nh are on the heap, and with body e.

Figure A5.1

A5.1.2. Local to Global Function Conversion

This process is standard *lambda lifting* [Peyton Jones 87]. In Fpm* local function definitions are indicated by *ev* function tags with constant arguments [4.2.3.5], these are simply lifted out and appended to the function list [4.2.1]. The *ev* tag is replaced by a *bi* one referring to the new global function, or by an *ev* tag containing an expression which constructs a closure if the constant function defintion refers to values in the enclosing environment. Note: if the local function definition is also partially applied [A5.1.1], then the two operations of closure generation and local to global lifting are combined and a *single* new function generated.

A5.1.3. Suspension Generation

To conform to Hope+C semantics the Fpm+ expression *ct* (construct tuple) must not evaluate its arguments when constructing the tuple [4.3]. This is made explicit by lifting out the sub-expressions of the *ct* expression and replacing them by expressions to construct suspension objects. This process is analogous to the previous two and is sketched in figure A5.1.

A5.2 Phase 2 – Fpm/AM Generation

The second phase converts the processed Fpm+ code into Fpm/AM instructions. As all partial applications and suspensions have been made explicit by phase one this process is straightforward. Each function definition in the function list [4.2.1] is compiled separately according to its type; standard, closure, or suspension; and named according to its position in the function list so that code for *bi* function tags can refer to them easily.

A5.2.1 Representation of the Environment

We assume local values are located on a downward growing stack [4.3.1] as is the case in the current Fpm/CM [4.4]. Function arguments are pushed in reverse order onto the stack and local values pushed in the order they are produced, the function return address is located between the arguments and local values on the stack. This is a standard stack layout but gives a non-contiguous mapping from local value number to a stack relative offset, which changes each time a local value is added or removed, when no frame pointer is used as in Fpm/CM [4.4]. We represent the current environment by a tuple, ρ:

(na, nh, nl, d, map)

where:

na	is the total number of function arguments
nh	is the number of arguments passed in the heap (closure/suspensions only)
nl	is the number of local values
map	is a mapping from local value number to stack offset from return address

The initial environment for a function is given by:

initialstate(na, nh) = (na, nh, 0, 0, [nh+1 ⇒ -1, ..., na ⇒ nh-na])

With this environment definition the current location of a local value *n* is given by the function:

loc(n, ρ) = if n < ρ.nh
then hp(n)
else sp(ρ.d - ρ.map[n])

Finally, the environment is extended to incorporate *n* new local values by the function:

extend(n, (na, nh, nl, d, map)) = (na, nh, nl+n, d+n, map + [nl+1 ⇒ d+1, ..., nl+n ⇒ d+n])

A5.2.2 The Rules

The translation rules $\mathcal{F}$, $\mathcal{C}$ and $\mathcal{S}$ are used to translate functions, closure functions and suspension functions respectively. These three setup the initial environment, output the function header and return instruction and use the $\mathcal{E}$ rule to translate the function body. The argument count (*ac*), heap argument count (*hc*), body expression (*body*), function number (*number*) and name (*name*) are extracted from the function list:

$$\mathcal{F}[\![(ac\ body\ name\ number)]\!] = \text{fun FUN}\ number, ac \\ \text{equate FUN}\ number, name \\ \mathcal{E}[\![body]\!]\ \text{initstate}(ac, 0), \text{work} \\ \text{retn}\ 0, ac$$

$$\mathcal{C}[\![(ac\ hc\ body\ number)]\!] = \text{fun FUN}\ number, ac \\ \mathcal{E}[\![body]\!]\ \text{initstate}(ac, hc), \text{work} \\ \text{retn}\ 0, ac - hc$$

$$\mathcal{S}[\![(hc\ body\ number)]\!] = \text{sfun FUN}\ number, hc \\ \mathcal{E}[\![body]\!]\ \text{initstate}(hc, hc), \text{work} \\ \text{sretn}\ 0, 0$$

The $\mathcal{E}$ rule translates Fpm⁺ expressions, together with the *cc* and *cs* extensions. In addition to the expression it takes the current environment, ρ, and the required destination of the expression result, δ, which is initially the *work* register [4.3.1]. The Fpm/AM code produced by these rules is correct but inefficient, standard code optimization techniques may be used to improve it. In the following rules all L_i are assumed to be taken from a source of unique label names.

Compilation of *lv*, *cv*, and *if* expressions is straight forward. Note the two rules for *if* as it is used for both conditional and multi-way branch forms:

$$
\begin{aligned}
\mathcal{E}[\![(\text{lv } n)]\!]\,\rho,\delta &= \text{move loc}(n,\rho),\delta \\
\mathcal{E}[\![(\text{cv } c)]\!]\,\rho,\delta &= \text{move lit}(c),\delta \\
\mathcal{E}[\![(\text{if } p\ e_1\ e_2)]\!]\,\rho,\delta &= \mathcal{E}[\![p]\!]\,\rho,\text{work} \\
&\quad \text{jumptrue work},L_i \\
&\quad \mathcal{E}[\![e_1]\!]\,\rho,\delta \\
&\quad \text{jump } L_j \\
&\quad \text{label } L_i \\
&\quad \mathcal{E}[\![e_2]\!]\,\rho,\delta \\
&\quad \text{label } L_j \\
\mathcal{E}[\![(\text{if } p\ e_1 \ldots\ e_n)]\!]\,\rho,\delta &= \mathcal{E}[\![p]\!]\,\rho,\text{work} \\
&\quad \text{case work},n,L_t \\
&\quad \text{label } L_t \\
&\quad \text{addr } L_1 \\
&\quad \ldots \\
&\quad \text{addr } L_n \\
&\quad \text{addr } \mathit{CaseError} \\
&\quad \text{label } L_1 \\
&\quad \mathcal{E}[\![e_1]\!]\,\rho,\delta \\
&\quad \text{jump } L_e \\
&\quad \ldots \\
&\quad \text{label } L_n \\
&\quad \mathcal{E}[\![e_n]\!]\,\rho,\delta \\
&\quad \text{label } L_e
\end{aligned}
$$

The next three rules cover the *ct*, construct tuple, expression and its two variations, *cc* for closure construction and *cs* for suspension construction. The value *SuspensionTag* represents the tag value for suspensions [4.4.5.7], closures and tuples have a tag of zero which therefore does not need to be added in!

$$
\begin{aligned}
\mathcal{E}[\![(\text{ct } e_1\ \ldots\ e_n)]\!]\,\rho,\delta &= \mathcal{E}[\![e_1]\!]\,\rho_0,\text{push} \\
&\quad \ldots \\
&\quad \mathcal{E}[\![e_n]\!]\,\rho_{n-1},\text{push} \\
&\quad \text{claim } n+1 \\
&\quad \text{move pop, pusheap} \\
&\quad \ldots \\
&\quad \text{move pop, pusheap} \\
&\quad \text{move } n, \text{pusheap} \\
&\quad \text{movehp } \delta \\
&\quad \textbf{where} \\
&\quad (na, nh, nl, d, map) = \rho \\
&\quad \rho_i = (na, nh, nl, d+i, map)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![(\text{cc } f\ p\ e_1\ \ldots\ e_n)]\!]\,\rho,\delta = {} & \mathcal{E}[\![e_1]\!]\,\rho_0, \text{push} \\
& \ldots \\
& \mathcal{E}[\![e_n]\!]\,\rho_{n-1}, \text{push} \\
& \text{claim } n+3 \\
& \text{move pop, pusheap} \\
& \ldots \\
& \text{move pop, pusheap} \\
& \text{move lit}(p), \text{pusheap} \\
& \text{move lab}(f), \text{pusheap} \\
& \text{move } n+2, \text{pusheap} \\
& \text{movehp } \delta \\
& \textbf{where} \\
& (na, nh, nl, d, map) = \rho \\
& \rho_i = (na, nh, nl, d+i, map)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![(\text{cs } f\ e_1\ \ldots\ e_n)]\!]\,\rho,\delta = {} & \mathcal{E}[\![e_1]\!]\,\rho_0, \text{push} \\
& \ldots \\
& \mathcal{E}[\![e_n]\!]\,\rho_{n-1}, \text{push} \\
& \text{claim } n+2 \\
& \text{move pop, pusheap} \\
& \ldots \\
& \text{move pop, pusheap} \\
& \text{move lab}(f), \text{pusheap} \\
& \text{move } n+1+\text{SuspensionTag}, \text{pusheap} \\
& \text{movehp } \delta \\
& \textbf{where} \\
& (na, nh, nl, d, map) = \rho \\
& \rho_i = (na, nh, nl, d+i, map)
\end{aligned}
$$

There are two cases for the *tv* (temporary value) expression, one for each of its two uses; defining a new local value, and unpacking a tuple object into its constituent values.

$$
\begin{aligned}
\mathcal{E}[\![(\text{tv } 1\ e_1\ e_2)]\!]\,\rho,\delta = {} & \mathcal{E}[\![e_1]\!]\,\rho, \text{push} \\
& \mathcal{E}[\![e_2]\!]\,\rho_1, \delta \\
& \text{add lit(1), sp} \\
& \textbf{where} \\
& \rho_1 = \text{extend}(1, \rho)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![(\text{tv } n\ e_1\ e_2)]\!]\,\rho,\delta = {} & \mathcal{E}[\![e_1]\!]\,\rho, \text{work} \\
& \text{index work, 1, push} \\
& \ldots \\
& \text{index work, } n, \text{push} \\
& \mathcal{E}[\![e_2]\!]\,\rho_1, \delta \\
& \text{add lit}(n), \text{sp} \\
& \textbf{where} \\
& \rho_1 = \text{extend}(n, \rho)
\end{aligned}
$$

Finally, there are three cases for function application. The rule for the *bi* tag does not handle the case where the builtin function exists as an inline Fpm/AM instruction, this is a trivial extension and is omitted here for clarity.

$$\begin{aligned}
\mathcal{E}[\![(\text{af (ud } m) \; e_1 \ldots e_n)]\!]\, \rho, \delta \;=\; & \mathcal{E}[\![e_1]\!]\, \rho_0, \text{push} \\
& \ldots \\
& \mathcal{E}[\![e_n]\!]\, \rho_{n-1}, \text{push} \\
& \text{call FUN}\, m, n \\
& \text{move work}, \delta \\
& \textbf{where} \\
& (na, nh, nl, d, map) \;=\; \rho \\
& \rho_i \;=\; (na, nh, nl, d+i, map)
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![(\text{af (bi } m) \; e_1 \ldots e_n)]\!]\, \rho, \delta \;=\; & \mathcal{E}[\![e_1]\!]\, \rho_0, \text{push} \\
& \ldots \\
& \mathcal{E}[\![e_n]\!]\, \rho_{n-1}, \text{push} \\
& \text{call BI}m, n \\
& \text{move work}, \delta \\
& \textbf{where} \\
& (na, nh, nl, d, map) \;=\; \rho \\
& \rho_i \;=\; (na, nh, nl, d+i, map)
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![(\text{af (ev } e_f) \; e_1 \ldots e_n)]\!]\, \rho, \delta \;=\; & \mathcal{E}[\![e_1]\!]\, \rho_0, \text{push} \\
& \ldots \\
& \mathcal{E}[\![e_n]\!]\, \rho_{n-1}, \text{push} \\
& \mathcal{E}[\![e_f]\!]\, \rho_n, \text{work} \\
& \text{apply work}, n \\
& \text{move work}, \delta \\
& \textbf{where} \\
& (na, nh, nl, d, map) \;=\; \rho \\
& \rho_i \;=\; (na, nh, nl, d+i, map)
\end{aligned}$$

A5.3. Optimizations

The above translation rules produce correct, if inefficient, code. A number of standard compiler techniques can easily be applied by the code generator to produce better code. Here we mention just three which are more functional language specific.

A5.3.1. Optimization of Head/Tail Accesses

The compilation of pattern matching often produces codes sequences of the form:

```
if isnil alist
then expression involving the head and/or tail of alist
else expression
```

Our compiler optimizes this code by combining the extraction, but not the evaluation if suspended, of the head and/or tail with the test for an empty list.

A5.3.2. Tuple/Closure/Suspension Construction

The translation rules given above first evaluate each element of a tuple and place its value on the stack; heap space for the tuple is then allocated and the values popped of the stack and onto the heap. This code sequence is safe as it ensures that if evaluating a tuple element entails heap allocation then this does not effect the construction of the tuple. It is however wasteful in the case where no heap allocation occurs when evaluating an element, our compiler detects this case and places the element value directly in the heap without using the stack as intermediate storage.

A5.3.3. Coalescing Claims

If the intervening code between two or more *claim* instructions does not alter the heap in anyway, then the *claim*s can be coalesced into a single claim at the start of the code sequence. As each claim involves checking that there is sufficient heap space available and initiating garbage collection if not, this optimization can save a large number of tests and conditional branches. This optimization was added to our compiler after it was first written and produced a marked increase in performance.