Theoretical Foundations for Practical

'Totally Functional Programming'

Colin John Morris Kemp

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in November 2007

School of Information Technology and Electrical Engineering

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the General Award Rules of The University of Queensland, immediately made available for research and study in accordance with the *Copyright Act 1968*.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material.

Statement of Contributions to Jointly Authored Works Contained in the Thesis

No jointly-authored works.

Statement of Contributions by Others to the Thesis as a Whole

The concept of TFP as described in Chapter 2 is the product of Professor Paul A. Bailes. Certain derivation techniques presented in Chapter 5 are the product of various researchers as cited. Other miscellaneous results from the literature appear throughout this work, and also have been cited where used.

Statement of Parts of the Thesis Submitted to Qualify for the Award of Another Degree

None.

Published Works by the Author Incorporated into the Thesis

P. Bailes and C. Kemp, Formal Methods within a Totally Functional Approach to Programming, in Formal Methods at the Crossroads: from Panacea to Foundational Support, B.K. Aichernig and T. Maibaum Eds., 10th Anniversary Colloquium of UNU/IIST, The International Institute for Software Technology of The United Nations University, Springer, 2003, 287-307

P. Bailes and C. Kemp and I.D. Peake and S. Seefried, Why Functional Programming Really Matters, in Proceedings of the 21st IASTED International Multi-Conference on Applied Informatics, Acta Press, 2003, 919-926

P. Bailes and C. Kemp, Obstacles to a Totally Functional Programming Style, in Proceedings of the 2004 Australian Software Engineering Conference, IEEE, 2004, 178-189

P. Bailes and C. Kemp, Fusing Folds and Data Structures into Zoetic Data, in Proceedings of the IASTED International Conference on Software Engineering 2005, ACTA Press, 2005, 299-306

The derivation techniques for TFP-style programs covered in the above papers also appear in Chapter 5.

Additional Published Works by the Author Relevant to the Thesis but not Forming Part of it

P. Bailes and C. Kemp, Integrating Runtime Assertions with Dynamic Types: Structuring Derivation From an Incomputable Specification, in Proceedings of the 27th Annual International Computer Software & Applications Conference, IEEE, 2003, 520-526

Acknowledgements

I would like to take this opportunity to express my profound thanks to Professor Paul Bailes, particularly for innumerable stimulating philosophical and technical discussions regarding Totally Functional Programming. His good humour, generosity, enthusiasm, and insightfulness have made my candidature under him truly privileged.

I would also like to acknowledge the patience of my advisory team, the University, and my family that has made this work possible. I also single out Dr Ian Peake and Dr Peter Robinson for special mention for their efforts in my aid.

Finally but far from unimportantly, this research was supported financially by an Australian Postgraduate Award, and by the School of Information Technology and Electrical Engineering, The University of Queensland.

Abstract

Interpretation is an implicit part of today's programming; it has great power but is overused and has significant costs. For example, interpreters are typically significantly hard to understand and hard to reason about. The methodology of "Totally Functional Programming" (TFP) is a reasoned attempt to redress the problem of interpretation. It incorporates an awareness of the undesirability of interpretation with observations that definitions and a certain style of programming appear to offer alternatives to it. Application of TFP is expected to lead to a number of significant outcomes, theoretical as well as practical. Primary among these are novel programming languages to lessen or eliminate the use of interpretation in programming, leading to better-quality software. However, TFP contains a number of lacunae in its current formulation, which hinder development of these outcomes. Among others, formal semantics and type-systems for TFP languages are yet to be discovered, the means to reduce interpretation in programs is to be determined, and a detailed explication is needed of interpretation, definition, and the differences between the two. Most important of all however is the need to develop a complete understanding of the nature of interpretation. In this work, suitable type-systems for TFP languages are identified, and guidance given regarding the construction of appropriate formal semantics. Techniques, based around the 'fold' operator, are identified and developed for modifying programs so as to reduce the amount of interpretation they contain. Interpretation as a means of language-extension is also investigated.

Finally, the nature of interpretation is considered. Numerous hypotheses relating to it considered in detail. Combining the results of those analyses with discoveries from elsewhere in this work leads to the proposal that interpretation is not, in fact, symbol-based computation, but is in fact something more fundamental: computation that varies with input. We discuss in detail various implications of this characterisation, including its practical application. An often more-useful property, 'inherent interpretiveness', is also motivated and discussed in depth. Overall, our inquiries act to give conceptual and theoretical foundations for practical TFP.

Keywords

interpretation, functional programming

Australian and New Zealand Standard Research Classifications (ANZSRC)

080203 Computational Logic and Formal Languages 75%, 080308 Programming Languages 25%

Table of Contents

Chapter 1

The problem of interpretation

Interpreters are prevalent, often implicit, and significantly complicate programming

We begin by introducing the concept of interpretation and supplying examples of it. We detail its undesirability and its prevalence in programming, and the existence of more-preferable alternatives to it. Subsequently, in the following Chapter, we discuss how the undesirability of interpretation and the existence of alternatives motivated the development of a novel methodology of programming which aims to be a solution to the problem of interpretation. This Thesis develops the necessary theoretical foundations to make this promising methodology practical. Our work on the methodology's lacunae (which are summarised in Chapter 3) involves both drawing from the relevant literature and significant amounts of original material, and is reported in the main body of this Thesis, Chapters 4 through 7. In those Chapters (respectively) we investigate how to type programs written using the methodology, how to derive those programs, how interpretation relates to language-extension generally, and determine the intrinsic nature of interpretation. The latter leads us to some interesting conclusions regarding the practicality of programming with less, and without any, use of interpretation; and much else besides. The Thesis concludes with a summary, Chapter 8, and a Chapter identifying avenues for further investigation, Chapter 9.

1.1 An introduction to interpretation

Interpretation is prevalent, implicit, and most-often presents as computing using symbols

As discussed below, interpretation is typically complex symbol-based computation, particularly symbols representing programs. However, neither symbolic representations nor complexity are actually required for interpretation; interpretation can be implicit. Furthermore, interpretation is demonstrably prevalent.

It should be noted that in this work we are focussed on interpreters in programs, rather than interpretation as a means of implementing the underlying programming language.

1.1.1 Explicit symbol-based interpretation

Typically, interpreters take as input symbols representing a program

Often, interpretation involves the 'enlivening' of representations, ie simulating execution of programs that are supplied as inert symbols. Consider the following interpretive program-fragment, which defines then uses a basic Reverse-Polish calculator:

```
Operations ::= clear | print | zero | one | add | sub | double

Calc (cons e r) ≙ Decode e; if r != nil then Calc r

Decode e:Operations ≙
        if e == clear then  S := emptystack
        else if e == zero then  S := (push 0 S)
        else if e == one then S := (push 1 S)
        else if e == double then S := push ((top S) * 2) (pop S)
        else if e == add then S := push ((top S) + (top (pop S))) (pop (pop S))
        else if e == sub then S := push ((top S) - (top (pop S))) (pop (pop S))
        else if e == print then printline (top S)

Calc [clear, one, zero, add, print, clear, one, zero, one, sub, add, double, print]
```

We assume that a variable 'S' has been defined previously, together with the operations 'push', 'pop', 'top', 'printline'; and also 'emptystack'.

In the above program-fragment, symbolic data, the 'Operations', are defined. An interpreter on them, 'Decode', is also defined. This interpreter takes a symbol, one of the 'Operations', and pattern-matches it to determine which semantics to execute. The function 'Calc' is defined in terms of this interpreter, which is applied to each member in turn of a list supplied as an argument. The last line in the program-fragment is a call to 'Calc' with a given list.

The above example reflects a common form of interpretation, where symbols given as input represent a fixed program ('clear; one; zero; add; print; clear; one; zero; one; sub; add; double; print'). These symbols are passed to a fixed interpreter ('Calc'), which acts (by use of 'Decode') on each of the symbols, to carry out behaviours identical to those of the constructs they represent (and, are named after).

1.1.2 Implicit symbol-based interpretation

Not all symbol-based interpretation is explicit

Some program-fragments may obviously contain interpreters, for example an interpreter for 'Pascal' written in 'C'; or the 'calculator' example above. However, interpretation can be more implicit than this. As we discuss later, there exist cellular automata that take symbols (cell states) representing functions and their inputs. While the computation done by each cell is very simple, combinations of identical cells need not be: even some very basic cellula automata have so-called universal interpretive power (see [Wol02]), ie can compute whatever a Universal Turing Machine can. Similarly, the ability exists to build a universal interpreter on symbolic representations of both a program and its input using a system based on Post Correspondence (see eg [CS68], [HU79]), or grammars ([HU79]). Additionally, the Turing-completeness of Partial Recursive Functions is also well-known. Finally, one may be surprised to learn that mere addition, multiplication and exponentiation on integers (which are symbols) can be used to form a polynomial equation, a Diophantine equation, of universal interpretive power (see for example [Jon82] or [Mat96]). One would not at first glance recognise instances of any of the above systems as being or containing interpreters, nor inputs supplied to them as being representations of programs.

Further, very common yet simple interpreters often go unrecognised as such. For example, each 'if..then..' or pattern-matching construct is conceptually an interpreter. These take symbols (eg 'true' or 'false') which can be viewed as representing programs whose semantics are the 'branches' of the constructs.

1.1.3 Asymbolic interpretation

The existence of asymbolic interpreters indicates that interpretation, while typically, is not always associated with symbol processing

So far in our presentation, interpreters have been presented as symbol-processing constructs (possibly constructed from simple parts) that take as input symbolic representations of programs, and then simulate the program. However, not all interpreters involve symbol processing. Combinations of simple asymbolic building-blocks can turn out to be interpretive. For example, simple combinatory functions (raw 'behaviours') can be used to form programs of universal interpretive power ([Bar84, pp. 179-183]). Further, one is able to simulate that classical interpreter, the Universal Turing Machine, using analog systems such as Ordinary Differential Equations (proof in [Bra95]).

1.1.4 Prevalence of interpretation

Interpretation is very common, and appears even in simple programs

Interpretation is exceedingly prevalent in today's programs, even simple ones. As discussed above, 'if..then..' constructs and pattern-matchers can be seen to be interpreters. Such constructs appear throughout programs in conjunction with data (ie symbols) and data-structures. Numbers are symbols, as are Booleans, lists, trees, etc; and require interpretation (ie pattern-matching) to use. For example:

add 0 n ≙ m

add (succ m) n ≙ add m (n+1)

is conceptually interpretive. A symbolic pattern-match is done on the first argument to 'add', 'zero' representing (and is interpreted to give) 'm', and non-zero values 'add m (n+1)'.

1.2 Interpretation is a problem

Interpretation is a significant source of unnecessary practical and conceptual complexity of software

Interpretation is a problem, at least when used where better alternatives exist. While interpretation gives a programmer access to maximally-powerful computation, in the sense of the Church-Turing Thesis, recourse to the power of interpretation (with its associated costs) is not always required. Interpretive code can be criticised on the grounds of transparency, maintainability, analysability and efficiency, when compared to the alternative of direct definition. To begin, we illustrate this by comparing the interpretive 'calculator' program given previously with a less-interpretive alternative.

1.2.1 Alternative 'Calculator'

Definitions are alternatives to interpreters

A token of program text, a construct, can be 'declared' (ie introduced by means of a data-type clause) to be an inert symbol, or 'defined' (ie bound by a definition) to have semantics, the semantics typically being that of a statement or function. For example, in the 'calculator' program-fragment presented earlier, 'clear' is declared to be a symbol, and is interpreted to give it semantics of clearing the stack. However, rather than being a symbol, it could have been defined to have the above semantics. More generally, the subprogram, 'clear, one, zero..' does not need to be represented by symbols, requiring interpretation, but instead can have its constructs defined so they immediately have their appropriate semantics:

```
clear ≙ S := emptyStack

zero ≙ S := (push 0 S)

one ≙ S := (push 1 S)

double ≙ S := push ((top S) * 2) (pop S)

add ≙ S := push ((top S) + (top (pop S))) (pop (pop S))

sub ≙ S := push ((top S) - (top (pop S))) (pop (pop S))
```

print ≜ printline (top S)

clear; one; zero; add; print; clear; one; zero; one; sub; add; double; print

This program we say is in ‘definitional’ style, as the constructs are directly defined to have their desired semantics, rather than the semantics arising from interpretation. In the sequel, we compare the above program-fragment to the earlier interpretive one, to illustrate the general differences between the two styles.

1.2.2 Practical problems with interpretation

Pragmatically, programs that use interpretation are more complex and harder to analyse than ones which use direct definition of semantics

Comparing the two versions of the ‘calculator’ program above, certain differences are apparent which hold true more generally as well. With the definitional version one can read the sub-program ‘clear; one; zero;..’ and understanding its meaning immediately from the definitions of each of those constructs. In contrast, with the interpretive version, one has to in effect manually execute the program to see what happens; to determine what order the subprogram will be executed in (eg right-to-left or left-to-right), and what semantics each construct will be interpreted to have. With interpretation, one must also convince oneself that a given symbol will be consistently interpreted to have some semantics, or determine how the semantics depends on the state.

While the interpreter in the simple ‘calculator’ program may not be particularly complicated to analyse, it is easy to imagine far more complex examples. Rather than writing a program directly in ‘Pascal’, one could write an interpreter for ‘Java’ in ‘Pascal’, and one for ‘Pascal’ in ‘Java’. Using the first to interpret the source-code of the second, one could then supply a symbolic representation of the program as input. The end result is a ‘Pascal’ program of significant complexity from the point of view of runtime analysis. Trying to understand what was ‘really going on’ from the symbols being processed would be nearly impossible. Analysis of interpreters can be very difficult (it is undecidable, in general) as one can need to prove properties of the runtime state. With definition, in contrast, the corresponding analysis is trivial.

The underlying reason for interpreters being harder to understand than definitions is that both the data-structure being interpreted, and the interpreters themselves, are able to be passed-around and even built at runtime. Interpreters follow the dynamic structure of the program, rather than its static structure (unlike definitions). This can result in the static structure of the program having little direct correlation with its runtime behaviour. Interpretation also adds an extra layer of abstraction and permits complicated dependencies and potential interrelations: the relationship between a construct (represented as a symbol) and its interpreted behaviour can be indirect and varying. With definition, the relationship is direct and constant.

These difficulties of analysis arguably have general impact: it is hypothesised in [HS93] that software quality is correlated with potential local verification effort. They also have more specific impacts with regards to the compiler or other tools. As discussed above, automatic analysis of interpreters is difficult and limited by decidability. Typically, program-analysers won't even try to determine what the interpreter does, even if the analysis would not be particularly difficult for the given interpreter. In contrast, the languages and tools readily handle definitions.

As well interpreters being typically opaque to languages and tools, the real 'meaning' of programs represented by symbols is invisible to them as well. Interpreting a program results in forgoing most type checking, compiler optimisations etc of it. The program being interpreted is merely a data-structure to the compiler. It will be only type-checked as such. The interpreted program won't be the target of optimisation done by the compiler. Debuggers for the language won't achieve their full functionality on the interpreted code; similarly for the profiler and other language tools. The symbols representing the program will be only treated as symbols.

Another, and well-known, practical difference between definition and interpretation is that interpreted programs will run slower. This is not always a significant concern, however; and is less so now than in the past ([KG95]). As per Moore's Law ([Moo65]), if an interpreted program is too slow by a factor of two to be acceptable, then in a mere 18 months time advances in hardware will mean it will run at an acceptable speed (assuming the criteria for acceptability has not changed).

Finally, writing interpreters is an often lengthy and error-prone process compared to writing equivalent definitions. For example, say one wanted to use a new construct in some (non-trivial) 'C' program. Let the construct be called 'GCD', which finds the greatest common divisor of two numbers. Simply adding a definition of the new construct, so that the token "GCD" refers to a function with the desired semantics, is only a small change. In contrast, writing an interpreter and representing your 'GCD'-containing program as data would involve essentially reimplementing all of 'C'! This reimplementation would likely also have to be verified to ensure it was correct, a non-trivial task. Using interpretation often results in having to re-implement parts or all of the existing language as well as any extensions, resulting in more code to write, maintain and reason about.

1.2.3 Interpretation, conceptually

Conceptually, interpretation is problematic due to the semantics of constructs being determined via an extra level of computation

Philosophically, interpreters for programming languages give semantics to things that are semantically inert, to symbolic data. Interpretation of programs turns the symbols representing the program into the desired behaviour, interpretation 'animates' ('makes zoetic', ie 'enlivens') the symbols. Definition acts differently as it results in constructs actually having the desired semantics (via static binding); a conceptually-preferable solution. What symbols, inert data, 'mean' varies depending on how they are interpreted. A single symbol can and often does 'mean' different things in different contexts; likewise a given interpreter can engage in vastly different semantics depending on the particular symbols passed in. This power is a strength, but also a weakness when only a simple 1:1 mapping between constructs and semantics is required. For such cases, inert symbols needing to be interpreted to give them semantics are less conceptually-desirable than constructs that are (by definition) the desired semantics.

There are additional reasons to view interpretation negatively on conceptual grounds. As discussed in the next Chapter, fault may also be found with interpretation on the grounds of a suspected parallel between programming and language-design. Negatives of interpretation in one domain imply corresponding negatives in the other. As also discussed next, interpretation

involves primitive data, which is a demonstrable redundancy in much of programming and adds unnecessarily complications to programming languages and programs.

1.3 Conclusion

Due to its various undesirable features, minimising the use of interpretation is desirable

Interpretation appears in numerous computational systems, and can take various guises (ie symbolic or asymbolic). Its prevalence in today's programs and numerous associated costs render it problematic, at least to the degree that more-suitable alternatives are not taken advantage of.

Due to the variety of forms in which interpretation appears and the lack of apparent commonalities between them, no characterisation of interpretation is immediately suggested. While interpretation is typically explicit and symbol-based, as we have discussed above this is not always the case and hence cannot form the basis of a characterisation. Consequently, until our detailed investigation of it in Chapter 7, interpretation necessarily remains an informal concept. In that Chapter a characterisation of interpretation is developed, one which confirms that what here and in later Chapters we consider to be interpreters, really are. Our intuitions regarding the existence and forms of alternatives to interpreters also find confirmation in Chapter 7.

Chapter 2

Totally Functional Programming: an introduction

TFP is a methodology that attempts to minimise interpretation in programming

The phrase "Totally Functional Programming" (TFP) is the name given to a particular methodology that is a promising solution to the problem of interpretation in programming. As a body of knowledge it contains principles, rules and methods developed primarily by P.A. Bailes from the mid-1980s to early-2000s. TFP was developed from a number of observations that illustrate interpretation's problematic nature and indicate the existence of alternatives to it. Its proponents expect TFP to, maximally, engender a fundamentally changed view of computing, programming, and programming languages; both in terms of theory and practice.

This Thesis documents certain work the author has carried out on outstanding topics in TFP. In this Chapter we introduce TFP as a synthesis of certain key observations. The core outcome of this synthesis is the primary thesis of TFP: a belief in the undesirability of interpretation despite its often-implicitness, and belief in the existence of alternatives (in at least some cases) to it. We then conclude the Chapter with a summary of the expected outcomes from development and application of TFP.

Having introduced TFP, in the subsequent Chapter we detail the outstanding issues and discoveries still required in order to fully realise it. The remainder of the Thesis offers solutions for selected of those outstanding issues. Portions of those solutions have previously found published form, specifically as part of [Bai01], [BK03a], [BKPS03], [BK04], and [BK05] which also contain the essentials of the TFP methodology.

Finally, we wish to point out that this 'totally functional programming' of Bailes should be distinguished from the 'total functional programming' of D. Turner ([Tur04]), which is concerned solely with totality, rather than interpretation. That paper does however independently uncover some of the same benefits for programming only with total functions as are in TFP, plus others besides.

2.1 Affinities of programming and language-design

Programming involves language-extension

A significant observation by Bailes that lead to TFP was a certain interesting fact, publicised in [BCP94]: definitions, under a denotational 'meaning' function representing a language, result in the remainder of the code having a modified 'meaning' function (ie modified language) applied to it. In other words definitions, nominally programming constructs, can be seen as specifying language modifications.

Taking this as evidence for a wider correspondence between what is 'program' versus what is 'language', Bailes proposed that programming could be seen as a language-design activity, ie that programming and language-design could to some extent be unified. Supporting this, apparent parallels had been noted ([Bai86]) between programs and languages, with regards to design and evaluation criteria. Additional parallels can also been drawn between program and language longevity, distribution and complexity. The full argument of Bailes can be found in [BKPS03].

A direct consequence of the postulated correspondence is that (at least some) programming techniques, such as interpretation, should be able to be evaluated against language-design criteria. Further, program-quality should be directly impacted by the quality of the language-extension activities undertaken by parts of the program.

2.2 Interpretation and language-extension

Interpretation is a relatively poor means of language-extension, hence is bad programming

It was observed by Bailes that language-extension by definition is to be preferred over that by interpretation, for reasons which have appeared in the previous Chapter. Given that programming is language-extension, and interpretation is by comparison to definition poor language-extension, the conclusion drawn is that interpretation is bad programming.

2.3 No data may mean no interpreters

Data complicates languages, and as it is essentially only used by interpreters, removing it from languages may remove the programmer's ability to write interpreters

Bailes has observed ([Bai01]) that, essentially, data is only used by interpreters. The exceptions are intrinsically data-centric, such as:

- Mathematical operations,
- Interfacing with systems that require data, and
- Operations that merely build or break down non-atomic data-types

Without data, ie symbolic representations, it is hard to conceive of how a symbol-processing function (an interpreter) can be written. As all common interpreters appear to use data, removing data from languages may remove the ability to write interpreters. Whether not being able to write interpreters is in some way undesirable is an interesting question: one may not often in practice implement a Turing Machine or a self-interpreter, but 'smaller' interpreters may perhaps be required on occasion.

Additionally, it should be noted that primitive data is superfluous: one can have a Turing-complete programming language that doesn't contain primitive data. An example of this ([Bar84]) is the lambda calculus, which contains only functions. The linguistic 'schism' (as per [GH80]) between primitive data and functions, is common in programming languages but is unnecessary and may permit interpretation.

2.4 Particular examples of relatively data-less functional programming

Certain interesting examples of higher-order functional programming appear to illustrate how to program without, or with less, primitive data and interpreters

Bailes has identified ([Bai01] etc) several unusual program fragments and techniques that do not appear to contain the interpreters one commonly sees in such situations. For many of them neither primitive data nor interpreters acting on such, or tests of any kind, are apparent. Instead, there is a reliance on first-class functions. Such examples are taken to be exemplars of how to program without (or at least, with less than the usual) interpretation. These interpretation-

eschewing programming exemplars are presented below, written in the lambda calculus. The lambda calculus will be used throughout this Thesis, and we have included a brief introduction to it as an Appendix.

2.4.1 Church Naturals

Programs written using Church Naturals appear to use less interpretation than usual

The long-known Church Naturals (also known as 'Church Numerals' or 'Church Numbers'), and related operations, were the first programming examples noted by Bailes to be apparently interpretation-avoiding. They were introduced by, and named after, A. Church in [Chu33], and are functional representations of the Naturals. They take the form:

zero ≙ (λf,x. x)

one ≙ (λf,x. f x)

two ≙ (λf,x. f (f x))

three ≙ (λf,x. f (f (f x)))

etc

In other words, 'zero' is represented by a function which applies its first argument to its second zero times; 'one' the function that does that application once, 'two' the function that does it twice, and so-forth. The original concept is credited to Wittgenstein, who published in 1921 the series:

x, Ω'x, Ω'Ω'x, Ω'Ω'x, ..

where

Ω'

denotes an operation. The explanation given of the above is (in translation, [Wit74]): "a number is the exponent of an operation" (6.021). However, it is certainly quite possible that the core idea predates him.

The link with TFP is as follows.

Usually, Naturals are interpreted to be iterators. For example, in the below a Natural, 'n', is interpreted by a 'for' loop to implement iteration:

```
iterate n initialstate changestate ≜
        state := initialstate
        For x = 1 to n
                state := (changestate state)
        Rof
        Return state
```

In a functional programming language the equivalent is:

```
iterate n changestate initialstate ≜    if n == zero then initialstate
                                        else if n == succ m then (changestate
                                                (iterate m changestate initialstate))
```

Or, utilising the common 'pretty-syntax' for pattern-matching:

```
iterate zero changestate initialstate ≜ initialstate

iterate (succ m) changestate initialstate ≜ changestate (iterate m changestate initialstate)
```

However, alternatively, Naturals can defined to be iterators, for instance:

```
zero ≜ (λchangestate, initialstate. initialstate)

one ≜ (λchangestate, initialstate. changestate (initialstate))

two ≜ (λchangestate, initialstate. changestate (changestate (initialstate)))

..
etc
```

These are the Church Naturals (note that variable name differences are not significant), and obviously do not need 'iterate' or any other interpreter applied to them to do iteration, they can be simply applied immediately. Note specifically the lack of symbol tests and recursion or loops in the Church Naturals, which are hallmarks of interpretation and present in the interpreter 'iterate'.

Rather than having to define each Church Natural explicitly, it is useful to define some functions that produce them from others.

To start with, one can define a Peano ‘successor’ function:

succ n ≜ (λf,x. n f (f x))

or, as per [Chu33]:

succ n ≜ (λf,x. f (n f x))

Addition can also be defined, also as per [Chu33]:

add a b ≜ b succ a

or, using a definition credited to J.B. Rosser in [Kle35a]:

add a b f x ≜ a f (b f x)

These two functions are quite distinct from the standard ways of implementing addition, which involve recursion, and symbol testing. They are instead ‘total’ and free of primitive data and tests on symbols. For example, addition is commonly implemented as:

add a b ≜ if a == 0 then b else add (decrement a) (increment b)

Note the symbol test (for zero), and the recursion: hallmarks of interpretation that are lacking in the Church Natural version. This apparent eschewing of interpretation is carried through to the other arithmetic operations as well. One can define multiplication as:

mul a b ≜ a (add b) zero

or, simply if obscurely: (also credited to Rosser in [Kle35a])

mul a b f ≜ a (b f)

One should note that the implementations of ‘add’ and ‘mul’ also have symmetrical forms, eg:

a succ b = add a b = add b a = b succ a

In general we will explicitly give only one of such variant forms, the other considered as being implied by the symmetric nature of the operation being implemented.

Another interesting operation is exponentiation, for as well as it being definable as:

pow a b ≜ b (mul a) (succ zero)

it can also, surprisingly, be simply implemented as (Rosser, as credited in [Kle35a] again):

pow a b ≜ b a

Note that these ‘pow’ have zero to the power of zero being one, as is common, rather than the equally-acceptable value, zero.

Further up the arithmetic hierarchy, Ackermann's function can also be defined relatively simply, as will be shown later. More complex implementations are required for predecessor and subtraction, definable as:

pred n ≜ secondelement (n (λt. <succ (firstelement t), (firstelement t)>) <zero,zero>)

sub a b ≜ b pred a

where 'firstelement' extracts the first element from a tuple:

firstelement <a,b> ≜ a

This is 'pred' as per its common definition on Church Naturals. It was first defined in [Kle35a], but slightly differently so that 'pred one = one', rather than 'zero'. The reason for this is that the original version of the lambda calculus required all variables of a 'λ' to appear in the body of the term, hence didn't permit 'zero' ie '(λf,x. x)'. Therefore, these operations were for positive integers only. The definition above for 'sub' is taken verbatim from [Kle35a], but here 'sub' can return 'zero'. Note that Church Naturals have no negatives and subtraction here of a greater from a smaller number will return zero. Such subtraction is known as 'proper' subtraction.

2.4.2 Church Booleans

Programs written using Church Booleans appear to use less interpretation than usual

Developed concurrently by A. Church with the Church Naturals (but not presented in [Chu40]), Church Booleans are functions ('(λa,b. a)' and '(λa,b. b)') which can be used to represent Booleans.

The usual way of writing the Boolean operations involves symbol-testing 'if' tests, eg:

or a b ≜ if a == true then true else b

or:

or a b ≜ IF a true b

where

IF x a b ≜ if x == true then a else if x == false then b

However, it is possible to write something equivalent, but without using primitive data or explicit tests. For Church Boolean 'x', as noted by A. Church:

IF x a b = x a b

ie

IF x = x

Hence, one can write:

true ≜ (λa,b. a)

false ≜ (λa,b. b)

or a b ≜ a true b

In other words, rather than interpreting Booleans using 'if' tests, which do testing, Booleans can be defined to be 'if' tests. The resultant Church Booleans don't contain any tests. Other Boolean operations on Church Booleans can also be implemented:

and a b ≜ a b false

not x ≜ x false true

eq a b ≜ a b (not b)

etc

Note that we have just shown that any function on the Booleans can be reimplemented for the Church Booleans, since 'not' and 'and' (which we have just defined for Church Booleans) form a complete basis, as is well-known.

Finally, like we've seen for the Church Naturals, generally multiple equivalent definitions are possible (not all of which involve symmetry). For example:

and a b ≜ a b a

not a ≜ (λx,y. a y x)

eq a b ≜ b (a true false) (b false true)

2.4.3 Church Lists

Programs written using Church Lists appear to use less interpretation than usual

There exists a functional representation for lists, commonly called 'Church Lists'. We are somewhat unsure as to when they were first constructed, but it was no later than [BB85]. Illustrating by example:

[1,2,3]

can be represented by the Church List function:

(λx,y. x 1 (x 2 (x 3 y)))

Church Lists can be constructed via the following functions:

nil ≜ (λx,y. y)

cons e r ≜ (λx,y. x e (r x y))

To extract the head of a list, one can apply (here, 'default' is the value to return if the list is empty):

head m ≜ m (λa,b. a) default

One can compare this to the operation defined for standard (dataful) lists, which uses symbol tests and recursion:

head m ≜ if m == nil then default else if m == cons e r then e

To find the 'tail' of a Church List is also possible albeit more difficult, requiring the same techniques as to define predecessor on Church Naturals.

Similarly, one can compare implementations of the operation called 'reduce', 'foldright', or simply 'fold'. Usually symbol tests and recursion are used, ie:

reduce op base m ≜ if m == nil then base
else if m == cons e r then op e (reduce op base r)

With Church Lists however, no such explicit interpretation is necessary:

reduce op base m ≜ m op base

Many other operations can also be implemented on Church Lists, also with apparently less use of interpretation than when the lists are data:

sum m ≜ m (+) 0

isempty m ≜ m (λe,r. false) true

isin m x ≜ m (λe,r. or (e == x) (r x)) false

append m n ≜ m cons n

reverse m ≜ m snoc nil

where:

snoc e r ≜ (λop,b. r op (op e b))

last m ≜ head (reverse m)

2.4.4 Characteristic Predicates

Programs written using Characteristic Predicates appear to use somewhat less interpretation than usual

The long-known 'Characteristic Predicate' representation of sets has been noted by Bailes as apparently avoiding some of the interpretation that goes into membership testing. Consider the following examples:

empty ≜ (λx. false)

evens ≜ (λx. x mod 2 == 0)

numbersabove5 ≜ (λx. x > 5)

singleton e ≜ (λx. x == e)

Here, each set is represented by a function that computes (usually using symbol tests) whether its argument is in that set. These Characteristic Predicates can be contrasted with a more standard representation of sets as strings, which are explicitly interpreted by a membership tester:

membershiptest s x ≜ if s = "empty" then false
else if s = "evens" then (x mod 2 == 0)
else if s = "numbersabove5" then (x > 5)
..

In this more-common implementation of sets, note the recursion and symbolic pattern-matching on the name of the set. It is this interpretation, of the set-representation, which Characteristic Predicates avoid while still interpretively symbol testing to check for membership.

One should also note that it is possible to come up with Characteristic Predicate implementations of many set operations:

union s1 s2 ≜ (λx. (s1 x) or (s2 x))

intersection s1 s2 ≜ (λx. (s1 x) and (s2 x))

diff s1 s2 ≜ (λx. (s1 x) and not (s2 x))

complement s ≜ (λx. not (s x))

These can likewise be contrasted with the standard way of combining sets possibly containing an infinite number of members. This is to combine two sets into one using a dataful node representing the form of the combination (eg 'union' vs 'intersection'). One must also ensure that the interpretive membership tester contains the appropriate recursive symbol pattern-matching to interpret these representations. As a simple example:

```
Set ::= empty | union Set Set | singleton Element

membershiptest : Set —> Element —> Boolean
membershiptest s x ≜ if s == empty then false
                     else if s == (singleton e) then (x == e)
                     else if s == (union a b) then
                                  (membershiptest a x) or (membershiptest b x)
                     ..
```

2.4.5 Combinator Parsers

Combinator Parsers appear to use somewhat less interpretation than usual

In Combinator Parsers (see eg [Hut89] for examples), each individual production-rule of a grammar is defined to be function, a parser. The alternation and sequencing operations are defined as higher-order functions which combine these individual parsers into a more complex parser. The end result is that the grammar is a parser for itself. In contrast, the standard way of parsing a grammar is to give to a function a piece of structured data representing a grammar, which it then interprets to 'animate' or 'enliven' that grammar into a parser.

2.4.6 Programmed Graph Reduction

Programmed Graph Reduction is a means of implementation of rewrite systems which is apparently less-interpretive than canonical means

Rather than executing lambda calculus programs by representing the terms by data and interpreting them, instead each term can be directly represented by executable code with semantics corresponding to the term, as demonstrated in [Pey87]. Some further details can be found eg in [BPL88]. For a simple example of the general concept, the term '(λa,b. a)' could be represented by code which has the semantics of taking two arguments and returning the first.

2.4.7 Functions representing Reals, for Exact Real Arithmetic

Exact Real Arithmetic can be implemented by representing reals as functions, rather than data-structures

As discussed in eg [BC90], one can represent (the constructive) Reals by functions that, when given a precision, return the real they represent to that precision. Specifically, if 'x' is a real, 'f_x' a functional representation of it, and 'P' a precision, one desires that:

$$| x - F_x(P) | \leq P$$

Rather than representing a Real by a data-structure requiring interpretation to, instead a Real is represented by a function, which returns the Real to as exact a precision as one desires. For general background and motivation for Exact Computation more generally, one can consult [YD95].

2.4.8 Summary

These interesting programming curios use higher-order functions in place of primitive data and tests thereon

What all of these representations have in common is that rather than being symbolic data which is required to be tested ie interpreted, they are instead defined to be functions. In addition, those representations are then something for which auxiliary operations (eg 'add', 'union') seem to be

able to be implemented using less symbol testing and recursion than normal. These exemplars of reduced-interpretation programming are suggestive that primitive data can be systematically replaced by functions, and that interpretation is unnecessary.

2.5 Platonic Combinators and Characteristic Methods

For every abstract class of objects, there exists a single operation capturing its semantic essence

There is a speculative philosophical assertion of Bailes, briefly covered in [Bai01]; that for every conceptually-atomic class of objects (be it Natural numbers, grammars, etc) there exists a single operation which is the sole 'natural' (and 'Platonic', following the philosophy of Plato) semantics for the class. For each instance of the class (eg individual Naturals) that semantics is presentable as a function, a combinator, which also incorporates that particular instance: we give examples below. Numerous concerns need to be addressed and difficult distinctions made for so-called 'Platonic Combinators' to move beyond the speculative stage however.

2.5.1 Examples of Platonic Combinators

Some examples of Platonic Combinators have been proposed, which also appear in the examples of how to program using less interpretation than usual

We have already met a number of Platonic Combinators. The claim made is that the 'natural' operation for Naturals is 'iteration'. Hence, the number '2', for example, has the 'Platonic semantics' of doing two-fold iteration. The function '(λf,x. f (f x))' is therefore a Platonic Combinator, as are all the other Church Naturals. For Booleans the 'natural' operation is 'selection' ('IF'); with the Church Booleans being the Platonic Combinators. For grammars the operation and Platonic Combinators are unknown, but are postulated to entail somehow both 'parsing' (concrete syntax to abstract syntax) and 'printing' (abstract syntax to concrete syntax), the two operations considered by Bailes central to the nature of grammars. For lists, which have an explicit ordering of elements, functions that expose that ordering seems indicated; and the Church Lists are suggested as the Platonic Combinators. Similarly, for sets, which have no ordering of elements, functions that likewise contain no ordering (chronological or otherwise) would intuitively be required, such as Characteristic Predicates.

Interestingly, these identified Platonic Combinators have also appeared in the exemplars of apparently- interpretation-eschewing programming.

2.5.2 Impure Platonic Combinators

Platonic Combinators may involve interpretation

The proposed Platonic Combinators for sets, the Characteristic Predicates, include interpretation in the form of the symbolic equality-test on elements. This is taken as evidence that some abstract object classes may have a Platonic behaviour that is interpretive. Such Platonic Combinators have been called by Bailes 'Impure Platonic Combinators', with the 'impurity' being the presence of interpretation.

2.5.3 Related semantics

Given some semantics, certain other semantics are postulated to be reachable only via interpretive means

Given certain Platonic semantics, it is suggested that any other semantics can be categorised as either 'related' (or 'cohesive'), or 'unrelated' ('non-cohesive'). Specifically, if the new semantics is defined in terms of the Platonic semantics, in some acceptable manner, the new semantics is 'related' to the Platonic semantics. What is 'acceptable' is best illustrated by counterexample. Consider:

(λx. if (f x) > 9 then (f x) else (g x))

By means of a test (ie interpretation), the function will sometimes act like 'f', and sometimes like 'g'; and hence isn't 'cohesive' with either despite being defined in terms of both.

The suggestion is that every Platonic semantics is surrounded by an envelope of 'related' semantics, namely semantics that can be reached from it without use of interpretation. One can also construct envelopes for each of those 'related' semantics, resulting in a hierarchy of 'related' functions, whose root is the Platonic semantics.

2.5.4 Relation to Object-Orientation

Cohesive objects should have one core method, reflecting the Platonic semantics; all other methods should be semantically related to that one

Bailes postulates that Platonic Combinators has overlap with Object-Orientation, in that they appear to share the same concept of 'cohesion'. In Object-Orientation, the more semantically-similar the methods of a class are to each other (or, the less similar to methods in other classes as per [YC79, p106]), the more 'cohesive' the class is. Class cohesion is a well-known quality measure for Object-Oriented systems. Applying the concept of Platonic Combinators tells us that for maximally-cohesive classes all the methods should be related semantically to the Platonic semantics. A method corresponding to the Platonic semantics itself could justly be termed the 'Characteristic Method' of the class, just as one has a 'Characteristic Predicate' for a set.

2.6 Synthesis

TFP is a methodology drawn from a number of precursors, which aims to help programmers produce better programs by minimising their use of interpretation

TFP contains critiques of interpreters (informally, 'symbol-based computation') from a number of angles. Firstly, in the topic of Platonic Combinators is found an interrelationship between interpretation and the quality of (Object-Oriented) code, by reference to the concept of semantic 'cohesion'. Secondly, interpreters require data, and data is an unnecessary complication in programming languages. Thirdly, and more importantly, in TFP we find the notion of a correspondence between programming and language-design combined with critiques of interpretation as a means of language-extension. Specifically, it is argued that all interpreters entail language-extension, not just those explicitly built to do so; and as interpretation is demonstrably a poor means of language-extension, all interpretation is to be avoided. Definition is identified as an alternative.

The claim that all interpretation is language-extension is more significant than might be realised initially.

Firstly, one can consider the results of applying it to typical programs. In every program each and every routine, subroutine, and even primitive that operates on data is, under the claim, doing language-extension: the input to each routine is an encoding of the source-code of some (perhaps quite basic) program, written in some notional programming language. Potentially, there may be as many of those notional programming languages to consider as there are constructs in the program: every 'if' expression and mathematical operation for example involves language-extension, and the language-extension done by a given routine arises from the language-extensions done by its subroutines. Further, note that the inputs to routines are often not literal values, but computed themselves. As the inputs are taken to represent the source-code of programs, the consequence is that those program-representations are being constructed, tested, destructed, and so-forth, at runtime. One has for example situations where a program-representation once produced is executed by an interpreter to produce a new program-representation, in a different language, that depending on some other program-representation may be modified and sent for execution to this other interpreter, or.. (etc). Linguistically things are often very complicated, even without allowing for the fact that the language-extension done by a routine can change during runtime: there may be state or free variables that affect its behaviour. Similarly, the program-representations at various points may contain unbound variables or references to state. Clearly then, interpreters permit a style of programming where the language-extensions are not well-structured.

Secondly, while for interpreters explicitly written to implement languages it may be obvious that definition is an alternative, for other interpreters the existence of definitional alternatives is not so clear. Nevertheless, TFP contains evidence that this form of interpretation can indeed be at least in some cases avoided: the functional-programming exemplars. The exemplars make use of definitions (ie do obvious language-extension) and are reduced-interpretation alternatives of code that used interpreters not obviously doing language-extension (such as the 'multiplication' operation). Hence they offer prima facie support for the idea that all interpreters do indeed do language-extension, and that definitions are available alternatives.

Further, TFP also contains the suggestion that such alternatives to interpretation may be able to be imposed. Interpretation might be able to be banished, its accidental or otherwise introduction into programs prevented, by such means as removing primitive data from programs. It is suggested that such may not lead to excessive restrictions on what can be programmed:

interpretation may be in general unrequired, as long as the language supports first-class functions as required by the exemplars.

As to a formal characterisation of interpretation, TFP contains suggestions as to the intrinsic nature of interpretation (beyond its informal characterisation as 'symbol-based computation'). In the topic of Platonic Combinators we find interpretation appearing as a means of producing 'uncohesive' functions. In addition the apparent overlap between the Platonic Combinators and the apparently- interpretation-avoiding exemplars may also be of relevance.

Finally, TFP recognises that there is a particular style of programming employed by interpretation-eschewing programs. This style as we've seen uses first-class functions instead of primitive-data, and does not use recursion or pattern-matching. This style of programming is claimed as Totally Functional Programming's own, to the extent that it takes its name from this style: TFP is programming that is

Totally

The exemplars of what appears to be interpretation-avoiding programming are taken as evidence that all interpretation-avoiding programs will not contain recursion. Moreover, they are expected to all be 'total', ie terminating on all inputs (perhaps also for all evaluation strategies, not just Normal Order).

Functional

The exemplars all use first-class functions, arguably functions in their 'purest' state.

Totally Functional

Only functions are used; preferably no primitive-data.

Note that Characteristic Predicates, Combinator Parsers and the like are considered to be 'TFP-style' in some impure sense, due to their internal use of some (but less than the usual amount of) data.

In summary, the precursors to TFP are combined to give the assertion that interpretation should be avoided, and that at least it sometimes can be. Interpretation is critiqued by treating it as a means of language-extension, by considering its practical effects, and also by considering 'cohesion'. Preferable alternatives to it were identified, namely definition and also the functional-programming exemplars, with their 'style'. The precursors also supply clues as to interpretation's essential nature.

2.7 Expected outcomes from TFP

There are numerous practical and theoretical outcomes expected to arise from TFP

We divide the expected outcomes of the TFP methodology into three: the practical, the theoretical, and the hardware-related; and detail them below. However, in order for theses outcomes to be achieved, the TFP methodology requires some significant additional development, as we detail in the next Chapter.

2.7.1 Expected practical outcomes from TFP

Practical outcomes expected from TFP include advice, languages, and tools; leading to better programs

Successful application of the TFP methodology is expected to lead to improvements in software quality, and easier programming. Powerful interpretation-aware programming languages and tools are anticipated to be developed.

2.7.1.1 Languages

Many different languages may be possible for practical TFP

There are a range of languages that could be developed based on TFP, depending on their feasibilities. Each language may also have developed a formal semantic model, type system, proof system, verification techniques, translators, libraries, compilation strategies etc.

Languages produced as part of TFP are expected to quite unusual as they are to be both general-purpose languages and sub-recursive. They should also permit some powerful analysis techniques, due to both termination and undecidability issues not arising. Also, importantly, semantic information is expected not be able to be hidden by being encoded as data.

Firstly and most importantly, it is desired to have a TFP language in which interpretive programs are not possible to write. This language may be termed 'sub-interpretive' or 'interpretation-free', as distinct from merely 'sub-recursive', which refers to only some (the universal) interpreters not being able to be written. There may perhaps be more than one such 'pure' language, reflecting different tradeoffs or programming paradigms.

Secondly, languages which do permit interpretation to some extent may also be desirable; especially if the set of interpretation-free functions is too small to be practical to program with. Such 'impure' languages might prevent access to certain interpretation; or could permit all programs but warn about undesirable interpreters.

These languages could be characterised by a number of properties, such as:

A) What constructs or other facilities the language offers programmers to help them build less-interpretive programs, such as libraries of interpretation-free or minimal-solutions.

B) How limited access to interpretation is in the language:

- No interpreters; ie the set of constructs offered by the language when validly combined forms only non-interpretive semantics

- Only the use of built-in (or library-supplied) interpreters, which are of just-sufficient power

- Programmer-built interpreters are permissible but are somehow limited in their complexity to prevent the 'worst' forms of interpretation; eg by being statically 'corralled' in scopes in which restricted rules apply, for example not having access to recursion or loops, or most of the program-state.

- No restrictions

C) What the language does when it detects undesirable interpretation.
It could offer:

- Less-interpretive alternatives

- General hints as to how to avoid the interpretation

- To relax its rules about interpretation, for this particular case

2.7.1.2 Tools

A number of TFP-based tools may be constructed

One may be able to produce tools that do similar jobs to the TFP languages identified above. Tools could be produced that automatically or otherwise detect and/or remove interpretation (or at least 'undesirable' interpretation) from programs. Other tools could help programmers in deriving less-interpretive programs.

Tools that mechanically convert interpretive code into interpretation-free code where possible would be particularly useful. These could form the basis of Design Recovery processes based on extracting semantics hidden as interpreted data from programs. The language-target of the tools could be an interpretation-free TFP language, for in such the use of data to represent semantics is impossible as noted in [BK03a]. Such 'pure' TFP languages could also act as universal Design Recovery target languages for the process of 'reverse-engineering' as well as a language for 'forward-engineering' (see eg [CCM92] for descriptions of both these terms).

It may also be desirable to have tools that do statistical analyses and/or calculate metrics for quality of code, especially the portion thereof that relates to interpretation. In particular, it may well be both possible and useful to rank particular uses or forms of interpretation on a scale, or at least partially-order them. The 'severity' of the interpretation is of interest for when there is no known interpretation-free solution the best interpretive solution should be chosen.

2.7.1.3 Documentation

Educational material will be produced

Documentation educating language designers and programmers as to the insights offered by TFP is anticipated to be produced. This may also include instructions on how to construct TFP languages from common existing programming languages, and also best-practice (least-interpretive) solutions to various problems.

2.7.2 Expected theoretical outcomes from TFP

There are significant theoretical outcomes expected from TFP

The core concern of TFP, interpretation, is somewhat fundamental to computing theory. Hence, significant theoretical outcomes from TFP seem likely. There may also be new discoveries made in the fields of language-design and language-extension while developing languages to support TFP. Specifically, it is expected that TFP languages will be unusually extensible in order to offer programmers maximal support for alternatives to interpretation.

2.7.3 Possible Hardware-based Outcomes

TFP may lead to hardware-based non-symbolic computation

There are possible applications of TFP (as noted in [BK03a]) to analog and hardware engineering and systems engineering in general. TFP languages, furthering existing work (eg [Ghi07]), might act as a design, specification, or prototyping language for digital or analog systems. Further, TFP may permit increases in efficiency, quality, and ease of design; due to the discovery of alternatives to interpreting symbols. Such may include perhaps having the 'wiring' between hardware components change during runtime, corresponding to first-class functions, rather than the components communicating via passing electronic representations of symbols.

More generally, alternatives to symbol-based computation, and non-symbolic representations of knowledge and information, are of broad relevance to Computer Science.

Chapter 3

Remaining work in TFP

TFP, while promising, contains a number of lacunae in its formulation to date

While TFP offers much promise, further research is required. In this Chapter, we discuss the outstanding items of research the author believes are required to be completed in order to fulfil TFP. This Chapter is intended to subsume the questions raised in [Bai01], [BK03a], [BKPS03], [BK04], and [BK05].

We conclude with mention of the particular issues subsequent Chapters work to resolve.

3.1 Principles

A number of theoretical discoveries are required to make TFP practical

Outstanding topics of research in TFP include some significant theoretical items, as we describe now.

3.1.1 Expressiveness

A characterisation is required of how much of programming is interpretive, in order to determine the limits of TFP

To begin with, it is desirable to characterise how much programming (and language-extension, if the two are distinct) can be done in the absence of interpretation. This is clearly an important concern as it determines the limits of TFP. Perhaps only an impractically-small set of input-output mappings are possible to implement without interpretation. Alternatively, perhaps quite a lot of the computational power in languages can be kept available without permitting interpretation.

3.1.2 Data

The concept of 'data' needs to be better understood, if one is to prevent interpretation by its removal

TFP contains the suggestion that removing data will prevent interpreters from being written. It remains to be shown that this is indeed the case. Further, to achieve success with this approach, a suitably-powerful concept of 'data' is required to be developed: for removing primitive data does not suffice. As discussed below, languages such as the lambda calculus contain no primitive data yet can denote interpreters by the use of first-class functions. Identification of the set of functions that can be used as data would be required as a first step.

Note that this concern about the nature of data need not be resolved if a way of removing interpretation other than by removing data can be found.

3.1.3 Interpretation

A characterisation of interpretation is essential for TFP to step from theory into practice

By far the most important discovery yet to be made is a characterisation of interpretation. Without such, we cannot be certain that any proposed TFP languages do prevent or detect all interpretation, or that any TFP-style programs are indeed interpretation-free. In TFP, interpretation is considered to be symbol-processing computation, exemplified by readily-detectable tests on inanimate symbols. However, there is also asymbolic interpretation, which is often less than obvious. A key example of this is the use of functions in place of symbols. The lambda calculus contains only first-class functions, yet has been shown ([Bar84]) to permit construction of a universal interpreter. This is based on the use of Church Naturals, which are shown to have every computable function on Naturals definable on them. Simple illustrations of such are tests for zero and for equality:

iszero n ≜ n (λx. false) true

equal a b ≜ and (iszero (sub a b)) (iszero (sub b a))

Here, 'true' and 'false' are Church Booleans, 'sub' is subtraction on Church Naturals, and 'and' takes and returns Church Booleans. It is also possible to define other arithmetic operations on

Church Naturals, which while eschewing primitive data, do recognisably-interpretive processing. An example of this is using Church Natural and Church Boolean operations ('IF', 'sizero', 'pred' and 'succ') to implement addition on Church Naturals in an interpretive manner:

add a b ≜ IF (iszero a) b (add (pred a) (succ b))

This should be compared to:

add a b ≜ a succ b

It seems that the Church Naturals merely permit less-interpretive implementations to be written. How, precisely, they (and TFP-style programs generally) do so needs to be determined.

The Church Booleans are also quite intriguing. Consider 'IF', perhaps the canonical interpreter, the simplest symbol test. It merely selects between its final two arguments depending on which Boolean symbol passed as its first argument. Yet we have an implementation of it which does not do a symbol testing; specifically 'IF b = b', where 'b' is a Church Boolean. Are Church Booleans therefore themselves interpretive, despite their simplicity?

Some subtle distinctions regarding interpretation in functional programming will have to be made. For example is the function '(λf,g,x. f g x)' an interpreter, or not? For some 'f' and 'g' the result of 'f g' (but not 'f' and 'g' themselves) can be clearly seen to be an interpreter.

Finally, note that one may need to do much more than simply identify the interpretive lambda-terms. Apart from application to other programming languages, there are a number of computational systems (of 'universal' ie Turing-complete computational power) outside of common programming experience. In particular, one has Diophantine equations ([Jon82]), analog (continuous, differential) systems ([Bra95]), and cellular automata (that some are universal is indicated in [BCG82], [Wol02], and [Coo04]). A suitable characterisation of interpretation should hold across this diverse range of systems.

3.1.4 Language-extension via interpretation and definition

An understanding of the interpretive and definitional styles of language-extension will help us understand in detail interpretation's undesirability and may aid its characterisation

The preference for definitional over interpretive language-extension is a key motivation of TFP, yet detail is lacking. A sign of this is the many questions regarding language-extension which remain to be answered. Specifically; what is it exactly that makes interpretive language-extension so bad? Is interpretation done for explicit language-extension distinct in any way from interpretation more generally? Can we determine objectively whether a given program does definitional rather than interpretive language-extension? Is it really true that all interpreters do language-extension; and hence that all interpreters (including even simple 'if' constructs) are able to be meaningfully criticised from the point of view of linguistics? Similarly, whether all (not just interpretive) programming involves language-design remains to be firmly established.

3.1.5 Other

There are numerous other discoveries that would permit a better understanding of some fundamental issues in TFP

A number of varied topics in TFP also require further development. A fuller explanation of how the interesting functional programming examples avoid interpretation is desirable. Additionally, it has been noted by Bailes that the Church Naturals are the only members of the 'Hindley Milner' type '$\forall\alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$', and a similar correspondence holds for the related Church Booleans and Church Lists. The relevance, if any, of this to interpretation remains to be determined. The subject of Platonic Combinators aka Characteristic Methods also needs to be more-fully developed, in particular in matters relating to 'cohesion'. When, exactly, are two methods 'cohesive'? Is there really a hierarchy pertaining to 'cohesiveness', such that eg there exists a method from which printing and parsing of grammars can both be derived? Can the link between Platonic Combinators and interpretation be made precise? Finally, justification is required for functional programming being the paradigm of choice for TFP, given the current popularity of Object-Oriented languages. Functional programming does however have some affinities with software at the enterprise level, as noted in [Mur07].

3.2 Techniques and methods

Certain discoveries would have direct practical applications

Practical discoveries are needed to be made for TFP, in order for its desired outcomes to be achievable. Chief among them is a test (preferably computable, and syntactic) for the presence of interpretation. With such, interpretation-aware languages and tools become possible. Further, if possible, a set of language primitives that can be combined into programs to span precisely the space of non-interpretive programs should be constructed: such can form the core of interpretation-free TFP languages. Algorithms for ranking or stratifying uses of interpretation would also be desirable.

Importantly, methods for derivation of alternatives to common uses of interpreters are needed. In particular, it is required to have techniques sufficient at least to derive the exemplars of TFP-style programming. These techniques could find practical form incorporated into languages or tools for automatic translation. Or, they could be used to generate online hints to for programmer when interpretation is detected; or be presented simply as documentation for the programmer. Knowing the limits of these methods, ie in which (if any) situations no alternatives to interpretation exist, is also required.

3.3 Concrete material

Languages, tools and documentation drawing on those theoretical and practical discoveries are to be constructed

The languages, tools and documentation previously-identified as potential outcomes of the TFP methodology await construction, due to unsatisfied preconditions. For example, without a formal characterisation of interpretation, producing a language known to be interpretation-free is not possible. Additionally, even the exact TFP languages required remains to be established, as exactly how much interpretation is required for practical programming awaits resolution.

Ancillaries to languages such as type-systems, formal semantics and the like can then be developed. The inability to encode semantics as data in an interpretation-free TFP language should be of benefit in these pursuits.

Below, we give guidance as to constructing some of these items.

3.3.1 Languages for TFP

TFP languages are likely to be produced by both restricting and extending the lambda calculus

Instead of constructing a TFP language from scratch, there may already exist languages which are 'halfway there', as it were, and could be suitably extended and/or restricted at low cost. A number of general-purpose programming languages extant today are perhaps suitable, the suitability criteria being that the exemplars of TFP-style programming can be written in them. The simplest such language is the lambda calculus; others include 'LISP', Object-Oriented languages (functions represented as methods on objects), and certain imperative languages which support passing procedures as parameters.

Alternatively, very restricted languages may be a better starting-point. Languages which may be of interest here are those which are already total and/or sub-recursive (ie not Turing-complete). Examples of some such languages are 'SQL', those based on automata theory ('push-down' automata etc; [HU79]), and the lambda calculus restricted with 'System F' typing as detailed later. A language which was already interpretation-free would be even better, but none have yet been identified.

Overall, we believe that constructing TFP languages is likely to be best-done by considering extensions and restrictions to the lambda calculus, a language in which all terms are first-class functions. There is no indication that any computing paradigm other than the functional one would be more suitable for TFP, given that the key programming exemplars are all and solely first-class functions. The lambda calculus is the simplest practical language in which the exemplars can be written. If features from other paradigms turn out to be required, they can be added via well-known techniques (see eg [Wad97] and [Bar97]) involving essentially programmer-discipline to hide variables, rather than addition of non-functional primitives. We prefer the lambda calculus over simpler but equivalent formulations such as 'SK', 'SKI', or 'X'

combinators (see [Bar84]), due to the lambda calculus having named variables, a necessity for a practical language.

3.3.2 Formal semantics for TFP

Formal semantics for TFP languages would permit formal reasoning about their programs, and a proof that the languages let programmers avoid interpretation as often as theoretically possible

Formal semantics are useful for proving properties regarding programs and have been produced for many languages, even those with powerful type-systems (such as 'System F', see eg [Mit84]). One sort of proof which is common is proving the semantic equivalence of two program-fragments. However, we are particularly interested in another: a proof that TFP languages permit the programmer as much as possible to avoid doing language-extension via interpretation. It has been proposed ([Bai01]) that by having an expressively-powerful language the need to use interpretation for language-extension can be reduced, if not eliminated. We build on this theme below, after first introducing the common kinds of formal semantics.

3.3.2.1 Kinds of formal semantics

Formal semantics are typically axiomatic, operational, or denotational in nature

There are three well-known kinds of formal semantics:

1. Axiomatic semantics. Properties are supplied for program-fragments (often only equalities between such, which are of limited expressive power), which can be used to infer properties for combinations of program-fragments

2. Operational semantics. Typically, execution-steps (or simulations thereof) are given, or a means of determining the return-value of a program or program-fragment

3. Denotational semantics. A mapping from programs and parts thereof to some model is given; the mapping is thought of as giving the mathematical 'meaning' of the term.

Axiomatic semantics is widely used for procedural programming, where assertions can be given relating the final to initial values of state variables. They can also be used for functional programming; for example one can specify that the semantics of a construct is that of Natural addition (we use upper-case to designate variables):

add zero B = B

add (succ A) B = add A (succ B)

Certain general properties such as linearity can also be expressed, ie for a function 'f':

f (add A B) = add (f A) (f B)

implies 'f' is linear. Finally, the simplicity of the lambda calculus leads to simple axiomatic semantics for that language:

$(\lambda x.M)\ N = M[x\backslash N]$

Often axiomatic equations can be converted directly into operational semantics by replacing equalities with directed evaluations. For an example of operational semantics, take the equation above and change the '=' to a '→' to form operational semantics for the lambda calculus.

Finally, a simple example of denotational semantics is the following mapping:

$M[\![one]\!] = 1$

$M[\![two]\!] = 2$

$M[\![add\ X\ Y]\!] = M[\![X]\!] + M[\![Y]\!]$

Many more detailed examples of denotational semantics can be found in eg [Sch86].

3.3.2.2 General uses for formal semantics

Formal semantics permit formal proof of program properties

Perhaps the most common use of formal semantics is to show that two program-fragments are semantically equivalent. Formal semantics that can be used to do so relatively easily are desirable, and the task is certainly not easy: even in the lambda calculus there are many examples of terms with observationally-equal semantics. These include (as detailed in [Wad76], among others) numerous terms which don't terminate ie have no Head-Normal Form (for a definition of this, see eg [Bar84, p31]), the existence a multiplicity of fixpoint combinators, and even terms observationally-indistinguishable from '$(\lambda x.\ x)$'.

An additional desideratum is that the semantics be verified against our conceptual view of how the language operates. These two items are somewhat at odds: the semantics which are closest to our conceptualisations are generally the most-difficult to use to prove equality of program-fragments. The canonical solution is to produce a pair of formal semantics (one for each use) and a proof that the two are in correspondence. The first semantics is typically based on computational steps, and is an operational semantics model. The second semantics is chosen for its ease of showing program-fragment equality. Axiomatic semantics is relatively little-used for this in functional programming. Equational axiomatic semantics is especially unsuitable as equations involving program-fragments are limited in expressive power; and further the set of equalities between programs is well-known not to be recursively enumerable. As for operational semantics, one generally needs to show that, in all contexts, the two fragments will produce the same (observable) results; which can be quite difficult. Instead, denotational semantics is the common choice. This is because the denotational mapping can identify some semantically-equivalent fragments; with only the remainder needing to be proved equivalent in the mapped-to model. While working out what element is mapped to is in general undecidable (as all non-terminating programs may map to the same element, 'solving' the mapping can be as hard as 'solving' the Halting Problem), often one is working with tractable program-fragments.

The correspondence desired between the first and second semantics, between operational semantics and denotational semantics, is usually one of 'full abstraction' (as first introduced in [Mil75]). The meaning of this is that two terms are equivalent in the operational semantics if and only if they map to the same element in the denotational model.

A few things should be said about full abstraction:

1. Historically, finding fully abstract semantics has been somewhat difficult.

2. It is applicable even for languages with quite expressive features, such as communicating stateful nondeterministic processes running in parallel ([Rus89]); and with languages with types ([Rey83])

3. Whether one has full abstraction or not is sensitive on the terms one has formalised as observationally-identical. There are choices regarding this to be made in the lambda

calculus. For example, whether one evaluates to Weak Head-Normal Form or Head-Normal Form is important ([Abr90]). Another interesting example is the difference between 'F', where 'F = (λx. F)' and 'G', where 'G = (λx. x x) (λx. x x)'. While one might consider 'F' and 'G' equally useless and indistinguishable computationally, they typically map to (very) different elements of the model ('top' and 'bottom'; [Abr90]).

3.3.2.3 TFP-specific uses for formal semantics

Formal semantics can be used to prove TFP languages require the programmer to take recourse in interpretation as infrequently as possible

For TFP, formal semantics could be used to prove that the programmer needs only to use interpretation in as few cases (hopefully, none) as possible. One can construct a model for denotational semantics which contains all possible computable semantics (ie includes a 'maximal' set of semantics) of particular chosen forms; historically this has primarily been done via 'Domain Theory' (see eg [Sco72], [Sto77], [AJ95], [Bar84], [Jun96]; [Rea89, p341 onwards] contains a gentle introduction). Interestingly, the construction is able to be carried out without reference to the Turing-computable functions but instead using fundamental notations of computation, namely 'monotonicity' and 'continuity'. It is not inconceivable that interpretation might be able to be formalised within Domain Theory, based on such primitive notions regarding computation.

With such a model, one can then show using standard techniques that every element in the model can be denoted by some program or program-fragment in the language, ie that one has 'expressive completeness'. If the language can indeed express all computable semantics, then the need for the programmer to interpret their programs is minimised. For example, in a language without parallelism, one represents programs which use parallel constructs as data and writes an interpreter which does 'time-slicing' (a real-life example is given in [BGM93]).

3.3.2.4 Problems with constructing an appropriate proof

There are a number of problems, some easily solvable and some not, with constructing an appropriate proof

A proof that a language does let programmers avoid needing interpretation (up to theoretical limits) will be non-trivial to construct, due to issues encountered when trying to construct a 'maximal' set of semantics. Firstly, a truly 'maximal' set of semantics is going to be very large, including for example semantics that let functions return information about the soure-code of their inputs (to avoid needing to interpret programs which use constructs that do such), etc. At the extreme, one requires the 'maximal' set of semantics to be such that one is able to simulate every possible language and indeed computer; not just in the weak, extensional, Church-Turing sense, but down to details of relative timing etc. Secondly, there are issues relating to support of first-class semantics. In the lambda calculus, the inputs to functions are functions: the type of a function is 'D —> D' but also 'D', where 'D' is the set of all functions. Cardinality problems arise by naively equating 'D' with 'D —> D'; however the well-known solution is to restrict the functions in 'D —> D' to a countable subset of functions, specifically, the computable ones ([Sco72]). A less-tractable problem is that a set of 'maximal' semantics cannot exist. As we present below, there are differing and incomparable notions of computability, based on what information functions can extract from their parameters.

To begin, one can note that commonly the inputs to functions are data, which can be equality-tested in finite time. In such systems (generally) one has the standard set of Turing-computable (ie 'recursive') functions. However, in systems such as those TFP is interested in (functional languages), it is not true that functions can investigate their inputs such that every input-value can be distinguished from every other input-value in finite time, and the set of computable functions is not the Turing-computable functions. For example, in the lambda calculus the equality test on terms (even for only those in Normal Form, see [Bar84]) is incomputable. Generally, the set of computable functions depends on what information functions can gain from their parameters (both individually and together).

Firstly, take the choice between permitting parallel versus (one of the many forms of; see [Lon02]) sequential investigation of parameters. There is difference between the two in terms of what information can be extracted from a set of parameters which include a non-terminating

computation ('⊥'). The essence of the matter is that in a sequential system investigating '⊥' makes the whole computation non-terminating. However, this is not the case in parallel systems. There are functions between the 'partial' Booleans (ie the set 'true', 'false', '⊥') which are computable in systems that permit parallel investigation of parameters but not when investigations must be sequential ([Plo77]). Of more relevance to TFP is that there are operations which take functions as input only (ie no ground data), such as the 'convergence testing' functions covered in [Abr90], that are incomputable in sequential systems but computable in parallel systems.

Secondly, one may have a situation where, no matter how long a function spends investigating a parameter, it will never be able to extract 'all the information' from it. Examples of such parameters are 'lazy lists', Combinator Parsers, and Characteristic Predicates. Evidence that finite time is insufficient comes from a demonstrable loss of distinguishability: one cannot computably determine whether two infinite 'lazy lists' are the same or different, as only a finite number of elements can be checked. Generally-speaking, depending on exactly what investigations can be made, one gets different sets of computable functions. Even for the simple case of parameters which are functions on Naturals, there are competing notions of computability ([Mit96, p106]).

Thirdly, we highlight specific work by J. Longley regarding varying investigations by functions. As detailed in [Lon99], functions can detect whether their parameters 'look at' supplied inputs, via use of exceptions or references (or the 'H' of [Lon02]). With such investigative powers available, the computable functions include non-monotonic functions. For details of varying notions of computability at higher order, [Lon05] can be consulted.

Unfortunately, some of the above investigations are incompatible; and hence there cannot exist a 'maximal' set of computable functions. Parallelism conflicts with functions that need sequentiality in order to be deterministic; there is no more general notion of computability that subsumes them both as shown in [Lon02]. Programmers may also want nondeterminism and probabilistic features, which can also be incompatible with parallelism. Not all semantics can be freely mixed.

However, we have identified a potential solution. One might be able to partition programs into regions, in each of which a particular notion of computation holds sway. Rather than requiring a

single ‘maximal’ set of semantics is available for the programmer, the requirement now is that the full set of computable functions for the chosen notion of computation in that part of the program is available. The practicality of this approach remains to be determined.

3.4 Tackled questions

In order to progress TFP beyond its current formulation, we pursue answers to a diverse, but interrelated, range of questions

In the remainder of this work, we tackle a chosen four of the major open questions, one per Chapter:

a. What are appropriate type-systems for TFP-style programming? (Chapter 4)

b. How can TFP-style programs be derived? (Chapter 5)

c. What fundamental differences are there between interpretive and definitional styles of language-extension, especially those that explain our preference for the latter? (Chapter 6)

d. What is the intrinsic nature of interpretation? (Chapter 7)

Determining the nature of interpretation is most-important for fulfilling TFP, and is dealt with last. Each of the Chapters prior gives us useful information for the determination, as well as being of value in their own right. In the Chapter on type-systems we determine whether types offer clues as to interpretation. Similarly, derivation of TFP-style programs is of interest as they are considered to be less-interpretive than programs written in the standard style. Finally, investigating language-extension, one use interpreters can be put to, enables linguistics issues to be pared away from interpretation proper.

Chapter 4

Type-systems for TFP

The 'System F' and above type-systems are suitable for TFP

One or more type-systems for TFP languages are required to be found. We survey the major (and some other) type-systems for the lambda calculus, the language we believe TFP languages will be based on and which the TFP-style programs are written in. Some of these type-systems are already known ([BK03a]) to be unsuitable for TFP, due to being unable to type certain TFP-style programs. Others however we find to be acceptable. Unfortunately, none of the type-systems under investigation prevent interpretation; and construction of a custom type-system which does so cannot be attempted due to the interpretation-free programs not yet being characterised.

This Chapter is organised into the following sections. Firstly, we discuss the desire for type-systems, both generally, and for TFP-specific reasons. We then detail the major type-systems, and evaluate them against general and TFP-specific criteria. After discussing other less well-known type-systems, we then give our concluding remarks. In subsequent Chapters we detail how TFP-style programs can be derived, investigate interpretation as a means of language-extension, and determine the intrinsic nature of interpretation.

We would like to thank S. Seefried for drawing our attention to the ability of 'FCP' to type TFP-style programs.

4.1 Desire for typing

Type-systems are desirable for programming languages, including and especially TFP languages

Type-systems are useful for TFP languages for the same reasons they are useful for other languages, as well as for TFP-specific reasons pertaining to interpretation.

4.1.1 General benefits of typing

Typing has practical benefits for TFP programming

Type-systems are considered to be of undoubted practical utility for programming languages. In a standard text such as [Rea89] we find stated that type-systems are indispensable for the construction of large-scale reliable software. Further, one finds claims such as by Cardelli and Wegner in [CW85], that in the absence of a type-system, programmers will create an informal one, based on classification of program-terms with regards to usage and behaviour.

Why are type-systems so useful? Firstly, the knowledge of a term's type can imply information which a compiler can make use of, not only for example to determine how much space needs to be allocated for that item, but also potentially for optimisations. Further, types are of use in proving properties of programs and program-fragments as well. One example of this is that from knowing the type of a function, theorems the function must satisfy can sometimes be determined ([Wad89]). Typing also acts as a guide to the programmer, indicating for example which uses of a function would be appropriate, and which would not.

Primarily however, the purpose of typing is to help detect errors, such as 'mis-applications' or non-termination, in otherwise syntactically well-formed programs. Type-systems usually let the programmer 'fine-tune' the error-detection somewhat, such as by assigning a 'narrower' type than necessary to a term, including by introducing a fresh type-constant. The type of a term is essentially a statement of properties of that term from which properties of the program as a whole can be established. Interestingly, in many type-systems, terms only have a type if and only if they have certain general properties. For example, in the major type-systems we consider below, lambda-terms can be given a type if and only if their execution terminates (regardless of the execution-strategy chosen).

Some type-systems have a perhaps-surprising expressive ability. Types exist, such that the terms of that type are precisely the lists whose lengths are prime numbers, or all square arrays, or all balanced trees, etc: see [Bay01], [Hin99], [Oka99], [BP99b], [BM98], among many others. Another example is the use of 'dependent types' ([XP99]) to ensure that a function returns a list

of the same length as its input list. Indeed, types can be so powerful by letting types vary with values that the distinction between types and specifications appears artificial.

At the other end of the scale, the requirement of 'syntactic well-formedness' itself can be used to impose many type-like constraints in a structured and efficient manner. Consider:

Bool ::= true | false

Nat ::= 0 | 1 | 2 | ..

NatExpr ::= Nat | NatExpr '+' NatExpr | ..

which can detect a 'type error' in:

true + false

Extensions to BNF ('attribute grammars' [AM91], 'facet grammars' [BC92], etc) extend this somewhat-limited ability, however in general to implement all desired constraints a more powerful static analysis is required: a 'type-system' as commonly understood.

Finally, it should be noted that functional languages based on the lambda calculus (such as those proposed for TFP) have a particular need for typing, as there are no syntax-based or other checks on function arities. If one maintains a functional program, and changes a function-definition so that the function takes fewer or more arguments, then all uses of that function also need to be changed. By checking parameter and result types, a type-system can often detect, indirectly, when these required changes have not been made.

4.1.2 Particular benefits of typing with regards to TFP

Types may be able to be used to prevent interpreters from being written, and may cast light on the intrinsic nature of interpretation

Below, we discuss various ways in which type-systems are of relevance to TFP's core interest, interpretation.

4.1.2.1 Types impose constraints on programs

The constraints that prevent interpretation may be implementable in a type-system, and might be derivable from the types of TFP-style programs

It is possible that some, if not all, of the constraints necessary to constrain a language from supporting interpretation may be able to be expressed as a static (ie syntactic) test, implemented using a type-system. Some evidence for the feasibility of this comes from papers such as [PL89], which suggests that for some type-systems writing a typed interpreter to handle programs with types of order 'n' requires order 'n+1' typing. As well, types have been explicitly used to enforce non-trivial properties. In [SB02] a higher-order programming language is constructed via typing and other static constraints to ensure all allowable programs have the property of being polynomial-time computable. Related work is presented in [HK96], and in [KV05], where certain restrictions of both functional and imperative languages are shown to correspond to complexity classes. Good overviews of this general topic can be found in [Imm87] and [Hof00].

In addition, clues as to the nature of interpretation might be gained from studying type-systems that seem to have a close affinity to TFP. Particularly, recall previous mention that certain of the TFP-style exemplars are the sole inhabitants of their types in some type-systems.

4.1.2.2 Sub-recursive computation

Type-systems can restrict languages so that they become sub-recursive, which may be of use as interpretation-free TFP languages will be sub-recursive

As mentioned earlier, we find in TFP an interest in sub-recursive ('total', preferably) computational systems. The nature of interpretation may perhaps be more readily established in such systems, given the restricted nature of computation possible. Type-systems are therefore of interest, as they can limit the programs writable to those in some sub-recursive set, often a set containing only 'total' programs. Clues as to the nature of interpretation might be gleaned by considering the programs typable under various type-systems.

In addition, interpretation can likely occur within types. Systems such as 'F_ω', introduced later, let the programmer write complex lambda-terms denoting types and functions between them. Considering the computations possible within types in such type-systems may also be fruitful grounds for investigation. Interestingly, the ability to compute within types implies that a 'TFP for type-systems' may also be desirable to consider, with benefits possibly including more decidable type checking and type inference.

4.1.2.3 Types, indirectly

One is able to view types as propositions and programs as proofs, which may permit questions relating to interpretation to be answered more easily

It is well-known that typed lambda calculi can be used to represent propositions (see eg [Far03] for an introduction). In addition, there is a deeper correspondence between proofs and types, one that views every type as a proposition, and programs of that type as proofs of that proposition (and type checking as proof-checking). One can consult [Bar97] for an overview, or for details see eg [SU98]. This correspondence, usually referred to as the 'Curry Howard' isomorphism, can be viewed as part of Constructive Mathematics. Specifically, a function-implementation is a 'constructive proof' that there exists a procedure by which an output with some property (type) may be produced from inputs with some properties (types). However, note that there exist proof-theoretic analogues of programs that are not Constructive (ie include the law of the excluded middle). For examples we refer the reader to [SU98], which also includes details of an interesting relation to the Continuation-Passing Style.

Given this correspondence, one might consider proof-systems as well as type-systems for clues as to interpretation. Perhaps the TFP-style exemplars have a previously unseen commonality which is exposed when viewed in the realm of logic. Additionally, perhaps a constraint to prevent interpretation would be best formalised in the language of proofs and propositions, from which an appropriate typing system can be derived (the proof-theoretic constructs used perhaps requiring the introduction of unusual typing constructs).

4.2 Details of the major type-systems

There are a number of type-systems available for data-free higher-order functional programming, such as to be done in TFP languages

As mentioned previously, TFP languages are expected to be based on the lambda calculus. The major (static) type-systems available for the lambda calculus are introduced briefly below. The papers [BH90] and [SU98] are good references for detailed formalities, and we have drawn on them for our presentation. We aim in this Chapter to give an accessible useful overview only; technical results of interest will be cited, and stated informally.

On matters of terminology, for brevity we will often refer to type-systems as if they incorporated programming languages. For example, we will say 'Hindley-Milner typing' instead of, more properly, 'the lambda calculus with Hindley-Milner typing'. The language, the lambda calculus, is to be understood. Secondly, while in the literature both 'x:t' and 'x^t' are common ways to denote a variable 'x' being of type 't', we will standardise on the former exclusively. Thirdly, we denote function-type construction using '─>', which brackets to the right. Finally, the reader will come across the phrases 'Church style' and 'Curry style'. These refer to two variants of typed lambda calculii, which differ in whether terms must include types or not. For example, the Curry style typed lambda calculus has terms which are just the terms of the lambda calculus. In contrast, Church style lambda calculus requires formal parameters of terms to have user-entered types. In more powerful type-systems, 'System F' and beyond, other type information is required to be added: abstractions and applications of types and type-constructors must be annotated. Note that for both Church and Curry styles, we let types be optionally added to terms, for type checking purposes.

4.2.1 Simple typing

The Simple typing is uncomplicated but inexpressive

The 'Simple typing' is the first of the type-systems for lambda calculus we consider. It comes in Church style ([Chu40]), and Curry style ([Cur34]). Presentations of it can be found in most

textbooks, including [Bar84]. In Church style, the lambda calculus with Simple typing can be presented as follows, using standard notation:

Term Language-

E	::=	x	*variables*
	|	E E	*application*
	|	λx:T. E	*abstraction*

Type Language-

τ	::=	t	*type constants*
	|	τ —> τ	*function type*

Typing Rules-

$$\frac{(x : \tau) \in A}{A \vdash x : \tau}$$

$$\frac{A \vdash E : \tau' \rightarrow \tau \quad A \vdash E' : \tau'}{A \vdash (E\ E') : \tau}$$

$$\frac{A, x : \tau' \vdash E : \tau}{A \vdash (\lambda x.E) : \tau' \rightarrow \tau}$$

It should be noted that:

1. For this and all the other type-systems presented in this Chapter, it is conventional to assume that 'λ'-abstracted variables cannot appear in types: it is assumed each has their own name-space; or each are drawn from disjoint sets.

2. The Simple typing in Curry style is the same as the above, but with 'λx:T.E' replaced with 'λx.E' so that the set of terms becomes those of the untyped lambda calculus.

3. The Simple typing is often presented as using type *variables* rather than (or as well as) type *constants;* the difference for Simple typing is not fundamental and type constants give, we feel, a simpler presentation with less potential for confusion.

An example of a (Church style) typed term is:

(λv:Nat, w:Nat. v)

The type of this term is 'Nat —> Nat —> Nat', ie 'Nat —> (Nat —> Nat)'. This type can be read as implying that the function it types takes an argument of type 'Nat', and returns a function, a function which takes a 'Nat' and returns a 'Nat'. The above term in Curry style is of course simply:

(λv, w. v)

In Simple typing, each function intuitively has a *single* (ie 'simple', 'monomorphic') type. In contrast, in the type-systems we consider below functions can have 'polymorphic' (also known as 'generic') types. This 'polymorphism' means the functions, their arguments and their results, have multiple 'simple' types.

4.2.2 Hindley-Milner typing

Hindley-Milner typing is based on the Simple typing, but has polymorphic definitions

Hindley-Milner typing was discovered by Hindley ([Hin69]) in the context of proof theory, and later rediscovered by Milner ([Mil78]). This type-system is an extension of the Simple typing, and approaches to a certain extent the power of the system 'F_2', below. Its claim to fame is that it permits fairly powerful polymorphism, but has practical algorithms for type checking and type inference; unlike more powerful type-systems. In essence, Hindley-Milner typing is the Simple typing plus support for polymorphic definitions and the ability to denote polymorphic types.

The lambda calculus with definitions (in the form of the 'let' construct) and the Hindley-Milner type-system (Curry style) can be presented succinctly as follows (reproduced from [Jon97], with minor alterations):

Term Language-

E	::=	x	*variables*
	|	E E	*application*
	|	λx. E	*abstraction*
	|	(let x = E in E)	*local definition*

Type Language-

σ	::=	∀t. σ	*polymorphic type*
	|	τ	*monotype*
τ	::=	t	*type constants*
	|	τ —> τ	*function type*

Typing Rules-

$$\frac{(x : \sigma) \in A}{A \vdash x : \sigma}$$

$$\frac{A \vdash E : \tau' \to \tau \quad A \vdash E' : \tau'}{A \vdash (E\ E') : \tau}$$

$$\frac{A, x : \tau' \vdash E : \tau}{A \vdash (\lambda x.E) : \tau' \to \tau}$$

$$\frac{A \vdash E : \sigma \quad A, x : \sigma \vdash E' : \tau}{A \vdash (\text{let } x = E \text{ in } E') : \tau}$$

$$\frac{A \vdash E : \sigma}{A \vdash E : \forall t.\ \sigma \ \text{(for fresh 't')}}$$

$$\frac{A \vdash E : \forall t.\ \sigma}{A \vdash E : [\tau/t]\sigma}$$

As is customary, variable renaming (only so far as it avoids capture) is also implicitly permitted.

In Hindley-Milner typing, types can involve what is known as 'parametric' polymorphism, as indicated by the last two rules. For example, the function:

(λx,y. y)

might be typed as any of:

Nat —> Nat —> Nat

Nat —> Bool —> Bool

∀d. d —> d —> d

∀e. ∀f. e —> f —> f

(Nat —> Bool) —> (Bool —> Nat) —> (Bool —> Nat)

∀j. ∀k. (j —> k) —> (j —> k) —> (j —> k)

etc

However, the typing rules result in this polymorphism being lost *once a function is supplied as a parameter*, ie:

(λg. (g (λw. add w 2) (λw. add w 3)) (g 4 5)) (λx,y. y)

is not typable in Hindley-Milner. Expressions can have a polymorphic type ('polytype'), but for application that type must first be narrowed to one of the so-called 'monotypes' which are members of that polytype. Hence all occurrences of a function parameter within a function body will have the same (mono)type. Naively, the overall type one would want for the operator above applied to '(λx,y. y)' is:

(∀a.∀b. a —> b —> b) —> Nat

but this isn't a valid Hindley-Milner type: in Hindley-Milner typing, all universal quantifiers must be *outermost* (and hence can be, and often are, elided). This restriction can be summarised by saying that 'first-class expressions are not polymorphic'; meaning that a function can have polymorphic type, but as an argument it is only monotyped. Such a restriction permits practical type inference and type checking algorithms, at a cost of reduced polymorphism.

However, Hindley-Milner typing does have definitions, each use of which is typed separately. Hence:

let g = (λx,y. y) in
(g (λw. add w 2) (λw. add w 3)) (g 4 5)

is able to be typed with Hindley-Milner typing.

4.2.3 System F typing

System F (F_2) typing lets terms have polymorphic type, and can type more programs than Hindley-Milner typing

The type-system of the system 'F' was developed by Girard ([Gir71], [Gir72]; [Gir86] is also useful, and in English) and rediscovered by Reynolds ([Rey74]). This system is commonly referred to just as 'System F' or 'F_2' (we use the two interchangeably). It is also known as the '2^{nd}-order (polymorphic) typing', 'λ2', and the 'Girard-Reynolds typing'. System F is more powerful than Hindley-Milner typing, which is has as a subset. System F has what is termed 'first-class polymorphism': functions can be explicitly parameterised on types, and remain polymorphic when supplied as arguments.

Since Girard's original presentation, it has become customary to use only the core of his original system, ie with the only type-constructors being '∀' and '—>', and with no recursion. We present lambda calculus with this reduced System F typing now, Church style as per [Jon97].

Term Language-

E	::=	x	*variables*
	|	E E	*application*
	|	λx:σ. E	*abstraction*
	|	E σ	*type application*
	|	Λt. E	*type abstraction*

Type Language-

σ	::=	t	*type variables*
	|	σ —> σ	*function types*

| $\forall t.\ \sigma$ *polymorphic types*

Typing Rules-

$$\frac{(x : \sigma) \in A}{A \vdash x : \sigma}$$

$$\frac{A \vdash E : \sigma' \rightarrow \sigma \quad A \vdash E' : \sigma'}{A \vdash (E\ E') : \sigma}$$

$$\frac{A, x : \sigma' \vdash E : \sigma}{A \vdash (\lambda x{:}\ \sigma'.\ E) : \sigma' \rightarrow \sigma}$$

$$\frac{A \vdash E : \sigma}{A \vdash (\Lambda t.\ E) : \forall t.\ \sigma \ \text{(for fresh 't')}}$$

$$\frac{A \vdash E : \forall t.\sigma}{A \vdash (E\ \sigma') : [\sigma'/t]\sigma}$$

Note:

1. While it was originally developed in Church style, presentations in Curry style are not uncommon.

2. Unlike in Hindley-Milner typing, '$\forall$' ranges over both monotypes *and* polytypes. Hence for example if a term is of type '$\forall t.\ t \rightarrow t$' then it is also of type, among others, '$(\forall u.\ u \rightarrow u \rightarrow u) \rightarrow (\forall u.\ u \rightarrow u \rightarrow u)$'.

3. System F is second-order (hence the moniker 'F_2') as one can abstract types using 'Λ', but not functions between types etc.

A example of a Church style term:

$$(\Lambda\alpha.\ \lambda x{:}\alpha.\ x)\ (\forall\alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha) : (\forall\alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha)$$

The same term, in Curry style, with a valid type added:

$$(\lambda x.\ x) : (\forall\alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha)$$

4.2.4 F_ω typing

F_ω typing is an extension of System F typing, and is the most powerful type-system we consider here

F_ω typing ('system F_ω') was introduced by Girard at the same time as F_2. This type-system is an extension of F_2. In F_ω, not only can one abstract types from terms, but also abstract types and functions between them *from types*. Hence one can do computations *within* the type language; as types are in F_ω essentially first-class citizens.

Adapted from [LLMP89], the lambda calculus with F_ω typing Church style:

Term Language-

E	::=	x	*variables*
	|	E E	*application*
	|	$\lambda x{:}\sigma.$ E	*abstraction*
	|	E σ	*type application*
	|	Λt. E	*type abstraction*

Type Language-

σ	::=	t	*type variables*
	|	$\sigma \rightarrow \sigma$	*function types*

	| ∀t:K. σ	*polymorphic types*
	| λt:K. σ	*abstraction of types*
	| σ σ	*type application*
K	::= ★	*the Kind of types*
	| K —> K	*functions of Kinds*

The typing rules for F_ω are elided due to being somewhat involved, having to deal with 'Kinds', various contexts, and both types and terms under type application and type abstraction.

The word 'Kinds' refers to the type-schema a type-variable can have. Having a Kind of '★' means the type-variable holds a type, for example 'Boolean' or 'Boolean —> Natural' or '∀a:★. a —> a', or even '∀a:★.∀f:★—>★. (f a) —> (f (f a))'. Similarly, having a Kind of '★—>★' means the type-variable holds a function from type to type, for example '(λt:★. t)' or '(λz:★. (λf:★—>★ —>★, x:★. f x x) (λa:★, b:★. b) z)'. Kinds are often described as being the 'type of types'.

4.2.5 $F_{n>2}$ typing

The $F_{n>2}$ typings are intermediate between System F and F_ω typing

Intermediates between F_2 and F_ω, ie 'F_3', 'F_4' etc, were also presented by Girard; see [Gir86]. Drawing from [HM91], one can summarise them by saying that:

F_2 is F_ω without 'abstraction of types', 'type application', and with the only Kind being '★'

F_3 (and higher) are F_ω with the Kinds being K_3 (and higher), where:

$K_2 ::= ★$

$K_{n+1} ::= K_n \mid K_n \text{ —> } K_{n+1}$

Hence F_2 does not have functions between types, while F_3 and higher support ever-more complex functions between types. Specifically, functions between types allowed in F_3 are those which take one or more types as input, and returning a type as output, ie are of the Kinds:

★

★ —> ★

★ —> (★ —> ★)

★ —> (★ —> (★ —> ★))

etc

For F_4, functions between types can take as inputs either types or those F_3 functions between types; and so forth for F_5 and greater.

4.3 General criteria and comparison of type-systems

A suitable type-system should satisfy a number of general criteria, the major type-systems do to varying extents

We establish a number of general criteria with which to evaluate candidate type-systems. These include whether the type-system is too restrictive in the programs it permits and whether one can have mechanical detection of type errors.

Beyond these criteria innumerable other ones of lesser importance exist, but will not be considered here. For example, one can ask how well the type-system can 'localise' errors, and whether type checking requires no more than polynomial time in the worst-case. Another question is how much programmers are able to 'fine-tune' the detection of program-errors by the type-system, eg by specifying 'narrower' types than otherwise necessary, introducing type-constants, or modify typing judgments.

Similarly, type-systems can have specific capabilities which find practical use, but these won't be covered in any detail. For example, linear ie 'uniqueness' typing (as briefly covered in [Bar97, p18]; see for theoretical details eg [GLT89, p149 onwards]) ensures that selected terms cannot be duplicated, which finds use for typing programs that interact with their environment. Likewise, certain type-systems are able to support such things as data-abstraction, object-orientation, etc in an elegant manner.

4.3.1 Typability

A suitable type-system will type the programs one wants to write, and whether a term can be typed should be computable: only Church style System F or higher is satisfactory in these regards

For a type-system to be suitable it should accept the programs one may reasonably want to write. In other words, each such program should be 'typable' using the type-system, either as-is (ie Curry style), or after user-addition of type annotations to it (making it Church style).

Note that when discussing typability, one does *not* identify terms up to '↔'-equivalence. For example, as noted in [Urz97], System F can type all terms *in* Normal Form, but cannot all terms which *have* a Normal Form.

We begin by noting the equivalence of typability between Church and Curry style typed lambda calculii, before moving on to the fundamental sameness of the Simple typing and Hindley-Milner typing. We then consider what terms can be typed in each of the type-systems, as well as whether determination of whether a type exists for a term is computable. We conclude with discussion of how certain features of type-systems lead to desirable classes of programs being typable.

4.3.1.1 Analysis: Church style versus Curry style

Church and Curry style typings type essentially the same programs

Church and Curry style do not entail differing typability. As shown in [SU98], specifically for the Simple typing and System F:

- For every typable Curry style program, type information can be annotated to give a typable Church style program

- For every typable Church style program, one can erase the type-information to give a typable Curry style program

4.3.1.2 Analysis: the Simple typing and Hindley Milner typing

The Simple typing and Hindley-Milner typing are essentially equivalent

It is widely recognised that the Simple typing and Hindley-Milner typing can type essentially the same programs.

Any program containing definitions, typable with Hindley-Milner typing, can be mechanically transformed into a semantically-identical program typable in the Simple typing. One simply treats definitions as macros, and expands the defined name 'in-line' (avoiding capture). Each use of the definition-body is then typed separately, with the existence of a Hindley-Milner type for the definition body, suitable for the program, meaning that Simple Types exist to type each occurance of the defined name (or its body, macro-expanded).

Conversely, any program typable with the Simple typing is, trivially, typable with Hindley-Milner typing.

4.3.1.3 Analysis: typability of total and partial programs

The major type-systems can type only, and then only some, 'total' programs; F_ω types the most

To begin with, none of the major type-systems permit any other than 'total' programs (ie those with a Normal Form) to be written. Moreover, the type-systems type only 'strongly normalizable' terms, meaning that if a term is typable then *any* sequence of reductions will terminate with the Normal Form (there is no infinite reduction sequence). A useful consequence of this is that eager evaluation can be used for execution, rather than lazy evaluation. Proofs that only strongly normalizable terms are typable in Simple typing and System F appear in eg [FLO83], and Girard's original works contain a proof for F_ω.

As a direct consequence of knowing a type-system types only strongly normalizable terms, we know that non-terminating terms such as:

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

cannot be typed by it.

For the Simple typing and Hindley-Milner typing, the *total* programs which are untypable can be surprisingly simple. The function '(λx. x x)' for example is well-known not to be typable ([Rea89, p392]); it is however typable in System F and higher (ie $F_{n>2}$ and F_ω). We give details as to why later. Another example is reported in [SU98]; the Simple typing (and, Hindley-Milner typing of course) cannot type:

(λf,x. f (f x)) (λg,h. g)

One would think System F could do significantly better, as its universally-quantified variables can range over polytypes, not just monotypes. Indeed it can. It is known that System F can type programs which can't be shown to terminate using (first-order) Peano arithmetic ([O'D79]). In fact System F types precisely the terms which can be proved to terminate in second-order propositional logic; this relationship was first shown in [Gir71]. A good reference for it is [Wad03], another is [GLT89]; similar relationships exist for certain restrictions of System F as detailed in [DL99]. As the terms in question include Ackermann's function ([Ack28]), this might be considered a somewhat weak restriction. However, one surprisingly-simple term (from [SU98] again) that System F cannot type is:

(λf,x. f (f x)) (λf,x. f (f x)) (λg,h. g)

An earlier one, which appeared in [GR88], is:

(λx,z. z (x (λf.f)) (x (λv,g. v)) (λy. y y)

As reported in [SU98], the untypability of the latter is due to there being no single type for '(λy. y y)' which lets it be applied to both '(λf.f)' and '(λv,g. v)'.

The type-system F_3 can type more programs than System F (aka F_2), F_4 more than F_3, etc ([Gir73]); and F_ω can type even more again. While no type-system can type all total programs (else it would solve the Halting Problem), even F_ω does not come close: the following is an example, given in [Urz97], of a simple term which can't be typed in it:

```
(λx. z
        (x (λf,u. f u))
        (x (λv,g. g v))
) (λy. y y y)
```

After a single reduction however, the result *is* reported to be typable in F_ω. This isn't surprising, given that all strongly normalizable terms have a Normal Form, and Normal Forms, reachable via reductions, can all be typed using System F as noted previously. Not all Normal Forms can be typed by Simple typing or Hindley-Milner typing of course, recall that '(λx. x x)', a term in Normal Form, can't be typed in either. Generally, reductions can only make a term *more*

typable. In [BH90] we find stated that all the major type-systems (Church style) considered here have the 'subject reduction' property with respect to types, namely that if 'M' has a type 'T', and 'M '$\rightarrow_\beta$* N', then 'N' also has type 'T'. One can also consider how eta-contraction affects typability. As noted in [SU98], there exist terms which are typable in Curry style System F, but on an eta-contraction the result is not. One can however add a new rule to fix this ([Mit88]). The same issue doesn't exist with Church style System F, simply because eta-contractions can't occur because type applications are in the way.

4.3.1.4 Analysis: computability of typability

Only the Simple typing and Hindley-Milner typing have computable typability for Curry style terms

An important question to ask is whether it is possible can determine whether a given term has some valid type, in the chosen type-system. In [Wel96] (superseded by [Wel99]) we find that whether a type exists for a (Curry style) term is in general decidable for the Simple typing (and, Hindley-Milner), undecidable for F_ω, and the paper itself shows that it is undecidable for F_2 (and hence also for F_3 etc). In contrast, typability is computable for Church style terms in all of these type-systems ([BH90]).

4.3.2 Polymorphism

A suitable type-system should support polymorphic types for definitions

Without polymorphic types programs may undesirably have to include multiple versions of the same function, versions which differ only in the type assigned to them by the programmer. For example, a program may have to include two instances of a sorting subroutine, one typed to sort integers, and the other for strings. As an even simpler example, without polymorphic typing, one needs to write:

f1 ≜ (λx,y. y): Nat —> Nat —> Nat

f2 ≜ (λx,y. y): (Nat —> Nat) —> (Nat —> Nat) —> (Nat —> Nat)

(f2 (+1) (-1)) (f1 4 5)

While with polymorphic typing, one can write, as the function operates 'uniformly' over a number of (mono) types:

f ≜ (λx,y. y): ∀a,b. a —> b —> b

(f (+1) (-1)) (f 4 5)

4.3.2.1 Analysis

Only the Simple typing is insufficiently polymorphic

We believe that some form of (parametric) polymorphic typing should be a required feature of any TFP type-system, due to the expected presence of definitions. All the type-systems detailed previously bar Simple typing support polymorphic typing; to varying levels of typability as discussed above.

4.3.3 Principal types

It is desirable for a type-system to have 'principal types' for all typable terms

In some type-systems whenever a type is correct for a term (a 'closed' term, ie one without free variables) there is also a certain 'most general' type the term can be given which also is correct. For example, in

(λx. x) true

the term '(λx.x)' has type 'Boolean —> Boolean', while in

(λx. x) id true

the same term has type '(Boolean —> Boolean) —> Boolean —> Boolean'. However, we can type '(λx. x)' as '∀a. a', with this type being acceptable in both of the above.

In those type-systems using 'most general' types means that one is guaranteed never to have to duplicate definitions and give them different types. Similarly, one will never have to modify the types of functions when their uses change.

This property of having a 'most general' type is implied by an often-stronger one, termed 'principal types'. A type-system has the 'principal types' property if and only if each term has a

'principal' type: informally, a type from which every other valid type for the term can be derived via substitutions. For details on principal types, see eg [DM82]. Principal types are to be distinguished from the 'principal typ*ings*' of [Wel02] etc.

4.3.3.1 Analysis

Only Hindley-Milner typing has principal types

While for Hindley-Milner principal types exist, and can be mechanically calculated, they don't exist for F_2 or F_ω (implied in [SU98] and [LLMP89]). They also don't exist for the Simple typing, as it isn't polymorphic. For example:

$(\lambda x.\ x) : Nat \multimap Nat$

and

$(\lambda x.\ x) : (Nat \multimap Nat) \multimap (Nat \multimap Nat)$

are both typed using the Simple typing, but the lack of polymorphism means that no Simple type entails both of the above types.

4.3.4 Ability to infer types

A suitable type-system should not place an onerous burden on the programmer to supply types

In order for a type-system to check that a program is type-correct, types must be known for each variable and term. In System F and higher, type applications and abstractions must be known as well. Programs written Church style have all or nearly all this type information supplied by the programmer. In contrast Curry style terms are type-free, hence all type information need to be determined by the type-system itself in a process called 'type inference' or 'type deduction'.

With TFP programs expected to use higher-order functional programming and eschew primitive data, the types involved would often be complex, hence difficult, error-prone, and time-consuming for the programmer to supply. Type inferencing would hence seem to be a requirement of any suitable type-system for TFP languages.

4.3.4.1 Analysis

Curry style System F and above do not have computable type inference

For all the major type-systems considered here, type inference (such that is required) is decidable for Church style terms, as stated in [BH90]. Type inference for the Curry style Simple typing is computable via first-order unification, see [Hin69] or [Mil78]. Hindley-Milner typing permits algorithms (see [Rea89, p382 onwards]) for full type inference, which construct principal types for untyped terms. Hindley-Milner also permits computable type inference when the programmer supplies some types.

For Curry style System F (and hence $F_{n>2}$) ([Wel96]), and F_{ω} ([Wel96], [Urz97]), type inference in general is not computable: any procedure must fail to terminate or give an incorrect answer, for some inputs. Note that type inference for these systems must not only infer types, but also type abstractions, type applications, etc. Inferring only types is known as 'type reconstruction', and is a simpler problem.

It should be noted however that theoretical intractability for type inference does not necessarily mean that approximations won't suffice, for 'most' programs.

Also, one should note that intermediates between the Church and Curry styles are also available: **t**here exists intermediates between Church and Curry style F_2, and also F_{ω}, which have computable type inference. In [Pfe93] has been shown that type inference for a certain intermediate between Church and Curry style System F is undecidable, even just for terms in Normal Form. This intermediate is interesting because in it only the problematic 'difficult to write' parts of the typings can be elided: type-applications and abstractions must be present but one need not specify the types involved, or the types of variables. This scheme can be considered to be only a little away from Church style, yet type inference is in general impossible. The paper also shows that this result holds even in a Hindley-Milner -like version where '$\forall$' only ranges over monotypes.

One should note however that other schemes based on System F exist in which some type information can indeed be successfully elided. The first system which offered System F power but didn't require programs typable in Hindley-Milner to have user-entered type annotations was

(the 'IFX' subset of) 'FX-89', as presented in [OG89]. An algorithm is given therein and its correctness proved. Some previous related works are referenced in the survey [Rey85]. There have also been a number of later systems, such as [BL03] and [Rem05]. However one which we will take a particular interest in later is the language 'FCP', as presented in [Jon97]. While at first glance it is an extension of Hindley-Milner typing with special typing rules for data-types, these result in the type-system that is in effect System F. Its author claims that any (Church style) System F program can be rewritten in it, by using data constructors and destructors instead of type application and abstraction. As can be seen in examples we give later, the annotations which are required are few and simple in nature. We consider it the most promising of the System F -based type-systems with computable type inference.

Finally, one also finds intermediates for F_{ω}. For Curry style F_{ω} programs which have had type placeholders added, type inference is not computable as reported in [LLMP89]. However, when certain additional type information is provided, as discussed in that work an algorithm exists which "in practice.. behaves well". Details can be found in [Pfe88], and are relevant to type reconstruction in $F_{n>2}$ as well. In addition, a programming language is also presented in [LLMP89], called 'LEAP' (see also [PL89] and [PDM89]). This language is based on F_{ω}, with type reconstruction as above but with the addition of an extra feature: type placeholders can be mechanically reconstructed based on extra information supplied with definitions.

4.3.5 Type checking

A suitable type-system should be able to mechanically detect type errors

The ability to mechanically detect mistyped or untypable programs, which are taken to indicate erroneous programs, is highly desirable in a type-system. Having an automated process for such 'type checking' is extremely important, as attempting to detect type errors 'by hand' is tedious and error-prone (details of a process for doing so can be found in [JMT02]).

4.3.5.1 Analysis

Type checking is incomputable for Curry style System F and above

Type checking algorithms for the Simple typing and Hindley-Milner typing are well-known (see eg [Rea89]); handling both Church and Curry style terms. The more powerful type-systems, System F and above, have undecidable type checking for terms in Curry style ([Wel96], [Urz97]). In contrast, as stated in [BH90], type checking is decidable for Church style terms.

Of relevance to this topic is the fact that particular (characterised in the next Chapter) TFP-style functions have an affinity with the polymorphic type-systems, in terms of detecting programming errors via type checking. Illustrating by example, say we wanted to ensure that the arguments to a particular function could only be Church Naturals. No type constants or similar need to be introduced in this case, as the Church Naturals have an *exact* match in terms of polymorphic types: as is well-known, the Church Naturals are the only terms (in Normal Form) of type '∀a. (a —> a) —> a —> a'. Hence if type inference of a function tells us that a function takes an input of type '∀a. (a —> a) —> a —> a' then we know that the function takes all, and only, Church Naturals as input.

4.3.6 Conclusion

Typability trades-off with the ability to do computable type inference and type checking

Generally, one desires as many reasonable programs as possible to be typable, to avoid programmers having to eg duplicate and re-type functions. Also, *only* reasonable programs should be typable else some program-errors may be missed. For example, having all terms being of type 'Λ' (the set of all lambda-terms) yields zero error-detection capability. None of the type-systems considered seem to type too many programs, but none type all reasonable programs. However this should not count against them for as previously-mentioned it is not possible for any type-system to type all and only the reasonable (total) programs. The question then is, do they type *enough* to be satisfactory. The answer depends on the particular programs one wishes to write; typability of TFP-style programs is discussed in the next section. However there is a general result as the type-systems can be ranked: the Simple typing and Hindley-

Milner typing can type the fewest lambda-terms, System F strictly more, $F_{n>2}$ more again, and F_{ω} the most.

Turning now to other criteria, the more type inference which can be done the better, especially if type expressions are long and hard to maintain. Not surprisingly, the more programs which are typable in a given type-system the harder it is to do type inference; such can be incomputable. Similarly, type-checking, an essential feature, can be uncomputable for powerful type-systems. Only the Simple typing and Hindley-Milner typing have computable type checking and type inference for Curry style terms; while all type-systems have both being computable for Church style terms. Note however that intermediates between Church and Curry style may offer computable type checking and inference. Further, when type checking is not computable, approximations may exist that suffice to check all programs one may reasonably ever write.

4.4 TFP-specific criteria and comparison of type-systems

A TFP type-system preferably should type most (if not all and only) interpretation-free programs, as well as possibly offer clues as to then nature of interpretation: none of these type-systems do

Above, we have compared the type-systems against a number of general criteria. For use with TFP however, there are additional criteria to be considered. The first pertain to interpretation. We then move on to the need for suitable type-systems to type the programs expected to be written in TFP languages, specifically the TFP-style programs.

As we discuss below, the type-systems under consideration do not seem to offer any help with characterising interpretation. They also do not prevent interpreters from being written, ie apparent interpreters are able to be typed. This cannot be rectified by restricting the permissible types: we show that for these type-systems no test for interpretation implemented as a test on the type of a program can avoid either letting through some apparent interpreters, or rejecting some apparent non-interpreters.

4.4.1 Help in characterising interpretation

A TFP type-system would, best-case, offer clues as to the nature of interpretation

In the best possible world, a type-system for TFP-style programming would have distinct features that are directly related to interpretation, and expose its intrinsic nature.

4.4.1.1 Analysis

The remarkable feature of the major type-systems, sub-recursiveness, appears unrelated to interpretation

In these major type-systems, only some computable input-output mappings are possible to implement, ie have a typable implementation. In other words, programming with these type-systems gives one only sub-recursive computational power. For example, only total programs can be typed. This however seems completely unrelated to matters of interpretation; instead it is type-theoretic restrictions which are to blame. These restrictions are on:

- The set of valid monomorphic types (eg recursive or not)

- What set of monomorphic types a function can have (ie polymorphism)

- How polymorphism varies with context, eg in Hindley-Milner typing functions lose their polymorphism when passed in as arguments

- How the type of a function depends on the type of other functions, eg via type-abstraction in F_ω and F_ω, or type-variables appearing in the types of multiple parameters in Hindley-Milner

- How the type of a function can depend on the computation so far, via functions between types as in F_ω

Hence we consider it unlikely that any information regarding interpretation can be discovered from looking at the restrictions on computation in these type-systems. For example recall the term from [Urz97] which is untypable even in F_ω:

(λx. z

(x (λf,u. f u))

(x (λv,g. g v))

) (λy. y y y)

This term doesn't seem to do equality-testing or pattern-matching or anything else that intuitively one would recognise as interpretation. It is simply a lack of polymorphism in F_ω which results in the term's untypability.

We can also discount computation *within* types as being a place to look for answers regarding interpretation. In $F_{n>2}$ and F_ω one can write functions between types (and also higher-order functions), however these are simply lambda-terms like any other. Further, the restrictions on these type-orientated functions are nothing special. While functions between types in F_ω may appear to be able to be quite complex, they are however restricted (due to each having to have a Kind) to being typable using essentially the Simple typing.

Overall therefore, the type-systems do not seem to offer any help with regards to characterising interpretation.

4.4.2 Ability to support a type-based test for interpretation

A TFP type-system may permit characterising programs into the interpretive and the interpretation-free by means of types

It would certainly be useful if the type-system chosen for TFP languages was able to detect interpreters; for example by their types in that type-system having some distinctive property.

4.4.2.1 Analysis

The major type-systems fail to type some non-interpreters and some interpreters, hence an accurate type-based test for interpretation is not possible using them

Consider now, if interpretiveness is a static property of programs, when that property could be formalised as a test on types in some type-system. Given that the major type-systems can only type strongly normalizable terms, some blanket judgement needs to be made for these non-

typable terms. Either all untypable terms are interpretive, or all are not interpretive. However, neither choice is correct. For example, neither '(λx. x x) (λx. x x)' or a simulation of a Universal Turing Machine in lambda calculus is strongly normalizable, but the first does not intuitively appear to do interpretation while the second does. Hence any type-based test for interpretation in these type-systems must be incomplete or sometimes give an incorrect answer.

Even worse, this is true even if we want a test for interpretation only for strongly normalizable terms. The major type-systems can only type *some* of the strongly normalizable programs; and the remainder do not seem to be all interpretive or all non-intepretive. The example from [Urz97]:

```
(λx. z
        (x (λf,u. f u))
        (x (λv,g. g v))
) (λy. y y y)
```

is apparently non-interpretive, is strongly normalizable, but is untypable in all the considered type-systems. Now, arbitrary interpretive code could be added to that example, to form an untypable program that does interpretation. For example, a semantically-redundant equality test on two Church Naturals (or anything else one considers interpretive) could be added. Hence not all untypable yet strongly normalizable programs are interpretive, and not all are non-interpretive.

4.4.3 Prevention of interpretation

A TFP type-system preferably should prevent all (but only) interpreters from being written

While not essential, we would prefer a type-system for TFP to type all and only interpretation-free programs. Hence TFP-style programs should be typable in it, but not known interpreters.

4.4.3.1 Analysis: typability of interpreters

The major type-systems all permit apparent interpreters to be written

The restrictions imposed by the major type-systems on programs do not prevent apparent interpreters from being written, ie typed. In particular, using F_2 or higher, one can define the full

range of symbolic-style tests on Church Naturals, eg ‘greater-than’ and ‘equality’. Even with the Simple typing and Hindley-Milner typing, symbolic-style tests on functions are typable. For example, one can define three symbols and equality on them as (the below is written for the ‘Haskell’ programming language, using Hindley-Milner types):

```
type Threesymbols t = t —> t —> t —> t
symbol1 :: Threesymbols t
symbol1 a b c = a

symbol2 :: Threesymbols t
symbol2 a b c = b

symbol3 :: Threesymbols t
symbol3 a b c = c

eq :: (Threesymbols (t —> t —> t)) —> (Threesymbols (t —> t —> t)) —> (t —> t —> t)
eq x y  = x (y true false false) (y false true false) (y false false true)
```

Similarly, the ‘iszero’ test on Church Naturals can be written, type-correctly, as:

```
type Church t = (t —> t) —> t —> t
iszero :: Church (a —> a —> a) —> a —> a —> a

cfalse x = false
iszero n = n cfalse true
```

4.4.3.2 Analysis: typability of TFP-style programs

None of the major type-systems can type all TFP-style programs

We now consider the typability of TFP-style programs in the various type-systems. The fact that the type-systems can only type strongly normalizable terms is not a constraint, as recall that TFP-style programs lack recursion etc; they are (by definition) total. However, the use of higher-order functions in TFP-style programs places significant demands on type-systems. We

show that none of the major type-systems can type all TFP-style programs; however F_ω can type the most.

We also note an affinity between TFP-style programs and polymorphic typing, which leads to good error-detection capabilities; and conclude with mention of the fact that programming languages could offer special support for TFP-style programs.

4.4.3.2.1 Hindley-Milner and the Simple typing

Hindley-Milner and also the Simple typing cannot type all TFP-style programs

Given that the Simple typing and Hindley-Milner typing type essentially the same programs as discussed previously, we consider them together here. We focus on computations on Church Naturals (which are typable in all the type-systems), as there exist non-contrived examples of considerable type complexity.

To program with the Simple typing or Hindley-Milner typing, one has the significant restriction that each variable bound to a parameter-supplied function must have a single monomorphic type sufficient for typing its every application. This lack of polymorphism makes it no surprise that not every TFP-style program can be typed in Hindley-Milner or the Simple typing. Citing results of Statman, [FLO83] states that functions for Church Natural equality, ordering and subtraction are not typable. The last of these particularly points to a serious lack of typability. Subtraction on Naturals can be implemented as a simple iteration:

```
subtraction a b = b predecessor a
```

A typable implementation of 'predecessor' is given in [FLO83]. Hence, while 'a', 'b', and 'predecessor' can all be typed, this application cannot. Some related typability results can be found in [HK96].

Due to the lengthy detail involved, we present the rest of our discussion of Hindley-Milner typing for TFP-style programming as an Appendix. In that discussion we show how some TFP-style programs untypable in Hindley-Milner can be mechanically transformed into semantically-equivalent typable ones. This is only a partial solution, however: it is demonstrated comprehensively that Hindley-Milner typing is inadequate for TFP-style programming.

4.4.3.2.2 System F Typing

System F cannot type all TFP-style programs

The ability of '∀' in System F to range over polytypes (implying impredicativity) means that System F can type more programs than Hindley-Milner. For example the 'subtraction' implementation above is typable in System F as shown in [FLO83]. The underlying reason for this is as follows. As discussed in detail in the Appendix, for a function to be iterated via a Church Natural the type of its input must be the same as the type of its output, since the type of a Church Natural is '∀a. (a —> a) —> a —> a'. No implementation of 'predecessor' in Hindley-Milner has the same output type as input type. However in System F more types are available, and 'predecessor' can be given a type such as '(∀b. Church b) —> (∀b. Church b)' which lets it be iterated.

Further, System F lets a single input be supplied as multiple functions' parameters in more cases than Hindley-Milner, as terms retain their polymorphic type after they are bound as arguments. There is no requirement as in Hindley-Milner typing that a variable have a single monotype sufficient for all of its applications.

The typable functions from Church Natural to Church Natural in System F are precisely the provably Total Recursive Functions of second-order Peano Arithmetic ([Gir73]). This includes not only functions such as subtraction, but also Ackermann's function: one definition of such using System F is (from [BB85], with some renaming):

aug = (λg:(Nat —> Nat), n:Nat. n Nat g (g cnone))

ack = (λn:Nat, m:Nat. n (Nat —> Nat) aug cnsucc m)

where:

Nat = (∀α. (α —> α) —> α —> α)

cnone = (Λα. λf:(α —> α), x:α. f x)

cnsucc = (λn: Nat. Λα. λf:(α —> α), x:α. f (n α f x))

Other examples of TFP-style programming in System F can be found in [Rey85], [BB85], [GLT89, pp84-94], [PDM89], and [Mai91]. No limitations of System F for such programs have been noted in these works.

The language FCP, introduced previously, has also been used to type TFP-style programs. Recall that FCP's use of data-types as type-annotations gives an intermediate between Church and Curry style System F with decidable type checking and type inference. One example given in the paper [Jon97] is, as slightly modified by us:

```
data ChurchNat = WrapCN ((a —> a) —> a —> a)
unWrapCN :: ChurchNat —> (a —> a) —> a —> a
unWrapCN (WrapCN n) = n
```

Breaking here, we note that 'WrapCN' is the name of a new data-constructor, which can be used to 'wrap up' a function into a data-structure. Extraction of 'wrapped' functions is done by 'unWrapCN' which is written as a pattern-match on the data-constructor. FCP has been implemented as part of the popular modern 'Haskell 98' language implementations 'GHC' and 'HUGS'. In these languages, one would write instead:

```
newtype ChurchNat = WrapCN (forall a. (a —> a) —> a —> a)
unWrapCN (WrapCN x) = x
```

Continuing:

```
cnzero :: ChurchNat
cnzero = WrapCN (λf,x. x)

cnsucc :: ChurchNat —> ChurchNat
cnsucc = WrapCN (λf,x. (unWrapCN n) f (f x))

cnadd :: ChurchNat —> ChurchNat —> ChurchNat
cnadd n m = (unWrapCN n) cnsucc m
```

Any qualms about introducing primitive data in TFP programs can be calmed by noting that the introduction of data can be considered to be merely an implementation detail, and hence data need not be part of a TFP language. As a first attempt, one could consider inferring the 'wrapping' and 'unwrapping' from the types, allowing one to write the above in the right language as:

```
TFPType ChurchNat = (a —> a) —> a —> a
cnzero :: ChurchNat
cnzero = (λf,x. x)
cnsucc :: ChurchNat —> ChurchNat
cnsucc = (λf,x. n f (f x))
cnadd :: ChurchNat —> ChurchNat —> ChurchNat
cnadd n m = n cnsucc m
```

The key language-capabilities required to support this are to call a 'WrapCN'-style operator whenever a 'TFPtype' is expected but not supplied; and to call a 'unWrapCN'-style operator when a 'TFPtype' is supplied but not expected.

Some other examples of TFP-style programming in System F using FCP:

1. Exponentiation, and self-exponentiation featuring self-application:

```
cnpow a b = WrapCN ((unWrapCN b) (unWrapCN a))
selfpow a = cnpow a a
```

2. Subtraction:

```
cnsub n m = (unWrapCN m) cnpred n
```

3. Ackermann's function:

```
aug g n = (unWrapCN n) g (g cnone)
ack m n = (unWrapCN m) aug cnsucc n
```

Note that no types are required for the above in 'Haskell', even for Ackermann's function; all are able to be inferred.

Unhappily, all the above suggests an unjustifiably rosy view of System F typing. As noted previously, the Simple typing cannot type:

```
(λf,x. f (f x)) (λg,h. g)
```

which is simply the Church Natural for '2' applied to the Church Boolean for 'true'. System F cannot do much better than this; recall that it fails to type:

```
(λf,x. f (f x)) (λf,x. f (f x)) (λg,h. g)
```

ie the Church Natural for 2 raised to the second power, applied to a Church Boolean: a TFP-style program. Even without making the connection to TFP-style programming, the simplicity of this

untypable program indicates that the limitations of the type-system will be reached sooner or later, when programming TFP-style.

Hence System F cannot type all TFP-style programs. Examples of terms untypable in F_ω and $F_{n>2}$ below can also be considered examples of terms untypable in System F, as System F is a fragment of those type-systems.

4.4.3.2.3 $F_{n>2}$ and F_ω Typing

$F_{n>2}$ and F_ω cannot type all TFP-style programs

As demonstrated in [PDM89], F_3 is a better fit for TFP-style programming than System F; and [LLMP89] indicates that F_ω is a better fit again (examples appear therein of TFP-style programs in F_ω).

In [Gir73] Girard shows that the functions between Church Naturals typable in 'F_x', 'x' being two or greater, are exactly the provably Total Recursive Functions of 'x'th order Peano Arithmetic on Church Naturals. Some further information about typability of functions on the Church Naturals can be found in [Urz97]. However, recall the term given in [Urz97], which is untypable in F_ω and hence in $F_{n>2}$:

```
(λx. z
        (x (λf,u. f u))
        (x (λv,g. g v))
) (λy. y y y)
```

From this example, TFP-style terms also untypable in F_ω and $F_{n>2}$ and System F can be derived, such as:

```
(λyyy. cnadd
        (yyy cnone)
        (yyy maketuple1 (maketuple2 cnzero cnsucc))
) (λy. y y y)
```

where we use TFP-style tuple representations:

```
maketuple1 = (λv,g. g v)
maketuple2 = (λv,w,g. g v w)
```

For the first argument to ‘cnadd’, note that:

cnone = (λf,u. f u)

yyy = (λy. y y y) = (λy. cnexp y (cnexp y y))

consequently:

yyy cnone = cnexp cnone (cnexp cnone cnone) = cnone

For the second argument, note that:

yyy maketuple1 = (λy. y y y) (λv,g. g v) = (λg. g (λv,h. h v))

maketuple2 cnzero cnsucc = (λj. j cnzero cnsucc)

(λg. g (λv,h. h v)) (λj. j cnzero cnsucc) = cnsucc cnzero

consequently:

yyy maketuple1 (maketuple2 cnzero cnsucc) = cnsucc cnzero

Hence, the term denotes simply ‘(λf,x. f (f x))’ ie ‘cntwo’.

Even if one rejects this example as contrived, the term’s simplicity leads one to expect that the limitations of the type-system will be reached sooner or later, when programming TFP-style terms. However, one would expect that with Ackermann’s function already being expressible in System F that surely ‘most’ TFP-style programming would be typable in F_ω.

4.5 Summary of suitability of the major type-systems

None of the major type-systems are perfectly suited for TFP, however some appear acceptable

All the major type-systems have an important affinity with TFP, namely in typing only strongly normalizable terms. However none help us characterise interpretation or can prevent interpretation. Also, none turn out to be perfectly suitable for TFP with regards to typability: none can type all TFP-style programs. As we’ve seen, the Simple typing and Hindley-Milner typing cannot type enough TFP-style programs to be considered acceptable; but have computable type inference and type checking for Curry style programs. The next system up, System F, appears to be a better choice. It can type Ackermann’s function, which may well indicate that an acceptably-large subset of TFP-style programs can be typed in it. And while Church style System F requires an impractical amount of type annotations from the programmer for TFP-style programming, and Curry-style System F has incomputable type checking,

intermediates between the two styles exist. Some of these intermediates as we've seen have computable type checking and type inference, and burden the programmer only a little with their requirements for type annotations. Using F_ω as a type-system would be even better, as it can type more TFP-style programs than System F. However we know of no intermediate between Church and Curry style of this most-powerful type-system which has not only has significant type reconstruction abilities but also computable type checking and type inference. LEAP is closest, but has in theory (but perhaps not practice) problematic type inferencing.

4.6 Additional type-systems

Certain extensions to these major type-systems appear suitable for TFP

There exist many other type-systems than the ones considered above. Many of them are extensions of the major type-systems formed by adding new type-constructors and/or new type-forming rules. These extensions typically attempt to be 'practical' (ie have computable type checking etc) yet type a large number of terms. We discuss now a handful of particularly-interesting extensions, and give a brief overview of some others. It should be noted that these extended type-systems impose restrictions on allowable programs, typically resulting in sub-recursive power. However, as with the major type-systems, these restrictions seem unrelated to anything to do with interpretation. Specifically, the type-systems below are typically extensions to the Simple typing, which we know can type some apparent interpreters.

Finally, we consider a different paradigm of type-system, 'dynamic typing'.

4.6.1 Rank 2 predicative and impredicative polymorphism

The Rank 2 restriction of System F has computable type inference and type checking, but not principal types and cannot type all reasonable TFP-style programs

By modifying the type-forming rules, the polymorphism of System F can be restricted. This can be a useful thing to do, for as we've seen System F is so powerful that type inference and type checking (for Curry style terms) is incomputable. A number of useful results can be found in [KW94], which discusses in detail the restriction of System F to so-called 'Rank 2' types via

appropriate changes to the type-forming rules. This restricted form of System F types strictly more terms than Hindley-Milner, but strictly less than full System F. Firstly, somewhat surprisingly, it is shown that precisely the same set of terms are typable whether universally-quantified variables range only over monotypes (predicativity), or also over polytypes (impredicativity). Secondly, as shown, both type inference and type checking are computable for Rank 2 System F, while they are incomputable for Rank 3 and higher. Hence in this specific sense, Rank 2 System F is as powerful as System F can be while remaining 'practical'. Finally, it is also shown that Rank 2 terms in general lack principal types, due to the limited polymorphism available.

We will show later that arbitrary-Rank predicative polymorphism fails to type some reasonable TFP-style programs. Hence as Rank 2 polymorphism is a fragment of such, it too is also inadequate for the needs of TFP.

4.6.2 Intersection type-constructors: Intersection typing

Intersection typing is an interesting alternative to polymorphism via universal quantification

Of all the extensions, adding to the Simple typing type-constructors for the 'intersection' of types is of particular interest due to its practicalities, as detailed below.

An example of a term typed with the intersection typing: (taken from [Jim95]):

$(\lambda x.\ x\ x) : (t \rightarrow t) \cap ((t \rightarrow t) \rightarrow (t \rightarrow t)) \rightarrow (t \rightarrow t)$

Intersection typing involves *explicitly listing* all the types a function is to have (ie entails 'ad hoc' or 'discrete' polymorphism). This is in contrast to '∀' ('parametric') polymorphism, which Intersection typing has been advocated (eg [CW05]) as a practical alternative to. Intersection typing is able to type more programs than even F_ω, is simpler than it, and also still only types (assuming no 'universal type') the strongly normalizable terms. This indicates that polymorphism based on universal quantification is actually *more* restrictive than 'discrete' polymorphism.

However, Intersection typing can be *too* powerful, and it is unclear if a form of Intersection typing exists which is suitable for typing TFP-style programs.

Firstly, as indicated eg in [Lei90], the canonical Intersection typing, ie Simple typing plus the '∩' type-constructor, is able to type *precisely* the strongly normalizable terms. Hence type inference is in general incomputable. This is also stated in [Wel96] and [BH90]; the latter of which indicates that this is due to intersection typing having the 'subject conversion' property (ie both 'subject reduction' and its converse, 'subject expansion'). As also stated in [BH90], type checking is also, as expected, undecidable. Interestingly, [Fla93] shows that a certain intermediate between Church and Curry style Intersection typing can have computable type inference, albeit the problem is still NP-hard.

Secondly, by typing all strongly normalizable terms Intersection typing does not permit much in the way of detection of program-errors via type inference: it can simply make a term have any required type via use of '∩'. Universally-quantified types in contrast require a 'uniformity' which is conducive to detecting mis-applications.

Thirdly, it should be noted that with Intersection typing a function may not have a type suitable for all its inputs (when infinite in number); a simple example is given in [BPS03]. Such functions are said to have 'non-uniform' type. However, by letting the list of types for a term be infinite [Lei90] shows how this 'non-uniformity' reduces somewhat, among other benefits.

Finally, of particular interest to us are the results proved in [BPS03], namely that the functions on Church Naturals with 'uniform' types (in various formulations of Intersection typing) are precisely the functions which are typable with the Simple typing. As the Simple typing, as we've seen, doesn't suffice to type all the TFP-style programs we would desire to write, use of Intersection typing will mean having 'non-uniform' types for some TFP-style functions.

4.6.3 Rank 2 Intersection typing

The Rank 2 restriction of Intersection typing has decidable type inference and also principal types, but cannot type all reasonable TFP-style programs

As discussed in [Jim95], the Rank 2 restriction of Intersection typing has computable type inference (for Curry style terms). Further, as stated in [KMTW99], type inference is computable for *any* finite Rank, but its complexity increases exponentially with the Rank. Also shown in [Jim95] is that, unexpectedly perhaps, Rank 2 Intersection typing types precisely the same

programs as Rank 2 System F. It also has principal types, unlike Rank 2 System F. However, as Rank 2 Intersection typing has identical typability as Rank 2 System F, it too must fail to type all reasonable TFP-style programs.

4.6.4 Arbitrary-Rank predicative polymorphism

Arbitrary-Rank predicative polymorphism cannot type some reasonable TFP-style programs; impredicativity is required

The popular 'Haskell 98' (see [Pey03]) language offers various extensions of the Hindley-Milner type-system such as 'type classes' and existential qualification. What is of interest to us here however is that the 'GHC' implementation of 'Haskell 98' also supports arbitrary-Rank (predicative) polymorphism, via the algorithm presented in [PVWS07]. As a result, it can type more terms than Hindley-Milner typing but less terms than System F. Below is an example of a program GHC can type. Note that user-entered type-annotations are required, specifically any types which include 'Polychurch' cannot be elided.

```
type Church t = (t —> t) —> t —> t
type Polychurch = (forall t. (t —> t) —> t —> t)

cnzero :: Church a
cnzero f x = x
cnone :: Church a
cnone f x = f x
cntwo :: Church a
cntwo f x = f (f x)

cnpow :: Polychurch —> Polychurch —> Polychurch
cnpow v w = w v

selfexponentiate :: Polychurch —> Polychurch
selfexponentiate n = n n

test :: Polychurch —> Polychurch
```

```
test n = (cnpow (cnpow n n) n)
```

A function expecting a ‘Polychurch’ can of course be supplied with a value of type ‘Church a’. It cannot however be supplied with a value of type ‘Church Polychurch’: being predicative, the ‘t’ in ‘forall t. (t —> t) —> t —> t’ cannot be ‘Polychurch’ as ‘forall’ (‘∀’) doesn’t range over polytypes. Hence, for example, given a function ‘cnpred’ of type ‘Polychurch —> Polychurch’, the function

```
cnsub n m = m cnpred n
```

needs to be typed as:

```
cnsub :: Polychurch —> (Church Polychurch) —> Polychurch
```

and *not* as:

```
cnsub :: Polychurch —> Polychurch —> Polychurch
```

Annyoingly however, while ‘cnsub’ can be typed as above, the lack of impredicativity results in ‘cnsub cntwo cntwo’ for example being untypable, where:

```
cntwo f x = f (f x)
```

or: (equivalently, given the type inferencing)

```
cntwo :: Church a
cntwo f x = f (f x)
```

or:

```
cntwo :: Polychurch
cntwo f x = f (f x)
```

or:

```
cntwo :: Church Polychurch
cntwo f x = f (f x)
```

The reason for this is that the first argument to ‘cnsub’ requires a ‘Polychurch’ while the second argument requires a ‘Church Polychurch’; no term can be both concurrently. Of course, ‘cnsub’ could be supplied with ‘(λf,x. f (f x))’ twice, as literals: each will then be typed separately.

An implication of the above is that one cannot define equality on Church Naturals as:

```
cneq a b = and (cniszero (cnsub a b)) (cniszero (cnsub b a))
```

since ‘a’ and ‘b’ need to be both concurrently of type ‘Church Polychurch’ and ‘Polychurch’.

It can be seen that the same typing issues as covered in the Appendix for Hindley-Milner typing exist here. And, as for Hindley-Milner, we have found that one can define an appropriate type-conversion function:

```
cnsucc n f x = f (n f x)
cnzero f x = x
typeconvert :: (Church Polychurch) —> Polychurch
typeconvert n = n cnsucc cnzero
```

The two arguments to 'typeconvert' are required, one cannot simply have 'typeconvert n = n'. To see that this works, note that:

```
Church Polychurch = ((forall a. Church a) —> (forall a. Church a))
                        —>  (forall a. Church a) —> (forall a. Church a)
```

Hence if we supply two arguments, we get '(forall a. Church a)' ie 'Polychurch'.

For an example, we can define:

```
cnsub n m = m cnpred (typeconvert n)
cnsub :: (Church Polychurch) —> (Church Polychurch) —> Polychurch
```

which makes 'cneq' type. However, similarly with Hindley-Milner typing, a type-conversion function in the opposite direction:

```
Polychurch  —> (Church Polychurch)
```

seems impossible.

In contrast, System F doesn't have these issues. For example, using FCP, one can write simply (and with type inference):

```
cnsub n m = (unWrapCN m) cnpred n

k a b = a
cniszero n = (unWrapCN n) (k false) true

cneq a b =  cnand (cniszero (cnsub a b)) (cniszero (cnsub b a))
```

Similarly, as presented previously, the standard definition of Ackermann's function can also be typed using System F, and specifically, FCP. In contrast, typing it using arbitrary-Rank polymorphism is problematic. From the definitions:

```
aug g n = n g (g cnone)
ack m n = m aug cnsucc n
```

one notes that ‘aug’ must take an input of the same type as its output, in order to be iterated: it must be of type ‘α —> α’, for some ‘α’. However, the type of ‘aug’ must be ‘(β —> β) —> ((Church β) —> β)’ for some ‘β’, based on its uses of ‘n’ and ‘g’. This cannot be unified with ‘α —> α’, without impredicativity.

In summary, arbitrary-Rank predicative polymorphism lifts some of the restrictions of Hindley-Milner typing, but not all: the same problems arise, just at a later stage. In contrast, System F can type strictly more programs, and implementations such as FCP have better type reconstruction than the implementation of arbitrary-Rank polymorphism in GHC. The standard implementation of Ackermann’s function on Church Naturals is not an unreasonable TFP-style program, hence as shown by trying to type it, TFP requires more than mere (arbitrary-Rank) predicative polymorphism: impredicative polymorphism is required, ie quantification over polytypes.

4.6.5 Other extensions of the major type-systems

Other extensions of the major type-systems may also be of interest

There are a large number of other extensions to the major type-systems, too many to investigate in detail here. Useful papers to consult are [Rey85], and especially [BH90] with its presentation of extensions of Hindley-Milner typing and especially its ‘generalised type-systems’ (see also [Bar91]). The paper [SU98] contains similar information on generalised type-systems (and the famous ‘Lambda Cube’). It also contains a good overview of the Curry Howard isomorphism, mentioned previously, and also typing generally. In [Wel96] can be found a table giving the decidability of type checking and type inference for various extended systems. A number of these extensions have found their way into practical programming languages, for example ‘QUEST’ (see [CL90]), which is an extension of F_ω with recursive types, subtypes, and more.

Many extended type-systems are simply Hindley-Milner typing or the Simple typing, with a new type-constructor added. The ‘intersection’ type-constructor has already been discussed. Other type-constructors include ‘union’, ‘subtyping’, and ‘recursive’ (‘μ’) types and even type *destructors* ([HP98], [Wei01]). In [BH90] we find stated that type checking of ‘recursive’ types

(as an extension to the Simple typing) is decidable; however *every* term has an inferable type. Decidability results are also given for other variant type-systems.

Other type-systems add a new quantifier to one of the major type-systems. An example of this is adding existential quantification. Existential typing helps support 'abstract data-types' (and more generally, 'objects'), see [MP88] and [CW85]. Using existential quantification, functions can take inputs which can't be directly (mis)used by the function but instead can only used with other supplied functions. Existential typing may be useful, but isn't a necessity for TFP languages.

Sometimes new type-rules are introduced. For example, practical implementations of Hindley-Milner and other type-systems often contain extensions such as support for recursion ([Jim95] has a brief but interesting discussion; see also [KTU93]). Other type-systems offer more fundamental extensions or changes to types. Some extensions offer modified forms of type-construction, such as bounded universal and/or existential qualification (see eg [CW85]).

Finally, some extensions focus primarily on making more properties of programs statically checkable via some fundamental extensions, for example by having types depend on the terms, ie 'dependent types' (see [XP99], [AMM05], [SU98], [BH90]). The utility of such for TFP languages remains to be established. However as pointed out in [AMM05], with dependent types one can express that not all values of a type can be used in the same way. This could potentially be of use with regards to matters of interpretation.

It should be noted that all these type-systems are extensions of the Simple typing, and hence will not prevent apparent interpreters from being written.

4.6.6 A different paradigm: dynamic typing

One can check things with dynamic typing that can't be checked using static analysis

All the major type-systems and extensions we discuss in this Chapter are 'static' in nature: all type checking is done via static analysis. In general this analysis can be quite complex, involving higher-order unification and theorem proving (cf [Ker94]). However other forms of type-system exist, specifically those which have a 'dynamic' component to type checking.

Practically, this permits more programs to be accepted. Static type checking aims to ensure that *all* executions of the program are correct with regards to typing, and this can cause undecidability to rear its head. Dynamic type-systems in contrast engage in runtime checks of types, ie they check *each actual* program-execution (the 'all' versus 'each' comparison was pointed out to us by Bailes, private communication). Dynamic typing is of lesser utility to the programmer because of this. In language implementations, it is not uncommon for the type-system to have both a static and a dynamic component; the static component being primary and dynamic typing used to detect errors when static analysis fails. Also, it is possible for many dynamic checks to be verified statically, see eg [AWL94]. Another interesting system is detailed in [ACPR95], which combines dynamic typing with polymorphic typing.

It may be worth considering if adding a dynamic component to type checking will be of benefit for TFP.

An interesting fact regarding dynamic typing is that it is possible to implement the typing as part of the program itself; in other words it is possible that the programmer can write their programs in such a way that they 'self type check'. An example of this in a functional setting are 'dynamic assertion types' ([BK03b]), co-developed by the author; which support both 'intersection' and 'union' types via parallelism. Another, rather different yet related system can be found presented in [Wei01]. Type checking is an apparently interpretive activity (one has to test to see if a term is a member of a set), hence would not be possible to do in this fashion in an interpretation-free TFP language. It should also be noted that the limited information type checking language-terms typically have available limits their expressivity: in a functional language the only information they may use is that which is supplied to the function. More generally, languages typically do not make available to terms full information about their static and dynamic context. In contrast, standard type-systems have available to them the full syntactic and static information of the program.

4.7 Conclusion

Some existing type-system are suitable for TFP languages, but there is no apparent relationship between typability and interpretation in the type-systems considered

None of the major type-systems are *perfectly* suitable for TFP. None seem to be able to prevent interpretation, and none can type all TFP-style programs. However, some of them seem acceptable.

At first glance, all of the major type-systems all show significant promise for TFP, as all have typable only total (indeed, strongly normalizable) programs. However, the less powerful two, the Simple typing and Hindley-Milner typing, have too-small a set of typable programs to be suitable for TFP. In the Appendix is discussed in detail how to get the most out of them by use of type-conversion functions. The remaining type-systems, System F through to F_ω, still cannot type all TFP-style programs. Those that can't be typed could be considered an acceptable loss, given that Ackermann's function is typable. Note however that the undecidability of type checking for Curry style terms in those systems means that some types will need to be supplied by the programmer. For System F, FCP acts as a witness that this need not be onerous; while LEAP does the same for F_ω (albeit with an approximation). LEAP is particularly interesting in this regard, as it supports special, near- type-free, syntax for functional representations of data (such as Church Naturals). This lifts from the programmer the burden of having to type these functions, for the relevant types, as demonstrated in [LLMP89], can be quite lengthy. From simple definitions, LEAP can generate automatically both the necessary types, *and the terms* (that such is possible is covered in detail in the subsequent Chapter); for example:

```
indtype Nat:* with
        zero : Nat
        succ : Nat ─> Nat
```

suffices to generate the Church Natural 'succ' and 'zero' functions, F_ω-typed. Also discussed in [LLMP89] is how type information can be used to display nicely such functional representations to the user, as well as how these representations can be recognised and optimised via compilation back into the corresponding data.

LEAP-style F_ω would seem to be the best choice for TFP, given its greater power and those features mentioned above. However, let us briefly make the case for FCP-style System F instead. Firstly, the extra typability in LEAP is unlikely to be needed in practice: Ackermann's function does not require it. That extra typability requires more-powerful type reconstruction and additional type annotations than required for FCP. Also, the additional type-annotations of FCP ('Wrap' and 'unWrap') could be potentially inferred from the types of functions when convenient. Finally, type inference and type checking in FCP is known to be (readily) computable, while for LEAP approximations (claimed to be acceptable in practice) are required.

Other type-systems may also be feasible to use with TFP languages. While arbitrary-Rank predicative polymorphism, Rank 2 System F (the highest Rank with decidable type checking), and Rank 2 Intersection typing have all been shown to be insufficient for the needs of TFP, unrestricted Intersection typing in particular has some nice properties and could be investigated further. In addition, dynamic typing could be of interest.

Finally, we have made some discoveries with respect to type-systems and interpretation. None of the major type-systems considered prevent simpler code that we intuitively consider to be interpretive from being used, for example, 'iszero', 'equality' and 'greater-than' tests on Church Naturals. They do however prevent universal interpreters from being used, as such are partial and hence not strongly normalizable. On the other hand, [PL89] indicates that some type-systems which type only strongly normalizable terms almost permit meta-circular interpreters.

We also noted that the sub-recursive (and total) nature of the terms typable by the major type-systems, as well as the ability to compute within types in certain of them, seems to have little to offer regarding the nature of interpretation. Instead, the limited typability is simply a direct consequence of the restricted polymorphism of the type-systems.

Chapter 5

Derivation techniques for TFP

Formal derivation of TFP-style programs is possible by application of both well-known and novel techniques

Type-systems for typing TFP-style programs have been discussed in Chapter 4. In this Chapter we consider the *derivation* of TFP-style programs, and the underlying nature of TFP-style programming itself. In subsequent Chapters we discuss how interpretation impacts on language-extension and consider the nature of interpretation.

There are a number of examples of programming presented in Chapter 2 that have been identified in TFP as apparently using less interpretation than one would expect for the operations involved. For the purposes of TFP, it is desirable to both be able to derive these examples and others in the TFP style, and to explain how they are able to avoid interpretation. In this Chapter we show how all the TFP-style functions of Chapter 2 can be systematically derived. In fact, we show how they can be (often mechanically) derived from more-usual symbol-testing implementations. The techniques involve replacing data, recursion, and symbol-tests with either definitions, or lambda-abstractions and β-reductions. A number of the techniques are taken directly from the functional programming literature.

Firstly, we categorise the various TFP-style functions presented in Chapter 2. Secondly, we identify the essential nature of the functions used as alternatives to data. Based on both of these, a series of derivation techniques are introduced which derive both TFP-style and non- TFP-style functions. These are then rationalised by recognising occurrences of a particular operation, 'fold'. We then present some additional derivation techniques that allow the results of such functions to be further transformed; this is followed by some derivations of particular interest. Subsequently we consider what can be learnt from the techniques about the nature of interpretation, how to remove it from programs, and how to eschew it in programming more generally. Finally, we give our concluding remarks, which include a summary of the techniques.

Throughout, we illustrate our techniques with worked derivations. To avoid excess (and possibly obscuring) verbosity, Combinator Parsers, Programmed Graph Reduction, and Exact Real Arithmetic are avoided as examples.

5.1 Categorisation of functions used in the identified TFP-style programs

The various functions used by the examples of TFP-style programming presented in Chapter 2 can be categorised by identifying correspondences between them and primitive data and operations thereon

Essentially, the various exemplars of apparently interpretation-eschewing programming given in Chapter 2 show how functions can be used as alternatives to primitive data. The functions involved can be categorised into three classes.

Firstly, specific functions in the exemplars act as replacements for individual items of data. Rather than Naturals, one has the Church Naturals ('(λf,x. x)' for zero, '(λf,x. f x)' for one, etc), rather than Booleans one has the Church Booleans, and rather than lists one has Church Lists. Similarly, instead of using data to represent reals, one represents reals with functions in Exact Real Arithmetic. In Programmed Graph Reduction, data-structures representing terms are represented by executable code acting as functions. Finally, in Characteristic Predicates and Combinator Parsers, functions take the place of specific data-structures representing sets and grammars respectively. We will refer to the set of functions that correspond to the members of a given data-type as a 'Data-Alternative'. For example, the set of Church Naturals forms a Data-Alternative. Each function of a Data-Alternative we will refer to as a 'Data-Alternative Element', ie each Church Natural is a Data-Alternative Element.

Secondly, the exemplars also contain functions that correspond to constructors of data-types; we call these functions 'Data-Alternative Constructors'. For example, in Chapter 2 are functions representing 'succ' and 'zero', the canonical constructors for the Natural data-type; and functions for 'cons' and 'nil', the usual constructors for lists. Similarly, there are functions corresponding to the alternation, concatenation, and literal-token data constructors for grammars in Combinator Parsers, and functions representing the data constructors we use for sets in this work, 'union', 'singleton', and 'empty'. Note that a data-type for sets can include other symbols

representing whole sets, eg ‘Evens’ for the set of even numbers. Corresponding to these are Characteristic Predicate functions of the type ‘E —> Boolean’, where ‘E’ is the chosen type of set-elements.

Third and finally, the exemplars also include functions that mirror operations on the data-type other than the data constructors. We term these functions ‘Data-Alternative Non-Constructors’. For example, in Chapter 2 there is a function that implements addition on Church Naturals, and a function that sums the elements of a Church List.

It should be noted that there is some overlap between the first and second categories. For example, the Church Natural for zero (‘(λf,x. x)’) is both a Data-Alternative Constructor, and a Data-Alternative Element. This occurs because at least one constructor for a given (practical) data-type must be ‘zero-arity’ (‘nullary’) in that type, ie requires no arguments of the type to be able to return a member of the type. The Boolean data-type is somewhat special as not being recursive all constructors of it are zero-arity; hence its corresponding Data-Alternative Elements are precisely its Data-Alternative Constructors.

5.2 Identification of the essential nature of the functional representations of data

Functional representations of data have a ‘trivial’ operation, implementable as the identity function

As noted in Chapter 2, some of the Data-Alternatives have an operation that can be implemented trivially on them. For example, for Church Boolean ‘b’, ‘IF b = b’. One could hypothesise that *every* Data-Alternative from Chapter 2 has an operation implementable as the identity function, and indeed, this is correct. We list now the Data-Alternatives from Chapter 2 and their trivially-implementable operation:

Church Natural: iterate
Church Boolean: IF
Church List: reduce
Characteristic Predicates: membershiptest
Combinator Parsers: parse

Programmed Graph Reduction: apply

Exact Real Arithmetic: valuewithinprecision

Note that we assume that these operations are defined so that they take the data or Data-Alternative Element as their first parameter. This is done so that when Data-Alternatives are used the operations are all implementable by the identity function ('id'), rather than a variety functions of the form '(λa,b,c,d. c a b d)' etc. The effect of this assumption is only to permit a simpler presentation.

Also, the operations 'valuewithinprecision', 'apply', and 'parse' were not introduced in Chapter 2, but are self-explanatory given their inputs.

Finally, note that if it is recognised that the Data-Alternative Elements 'are' something, then that some operation has a trivial implementation on them follows immediately. For example, Church Naturals 'are' iterators, eg '(λf,x. f (f x))' is two-fold iteration. This means that Church Naturals 'are already' iterators, ie the identity operation suffices to produce iterators from them.

5.3 Deriving TFP-style functions from data-oriented code

A number of derivation techniques, many well-known in the literature, can be used to derive various TFP-style functions (and others besides) from code that does symbol-processing

We consider below a number of techniques, many of them well-known in the literature, which can derive many of the TFP-style functions identified in Chapter 2 from data-centric implementations. Deriving certain of the TFP-style functions however requires additional techniques, presented in the subsequent section.

It should be noted that the Data-Alternative Elements, Constructors, and Non-Constructors able to be derived using these techniques are not all in the TFP-style; some contain symbols, tests on symbols, and recursion.

5.3.1 Derivation technique: simplifying applications

The functional representations of data can be derived by simplifying applications of functions to data

The existence of an operations trivially-implementable for a given Data-Alternative, or equivalently, that the Data-Alterative Elements 'are' something, is we have discovered sufficient for derivation of the Data-Alternative Elements. One simply applies an implementation of the operation to each member of the chosen data-type, and simplifies.

For example, to produce the Church Natural for 'two', the function which does two-fold iteration, all one has to do is take an implementation of iteration on Naturals, say:

iterate ≙ (λn,f,x. if n == zero then x else if n == succ m then f (iterate m f x))

and apply it to the data 'two' ('succ (succ zero)'):

iterate (succ (succ zero))

This term contains symbols, tests, and recursion. However simplification, evaluation suffices in this case, gives the desired TFP-style function:

iterate (succ (succ zero))
= (λn,f,x. if n == zero then x else if n == succ m then f (iterate m f x))
→* (succ (succ zero))
→* (λf,x. f (f x))

All the Data-Alternative Elements appearing in Chapter 2 can be derived by forming and then simplifying such applications; all one has to do is identify which operator has a trivial implementation on the Data-Alternative Elements. The simplification can often remove all symbols, tests, and recursion, ie form TFP-style functions.

This technique can be seen to 'enliven' inert data ie symbols, by combining them with semantics. One can additionally note that by the above derivation technique, the trivially-implementable operation is in some sense the *intrinsic* operation of the Data-Alternative. As we discuss later, the choice of intrinsic operation can result in certain operations on the data (constructors and/or non-constructors) being non-implementable on the Data-Alternative.

In terms of novelty, the link between the Data-Alternative Elements and operation-implementations on data has occasionally been independently noted in the literature, but only for individual examples of higher-order programming. We have not seen elsewhere a table such as the one given previously relating Data-Alternatives to intrinsic operations, nor found recognition of a general class of functions derivable from simplifications of applications.

Finally, we illustrate use of this derivation technique to derive Elements of a new Data-Alternative. The Data-Alternative we want is for lists, so that lists 'are' membership tests. Given an implementation of the desired membership operation:

isin ≙ (λm,x. if m == nil then false else if m == cons e r then (x == e) or (isin r x))

one simply applies the function to members of the data-type, lists, and simplifies. For example, for the list containing the number 3:

isin (cons 3 nil)
= (λx. if (cons 3 nil) == nil then false else
if (cons 3 nil) == cons e r then (x == e) or (isin r x))
= (λx. if false then false else if (cons 3 nil) == cons e r then (x == e) or (isin r x))
= (λx. if (cons 3 nil) == cons e r then (x == e) or (isin r x))
= (λx. (x == 3) or (isin nil x))
= (λx. (x == 3) or false)
= (λx. x == 3)

5.3.2 Derivation technique: homomorphisms

The correspondence between TFP-style functions, and data and operation-implementations thereon, can be formalised and used immediately as the basis of a derivation technique

We begin by formalising as a homomorphism what it means for functions to 'represent' data and operation-implementations thereon. We then use that formalisation as the basis of a derivation technique for deriving Data-Alterative Constructors and Non-Constructors. We illustrate the technique with a number of examples and also use it to determine when such functions can exist.

5.3.2.1 Formalisation of 'represents'

What it means for a set of functions to 'represent' data and operation-implementations thereon can be formalised as a homomorphism

Homomorphisms are well-known means of formalising when one system 'represents' another. For a homomorphism to exist where Data-Alternatives are concerned, there must exist a function '$M_{d\rightarrow a}$' such that for every implementation 'O_d' (on data-type 'DT') of an operation taking 'n' arguments, one of the functions on the Data-Alternative must implement the same operation (on the Data-Alternative), ie be the 'O_a' in:

$$\forall D_1..D_n{:}DT.\ O_a\ (M_{d\rightarrow a}\ D_1)..\ (M_{d\rightarrow a}\ D_n) = M_{d\rightarrow a}\ (O_d\ D_1..\ D_n)$$

Usually one has a particular set of 'O_d' in mind, which includes at a minimum the constructors for the data-type. Also, note that '$M_{d\rightarrow a}$' is a function that maps from items of data of the chosen data-type 'DT' to Data-Alternative Elements. Hence it is the intrinsic operation of the Data-Alternative, implemented on data.

The equation above is the typical one, however sometimes some modifications are required. For example, some data-types are what we call 'container types': lists and sets are examples of these. For 'container types', typically the elements the 'container' holds are from another data-type and are not required to be different in the Data-Alternative compared to the data-structure. Hence in the equations, they do not have '$M_{d\rightarrow a}$' applied to them.

5.3.2.2 From equations to derivations

From the equations of the homomorphism, TFP-style functions can be derived

We have discovered that the equations of the homomorphism can be used to derive the various Data-Alternative functions 'O_a'. One takes each equation and simply solves for the 'O_a' as the unknown, as follows.

Firstly, when 'n' is zero the equation is simply:

$$O_a = M_{d\rightarrow a}\ O_d$$

An implementation of 'O_a' can be immediately read-off, but the right-hand side will usually be desired to be simplified as far as possible.

Secondly, for arbitrary 'n', the equation is:

$$\forall D_1..D_n{:}DT.\ O_a\ (M_{d\to a}\ D_1)..\ (M_{d\to a}\ D_n) = M_{d\to a}\ (O_d\ D_1..\ D_n)$$

One tries to transform (eg by evaluating) the right-hand side so that the 'D_1' through to 'D_n' appear only as operands to '$M_{d\to a}$'. One can then abstract '$(M_{d\to a}\ D_1)$' through '$(M_{d\to a}\ D_n)$' from both sides, giving:

$$\forall D_1..D_n{:}DT.\quad \text{let } Z_1 = (M_{d\to a}\ Z_1),..\ Z_n = (M_{d\to a}\ Z_n) \text{ in}$$
$$O_a\ Z_1..\ Z_n = \ldots$$

We can rewrite this to:

$$\forall Z_1..Z_n{:}\{Z \mid \exists D{:}DT.\ Z = M_{d\to a}\ D\}.\ O_a\ Z_1..\ Z_n = \ldots$$

Now, '$\{Z \mid \exists D{:}DT.\ Z = M_{d\to a}\ D\}$' is just the Data-Alternative Elements. Hence the equation is equivalently:

$$\forall Z_1..Z_n{:}\text{Data-Alternative}.\ O_a\ Z_1..\ Z_n = \ldots$$

An implementation of 'O_a' can now be read off. Again, one may wish to simplify as far as possible the right-hand side; this can be difficult to do.

This technique can be used to derive Data-Alternative Constructors and Non-Constructors. It can also be used to derive the Data-Alternative Elements; by notionally introducing as required corresponding zero-arity data constructors 'O_d':

$$O_a = M_{d\to a}\ O_d$$

This is however just a restatement that application of an implementation of the intrinsic operation to the members of the data-type gives Data-Alternative Elements.

5.3.2.3 Examples

Examples illustrate how the derivation technique can derive various TFP-style functions

As an example of solving the homomorphism equations via the above process, consider the primitive data-type of Natural numbers and its data constructors. With iteration as the intrinsic operation, implemented on data as 'iterate' (as defined previously), and the only operations on the data we consider the two data constructors, the equational requirements for a homomorphism are:

$\text{zero}_{\text{Church Natural}} = \text{iterate zero}_{\text{primitive Natural}}$

$\forall D_1$:primitive Natural. $\text{succ}_{\text{Church Natural}}$ (iterate D_1) = iterate ($\text{succ}_{\text{primitive Natural}}$ D_1)

The implementation of '$\text{zero}_{\text{Church Natural}}$', can be immediately read off as 'iterate $\text{zero}_{\text{primitive Natural}}$'. Simplifying this gives the TFP-style implementation appearing in Chapter 2, '(λf,x. x)'.

For the data constructor '$\text{succ}_{\text{Church Natural}}$', first evaluate the right-hand side:

iterate ($\text{succ}_{\text{primitive Natural}}$ D_1)

→* (λf,x. f (iterate D_1 f x))

This has fortuitously made 'D_1' now an operand of 'iterate'. The equation is now:

$\forall D_1$:primitive Natural. $\text{succ}_{\text{Church Natural}}$ (iterate D_1) = (λf,x. f (iterate D_1 f x))

We then abstract 'iterate D_1' from both sides, giving:

$\forall D_1$:primitive Natural. let Z_1 = iterate D_1 in

$\text{succ}_{\text{Church Natural}}$ Z_1 = (λf,x. f (Z_1 f x))

ie:

$\forall Z_1$:{Z | ∃D:primitive Natural. Z = iterate D}. $\text{succ}_{\text{Church Natural}}$ Z_1 = (λf,x. f (Z_1 f x))

Recognising '{Z | ∃D:primitive Natural. Z = iterate D}' as being the Church Naturals, we have:

$\forall Z_1$:Church Natural. $\text{succ}_{\text{Church Natural}}$ Z_1 = (λf,x. f (Z_1 f x))

In other words:

$\text{succ}_{\text{Church Natural}}$ = (λn,f,x. f (n f x))

If we had chosen a different implementation for 'iterate', say:

iterate n f x ≙ if n == zero then x else if n == succ m then iterate m f (f x)

then after going through the above processes, we would have ended up with a different implementation of the Constructor, in this case:

$\text{succ}_{\text{Church Natural}}$ = (λn,f,x. n f (f x))

As a second example, we derive Data-Alternative Constructors for Characteristic Predicates. As an implementation of their intrinsic operation on data, we use:

membershiptest d e ≜

if d == empty then false
else if d == singleton x then (e == x)
else if d == union s t then
or (membershiptest s e) (membershiptest t e)

The equations of homomorphism are:

$\text{empty}_{\text{Char Pred}}$ = membershiptest $\text{empty}_{\text{Data-structure}}$

$\forall D_1,D_2$:Data-structure. $\text{union}_{\text{Char Pred}}$ (membershiptest D_1) (membershiptest D_2)
= membershiptest ($\text{union}_{\text{Data-structure}}$ D_1 D_2)

$\forall D_1$:Element. $\text{singleton}_{\text{Char Pred}}$ D_1 = membershiptest ($\text{singleton}_{\text{Data-structure}}$ D_1)

Note that 'D_1' in the equation for 'singleton' doesn't have 'membershiptest' applied to it, the elements of the set we permit to be the same, drawn from set 'Element', for both the Data-Alternative- and the data- centric versions of the operation.

Solving for '$\text{empty}_{\text{Char Pred}}$' is trivial. For '$\text{union}_{\text{Char Pred}}$', note that

membershiptest ($\text{union}_{\text{Data-structure}}$ D_1 D_2)
→* (λe. or (membershiptest D_1 e) (membershiptest D_2 e))

hence:

$\forall D_1,D_2$:Data-structure. $\text{union}_{\text{Char Pred}}$ (membershiptest D_1) (membershiptest D_2)
= (λe. or (membershiptest D_1 e) (membershiptest D_2 e))

which is rewritable to:

$\forall Z_1,Z_2$:Characteristic Predicate. $\text{union}_{\text{Char Pred}}$ Z_1 Z_2 = (λe. or (Z_1 e) (Z_2 e))

from which an implementation of '$\text{union}_{\text{Char Pred}}$' can be read off.

Finally, for 'singleton', we have:

$\forall D_1$:Element. $\text{singleton}_{\text{Char Pred}}$ D_1 = membershiptest ($\text{singleton}_{\text{Data-structure}}$ D_1)

We can now if we choose simplify the right-hand side, resulting in:

$$\text{singleton}_{\text{Char Pred}} = (\lambda D_1.\ (\lambda x.\ x == D_1))$$

5.3.2.3 Derivability

The homomorphism equations do not always have a solution, meaning that a different functional representation of the data should be chosen

All the steps involved in solving a homomorphism equation are mechanical, apart from transforming the right-hand side of the equation into the right form for the abstraction. The ease of this transformation depends on chosen implementation of the given operation, as well as of the intrinsic operation, which appear in the right-hand side. Note that if the former is written such that those arguments which are to be bound to supplied data appear in the body only as operands to the implementation of the intrinsic operation, then the transformation is particularly simple: just instantiate those arguments with the data.

It is important to be aware that a given homomorphism equation may not *have* a solution. An example of such involves an intrinsic operation called 'isone', an operation on Naturals that returns true if its argument is one, false otherwise. The homomorphism equation for the 'successor' data constructor is:

$$\forall D_1\text{:primitive Natural.}\ \text{succ}_{\text{isone Natural}}\ (\text{isone}\ D_1) = \text{isone}\ (\text{succ}_{\text{primitive Natural}}\ D_1)$$

Now consider the second equation when 'D_1' is zero, it simplifies to:

$$\text{succ}_{\text{isone Natural}}\ \text{false} = \text{true}$$

And, when 'D_1' is two, one gets:

$$\text{succ}_{\text{isone Natural}}\ \text{false} = \text{false}$$

These two are contradictory; no implementation (actually, this argument is limited to functional programming) of 'succ' on the chosen Data-Alternative can exist. In this situation, one can switch to a different intrinsic operation to allow the operation to be implementable. More examples of unimplementability can be found (without the intrinsic operations being explicit, however) in [Int94].

5.3.2.4 Another way of solving the equations

Particular derivations can be done mechanically when appropriate inverses exist

For a given Data-Alternative, recall that the homomorphism equations to solve are of the form:

$$\forall D_1..D_n{:}DT.\ O_a\ (M_{d\to a}\ D_1)..\ (M_{d\to a}\ D_n) = M_{d\to a}\ (O_d\ D_1..\ D_n)$$

If '$M_{d\to a}$' is an inverse of some function '$M_{a\to d}$', ie:

$$\forall D{:}DT.\ M_{a\to d}\ (M_{d\to a}\ D) = D$$

then:

$$\forall D_1..\ D_n{:}DT.\ \text{let } Z_1 = M_{d\to a}\ D_1,..\ Z_n = M_{d\to a}\ D_n \text{ in}$$
$$O_a\ (M_{d\to a}\ (M_{a\to d}\ Z_1))..\ (M_{d\to a}\ (M_{a\to d}\ Z_n))$$
$$= M_{d\to a}\ (O_d\ (M_{a\to d}\ Z_1)..\ (M_{a\to d}\ Z_n))$$

which simplifies to:

$$\forall D_1..D_n{:}DT.\ \text{let } Z_1 = M_{d\to a}\ D_1,..\ Z_n = M_{d\to a}\ D_n \text{ in}$$
$$O_a\ Z_1..\ Z_n = M_{d\to a}\ (O_d\ (M_{a\to d}\ Z_1)..\ (M_{a\to d}\ Z_n))$$

ie:

$$\forall Z_1..Z_n{:}\text{Data-Alternative}.\ O_a\ Z_1..\ Z_n = M_{d\to a}\ (O_d\ (M_{a\to d}\ Z_1)..\ (M_{a\to d}\ Z_n))$$

from which an implementation of 'O_a' can be read off:

$$O_a = (\lambda Z_1..Z_n.\ M_{d\to a}\ (O_d\ (M_{a\to d}\ Z_1)..\ (M_{a\to d}\ Z_n)))$$

Hence, one can mechanically derive 'O_a' as long as such an '$M_{a\to d}$' can be found. It may not exist for a given intrinsic operation however.

Equivalently, one can start with the reverse homomorphism equations, ie with the role of data and Data-Alternative reversed. The equations are now of the form:

$$\forall Z_1..Z_n{:}\text{Data-Alternative}.\ O_d\ (M_{a\to d}\ Z_1)..\ (M_{a\to d}\ Z_n) = M_{a\to d}\ (O_a\ Z_1..\ Z_n)$$

If '$M_{a\to d}$' has an appropriate inverse '$M_{d\to a}$', then we can apply that inverse to both sides to give:

$$\forall Z_1..Z_n{:}\text{Data-Alternative}.\ M_{d\to a}\ (O_d\ (M_{a\to d}\ Z_1)..\ (M_{a\to d}\ Z_n)) = M_{d\to a}\ (M_{a\to d}\ (O_a\ Z_1..\ Z_n))$$

ie:

$$\forall Z_1..Z_n{:}\text{Data-Alternative}.\ M_{d\to a}\ (O_d\ (M_{a\to d}\ Z_1)..\ (M_{a\to d}\ Z_n)) = O_a\ Z_1..\ Z_n$$

From this, the same implementation as before of 'O_a' can be read off. Note that the implementation is somewhat complex, containing both a mapping from data to Data-Alternative,

the reverse, and the data-centric operation-implementation 'O_d'. Significant transformation may be required to simplify it to a TFP-style function.

The trick of using inverses was suggested to us by P.A. Bailes (private communication, October 2005); and that it works generally for any (strict) function with appropriate inverse can be found stated in [MFP91].

5.3.3 Derivation technique: direct replacement

Replacing all the data-centric constructs in a function by ones that take and return functional representations of data instead, results in the function also taking and returning functional representations of data

Somewhat obviously, we have found that if we directly replace in some function sub-terms that take and return data (eg primitive Naturals) with equivalent sub-terms that take and return a Data-Alternative (eg Church Naturals), then the function itself will now take and return a Data-Alternative (Church Naturals, in this case). A number of the Data-Alternative Non-Constructors can be seen to be mechanically derivable in exactly this fashion.

One common case is a function, acting on data, which makes use only of primitive data constructors and a term implementing on data the intrinsic operation of some Data-Alternative. That function can be rewritten so as to act on Data-Alternative Elements instead. Simply replace the data constructors with Data-Alternative Constructors for the Data-Alternative, and replace the term implementing the intrinsic (ie trivially-implementable, on the Data-Alternative) operation with the identity function, which can then be evaluated-away.

5.3.3.1 Examples

Examples illustrate the simplicity of the technique

As an example of this technique, consider the following implementation of addition on primitive Naturals:

add a b ≜ iterate a succ b

This function uses the data constructor ‘succ’, and the implementation of iteratation on data given previously; iteration being we recognise the intrinsic operation of Church Naturals. The replacement we will do can be semantic only, via overriding definitions; or syntactic as well as semantic, ie by replacing the construct with a lambda-term. Using the former, one gets:

iterate ≙ id

succ ≙ (λn,f,x. f (n f x))

add a b ≙ iterate a succ b

Using the latter, one has:

add a b ≙ id a (λn,f,x. f (n f x)) b

On simplification, one gets:

add a b ≙ a (λn,f,x. f (n f x)) b

Hence, we have derived an implementation of addition for Church Naturals from an implementation of addition for primitive Naturals. One should note that deriving the same result by solving homomorphism equations would require a number of non-trivial transformation steps.

Once a Data-Alternative function has been derived by the above technique, it can be used in later derivations; one can now for example derive

mul a b ≙ a (b (λn,s,z. s (n s z))) (λf,x. x)

from

mul a b ≙ iterate a (add b) zero

and the above implementation of addition. Interestingly, one doesn’t need to have Data-Alternative Elements to start a chain of derivations via direct replacement, Data-Alternative Constructors suffice. For example, via direct replacement of the data constructors that form a given item of data with Data-Alternative Constructors, we can go from:

succ (succ zero)

directly to:

(λn,f,x. f (n f x)) ((λn,f,x. f (n f x)) (λf,x. x))

which then reduces to:

(λf,x. f (f x))

ie the Church Natural for ‘2’. This isn’t surprising, as the purpose of Data-Alternative Constructors is after all to *construct* Data-Alternative Elements.

For an example on other than Naturals, consider the following data-centric function:

or x y ≜ IF x true y

Replace those data-centric implementations of 'IF' and 'true' with Church Boolean-centric implementations, and evaluating-away the identity function, gives the Church Boolean-centric function:

or x y ≜ x (λa,b. a) y

5.3.3.2 Partial replacement

It is often possible to replace only some of the constructs of a function, to change what it takes as input, or what it gives as output

One should note that one can sometimes do *partial* replacement. If the *output* of the function is produced using only terms we have Data-Alternative versions of (and these are not used for other purposes), then replacing only those terms with Data-Alternative versions will produce a modified function that takes the same input, but now gives Data-Alternative Elements as output. Similarly, if the *input* to a function is processed only by terms (which are not called on other things) we have Data-Alternative versions of, then replacing those terms gives a modified function that returns the same output, but takes now Data-Alternative Elements rather than data as input. For example:

(λm. reduce m (λe,r. add one r) zero)

is a function which calculates the length of a list. Its result is built only using 'add', 'one', and 'zero', hence replacing these by Church Natural versions will result in a Church Natural being produced but still a dataful list being taken as input. Likewise, we could note that we could instead replace 'reduce' by the identity function and give the function Church Lists instead of lists, while having the same output produced as before.

5.3.3.3 Derivability

The derivation technique cannot derive all the TFP-style functions in Chapter 2

The Data-Alternative Elements and many of the Data-Alternative Non-Constructors of the exemplars of TFP-style programming in Chapter 2 can be derived via this derivation technique, from symbol-testing functions acting on primitive data. However, the derivations typically require Data-Alternative Constructors to be known, and these must be derived using some other technique. Some of the Data-Alternative Non-Constructors from Chapter 2 that cannot be derived using this technique are:

```
add a b f x = a f (b f x)
mul a b f = a (b f)
pow a b = b a
and a b = (λx,y. a (b x y) y)
```

5.3.4 Derivation technique: parameterising the constructors of data

Parameterising the data-type constructors from an item of data mechanically derives a functional representation of it

The work [BB85] is an oft-cited paper detailing how some of what we call Data-Alternative Elements, Constructors and Non-Constructors can be derived. We begin below by considering derivation of Data-Alternative Elements (equally, zero-arity Constructors). It should be noted that [BB85] uses an expansive view of what constitutes a data-type, including support for 'container types'; and also presents typed functions. Here, we present functions untyped and give illustrative examples, the reader being referred back to [BB85] for the generalised, typed, treatment.

In [BB85], it is stated that one can take any item of data, and parameterise (ie turn into lambda-bound variables) all the constructors of its data-type to form a function representing the data. For example, given a Natural:

```
succ (succ zero)
```

one can parameterise the constructors of Naturals, 'succ' and 'zero', to give the function:

```
(λsucc,zero. succ (succ zero))
```

This is of course a Church Natural. Similarly, given:

zero

one gets:

(λsucc,zero. zero)

As our second example, given the data-type:

Colour ::= red | green | blue

one gets for the constructors after parameterisation:

red ≜ (λred,green,blue. red)

green ≜ (λred,green,blue. green)

blue ≜ (λred,green,blue. blue)

Thirdly, consider integers constructed using some given set of data constructors; one can parameterise those constructors to give functional representations of integers. Hence there is an alternative to the very common representation of integers as a pair of Naturals (Church Naturals, in [Kle35b]) whose difference is the integer's value.

It should be noted that one can treat even an entire program's source-code as a data-structure and parameterise its constructors, ie the programming constructs. At an intermediate scale, one can parameterise data constructors from sub-languages. For example:

(λsucc,zero,true,false. true)

(λsucc,zero,true,false. succ (succ zero))

are Church Object Elements, for the data-type which is the union of Booleans and Naturals. Similarly, one can consider Church Object Elements for 'lists of Naturals', eg:

(λcons,nil,succ,zero. cons (succ (succ zero)) (cons zero nil))

These examples are not intrinsically special, they merely involve somewhat unusual data-types.

In Chapter 2, only the Church Naturals, Church Booleans, and Church Lists can be seen to be derivable via parameterising constructors. We term those particular Data-Alternatives that can have their Elements derived in this fashion the 'Church Objects'; they are also known in the literature as 'Church Encodings'.

To finish, we note that one can derive functional representations of tuples, a 'container type', from the (often implicit) data-type of tuples, ie:

Tuple ::= maketuple A B

for any 'A', 'B', can be used to produce:

maketuple A B ≜ (λmaketuple. maketuple A B)

Somewhat surprisingly, we have not come across evocative variable-naming such as the above in published works, indicating perhaps a general lack of awareness that these lambda-terms for tuples ('Church Tuples', in the literature) are the result of parameterising a data constructor. Church Tuples with other than two elements can also be readily derived. Indeed, n-ary Church Tuples were defined (without derivation, and not under that name) back in [Kle35a].

5.3.5 Derivation technique: algorithm for writing-out functions to replace data constructors

One can mechanically derive, from the signature of a data-type, particular functions that can take the place of the data constructors

In [BB85] it is shown that Data-Alternative Constructors for the Church Objects can be mechanically written-out in a single step from only knowledge of the data-type constructors. Take a non-'container type' that has 'm' data constructors, '$c_1..c_m$'. Below is 'f_i', a Data-Alternative Constructor corresponding to data constructor 'c_i', taking the same number, 'n', parameters:

$f_i\ a_1..a_n \triangleq (\lambda c_1..c_m.\ c_i\ (a_1\ c_1..c_m)\ ..\ (a_n\ c_1..c_m))$

For example, the Data-Alternative Constructors for Church Naturals from this single-step algorithm are:

zero ≜ (λsucc, zero. zero)

succ n ≜ (λsucc,zero. succ (n succ zero))

Note that:

succ n ≜ (λsucc,zero. n succ (succ zero))

is not derivable using this technique, nor are Constructors for other than the Church Objects.

If the data-type is a 'container type', then the 'a's representing what is contained by the type do not have '$c_1..c_m$' applied to them; the presentation in [BB85] covers such cases. Hence, for Church Lists, one ends up with:

nil ≜ (λcons,nil. nil)

cons e r ≜ (λcons,nil. cons e (r cons nil))

5.3.6 Derivation technique: algorithm for writing-out functions to replace Structural Recursions

One can mechanically derive recursion-free functions which take functional representations of data as input, from functions on data defined as Structural Recursions on their input

It is shown in [BB85] that any function on data defined by Structural Recursion can be mechanically re-written to take a Church Object as input instead. A related version of this was shown in [Chu41], for Church Naturals.

The general form of Structural Recursion involves a number of indexes and hence is a little involved; it is presented fully in [BB85]. Briefly, it involves top-down pattern-matching and recursion on the data, and can be illustrated well by examples. For Naturals, 'F' below is the general pattern, involving arguments '$a_1..a_n$', which are passed unchanged down the recursion, and arbitrary functions 'h_1' and 'h_2':

F $a_1..a_n$ (succ x) ≜ h_1 $a_1..a_n$ (F $a_1..a_n$ x)

F $a_1..a_n$ zero ≜ h_2 $a_1..a_n$

Note that we use here the common pretty-syntax for parameter pattern-matching, rather than '=='.

In comparison, in the related but more-powerful Primitive Recursion, the 'h_1' function also receives 'x' ('r') as a parameter.

The derivation technique specified in [BB85] takes any function defined as a Structural Recursion on data and produces an equivalent function that takes a Church Object Element as

input (the output and other inputs are unchanged), by a one-step mechanical rewrite. As an illustrative example, for the 'F' for Naturals above, one writes-out:

$$G\ a_1..a_n\ x \triangleq x\ (h_1\ a_1..a_n)\ (h_2\ a_1..a_n)$$

Wherever 'F' took a Natural and produced some output (via recursion), 'G' takes the corresponding Church Natural to the same output (without use of recursion).

To illustrate the limitations of this derivation technique, consider:

add a b ≜ if b == zero then a else if b == succ m then succ (add a m)

One can rewrite it as a Structural Recursion:

add a (succ m) ≜ succ (add a m)

add a zero ≜ a

and then immediately write-out an implementation of addition that takes a Church Naturals and a primitive Natural, and returns a primitive Natural:

add a b ≜ b succ a

The derivation technique is however quite suitable for operations that take a single input and return an output of different type. For example, for lists (a 'container type') one has that from:

$$F\ a_1..a_n\ (cons\ e\ r) \triangleq h_1\ a_1..a_n\ e\ (F\ a_1..a_n\ r)$$

$$F\ a_1..a_n\ nil \triangleq h_2\ a_1..a_n$$

one can write-out:

$$G\ a_1..a_n\ x \triangleq x\ (h_1\ a_1..a_n)\ (h_2\ a_1..a_n)$$

As a concrete example:

sumList (nil) ≜ 0

sumList (cons e r) ≜ (λe,s. e + s) e (sumList r)

which acts on primitive lists, rewrites to:

sumChurchList m ≜ m (λe,s. e + s) 0

which acts on Church Lists.

Finally, we note that Structural Recursions can be 'degenerate', ie not actually recursive. The following two functions are in the form of a Structural Recursion:

firstelement (maketuple m n) ≙ (λm,n. m) m n

secondelement (maketuple m n) ≙ (λm,n. n) m n

From them can be derived tuple-selectors for Church Tuples:

firstelement t ≙ t (λm,n. m)

secondelement t ≙ t (λm,n. n)

5.3.7 Derivation technique: programs from proofs

TFP-style functions can be derived from proofs

Predating [BB85], it was demonstrated in [Lei83] that many Church Object Elements, Constructors, and Non-Constructors (together with appropriate types) can be mechanically derived from certain natural-deduction proofs. For a full description, the reader is referred to that paper; the method is overly lengthy to reproduce here. One can also consult [Mai91] for the general idea and examples. Roughly, it is noted that certain theorems are 'uniform' with regards to a proof of a particular property. By parameterising these proofs, one can form a 'function' from proofs to proofs. By use of the Curry-Howard isomorphism, actual lambda calculus functions can be produced.

5.3.8 Unification of derivation techniques

Primarily via reference to parameterising constructors, apparently disparate derivation techniques can be unified

Above, we have presented a number of derivation techniques. Interestingly, there are overlaps.

We have three distinct methods for producing Constructors for the Church Objects. The first is via solving homomorphism equations involving an implementation of the intrinsic operation of the Data-Alternative, the second is via a mechanical writing-out from the data-type's signature, and the third is via proofs. We also have three distinct methods for producing Church Object Elements: by simplifying applications of an implementation of the intrinsic operation to data, by parameterising constructors from the data, and from proofs. Finally, we can produce Data-

Alternative Non-Constructors via direct replacement, or via writing-out from Structural Recursive definitions. Church Object Non-Constructors can also be derived from proofs.

These apparently disparate sets of techniques can be unified, via reference to parameterising constructors.

5.3.8.1 Parameterising constructors, and direct replacement

The same result as parameterising data constructors from data can be achieved by direct replacement of the data constructors with certain functions

Recall that application of a Church Object Element to the Church Object's Constructors gives back the same Element. This holds because Data-Alternative Constructors construct Data-Alternatives, and Church Object Elements are (by definition) items of data with their constructors parameterised. For example, one can form the Church Natural for '4' in two ways: one can take 'succ (succ (succ (succ zero)))' and parameterise the data constructors; or one can apply the Church Natural Constructor for 'succ', 4 times, to the Church Natural Constructor for 'zero', and reduce.

Say we parameterise the constructors of an item of data, to form a Church Object Element. As per above, we could then apply that Element to the Church Object Constructors without changing its semantics. On reduction, those Constructors will be bound in where the data constructors used to be. This is the same result as if we had used direct replacement to substitute for the data constructors Church Object Constructors. Hence direct replacement of data constructors with Church Object Constructors is semantically identical to parameterising the data constructors (parameterising is more efficient and leads to simpler terms however).

Actually, it is apparent that it not just data constructors can be parameterised to produce Church Object Elements; Data-Alternative Constructors can be parameterised as well. For example, parameterising 'succ' and 'zero' from

succ (succ zero)

will give a Church Natural, regardless of what 'succ' and 'zero' are: data constructors, Church Natural Constructors, or otherwise. Naïvely, the condition is that the term is *solely* composed of the constructors. However, parameterising constructors can be done to a term that is *convertible*

(‘↔’) or can be *reduced* (‘→’), once free variables are bound, to a term which consists solely of constructors applied to each other. For example, consider:

(id succ (id zero))

The two constructors can be parameterised to produce a Church Natural. Further, whenever the output of some function is produced *only* using some constructors, those constructors can be parameterised, causing the output to be a Church Object. The condition is, informally, that those constructors shouldn’t be used for other purposes and the output while being accumulated is not tested on etc. As an example where this condition isn’t true, consider the following function that takes three primitive Naturals as input:

add3 a b c = iterate a succ (iterate b succ c)

Parameterising ‘succ’ and ‘zero’ will break the function, as the outermost ‘iterate’ requires primitive Naturals to operate. Quite usefully, the condition in question can be formalised in terms of types. It is well-known that if and only if the function operates ‘uniformly’ ie is ‘parametrically polymorphic’ with regards to the constructors (as per [Rey83] or [ACC93], and as discussed in the well-known work [Wad89]), then those constructors can be parameterised to give a semantically equivalent function, with a particular polymorphic type. Hence, to make sure that parameterising constructors will work in a given instance, one can go ahead and parameterise, and then check to make sure the resultant function is of the expected type. However, there can be a choice of which constructors to parameterise; [Chi99] suggests an efficient, type-based, procedure.

5.3.8.2 Parameterising data constructors, and the algorithm for writing-out functions to replace data constructors

Being able to mechanically write out functions to replace data constructors follows directly from the homomorphism equations, and an operation that parameterises constructors

Recall that in [BB85] it was shown that certain Church Object Constructors can be written-out from data-type signatures. We now show that this is a consequence of the relevant homomorphism equations, and the fact that Church Object Elements are data with data constructors parameterised. We use Naturals as our example, without loss of generality.

The homomorphism in question shows that Church Object functions can systematically replace data and operation-implementations thereon, ie it relates Church Object Elements (data which

has had its constructors parameterised) to primitive data, and Church Object Constructors to data constructors. To begin, we introduce an operation that parameterises constructors, called 'parameteriseconstructors'. On primitive Naturals, it can be implemented as:

parameteriseconstructors n ≜ (λs,z. if n == zero then z
else if n == (succ m) then
s (parameteriseconstructors m s z)
)

Using this, the homomorphism equations are:

$zero_{Church\ Natural}$ = parameteriseconstructors $zero_{primitive\ Natural}$

$\forall D_1$:Natural. $succ_{Church\ Natural}$ (parameteriseconstructors D_1) =
parameteriseconstructors ($succ_{primitive\ Natural}$ D_1)

These can be solved as per usual. For the first, simple evaluation gives '$zero_{Church\ Natural}$' being '(λs,z. z)', and for the second via partial-evaluation and abstraction one gets:

$\forall Z_1$:Church Natural. $succ_{Church\ Natural}$ Z_1 = (λs,z. s (Z_1 s z))

Note that the right-hand sides of the initial equations have been able to be simplified via mechanical evaluation so that all occurrences of 'D_1' have 'parameteriseconstructors' applied to them. This is due to the particular implementation of 'parameteriseconstructors' chosen. For any data-type such a suitable implementation can be written-down, following the scheme for Naturals above. Hence, the above steps can be seen as a mechanical derivation of Church Object Constructors from data-types, one that gives exactly the same results as the one-step derivation of Church Object Constructors in [BB85].

We now present a concrete example of a derivation done explicitly by parameterising constructors. Consider 'Sum Types' (tagged unions). Sum Types (see eg [Mit96, p122]) involve the constructs 'inleft' and 'inright', which act as injections and are intuitively conceived of as data constructors, ie:

Sumtype ::= inleft X | inright X

for some 'X'. One can parameterise 'inleft' and 'inright' from the two data constructors of this 'container type', to give:

inleft x ≜ (λinleft,inright. inleft x)

inright x ≜ (λinleft,inright. inright x)

Using this representation, the 'case' construct of Sum Types, which has axiomatic semantics:

case (inleft X) L R = L X

case (inright X) L R = R X

can be implemented trivially. Hence Sum Types can be implemented without primitive data. We have found on occasion these definitions in the literature (eg in [SU98]) although not with variable names reflective of an understanding that data constructors have been parameterised.

5.3.8.3 The dual nature of certain operations

Particular functional representations of data can be derived by use of an operation that parameterises constructors from data

We now have two derivation techniques for deriving Constructors for Church Objects using homomorphisms: one using their intrinsic operation ('IF', 'iterate', 'reduce'), and one using 'parameteriseconstructors'. Similarly, we now have two methods for deriving Church Object Elements: one involves parameterising constructors (equivalently, application of 'parameteriseconstructors'), and the other application of their intrinsic operation. The two derivation techniques in each case are in fact one: the intrinsic operations for Church Objects, despite being apparently disparate, are in fact semantically identical to 'parameteriseconstructors'!

For Church Booleans, the intrinsic operation is 'IF'. On data, it can be implemented as:

IF x m n = if x == true then m else if x == false then n

Compare this to 'parameteriseconstructors' on primitive Booleans:

parameteriseconstructors x t f = if x == true then t else if x == false then f

Similarly, for Church Naturals:

```
iterate n = (λf,x. if n == zero then x else if n == succ m then f (iterate m f x))
parameteriseconstructors n =  (λs,z.
                                    if n == zero then z
                                    else if n == succ x then
                                                s (parameteriseconstructors x s z)
                              )
```

Also, Church Lists:

```
reduce m = (λo,b. if m == nil then b else if m == cons e r then o e (reduce r o b))
parameteriseconstructors m = (λc,n.
                                    if m == nil then n
                                    else if m == cons e r then
                                                c e (parameteriseconstructors r c n)
                             )
```

The ability to write syntactically-identical (modulo variable renaming) implementations for each pair of operations shows that they are semantically identical.

As another verification, note that 'parameteriseconstructors' should be the intrinsic operation for Church Objects, and hence should be trivially-implementable on them, ie:

parameteriseconstructors x = x

where 'x' is a Church Object Element. This implies that Church Object Elements 'are' items with their constructors parameterised, which indeed is the case, by the definition of Church Object.

The common name in the literature for 'parameteriseconstructors' is 'fold', generalised from the 'fold' (ie 'reduce') operation on lists. We will use this name rather than 'parameteriseconstructors' for the remainder of this work. Note that 'fold' as usually-defined takes arguments in a different order than here. As earlier discussed, this difference is not of any semantic importance but does permit us in this Chapter somewhat simpler presentations.

5.3.8.4 The algorithm for writing-out functions to replace Structural Recursions

Parameterising constructors is the essence of Structural Recursion, and Primitive Recursion

Structural Recursions can be expressed as parameterisation of constructors, ie using 'fold'. We illustrate this on Naturals. As can be proved by induction:

$F\ a_1..a_n\ (succ\ x) \triangleq h_1\ a_1..a_n\ (F\ a_1..a_n\ x)$

$F\ a_1..a_n\ zero \triangleq h_2\ a_1..a_n$

gives 'F' the same semantics, for Natural 'x', as:

$$F\ a_1..a_n\ x \triangleq\ fold\ x\ (h_1\ a_1..a_n)\ (h_2\ a_1..a_n)$$

In effect, we have abstracted the recursion out into 'fold'. Now, when 'x' is a Church Object 'fold' can be implemented as the identity function, hence by the technique of direct replacement this becomes:

$$F\ a_1..a_n\ x \triangleq\ id\ x\ \ (h_1\ a_1..a_n)\ (h_2\ a_1..a_n)$$

which reduces to:

$$F\ a_1..a_n\ x \triangleq\ x\ \ (h_1\ a_1..a_n)\ (h_2\ a_1..a_n)$$

This is the same result as the single-step write-out from the original presented in [BB85].

An equivalent formulation of the equivalence of Structural Recursion and 'fold' is known as the 'universal property' ([Hut99]) of 'fold'. It tells us that all Structural Recursion can be defined as a call to 'fold' (ie is 'fold'-expressible), and vice-versa. For Naturals, it states that there exists (unique) 'f' and 'x' such that:

$$g\ zero\ = x$$

$$g\ (succ\ n) = f\ (g\ n)$$

$$\Leftrightarrow$$

$$\forall m.\ g\ m = fold\ m\ f\ x$$

Finally, it is also important to note that, as reported back in [Chu41] for Primitive Recursion over the positive Naturals, any Primitive Recursion can be implemented using tuples, tuple-selectors (ie 'projections'), and Structural Recursion. Hence 'fold' also can be used to encapsulate the recursion in Primitive Recursion. Recall that the difference between Structural and Primitive Recursion is that in Primitive Recursion an additional parameter is passed-down the recursion. This additional parameter can be combined with the return-results and a projection function used to return the overall value. In other words, if we define (we work with Naturals here):

$$F\ a_1..a_n\ x \triangleq (\lambda\langle r,y\rangle.\ r)\ (J\ a_1..a_n\ x)$$

where:

$$J\ a_1..a_n\ (succ\ x) \triangleq j_1\ a_1..a_n\ (J\ a_1..a_n\ x)$$

$$J\ a_1..a_n\ zero \triangleq j_2\ a_1..a_n$$

and:

$j_1\ a_1..a_n <r,x> \triangleq <h_1\ a_1..a_n\ r\ x,\ succ\ x>$

$j_2\ a_1..a_n \triangleq <h_2\ a_1..a_n,\ zero>$

then 'J' is a Structural Recursion (by definition), and 'F' implements a Primitive Recursion, ie satisfies:

$F\ a_1..a_n\ (succ\ x) = h_1\ a_1..a_n\ (F\ a_1..a_n\ x)\ x$

$F\ a_1..a_n\ zero = h_2\ a_1..a_n$

as can be readily verified via proof.

Hence, given our work with Structural Recursion above, one has the corresponding implementation of Primitive Recursion, on Church Naturals:

$F\ a_1..a_n\ x \triangleq (\lambda<r,y>.\ r)\ (x\ (j_1\ a_1..a_n)\ (j_2\ a_1..a_n))$

For an example of the above, one has the well-known implementation of 'predecessor' on Church Naturals as:

pred n = (λ<a,b>. a) (n (λ<a,b>. <b, succ b>) <zero, zero>)

An equivalent, based on a slightly different way of doing Primitive Recursion with Structural Recursion, is:

pred n = (λ<a,b>. b) (n (λ<a,b>. <succ a, a>) <zero, zero>)

Note that the tuples here can be represented by functions, as done in eg [Sto77] and (nearly) [Kle35a], and as discussed earlier.

5.3.8.5 Relationship between certain functional alternatives to data

Certain functional alternatives to data can be shown to be related to each other in terms of an operation that parameterises constructors from data

In the above sections, we have primarily been dealing with Church Objects. Interestingly, Data-Alternatives are not much more work. Firstly, note that for an arbitrary Data-Alternative, one wants applications of its Constructors to construct an Element, and Elements can also be produced by application of the intrinsic operation to data. Now, one way of producing a term consisting of applications of the Data-Alternative Constructors is to take an item of data and replace the data constructors with the Data-Alternative Constructors. This can be expressed

using ‘fold’. Therefore, for any given Data-Alternative (including as a special case, Church Objects) corresponding to data-type ‘DT’ and with Constructors ‘$cons_1$.. $cons_n$’:

$\forall D{:}DT.\ fold\ D\ cons_1 .. cons_n = M_{d \to a}\ D$

where ‘$M_{d \to a}$’ an implementation of the intrinsic operation. There are a few interesting things which can be said regarding this.

Firstly, combing this equation with the ‘universal property’ implies the desired homomorphism between the Data-Alternative and its corresponding data. For example, the ‘universal property’ on Naturals is:

$g\ zero = x$

$g\ (succ\ n) = f\ (g\ n)$

$\Leftrightarrow$

$\forall m.\ g\ m = fold\ m\ f\ x$

Putting in the Constructors for ‘f’ and ‘x’ gives:

$g\ zero = cons_2$

$g\ (succ\ n) = cons_1\ (g\ n)$

$\Leftrightarrow$

$\forall m.\ g\ m = fold\ m\ cons_1\ cons_2$

Now, when ‘g’ is ‘$M_{d \to a}$’ the last line matches what we have above, and we are left with:

$M_{d \to a}\ zero = cons_2$

$M_{d \to a}\ (succ\ n) = cons_1\ (M_{d \to a}\ n)$

which are the relevant homomorphism equations for the Data-Alternative. Hence all derivation techniques based on the homomorphism equations can be seen to be instances of ‘fold’-based reasoning.

Secondly, one can recognise that ‘fold D’ is simply the Church Object Element corresponding to the data; hence to turn a Church Object Element for an item of data into a given Data-Alternative Element for the same item of data, all one has to do is apply it to the Constructors for that Data-Alternative. This makes Church Object distinguished, for whatever representation one wants to use, that representation can be produced from the Church Object via application of it to the relevant Data-Alternative Constructors or data constructors. This later point is of interest when considering the derivation technique presented earlier which is based on finding inverses to the intrinsic operation. For Church Objects, such an inverse always exists: it is simply the function

that applies its input to the data constructors, to regenerate back the data those constructors were parameterised from.

Thirdly, for a given Data-Alternative, the equation tells us that if we express the implementation of intrinsic operation in terms of 'fold' (equivalently, a Structural Recursion, or the equations from the 'universal property'), then we can read out implementation of the Constructors. Examples of working out an implementation in such a form can be found in eg [Hut99].

Fourthly and finally, the equation tells us that whenever a Data-Alternative has implementable Constructors, the intrinsic operation can be 'fold'-expressed. The converse also holds: 'fold'-expressibility implies implementable Constructors, as witness-Constructors appear within the 'fold'-expression. Hence, a Data-Alternative has implementable Constructors if and only if its intrinsic operation can be 'fold'-expressed. All operations on 'flat' data-types eg Booleans are 'fold'-expressible, hence Constructors of corresponding Data-Alternatives are always implementable. For recursive data-types, the situation is more interesting. As shown in [GHA01] (refined in [WC04]), whether an operation is 'fold'-expressible or not can be determined from its graph (ie mapping). Consider, as stated earlier, that a function 'f' on lists is 'fold'-expressible if and only if it can be expressed in the form:

$$f\ nil = b$$

$$f\ (cons\ e\ r) = op\ (f\ r)$$

Roughly, the application of 'f' to 'r' can't lose any information from 'r' which is needed by 'op'. If 'f' gives the same output for two 'r', but 'f (cons e r)' returns different output for the two 'r', then the equations can't hold and 'f' isn't 'fold'-expressible. Based on this, simple tests were constructed in [GHA01]. For lists, a function 'f' is 'fold'-expressible if and only if, for all 'r', 's' and 'e':

$$f\ r = f\ s \Rightarrow f\ (cons\ e\ r) = f\ (cons\ e\ s)$$

in other words, if 'f' identifies two lists, then it identifies the result of 'cons' on those lists. For Naturals, a function 'f' is 'fold'-expressible precisely when, for any Naturals 'n' and 'm':

$$f\ n = f\ m \Rightarrow f\ (succ\ n) = f\ (succ\ m)$$

Using these sorts of tests, one can determine whether a given choice of intrinsic operation will lead to non-implementable Constructors. One could then switch to a different intrinsic operation, or one could choose a different set of data constructors for the data-type and try again. Equivalently to the last of these, one may be able to find Non-Constructors which let one denote all the Elements.

5.3.8.6 Parameterising constructors, and programs from proofs

One can parameterise constructors from proofs as well as from data

The derivation technique mentioned previously which derives programs from proofs can only derive Elements, Constructors, and Non-Constructors for Church Objects. Essentially, it can be seen that the derivation technique is no different from the one of parameterising constructors. If some proofs operate 'uniformly' with regard to included assumptions (the 'constructors' in this case) then on parameterising those assumptions (and applying the Curry-Howard Isomorphism) one forms a function, a function that can be equally read as constructing data or Data-Alternatives as constructing proofs.

Setting up a proof and then producing a program is a rather round-about way of deriving Data-Alternative Elements, Constructors and Non-Constructors. This derivation technique doesn't seem to be able to derive anything parameterisation of data constructors can't. Indeed, as the proof-based technique involves parameterising *all* the assumptions without condition, it can be seen to be relatively inexpressive.

5.3.8.7 Conclusion

Reasoning based on parameterising constructors from data simplifies the derivation techniques

We have shown how recognition of parameterisation of constructors enables collapse of the various derivation techniques. As a result, the set of derivation techniques can be simplified to the following.

To derive Data-Alternative Constructors one needs a 'fold'-expressed implementation of the intrinsic operation, they then can be read off. For Church Objects, the Constructors always exist and can be mechanically produced, based on reduction of applications of 'fold', appropriately-implemented. For Data-Alternative Non-Constructor functions, direct replacement can be used; also the equations of homomorphism (or, the 'universal property') may find use. Finally, for Data-Alternative Elements, one can use application of the intrinsic operation; or apply the

Constructors to the appropriate Church Object Element. The intrinsic operation for Church Objects is ‘fold’.

One should also add to these abstraction of ‘fold’ (Structural Recursion), to encapsulate recursion from data-centric implementations before they are turned into Data-Alternative - centric implementations. Without this step, the recursion will persist resulting in a non- TFP-style term. For example, take the following implementation of addition on primitive Naturals:

add a b = if (iszero a) then b else (add (pred a) (succ b))

We can use the technique of direct replacement, to produce a recursive implementation of addition that operates on Church Naturals. However if we take the above and rewrite it as the equivalent:

add a b = if (iszero a) then b else succ (add (pred a) b)

then we can recognise Structural Recursion, and abstract ‘fold’:

add a b = fold a succ b

and then use direct replacement to give a TFP-style function.

It should be noted that some additional derivation techniques are still required, specifically:

pow a b = b a

mul u v = (λf. u (v f))

cannot be derived satisfactorily using only the above techniques.

5.4 Deriving TFP-style functions from other TFP-style functions

After deriving functions to replace data and data-centric constructs, one can then derive equivalent but different functions by application of some additional derivation techniques

Below, we discuss some particular means by which functions derived by the above derivation techniques can transformed. In other words, given some Data-Alternative functions, we can derive new Data-Alternative functions. This lets us in particular derive certain Data-Alternative Non-Constructors, ie ‘pow a b = b a’ and ‘mul u v = (λf. u (v f))’, from common data-centric implementations of the same-named operations.

5.4.1 Removing certain redundant functions

One can often simplify TFP-style functions by removing certain functions which typing indicates to be redundant

A derivation technique for deriving some Church Object Non-Constructors from each other can be found in [BB85]. As proved (and earlier discussed), applying a Church Object Element to its Constructors gives back the same Church Object Element. For example:

(λsucc,zero. succ (succ zero)) (λn,s,z. s (n s z)) (λs,z. z)

gives

(λsucc,zero. succ (succ zero))

Also, as is also shown therein, the Church Object Elements are the sole inhabitants of their corresponding polymorphic type. Hence, a function (perhaps with free variables) has the same type as a Church Object Element, then (when those free variables are bound) it *is* an Element, modulo '↔'. For example, Church Naturals and terms convertible to such are the only inhabitants of the type '∀a. (a —> a) —> a —> a'. A function of this type takes an argument of type 'a —> a' and an argument of type 'a' and returns a term of type 'a'. One can think of 'a' as the type of some Data-Alternative (or, data), whose Constructors (resp. data constructors) are those two arguments: the first corresponds to 'succ', hence the 'a —> a', and the second corresponds to 'zero', hence the 'a'. As the function operates for all 'a', all it can do with its arguments is apply them to each other. These polymorphic types are a direct consequence of Church Object Elements being terms with constructors parameterised, and can be written out from the corresponding data-types.

Combining the two above results gives us that if a term has the type of some Church Object, and the term appears applied to the full list of Constructors of that Church Object, in the correct order, then those Constructors can be removed. Note that commonly such applications are formed by explicitly abstracting the Constructors from some term.

An example given in [BB85] is the function to append two Church Lists:

cat = (λu,v. (λc,n. u c (v c n)) cons nil)

where 'cons' and 'nil' are the Church List Constructors, and 'u' and 'v' are Church Lists. This can be rewritten to:

cat = (λu,v. (λc,n. u c (v c n)))

as '(λc,n. u c (v c n))' has the type of a Church List.

The derivation technique can be seen to be a specialisation of a general derivation method discussed earlier. Recall that parameterising constructors from a term results in the term returning a Church Object, on the condition that the term uses those constructors only (and with nothing else) to build its output. Here, one considers only terms that are applications of a function to Constructors, which lets the type of the function be used to determine if the above condition holds. To parameterise one need not then bind those Constructors back into the function, instead one can parameterise them as operands, with an eta-contraction then giving the same overall result. For example, given:

F succ zero

where 'F' is appropriately typed, parameterising the constructors gives:

(λs,z. F s z)

which then simplifies (eta) to:

F

The end result is the same as if the constructors were simply removed. Essentially, parameterisation is equivalent to removal, when operands are the target.

Note also that as the constructors in question here are Church Object Constructors, this derivation technique can be seen to be an *optimisation* technique: we know (given the type of the function) that on execution those Constructors will be bound to variables, and then reductions will give (equivalent) variables back again. Hence the Constructors can be removed, even when the application can't be evaluated statically to the same result eg because the operand is a, or contains, unbound variables.

We can combine this derivation technique with the one of direct replacement of data-oriented functions with Data-Alternative -oriented functions. For example, using direct replacement, we are able to start from an implementation of multiplication on primitive Naturals:

mul u v = iterate u (add v) zero

Replacing with Church Natural versions of 'zero', 'add', and 'iterate' gives the Non-Constructor:

mul u v = (λm.m) u (add v) zero

Reduction then gives:

mul u v = u (add v) zero

From this, we produce an equivalent but different Non-Constructor as follows. First, by the definition of 'add':

mul u v = u (λx. v succ x) zero

eta-contraction gives:

mul u v = u (v succ) zero

We then abstract the constructors, so we can check the type of the function formed and potentially remove the Constructors:

mul u v = (λs,z. u (v s) z) succ zero

Eta-contraction gives:

mul u v = (λs. u (v s)) succ zero

Finally, recognising that '(λs. u (v s))' is typed as a Church Natural, we remove the Constructors:

mul u v = (λs. u (v s))

For some more examples of what can be derived, let 'A ≫ B' mean that this combination of derivation techniques can derive 'B' from 'A'. Then:

add a b = iterate a succ (iterate b succ zero))

≫ add a b f x = a f (b f x)

and a b = IF a (IF b true false) false

≫ and a b = (λx,y. a (b x y) y)

not b = IF b false true

≫ not b = (λt,f. b f t)

iszero n = iterate n (λx. false) true

≫ iszero n = (λt,f. n (λx. f) t)

However, 'pow a b = b a' cannot be derived in this fashion, from a reasonable implementation of exponentiation on Naturals.

5.4.2 Fusion

The 'fusion law' is useful for proving that transformations are semantics-preserving, and for deriving certain functions

There is a large number of theorems pertaining to 'fold' ([MJ95], [Joh05], [JG07] etc), especially using Category Theory in which 'folds' are 'catamorphisms' in appropriate Categories (see eg [MFP91]). Many of these theorems are directly deducable from the types involved ([Wad89]), and certain of them have even been brought across into the logical programming paradigm (eg [SS00]); indicating that their utility isn't limited to functional programming.

Some theorems are motivated by the desire to show program-equality and the like. Abstracting out the pattern of recursion used, for example writing Structural Recursion using 'fold', demonstrably makes proving properties of programs easier. See eg [MFP91], or for a more gentle presentation, [Hut99]. Other theorems concern themselves with optimisation, especially 'deforestation': the removal of intermediate data-structures from programs. For an introduction to this one can consult [Wad90], [LS95], or [TM95].

There is one well-known theorem we find particularly useful, it is the so-called 'fusion property' (or 'law') of 'fold'. Illustrating by example on lists, as described (and proved) in [Hut99], for any list 'm':

$h\ b_1 = b_2$

$\forall e,r.\ h\ (op_1\ e\ r) = op_2\ e\ (h\ r)$

$\Rightarrow$

$h\ (fold\ m\ op_1\ b_1) = fold\ m\ op_2\ b_2$

Hence, if one wants to prove that '$h\ (fold\ m\ op_1\ b_1) = fold\ m\ op_2\ b_2$', proving '$h\ b_1 = b_2$' and '$\forall e,r.\ h\ (op_1\ e\ r) = op_2\ e\ (h\ r)$' suffices.

Similar instances of this law exist correspondingly for other data-types, as 'free theorems' as per [Wad89] and [Mai91]. The 'fusion law' can be seen to be a generalisation in some sense of the 'universal property' and can be used often instead of it, but at the cost of the 'iff' becoming an

implication. This can be shown by setting 'op_1' and 'b_1' to the corresponding data constructors, and simplifying.

This theorem shows that (where the left-hand side of the implication holds) the semantics of 'h' can be incorporated into the arguments of 'fold', ie how 'h' can be combined ('fused' is the word often used in the literature) with 'op_1' to give 'op_2', and 'h' combined with 'b_1' gives 'b_2'. This is of particular relevance to us as follows. First recognise that 'fold m' is in this case a Church List; and write 'C' for it:

$$h\ b_1 = b_2$$
$$\forall e,r.\ h\ (op_1\ e\ r) = op_2\ e\ (h\ r)$$
$$\Rightarrow$$
$$h\ (C\ op_1\ b_1) = C\ op_2\ b_2$$

In other words, using 'op_1' and 'b_1' to construct an item (of data, or a Data-Alternative Element) followed by application of 'h', can instead sometimes be replaced by the same item constructed using 'op_2' and 'b_2'. For example, rather than constructing a data-structure using 'singleton', 'union' and 'empty', then applying 'membershiptest', one can instead construct a Characteristic Predicate using constructs of the same name but different semantics. Assuming 'op_1', 'b_1', 'op_2', 'b_2' are Data-Alternative Constructors, the above equation can be seen relate two Data-Alternatives. Writing 'DA_1' for the Data-Alternative with Constructors 'op_1' and 'b_1', similarly 'DA_2' for the Data-Alternative with Constructors 'op_2' and 'b_2', we get:

$$h\ b_1 = b_2$$
$$\forall e,r.\ h\ (op_1\ e\ r) = op_2\ e\ (h\ r)$$
$$\Rightarrow$$
$$h\ DA_1 = DA_2$$

Hence, the 'fusion law' relates two Data-Alternatives and their Constructors, via a function 'h' that turns one Data-Alternative into another. The equations can be seen to be those of a homomorphism between the two Data-Alternatives, with regards to their Constructors.

We note that the 'fusion law' can be used to derive implementations of Data-Alternative Constructors, using the methods given previously for solving homomorphism equations. The derivation is from implementations of the Constructors of some other Data-Alternative, whose intrinsic operation is related to that of the first Data-Alternative via 'h'.

It should be noted that we find we do not need to use the 'fusion law' to derive the functions given in Chapter 2.

5.4.3 Binding variables earlier and applying to arguments earlier

Two useful theorems let programs be transformed so that things we know will happen at runtime can be made to happen statically

None of the above derivation techniques are adequate for a satisfactory derivation of 'pow a b = b a'. One could start with (found by reverse-engineering):

pow a b = iterate b (λz. iterate a z) succ zero

and derive, via direct replacement then removal of Constructors:

pow a b = b a

However, that first definition is highly uncommon. We would prefer if it was possible to derive 'pow a b = b a' from a common definition of exponentiation on Naturals, without any highly-non-mechanical transformations. This is our motivation for the 'BVE' theorem below, which lets us carry out just such a derivation. Subsequently, we present the related 'AAE' theorem.

We hesitate to claim novelty for the 'BVE' and 'AAE' theorems, as they are, as will be shown, specialisations of the 'fusion law'. It also seems likely that they would have been constructed before. In [Hin05], for example, similar ideas (but not the theorems) can be found.

5.4.3.1 BVE

One can sometimes transform terms so that variables become bound earlier in their execution

As discussed earlier, one can take an application of a Church Object Element (or something of Church Object Element type) to the Constructors of the Church Object (given in the right order), and remove those Constructors without affecting the semantics. Without implying loss of generality, we use Naturals for our examples. Writing '(λsucc,zero. M)' for the Element, the equality can be written as:

(λsucc,zero. M) (λn,s,z. s (n s z)) (λs,z. z) = (λs,z. M[succ\s, zero\z])

Now, note that after reductions in the left-hand side, the first occurrence of ‘s’ in the body of ‘(λn,s,z. s (n s z))’ will appear in ‘M’ whenever ‘succ’ used to appear, similarly for ‘z’ in ‘(λn,s,z)’. This suggests a generalisation, namely that for any ‘A’ and ‘B’:

(λsucc,zero. M) (λn,s,z. A (n s z)) (λs,z. B) = (λs,z. M[succ\A, zero\B])

ie:

C (λn,s,z. A (n s z)) (λs,z. B) = (λs,z. C A B)

for Church Natural ‘C’. Rather than occurrences of ‘succ’ and ‘zero’ in ‘M’ being replaced by ‘s’ and ‘z’, they are now replaced by ‘A’ and ‘B’. For this equality to hold, we require that ‘A’ not have ‘n’ as a free variable, and ‘C’ not have ‘s’ or ‘z’ as a free variable. Corresponding equalities can be constructed for the other Church Objects.

We can prove the above equality using the ‘universal property’ or the ‘fusion law’. We choose here the former. We first put the equality into the appropriate form:

(λx. (λs,z. fold x A B)) m = fold m (λn,s,z. A (n s z)) (λs,z. B)

The two proof obligations are:

(λx. (λs,z. fold x A B)) zero = (λs,z. B)

and:

(λx. (λs,z. fold x A B)) (succ n) = (λn,s,z. A (n s z)) ((λx. (λs,z. fold x A B)) n)

The first reduces to:

(λs,z. fold zero A B) = (λs,z. B)

which is true by the definition of ‘fold’.

The second reduces to:

(λs,z. fold (succ n) A B) = (λn,s,z. A (n s z)) (λs,z. fold n A B)

then to:

(λs,z. fold (succ n) A B) = (λs,z. A ((λs,z. fold n A B) s z))

ie:

(λs,z. fold (succ n) A B) = (λs,z. A (fold n A B))

which is true, by the definition of ‘fold’.

We can explain conceptually why ‘C (λn,s,z. A (n s z)) (λs,z. B) = (λs,z. C A B)’ holds as follows. During execution of the left-hand side, every reduction which binds an ‘s’ to a value will bind it to the same value; similarly for ‘z’. Hence the ‘s’ (and ‘z’) in every occurrence of

'A' and 'B' in the result will have the same value. Rather than binding the 's' and 'z' in 'A' and 'B' during execution of 'C', they could instead be bound earlier, before the execution. This is what is done in the right-hand side, with 'A' and 'B' replacing 'succ' and 'zero' in 'C'.

Hence, we call this (class of) theorem 'Bind Variables Earlier', or 'BVE'. Instances of it exist for all Church Objects. For example, for Church Booleans, the theorem is simply ('C' is now a Church Boolean or something equivalently typed):

C (λs. A) (λs. B) = (λs. C A B)

assuming 'C' doesn't have 's' free. For practical reasons, we also include as instances of 'BVE' variants that involve more or fewer arguments. For example, for Church Natural 'C', it can be readily verified that:

C (λn,s. A (n s)) (λs. B) = (λs. C A B)

C (λn,s,z,w. A (n s z w)) (λs,z,w. B) = (λs,z,w. C A B)

etc

given side-conditions on free variables.

Note that 'BVE' makes terms simpler. Also, in the left-hand side the arguments to 'C', a Church Object, can be seen to be Data-Alternative Constructors. Hence 'BVE' could be used to optimise program-fragments containing appropriate Data-Alternatives.

5.4.3.2 Applications of BVE

Being able to bind variables to their values earlier makes possible relatively simple derivations of certain TFP-style functions

As a derivation technique, 'BVE' is quite useful. For example, while one can derive:

add a b = (λf,x. a f (b f x))

from:

add a b = iterate a succ (iterate b succ zero)

by the techniques of direct replacement and Constructor-removal, that precursor is somewhat unnatural. Using 'BVE' however, we can proceed from:

add a b = iterate a succ b

Applying the derivation technique of direct replacement to the above gives:

add a b = (λx.x) a (λn,f,x. f (n f x)) b

ie:

add a b = a (λn,f,x. f (n f x)) b

If we eta-expand ‘b’:

add a b = a (λn,f,x. f (n f x)) (λf,x. b f x)

then the left-hand side is now in appropriate form for ‘BVE’, which gives:

add a b = (λf,x. a f (b f x))

as desired. Note that if we could not have successfully used ‘(λn,f,x. n f (f x))’ instead of ‘(λn,f,x. f (n f x))’.

As a second example, recall that by using direct replacement and Constructor-removal, we derived:

mul a b = (λs. a (b s))

from :

mul a b = iterate a (add b) zero

This involved implementing ‘add’ as ‘add a b = a succ b’; note that ‘add a b = (λf,x. a f (b f x))’ wouldn’t have worked as it doesn’t contain any Constructors to abstract or parameterise. However, ‘BVE’ is a generalisation of Constructor-removal, and is able to progress when this later ‘add’ is used. We begin with the canonical (modulo symmetry) definition of multiplication on primitive Naturals:

mul a b = iterate a (add b) zero

After direct replacement we get:

mul a b = (λx.x) a (λx. b s (x s z)) (λs,z. z)

ie:

mul a b = a (λx. (b s) (x s z)) (λs,z. z)

Applying ‘BVE’ to the above gives:

mul a b = (λs,z. a (b s) z)

Finally, an eta-contraction yields:

mul a b = (λs. a (b s))

For our last example, we show how using ‘BVE’ we can now finally derive ‘pow a b = b a’ from a reasonable data-centric implementation. We start with the following definition of exponentiation on primitive Naturals:

pow a b = iterate b (mul a) one

We then use the technique of direct replacement, with very particular implementations, giving:

pow a b = (λx. x) b ((λa,b,s. a (b s)) a) (λs,z. s z)

This reduces to:

pow a b = b (λr,s. a (r s)) (λs,z. s z)

The right-hand side is now in the right form for application of 'BVE', the result of which is a new implementation of 'pow' on Church Naturals:

pow a b = (λs. b a (λz. s z))

This can be further simplified by eta-contraction to:

pow a b = (λs. b a s)

and after a further contraction:

pow a b = b a

as desired. One may note that we have not come across a derivation of 'pow a b = b a' before; instead that term is usually given *ex nihilo* and proved correct.

5.4.3.3 AAE

Terms can sometimes be transformed so that applications happen earlier in their execution

There are a number of theorems related to 'BVE'. For example, the two most-common fixpoint functions, used to implement recursion:

(λf. (λy. f (y y)) (λy. f (y y)))

and:

(λy,f. f (y y f)) (λy,f. f (y y f))

are clearly related by a 'BVE'-style theorem. As an aside, note that these can be approximated by:

(λf. C f X)

for arbitrary 'X' and Church Natural 'C' representing a value larger than the depth of recursion that will be reached. We suggest using an exceedingly-large Church Natural, perhaps 'pow (pow 99 99) (pow 99 99)'. Using such instead of the fixpoint functions, a program that would be TFP-style if it wasn't for recursion can be turned into an equivalent (for all practical purposes) program that is in the TFP style.

One particularly-useful (class of, one for each Church Object) theorem related to 'BVE' that we have developed we call 'Apply to Arguments Earlier', or 'AAE'. Instead of binding variables

earlier, in ‘AAE’ applications to arguments are made to occur earlier; indeed the applications are formed statically instead of dynamically.

The derivation of the theorem for a given Church Object begins with the ‘BVE’ equation for that Church Object. Illustrating using Church Booleans, ‘BVE’ is:

C (λs. A) (λs. B) = (λs. C A B)

If we now apply some argument ‘Z’ to both sides, we get:

C (λs. A) (λs. B) Z = (λs. C A B) Z

ie (as ‘s’ is not free in ‘C’, a side-condition of ‘BVE’):

C (λs. A) (λs. B) Z = C A[s\Z] B[s\Z]

or, equivalently:

C (λs. A) (λs. B) Z = C ((λs. A) Z) ((λs. B) Z)

Now, introduce ‘S_1’ to stand for ‘(λs. A)’, and ‘S_2’ for ‘(λs. B)’. The result is the ‘AAE’ equation:

C S_1 S_2 Z = C (S_1 Z) (S_2 Z)

Applications of ‘S_1’ and ‘S_2’ to ‘Z’, which we know will happen at runtime on execution of ‘C’, are done earlier, statically, in the right-hand side.

For recursive types, the derivation yields a slightly different form of equation. Switching to Church Naturals, one starts with (unary) ‘BVE’, with both sides applied to ‘Z’:

C (λn,s. A (n s)) (λs. B) Z = (λs. C A B) Z

This equals:

C (λn,s. A (n s)) (λs. B) Z = C ((λs. A) Z) ((λs. B) Z)

We can rewrite ‘A’ to ‘(λs. A) s’ in the left-hand side, giving us:

C (λn,s. ((λs. A) s) (n s)) (λs. B) Z = C ((λs. A) Z) ((λs. B) Z)

Then, by calling ‘(λs. A)’ ‘S_1’ and ‘(λs. B)’ ‘S_2’ we get the ‘AAE’ equation for Church Naturals:

C (λn,s. S_1 s (n s)) S_2 Z = C (S_1 Z) (S_2 Z)

Again, applications of ‘S_1’ and ‘S_2’ to ‘Z’ that we know will happen during execution, are present statically in the right-hand side.

One can construct instances of ‘AAE’ for the other Church Objects in a similar fashion. One can also construct instances of ‘AAE’ for when multiple operands are to be applied to earlier.

5.4.3.4 Applications of AAE

Some functional alternatives to data can be derived from others, by making applications happen earlier

The 'AAE' (class of) theorem can be used to simplify programs. They can also be seen to describe a way of producing a new Data-Alternative based on an old Data-Alternative (of the correct form); where the new Elements are to be equal to the old Elements applied to some 'Z'. Hence, 'AAE' appears complementary to the 'fusion law', which involves applying *to*, not applications *of*, Data-Alternatives. However, 'AAE' is in fact a *specialisation* of the 'fusion law'. On Naturals, the 'fusion law' is:

$h\ b_1 = b_2$

$\forall r.\ h\ (op_1\ r) = op_2\ (h\ r)$

$\Rightarrow$

$h\ (fold\ m\ op_1\ b_1) = fold\ m\ op_2\ b_2$

Now, set:

$h = (\lambda x.\ Z)$

$op_1 = (\lambda n,s.\ S_1\ s\ (n\ s))$

$b_1 = S_2$

$op_2 = S_1\ Z$

$b_2 = S_2\ Z$

The implicand is now 'AAE', and the proof-obligations hold.

5.5 Relation to removing interpretation

The techniques in this Chapter can be used to lessen the amount of interpretation in programming, assuming interpreters can be identified

We begin by discussing how interpreters can be removed from programs, via 'pre-interpreting' inputs to functions to form Data-Alternatives. We then present what can be seen as a more general argument, namely that the *simplifications* done by the derivation techniques can remove interpreters. Subsequently we focus on what TFP considers to be interpretation: symbol-

oriented programming. We cover what symbol-tests are replaced with in when applying the derivation techniques, how best to avoid the need for symbol-tests, and the curious fact that all tests on symbols can be removed from a program-fragment but tests can remain.

5.5.1 Avoiding interpretation via choice of input-representation

Functions can avoid needing to interpret their arguments if those arguments are pre-interpreted

Consider Data-Alternatives formed by simplifying applications of interpreters (however identified) to their input. Such Data-Alternatives can be considered to be that input 'pre-interpreted'. For example, if all one does with grammars is call 'parse' on them, then one can use Combinator Parsers instead of grammars. The result is a program that doesn't call 'parse'; instead an elidable call to the identity function suffices to produce a parser from a Combinator Parser.

Note that if one wants to use pre-interpreted input, functions in the program will have to be modified to take and/or return pre-interpreted values. Hence if a certain construct processes some values before they reach the interpreter, values one wants to pre-interpret, that construct will require modification. We have given means for how this can be done earlier in the Chapter, including for the very common case when the construct in question is a data constructor.

Below, we consider a more general view of how interpretation can be avoided.

5.5.2 Interpretation-removal via simplification

Evaluation can evaluate-away interpreters; more generally simplification can remove interpreters

Simplifying applications of functions to their input can remove interpretation. For example, consider some function, interpretive in some chosen characterisation of interpretation. The Data-Alternative Elements one derives by simplifying applications of the function will generally be *less* interpretive than the function because parts of the function have been evaluated or otherwise simplified-away. Generally, the more simplification, the less interpretation. One can

often view the situation as the interpreter no longer executing at runtime, but instead being 'executed' by the programmer.

For concrete illustrations, we note that engaging in symbol-tests is considered interpretive in TFP. Computing with primitive data requires symbol-tests, but computing with Data-Alternative functions may not. As we've seen, symbol-tests on data constructors are evaluated-away during the derivation process. Where data constructors once were one now has the *result* of the tests on those constructors. We discuss further how our derivations of TFP-style programs involve removing symbols and tests thereon in the sequel.

To remove as many as possible occurrences of (whatever one considers to be) interpreters via simplification of applications, two things are important.

Firstly, one requires techniques to transform programs, so as to form applications able to be simplified. Using 'AAE', we can transform certain programs so that the rather than the interpreter and its input coming together at runtime, that application is instead statically present and hence able to be simplified. The 'BVE' transformation is similarly useful. Recall 'BVE' for Naturals:

C (λn,s. A (n s)) (λs. B) = (λs. C A B)

If we had in a program:

x (λn,s. A (n s)) (λs. B) Z

where we did not know the value of 'x', no simplifications can take place. But if we knew 'x' would only be bound to a Church Natural, we could use 'BVE' to transform this to:

(λs. x A B) Z

After a beta-reduction, simplifiable applications in 'A' and 'B' may be formed. Another technique is to transform programs by 'pushing' interpreters outwards to their inputs, so as many interpreters as possible appear only in simplifiable applications with their inputs. This is 'pre-interpretation'. For example, consider a program of the form:

x := 4

y := 6

..

..interpreter x..interpreter y..

If we know that the values of ‘x’ and ‘y’ are never modified once assigned, and that ‘x’ and ‘y’ are only ever given as arguments in that final line, then one can rewrite the above program, pushing the interpreters back in time as it were, to:

interpreterx := interpreter 4

interpretery := interpreter 6

..

..interpreterx..interpretery..

As all interpreters now appear only in simplifiable applications, they are able to be evaluated-away. In general, one can only remove in this way interpreters which can be appropriately matched-up with their input. For example, a given item of data may be interpreted by *multiple* interpreters (or, vice-versa), meaning that the relevant applications cannot be formed. Note however that programs can sometimes be rewritten so that only a single interpreter is applied to given data; for example:

add a b ≙ if b == zero then a else succ (add (pred b) a)

mul a b ≙ if b == zero then zero else add n (mul (pred b) a)

can be rewritten as we’ve seen in terms of a common function, ‘iterate’. There are no doubt a large number of other transforms that could be used to form simplifiable applications of interpreters to their input.

Secondly, as well as means of producing simplifiable applications, one also requires powerful means of simplification; more than partial-evaluation ie beta-reductions (‘→’) is required. For a simple illustration of this, given the code:

(λn,x. n (λr,y. 4) (λy. 4) (interpret x))

if we know that ‘n’ will be a Church Natural, then we know that ‘interpret x’ can be removed as it will be only ever bound to ‘y’ and then ignored. Generally, it can be quite difficult when simplifying to work out what code could be removed, and then to remove it while preserving semantics. A body of work ([Bon89], [DMP96], [SGT96], [CDP99] etc) has been developed on precisely on how to simplify as far as possible applications of functions to their input data, especially using types as an aid (see particularly [Hug96]).

Finally, it should also be noted that one can remove interpreters by simplify not just single applications, but also terms throughout a program. Particularly, switching to a good choice of Data-Alternative can lead directly to simplified code, by making certain operations simpler (or

even trivial) to implement, as we've seen. For example, 'fold' can be implemented as the identity function when its argument is a Church Object, and hence the pattern-matching 'fold' would normally do no longer appears: this turns out to have been known previously, it can be found stated in [WW03].

5.5.3 Symbols, tests, and recursions are replaced in TFP-style programs

The derivation techniques can be seen to show how recursion, data. and tests can be replaced by variables and beta-reductions, or sometimes, definitions

In TFP, tests on symbols are considered to be interpretive, especially when accompanied with recursion. It is therefore of interest to consider what replaces symbols and tests during derivation of recursion-, symbol-, and test- free functions, ie TFP-style functions.

Many operations are implemented using recursive tests on data constructors, tests that execute the same code for each instance of a given data constructor. These tests correspond to Structural Recursions (which effectively includes Primitive Recursion, as discussed previously). Such recursive tests can be abstracted out into 'fold', and be implemented by the identity function when the data is exchanged for Church Object Elements. In effect, the recursion has been removed, the data constructors have been replaced by variables, and the tests replaced by beta-reductions, eg:

d ≜ succ zero

..

fold d doifsucc doifzero

turns into:

d ≜ (λsucc,zero. succ zero)

..

d doifsucc doifzero

This is the essence of TFP-style programming.

It should also be noted that definitions can take the place of the beta-reductions. If the variables of some Church Object are consistently bound to the same values, definitions can be introduced.

One first transforms the program if needed using eg ‘AAE’ to make those binding happen in a static application, eg:

(λsucc,zero. M) doifsucc doifzero

and then replaces that application by definitions, renaming or using scoped definitions as required to avoid undue capture; eg:

succ ≜ doifsucc

zero ≜ doifzero

M

Hence, symbol-tests on data constructors, whose branches are replacement semantics for those data constructors, can be replaced by definitions in some cases. Note that all we have really done here is introduce a non-Church Object Data-Alternative, whose Constructors ‘succ’ and ‘zero’ have the semantics of ‘doifsucc’ and ‘doifzero’.

As a final example, consider that in many programs functions return an error-code which the caller then tests and calls the appropriate (possibly ‘no action’) error-handler. Now, these error-codes can be replaced by variables, to be bound via beta-reductions to the appropriate error-handler. First, take the general pattern:

Errorcodes ::= $errorcode_1$ | .. $errorcode_n$

..

F ≜ (λx.M)

..

(λr.

if r == $errorcode_1$ then S_1

..

else if r == $errorcode_n$ then S_n

) (F X)

Recognising ‘fold’, the above can be rewritten as:

Errorcodes ::= $errorcode_1$ | .. $errorcode_n$

..

F ≜ (λx. M)

..

(λr. fold r S_1.. S_n) (F X)

Direct replacement, followed by abstracting Constructors then removing them (or, parameterising them), gives:

$errorcode_1 \triangleq (\lambda errorcode_1,..errorcode_n.\ errorcode_1)$

..

$errorcode_n \triangleq (\lambda errorcode_1,..errorcode_n.\ errorcode_n)$

..

$F \triangleq (\lambda x, errorcode_1,..errorcode_n.\ M)$

..

$F\ X\ S_1..\ S_n$

Finally, as the definitions of '$errorcode_1$' through '$errorcode_n$' are never used, they can be removed. In essence, what we have done here is replace the error-codes by 'continuations' (see eg [Wan80]).

5.5.4 Best choice of functional representation to avoid symbols and tests thereon

Certain functional representations for data are often the best choice to make programs less symbol-oriented

Extending the observation made in [DN01] for Church Objects, Data-Alternatives can be thought of as inverses in some sense of 'defunctionalization' (see eg [BBH97]), a technique where functions are represented by data. However, one might ask, what is the *best* Data-Alternative to use to replace data with in a given program? We suggest the answer is the one whose intrinsic operation most, if not all, symbol-consumers used in the program can be written in terms of (and has required Constructors and Non-Constructors implementable). Usually the best choice for such an intrinsic operation is 'fold', which makes the Data-Alternatives Church Objects. As 'fold' also encapsulates Structural Recursion, a very common form of recursion (as evidenced by the prevalence of 'IF', 'reduce' and 'iterate'), this makes Church Objects a good choice as the resultant programs will be simpler and less recursive as a bonus. Additionally, if any other Data-Alternative turns out to be suitable in certain locations in the program, then nothing is lost by using Church Objects: to turn the Church Object into the Data-Alternative all one has to do is apply it to the relevant Constructors. Consequently, one expects when Church Objects are used that Non-Constructors will always be implementable. A proof of this should be

able to be constructed, based on functions (and their inverses) that injectively map from Church Objects to Church Naturals, an appeal to the Church-Turing Thesis, and the ability to find lambda-terms to implement the Partial Recursive Functions ([Bar84]). Hence if Church Objects are used in programs that then require modification, one will never be forced to switch to a different Data-Alternative in order to make new operations implementable. In contrast, recall that other Data-Alternatives do not always have implementable Non-Constructors. For example, the operation that returns the cardinality of a set is generally impossible to implement on Characteristic Predicates.

However, some Data-Alternatives do permit more symbol-tests to be removed (via simplification of applications) than Church Objects. For example, consider a Data-Alternative for sets, whose intrinsic operation is 'contains8'. Implementations of the Constructors and intrinsic operation are:

empty ≜ false

singleton x ≜ (x == 8)

union a b ≜ or a b

contains8 s ≜ s

In a given program, if 'singleton' only appears in static applications to literal data, then those applications (and hence the '==' symbol-test) can be evaluated away. Hence no '==' test will remain. In contrast, with Characteristic Predicates, one has the Constructors:

empty ≜ (λe. false)

singleton x ≜ (λe. x == e)

union a b ≜ (λe. or (a e) (b e))

and 'contains8' can be implemented as:

contains8 s ≜ s 8

Here, the test '==' cannot be easily removed, it is executed only when 'contains8' is called. Both of these Data-Alternatives remove the symbol-testing of the data-structure representing the set, but only in the first Data-Alternative is the application of 'x ==' to '8' static and hence reducable. In the other Data-Alternative, the application is formed during runtime, and hence not readily removable (some form of 'BVE' or 'AAE' would be required). More research could be conducted on these issues.

5.5.5 Tests surviving simplifications

Evaluating-away all the symbol-tests still seems to leave a test, as intuitively understood

Surprisingly, while we can evaluate-away symbol-tests from functions, the functions will still seem to *do* tests. For example, consider the program-fragment:

b := true

IF b x y

We can convert this to:

true ≙ (λx,y. IF true x y)

false ≙ (λx,y. IF false x y)

b := true

(λm. m x y) b

Completely evaluating the applications of the symbol-tester 'IF', gives:

true ≙ (λx,y. x)

false ≙ (λx,y. y)

b := true

(λm. m x y) b

Now, note that '(λm. m x y)' still intuitively does a test on 'b': if 'b' is the representation for 'true' then it executes 'x'; else if 'b' is the representation for 'false', then it executes 'y'. However, it does not appear to actually *do* a test. Hence, if one considers tests to be interpretive, the function is still apparently interpretive: only it now interprets on Church Booleans rather than primitive Booleans.

Similarly, consider:

(λa. a (λb,c,d. d) (λe,f. e))

and:

(λa,g,h. a (λb. g) h)

These functions do not appear to do tests, yet in fact they implement the 'iszero' test on Church Naturals.

As a result of the apparent presence of interpretation in these examples, the overall utility of the use of Data-Alternatives to avoid interpretation, as opposed to merely symbol-tests, is in doubt at this point. Perhaps all use of Data-Alternatives does is cause interpreters acting on symbolic data to become equivalent interpreters acting on functional representations of data; ie a change from symbolic to asymbolic interpretation? Recalling as well that Church Naturals can replace recursion, is TFP-style programming really less-interpretive programming?

We resolve these apparent paradoxes between our intuitions regarding interpretation and testing, and interpretation and TFP-style programming, in Chapter 7.

5.6 Conclusion

TFP-style functions can often be easily, and something mechanically, derived

In this Chapter, we have presented derivation techniques able to derive the exemplars of TFP-style programming from Chapter 2, and others (not necessarily in the TFP style) besides. We have also discussed the relationships between these techniques, data- and recursion- free programming, and interpretation. Overall, we believe that (true or 'pure', ie completely data-free) TFP-style programming should be identified with 'programming with Church Objects'; while TFP-style programming in the weaker sense (ie that which permits some data, for example as in Characteristic Predicates) should be identified with 'programming with Data-Alternatives'.

A number of the derivation techniques we have presented are relatively well-known in the literature, as are Church Objects generally. However Data-Alternatives as a class appear not to have been fully recognised. Further, published derivation techniques are narrowly focussed on purposes such as 'deforestation', rather than on deriving functional representations in their own right, or on removing symbol tests from programs. We therefore believe that we have given the first general methodology for deriving functional alternatives to data. Note that determination of precise limits of applicability etc remains as future work.

By means of summary, below we give a core set of techniques for each of the three forms of Data-Alternative function.

Firstly, derivation of the Data-Alternative Elements can be done in a number of differently-motivated ways. One can apply an operation-implementation to literal data, and optionally simplify. Alternatively, one can replace data constructors by Data-Alternative Constructors (and again, optionally, simplify) if such are known; or if the Data-Alternative is a Church Object, instead parameterise all data constructors of the data-type.

Secondly, given an implementation of the intrinsic operation in the correct form (ie Structural Recursive), Data-Alternative Constructors can be mechanically derived. For Church Objects, one can mechanically write-out an implementation of the intrinsic operation, 'fold', in just such a form. Other Data-Alternatives use other intrinsic operations, some of which cannot be implemented in the required form (in a functional programming language, at least) and are hence have unimplementable Constructors.

Thirdly, deriving whatever Data-Alternative Non-Constructor functions are desired can be done by replacing some or all data-centric sub-terms with Data-Alternative -centric sub-terms, earlier-derived. As with the Constructors, desired Non-Constructors are not necessarily implementable.

Finally, note that once a Data-Alternative function has been derived by some method, it can often be transformed into a semantically equivalent but different function, by use of 'BVE' or 'AAE'.

Generally, if one is interested in finding all the different ways an operation can be implemented on a given Data-Alternative, one would proceed as follows. First, write out all the ways of implementing the operation on data. Then, convert those implementations to take and return the Data-Alternative, using the derivation techniques of this Chapter (including use of 'BVE' and 'AAE' where possible). As part of this, basic rewrites (such as eta-contraction or eta-expansion) may be required. If one is interested in deriving TFP-style functions only, then one should only consider initial implementations that have all recursion abstracted-out into an operation, the intrinsic operation of the Data-Alternative. One needs to ensure this operation includes all the recursions and tests on symbols done by the implementation, in order to end up with TFP-style functions. However, note that one can instead introduce eg Church Objects (which can be turned into any other desired representation) to take the place of data where appropriate, to give symbol-free results. For example, rather than lists of Naturals one can have Church Lists of Church Naturals.

With regards to interpretation, the derivation techniques can be viewed as simplifications (implying TFP-style programming is *simpler* programming), and the simplifications can involve removing interpretation. Note that some significant simplifications are possible due to the applicative nature of Data-Alternatives, to be contrasted to inert data. The function 'pow a b = b a' is a good example. However, to produce some of these simpler implementations, felicitous choices need to be made regarding implementations and derivation steps. Also, we conjecture that major simplifications (eg 'pow' above) may require there be a hierarchy of operations, each implementable in terms of operations below it.

Finally, we have in this Chapter concerned ourselves with only what are called in the literature 'regular' data-types, which include 'container types'. These include perhaps every data-type one might encounter in practice. They even include data-types for program source-code; the so-called "higher-order abstract syntax" of [PE88] can be seen to be a Data-Alternative for such. On this topic, we note that we have discovered (or rather, rediscovered: [Hut98]) that the denotational semantics mapping, in formal semantics, can also be expressed as a 'fold' on a data-structure representing a program. This is because in denotational semantics each occurrence of a program-construct is given the same semantics under the mapping. Amusingly, this makes the corresponding Data-Alternative for a program's source-code the program itself, semantically-speaking.

However, more exotic forms of data-types do exist, and there is potential to extend our techniques to them. For example, one can conceive of data-types which have rules regarding what constructors can be applied when (as expressed eg by a Context-Free Grammar). An example are the so-called 'nested data-types'. For nested data-types, 'fold' and 'fold'-like operations have been constructed for typed languages in eg [Hin00] and [Bay01] (see also [BP99a], for an interesting application). This indicates that there is potential for application of our derivation techniques to such data-types. Further, certain other unusual data-types, such as those which include functions as components, also have 'fold'-like operations definable, see eg [SF93], [MH95], [FS96], and [WW03]. For graphs, 'fold' and 'fold'-like operations have been considered in [Erw97]; and for cyclic structures, in [TW01]. Finally, one can even implement 'fold' over types themselves, see eg [Wei01].

Chapter 6

Definitional and interpretive styles of language-extension

Theoretical rationales exist for preferring definition over interpretation; these include the typically-limited expressivity of interpreters with regards to language-extension

In the previous Chapter we discussed how, once interpreters are identified, programs can be transformed to often-times *remove* them; while the earlier Chapter on type-systems discussed how those transformed (ie TFP-style) programs can be *typed*. In this Chapter we concern ourselves with the *use* of interpreters as a language-extension technique, and compare them with definitions ('≙').

Interpretation is intuitively characterised as symbol-orientated computation; especially pattern-matching on symbols. One specific application of interpreters is to implement programming languages, and it is such interpreters that are the focus of this Chapter. Interpreters for programming languages are intuitively characterised by a recursive loop, which pattern-matches on constructs of a program (the 'guest') supplied as a data-structure, and simulates each such construct. The alternative to programming language interpretation identified in TFP is definition. It can be seen that rather than using pattern-matchers to convert a symbolic representation of a construct to its intended 'meaning' at runtime, definitions bind constructs to their 'meaning' statically. Based on that essential difference, we have already covered in this work much of relevance, specifically in Chapter 5.

Firstly, it is immediately apparent that 'fold'-expressed functions are the quintessential programming language interpreters, as they do systematic replacement of data constructors (which can represent program constructs) by supplied semantics. Since definitions do systematic static binding of program constructs to supplied semantics, one would expect that it is possible to express a correspondence between the two.

This can indeed be done:

fold M f_1.. f_n

=

$c_1 \triangleq f_1$

..

$c_n \triangleq f_n$

M

where 'n' is a Natural and 'M' stands for a term consisting solely of 'c_1' through 'c_n'. The primarily caveat here is that the definitions can't be recursive or refer to each other. Note that the above is familiar: we are simply dealing with a Data-Alternative, one with intrinsic operation '(λx. fold x f_1.. f_n)' and Constructors called 'c_1' through 'c_n'. To produce an Element of the Data-Alternative, we know we can apply the intrinsic operation, or directly replace (eg by using definitions) the data constructors with the Data-Alternative Constructors: this equivalence is precisely what is expressed above. Chapter 5 hence tells us that one can transform all, and only (recalling when Data-Alternatives have implementable Constructors), interpreters able to be expressed as Structural Recursions into equivalent definitions, and vice-versa. Interestingly for TFP, this means that interpreters can be used in strictly more situations than definitions: definitions do *not* always suffice, and one *does* on occasion need to resort to interpreters. Interpreters are possible whenever the desired mapping from symbols to output is Turing-computable (assuming no limitations of the programming language). In contrast, definition is possible only when the desired mapping is 'fold'-expressible.

In addition, Chapter 5 demonstrates concretely that there is more to programming language interpretation than recursive pattern-matching on data representing program constructs. The Chapter shows how functions can be used as alternatives to data, and how Structural Recursion can be 'pre-packaged' into the structure being traversed. It also shows how programs that contain pattern-matchers and data can be transformed into higher-order functions, free of any primitive data or primitive constructs for testing such.

Certainly, Chapter 5 tells us much about interpretation and definition. However, what is lacking is a more fundamental investigation of the interpretive and definitional approaches, to fill some specific lacunae in the current formulation of TFP. As indicated previously, as part of TFP some general reasons why interpretive-style language-extension is best avoided have been identified,

but little is offered in the way of a detailed explanation. Mention is made of topics such as type checking and comprehension, but no underlying reason is given for the apparent differences. Furthermore, a framework sufficient to properly discuss such issues is lacking. By going back to first principles we develop a suitable framework as our first step, which is general enough that it is not restricted to functional programming. While the exact nature of interpretation is not yet established we are still, via this framework, able to compare language-extension done via the use of what are clearly interpreters to language-extension done via the use of definitions. We show that a hard distinction between the two styles of language-extension cannot be satisfactorily made, as either subjectivity or knowledge of the programming language's implementation is required. Additionally, the apparent differences between the two styles are explicated, and the fundamental reasons why definition is preferred over interpretation (when both are possible) in today's common programming languages are identified. We additionally derive useful information for a later establishment of the nature of interpretation.

6.1 Framework

There is a simple and general framework that can be used to investigate the differences between the definitional and interpretive styles of language-extension and other concerns

To help answer the open questions in TFP pertaining to language-extension, *vis-a-vis* interpretation, we make use of a suitable framework, detailed below. The framework is targeted to match reality at an appropriate level of detail, and is based on first principles to ensure no unwarranted assumptions are made. The key concepts of our framework are 'Programming Language Model', 'Conceptual Model', 'Design', and 'Hosting'.

6.1.1 Programming Language Model

A given programming language can be modelled as a tuple consisting of a model of its syntax, and a model of its semantics

For a given programming language 'X', one can construct a tuple '$<\mathcal{L}_X, [\![\ ]\!]_X>$' (the 'X's being elided when clear from context) which we term a 'Programming Language Model' or 'PLM' for

'X'. As described below, each choice for the particular'$\mathcal{L}_X$' and '$[\![\ ]\!]_X$' will lead to a different PLM.

The first element of the tuple, '$\mathcal{L}_X$', is the set of what one considers to be the syntactically well-formed programs of programming language 'X'. For example the '$\mathcal{L}$', the syntactic 'language' as it is termed in the literature, for some programming language might be declared to be the subset of finite strings, of tokens drawn from a particular alphabet, able to be parsed by a particular Context-Free Grammar. It should be noted that elements of '$\mathcal{L}_X$' and their constituents have no *intrinsic* meaning: it is only convention, experience, and likelihood that leads one to think that "1+1" has 'twoness', rather than it denoting eg the concatenation of the first elements from two lists given elsewhere.

The second element of the tuple, '$[\![\ ]\!]_X$', is a model of the semantics of programming language 'X'. It is an infix total function, known as a 'semantic function' or 'meaning function' in the literature. It maps from its argument, a program drawn from '$\mathcal{L}_X$', to a model of the observable results of applying the programming language to that program. What observable results are considered depends on how one wishes to model the programming language, but '$[\![\ ]\!]$' would typically produce a semantic model of the program such as an input-output mapping or the denoted element in a chosen Domain. In the literature '$[\![\ ]\!]$' generally returns *solely* such a model of the program, but there are many other possibilities that can be returned in addition or instead. For example, information regarding the internal consistency (eg type-correctness) of the program, or a machine-code or assembly-code equivalent of the program, might also be returned. Real-life programming language implementations today can return a wealth of outputs, and us leaving unconstrained the output of '$[\![\ ]\!]$' permits them to be modelled in as much detail, or as abstractly, as one may desire.

6.1.2 Conceptual Model

Programs and languages can be modelled conceptually

A model is intended (the model may be flawed or incorrect) to be an abstraction of a particular system. This view is in alignment with current standards (eg [OMG03]). Commonly, models are textual in nature. However, because our work in this Chapter is intimately concerned with distinctions between programs and what programs denote, we find textual models unsatisfactory. Being textual, the models are meaningless in and of themselves: one does not know the associated semantics. A mapping to semantics could of course also be communicated, but that would require additional information to be supplied to determine its own meaning. To break out of this circularity commonly one simply *assumes*, implicitly, that at some stage the meaning can be determined. However to do things formally and objectively one must instead introduce a function to meaning. Having to pair a textual model with such a function is though somewhat clunky.

Fortunately there is one class of models for which mappings to meaning are not required, because all 'meaning' is built-in. These are what we term here 'Conceptual Models'. Conceptualisations (comprehensions) of systems are certainly models. In some sense, conceptualisations are the only true meanings and Conceptual Models are the only true models: other models can be considered to merely be intended to evoke some particular Conceptual Model on reading.

Conceptual Models may seem somewhat exotic, but they are recognised in the literature: see eg [NM07] where they appear under the names 'views' and 'mental representations'. Their importance to mathematics generally is argued in [LN00].

The primary use we put Conceptual Models to in this Chapter is to invoke them when discussing the meaning of programs and parts thereof. Making explicit reference to Conceptual Models in these cases avoids the alternative of having implicit assumptions regarding such meanings. In most works such assumptions are harmless, ie that some program *could* execute as assumed is sufficient. However in this Chapter we are particularly concerned with what is *objectively true*, and the appearance of a Conceptual Model immediately tells us that the meaning in question is *subjective*.

Generally, a Conceptual Model can be considered a valid model of a system if it agrees with it on selected properties. When discussing programs, one would require agreement on at least overall computational semantics (ie consistency with the nominal input-output mapping of the program).

6.1.2.1 Common nature of Conceptual Models

Conceptual Models typically involve sets of discrete interacting entities which carry out actions over time

There are some apparent commonalities across Conceptual Models. Firstly, systems always seem to be Conceptually Modelled as discrete entities. Even continuous surfaces, for example, are typically viewed as either a single entity or as an infinite number of point or surface-element entities. Secondly, in Conceptual Models each entity typically has associated with it a set of properties, information about the entities. Aggregate and hierarchical properties are also typically present, for example we may conceptualise a sorting algorithm in terms of individual instances of operations ('partition', 'append', 'isgreaterthan' etc), yet also be aware that the whole aggregate will result in sorted output. Thirdly and finally, Conceptual Models of programs typically include a temporal component, ie contain entities whose existence or properties are conceptualised to change over time, typically corresponding to individual execution-steps. Hence, a program with its input can be conceptualised as a Conceptual Model which includes how an idealised execution progresses. For example, a Conceptual Model of "(λx. x) 4" may contain two entities, representing "(λx. x)" and "4", as well as knowledge that the "x" will be replaced by "4", which then becomes the end result of the execution.

6.1.2.2 'Ambient' semantics in Conceptual Models

Not all computational semantics may come from the program

There may be in Conceptual Models what we term 'ambient' semantics: actions which do not correspond to any entity in the system being modelled. For example, in a Conceptual Model for a program each term might have its own meaning but there may be 'ambient' semantics not associated with any particular terms, such as incrementing a counter every execution-step, doing

garbage-collection, etc. 'Ambient' semantics may (or may not) be ascribed to a distinguished 'runtime environment' or 'master' entity added to the Conceptual Model.

6.1.2.3 Levels of Detail

Conceptual Models can be at varying and multiple levels of detail

Like all models, Conceptual Models vary in the information they include about the system being modelled. Conceptual Models can consist of a single atomic entity representing the entire system being modelled, through to having one entity for each token in the program (as evidenced by our ability to read and 'visualise' code, line by line).

For example, given a program with classes and methods, we can view every instance of a class as an atomic entity in the Conceptual Model of the program. Or, we could view those instances as being non-atomic, a combination of state and an implementation of each method. Similarly, we can view the contents of 'sub main' in a program as the atomic entities of the Conceptual Model of the program, with the rest of the program just defining their properties. Or the Conceptual Model may contain of only a single entity representing the entire program.

Conceptual Models can also include information relating to *multiple* levels of detail. For example a particular Conceptual Model may contain information about individual constructs used in a program, as well as information about what entire methods do, what particular classes are meant to encapsulate, and what the entire program does.

6.1.2.4 Good Conceptual Models

Some Conceptual Models appear to be better than others

While not important that we do so, we propose that the underlying reason that makes one Conceptual Model appear better than another is that it permits likely questions to be able to be answered with lower total effort. One particularly-common question asked regarding programs is: "What does this part of the program do?". Another is "What are the best modifications to make to the program to achieve this desired change in behaviour?".

6.1.3 Hosting

Program source-code which includes a hole, plus a programming language, equals a new programming language

Consider the situation where one has a program (or programs) written in a particular programming language, and the desire to run them in some other chosen programming language. Both definitions and interpreters are potential means of achieving this. One simply takes the program to execute and wraps it in appropriate program-text: definitions, or terms that results in the wrapped program being treated as data and passed as a parameter to a suitable interpreter. The result is something that can be run in the chosen programming language. We can formalise the above situation as follows, using the previously-introduced concept of a Programming Language Model. This formalisation also can be used as a foundation for the previous work in TFP ([BCP94] and [Bai86]) on language-extension via definitions.

Consider a particular programming language (which acts as we term it as a 'host programming language') modelled as the tuple '$<\mathcal{L}_H, [\![\ ]\!]_H>$'. Based on this tuple, new PLMs (for what we call 'guest programming languages') can be formed as:

$$<\mathcal{L}_G, [\![\ ]\!]_G>$$

where:

$$\mathcal{L}_G \subseteq \{c \mid ..c.. \in \mathcal{L}_H\}$$

$$[\![\ ]\!]_G = (\lambda c.\ [\![..c..]\!]_H)$$

and where '.. ..' stands for some fixed program-text (the 'hosting code') with a hole in it, a hole that 'c' (the 'guest code') fills. Note that the above basic scheme could be extended to permit non-contiguous holes with 'c' being correspondingly compound. Also, note that the hosting code may consist solely of the hole. We say that the pair of hosting code and host programming language 'hosts' (or, is a 'hosting of', or 'implements') the guest programming language. Similarly, we say that the guest code placed into the hole is 'hosted' by the pair. The 'guest' and 'host' terminology we are using here is drawn from [O'D85, p235]. There does exist competing terminology, eg 'object language' and 'meta language'.

In terms of programming languages rather than models thereof, the above corresponds to taking a programming language ‘H’, writing some program-text (for example, some definitions) and marking a location where the guest code is to be inserted. The result, once decisions have been made regarding any additional syntactic restrictions (ie the ‘⊆’), is a new programming language ‘G’.

Production of new programming languages via use of a host programming language and hosting code is one means of doing language-extension. Other means exist, for example the ‘orthophrastic’ extensions identified in [Sta75] which modify the implementation of the host programming language.

6.1.3.1 Examples

Simple examples illustrate the concepts regarding hosting

As an example of applying the above concepts, consider the following code in some functional programming language (call it ‘P’):

add ≙ (+)

4 add 6

Here we can recognise that programming language ‘P’ acts as a host programming language and that “4 add 6” is guest code, written in some unnamed programming language. If this unnamed programming language has been hosted correctly by the definition (the hosting code) and ‘P’, then it follows that it is the same as ‘P’ but supports an additional construct. From reading the definition, this construct, ‘add’ has semantics indistinguishable from that of ‘(+)’. Note that possibly the unnamed programming language might permit strictly fewer programs than ‘P’ (augmented with ‘add’): we have no way of telling. Overall, the above can be said to be an example of language-extension, of extending programming language ‘P’ with a new construct (‘add’) which permits it to host programs that use that construct.

If we include additional definitions then there will be multiple host and guest combinations; for example consider:

minus ≙ (-)

add ≜ (+)

4 add 6

The first line can be seen as hosting code, which in conjunction with the host programming language ‘P’ hosts the program formed by lines 2 and 3. Alternatively, the second line can be seen as hosting code for a hosting of line 3, written in a language equal to ‘P’ but with a ‘minus’ construct added due to line 1. Further, lines 1 and 2 can be seen as hosting code for a hosting of line 3, written in a host programming language equal to ‘P’ but with support for additional constructs as specified by line 1 and line 2.

As another example of hosting, consider the following hosting code, in some appropriate host programming language:

succ ≜ (λn,s,z. n s (s z))

zero ≜ (λs,z. z)

add ≜ (λa,b. a succ b)

print *[hole]*

Here ‘*[hole]*’ denotes the required hole in the program.

Let ‘print’ denote a routine that displays Church Naturals as base-ten numerals. Among others, the above code is a hosting of the particular guest programming language ‘E’ defined by the grammar:

E ::= (add E E) | N

N ::= (succ N) | zero

and characterised by the axiomatic semantics (‘+1’ being the base-ten increment function, and ‘0’ base-ten zero):

add zero Y = Y

add (succ X) Y = succ (add X Y)

succ X = (+1) X

zero = 0

For our final example, as a contrast to the definitional-style hosting above, we present an equivalent interpretive-style hosting. Using some hypothetical host programming language supporting string-based pattern-matching, one can write:

interpret (“zero”) ≙ 0

interpret (“succ ”++N) ≙ 1 + (interpret N)

interpret (X++“ add ”++Y) ≙ (interpret X) + (interpret Y)

interpret (“(” ++Z++“)”) ≙ (interpret Z)

print (interpret “*[hole]*”)

Here, ‘++’ is intended to have the semantics of string concatenation. This is again a hosting of (among others) the same programming language ‘E’, assuming the host programming language is appropriate. Placing around the hole string-quotes as we do here is generally the most flexible option for hosting, as then the host programming language places the fewest restrictions on what can go in the hole and also (as discussed later) what programming languages may be hosted. For example, in most functional programming languages, presenting:

44 $$55 + 66$$ ^ 1EZ4Z0.3.4 andor then do 2X &=&])(1X+

outside of string quotes would cause a parse error, regardless of what else surrounds it (excluding comment-markers). However, if presented inside string-quotes in a program, the program will parse.

6.1.4 Designs

Designs are distinguished Conceptual Models that are intended to represent abstract solutions to problems

It has long been recognised that we conceive solutions to problems at a relatively high level of abstraction and then use that conceived solution to produce ‘low-level’ program source-code. We term these distinguished, high-level, Conceptual Models ‘Designs’.

In the classical software-engineering process, one first produces a Design, which embodies abstract algorithms. One then implements the Design by selecting or constructing an acceptable

programming language, program pair. Commonly the choice of programming language is highly constrained. This is made up for however to some extent by the ability to perform language-extension by use of hosting code.

Note that low-level implementation details which while necessary are not specific to the chosen solution and hence not in the Design will appear in the pair. Conversely, high-level information (aggregate properties of program constructs) in the Design will typically not be explicit in the pair; hence going from the pair back to the Design will require non-trivial 'reverse engineering'.

6.2 Objective differences, not dependent on how the programming language is implemented, between definition and interpretation

Whether the language-extension which occurs is definitional or interpretive in style cannot be determined objectively without use of details of the programming language's implementation

Based on the above framework, we now discuss differences pertaining to interpretive- versus definitional- style hosting which are both objective and independent of the implementation of the programming language. In the sequel the converse is covered, ie subjective and/or implementation-dependent differences.

6.2.1 Objectivity and independence from implementation details of the programming language

We desire objective differences between the two styles of language-extension, differences which are not dependent on implementation-details of programming languages

Of all differences between interpretation and definition with regards to language-extension, the ones of the most interest are objective in nature, and also independent of the details of the host programming language's implementation. Objectiveness is clearly desirable; and independence from the host programming language's implementation can be seen to be so too: for a given hosting, details of the programming language's implementation may not be available, or even exist. One can consider dreamt-up abstract programming languages which don't even have concrete implementations.

The differences in question are hence *consistent* differences: they are independent of the host programming language's implementation and of the person doing the analysis.

Without needing to attempt to formalise 'interpretation', we are able to achieve some results in this area.

6.2.2 What is known objectively and is not dependent on the programming language's implementation

Only the observable results of applying the programming language to the text of a program, and the text of the program itself, are both known objectively and are consistent across implementations of the programming language

The form of language-extension considered in this Chapter consists of a host programming language and some hosting code being paired to form a new programming language, one which is supplied with guest code. All we know objectively about the host programming language (without dependence on particular implementations of it) is extensional information: this is modeled as the mapping '⟦ ⟧'. Turning now to the hosting code and guest code, we note that they are simply text and hence our knowledge regarding them is both objective and, trivially, independent of the host programming language's implementation.

Hence, objectively and independently of the host programming language's implementation, we know when using this form of language-extension only:

- The mapping '⟦ ⟧' of the host programming language, from program-text to observables
- The guest code
- The hosting code

6.2.3 Determination of the style of language-extension occurring

Extensional information regarding the programming language, and the program source-code, do not suffice to let one objectively, and without reference to details of the implementation of the programming language, determine whether an interpretive or definitional style of language-extension is occurring

Using what is known objectively and independently of the guest programming language's implementation, ie the hosting code, guest code, and the mapping '⟦ ⟧' of the host programming language, one cannot distinguish between interpretive-style language-extension and definitional-style language-extension. For any given triple of guest code, definitional-style hosting code, and mapping, there exists an implementation of the host programming language which detects via pattern-matching that specific guest code and hosting code combination. On detection it then internally executes particular *interpretive*-style hosting code and guest code resulting in the same return-values from '⟦ ⟧' as if the original hosting code and guest code were executed. Likewise, there exists an implementation of the host programming language which on detecting chosen interpretive-style hosting code and guest code, executes *definitional*-style hosting and guest code to give the expected return-values. With the information available one simply doesn't know the actual 'meaning' of the program: as discussed previously, source-code has no objective 'meaning' by itself; and the mapping, being extensional, does not tell us about what happens during execution. Hence, we cannot know whether what looks to us to be definitional-style hosting code actually leads to definitional-style language-extension, and similarly for interpretive-style hosting code.

Note that the above also holds for the weaker case, where rather than a triple, we are presented with the text for the whole program, with hosting code being undifferentiated from guest code.

6.2.4 Correlations

Some programming languages objectively and implementation-independently distinguish between definitional and interpretive style hosting code

Above we discussed how the language-extension actually performed when executing a program cannot be determined objectively and without reference to the programming language's implementation. However, one may note that Programming Language Models might distinguish between interpretive and definitional styles of hosting code by their '⟦ ⟧' returning different values on the two cases: there can be a correlation between the style (as we subjectively judge it) of the hosting code, and these values. What language-extension is *actually* occurring is as previously discussed not objectively and implementation-independently knowable, but the '⟦ ⟧' can produce values *consistent with*, ie *as if*, a particular style of language-extension was occurring. Being only a correlation little more can be said unfortunately; but programming languages commonly encountered in practice today do distinguish (objectively and implementation-independently) in this way between definitional and interpretive styles of hosting code. We discuss this further below.

Before continuing, we introduce some additional notation. We say hosting code 'a' and hosting code 'b' are 'equivalent' (assuming some host programming language 'H' and guest programming language 'G') iff for all guest code 'c' in '$\mathcal{L}_G$':

- placing 'c' into the hole in 'a' gives a program in '$\mathcal{L}_H$'
- placing 'c' into the hole in 'b' gives a program in '$\mathcal{L}_H$'
- '$⟦ ⟧_H$' returns for both of the above programs identical information about their overall computational semantics (eg input-output relationship)

Consider a given host programming language and two 'equivalent' (in the above sense) hosting codes: one definitional in style, and the other interpretive. Common host programming languages will exhibit differences in the return-values of '⟦ ⟧' depending on which is used in a program. For example, differing type information about the guest code will typically be produced. Generally-speaking, the return-values when definitional-style hosting code is used will be consistent with the guest code being considered 'real code' for which full typing,

optimisations etc are done via the host programming language's built-in static analysis abilities. This stands in contrast with interpretive-style hosting code, for which the return-values will typically be completely consistent with the guest code being treated as if it were data to be processed at runtime. In today's common programming languages, no interpretive-style hosting code will give return-values consistent with the guest code being treated as 'real code' rather than data. Essentially, in these programming languages, interpretive-style hosting code can be used to host guest programming languages only as far as those return-values of '⟦ ⟧' pertaining to overall computational semantics.

Note that we say "consistent" in various places above because nothing stronger holds, objectively and independently of programming language implementation details.

A conclusion we may draw from all this is that use of definitional-style hosting code yields guest programming languages with *a greater number of desirable* return-values of '⟦ ⟧' than interpretive-style hosting code, when using common host programming languages.

Conclusions we can't draw include any to do with the nature of interpretation. One might consider the possibility of characterising interpretation as occurring in a hosting precisely when those return-values from '⟦ ⟧' pertaining to overall computational semantics are as desired, but other return-values are not. However, one must then determine what other return-values to consider; the choice of such is subjective. Further, this characterisation would be based on *results* of doing interpretation, when we are after what *makes* something interpretive.

6.3 Differences that are subjective and/or dependent on how the programming language is implemented, between interpretation and definition

The conceptual difference between definitional and interpretive styles of language-extension lies in their differing uses of, and additions to, properties of the guest code assigned by the host programming language

We now further our discussion of the differences between interpretation and definition by considering the complement of the requirement that differences be objective and independent of

the programming language's implementation. The differences which are now considered are those that are:

> true objectively, but only in *expected*, *typical* or *canonical* implementations of the host programming language;

and/or

> are true subjectively, ie in some Conceptual Model of the host programming language consistent with its PLM.

We begin by conceptualising how guest programming languages are hosted, as three distinct stages. We then outline Conceptual Models for the two styles of language-extension based on those three stages. The differences between the two styles are then detailed, including how they impact on programming and how they may be related to the nature of interpretation. Interestingly, we conclude that the fact that the two differing styles of language-extension seem to exist with no immediately-apparent intermediates is due to the nature of today's common programming languages.

6.3.1 Conceptualising the form of language-extension

Conceptually, language-extension done using hosting code is a three-stage process involving assignation of properties to the guest code by the host programming language, then by the hosting code, before finally the observable outputs are produced

We now present a reasonable conceptualisation of the general process of language-extension as done using hosting code. It is of course subjective, however it can be objective if the chosen host programming language implementation actually does proceed along these lines.

Firstly the host programming language reads in the guest code and does some initial processing of it, assigning it and parts thereof various properties (semantics, types, static relationships etc). Note that this may involve, passively, part of the hosting code: the host programming language may recognise data-type introductions, literal-quotes, etc which act to modify its parsing of the guest code.

Secondly, the hosting code (as an active agent, 'enlivened' by the host programming language) assigns properties to the guest code of its own.

Third and finally, the host programming language takes the guest code, with its newly-assigned properties, and produces outputs corresponding to the return-values of '⟦ ⟧' from the PLM of the host programming language. Some of these outputs may be produced from the program-text more generally, for example one such output is a count of the total number of lines of code, hosting code included.

It is important to keep in mind that what we discuss here is subjective or is based on the programming language's implementation. Just because it happens in a chosen Conceptual Model or some particular programming language implementations doesn't mean it will happen in others. On the other hand, programming language implementations would typically be *consistent* with this conceptualisation, ie give outputs *as if* they had proceeded as described above.

Next we give outlines of Conceptual Models, following the above conceptualisation, of the two forms of language-extension of interest.

6.3.2 Interpretive and definitional hosting conceptualised

There are a number of conceptual differences between definitional-style and interpretive-style hosting code

In today's common programming languages, there are two identifiable, distinct, styles of hosting code: the interpretive and the definitional. For each we present an outline of a Conceptual Model of the host programming language's execution based on the above three-stage conceptualisation. The two Conceptual Models will be seen to differ from each other in the first two of the stages, and also in the resultant outputs.

Later, based on the above, we will contend that there is no true dichotomy between the two styles of hosting: they appear as distinct alternatives only due to the nature of today's common programming languages.

6.3.2.1 First stage

The guest code is treated by the host programming language as either code or data; depending on the hosting code

The first stage of our Conceptual Models consists of the host programming language assigning properties to the guest code. What properties are assigned varies between the two styles of hosting code.

In typical interpretive-style hosting, literal-quotes or data-type introductions in the hosting code are recognised by the host programming language. This results in the guest code being read in as a data-structure with only basic, generic, properties (ie those pertaining to a data-structure) assigned.

In contrast, in definitional-style hosting, the guest code is read in not as data but as a program-fragment ie 'real code'. The definitions in the hosting code are used to aid concrete syntax parsing. Occurrences of host programming language primitives are assigned properties pertaining to their semantics, statics, etc. For the new constructs introduced by definitions, only properties relating to statics are assigned.

6.3.2.2 Second stage

With definitional-style hosting code individual constructs in the guest code are given different properties than with interpretive-style hosting code

In the second stage of our Conceptual Models, the hosting code acts as a function which takes guest code plus properties as input and returns the guest code with possibly-modified properties as output. The combination of guest code and properties we will refer to as a 'guest object'. This 'guest object' is used in the next stage to produce the observable outputs.

In definitional-style hosting code, new *individual* constructs, those being 'defined', are given computational semantics, types, etc. In contrast, in interpretive-style hosting code, the interpreter assigns overall computational semantics to the 'guest object' *as a whole*. Interpreters

as we conceptualise them take as input a ‘guest object’ which has basic properties, ie that of a data-structure or similar. They then carry out computations on it, building up computational semantics for it; traditionally by use of pattern-matchers. The semantics of individual constructs are not set or modified; instead the ‘guest object’ as a whole is given computational semantics.

Note that we consider empty hosting code to be definitional in style. Also, note that in our conceptualisation introducing definitions of the form ‘A ≙ A’ (this is not intended to be recursive; but rather ‘let the new value of A be what its old value was’, ie semantically null) will vary when semantics are assigned to occurrences of the construct ‘A’ in the guest code, from the previous stage to this stage. Nothing in this Chapter will be based on this variation, as whether the guest code uses constructs which happen to exist in the host programming language with the same lexical token and semantics, or not, is not of concern.

Overall, with definitional-style language-extension, most of the properties of the guest code are assigned by the host programming language. Only a few properties are assigned by the definitional-style hosting code; specifically semantics, types etc for the constructs being defined. In contrast, with interpretive-style hosting code, the host programming language reads the guest code in as data and assigns the guest code properties that are correspondingly basic. All computational semantics needs to be assigned by the interpretive hosting code.

6.3.2.3 Third stage

As a final step, outputs are produced

In the third and final stage of our Conceptual Models the host programming language comes into play again, generating outputs based on, primarily, the ‘guest object’ produced by the second stage. With definitional-style language-extension, the ‘guest object’ is rich in properties from which to generate useful outputs. The outputs can contain information about individual constructs and parts thereof as if the guest code were treated as ‘real code’, as indeed it is in our conceptualisation. If instead an interpretive-style hosting is chosen, there is no possibility of generating the same or equivalent outputs, as the ‘guest object’ has only the properties of a data-structure, plus overall computational semantics.

6.3.3 Result of the differences: interpretive-style language-extension is demanding to write

Interpretive-style hosting code is typically quite involved to write, as it often involves significant reimplementation rather than reuse of host programming language functionality

A choice typically needs to be made when constructing hosting code in today's common programming languages, between source-code that is interpretive versus definitional in style. The latter is usually preferable, for as we've seen, useful outputs (other than overall computational semantics) pertaining to the guest code are available. In addition, interpretive-style hosting code is generally somewhat complex to write, because overall computational semantics and the like need to be produced from the basic properties (those of a data-structure) initially assigned to the guest code. In contrast, use of definitional-style hosting code results in outputs such as overall computational semantics being able to be produced based on quite 'rich' properties produced from the static analysis done by the host programming language. This means that interpretive-style hosting code has more work to do than equivalent definitional-style hosting code.

To begin with, note that with a typical host programming language interpretive hosting code has only limited control over the grammar used to parse the guest code in as data. As a result, the interpretive hosting code has to do significant amounts of additional parsing (involving recursion, one of the hallmarks of interpretation claimed in TFP) as part of the implementation of the desired guest programming language. Similarly functionality for static scoping, name-binding etc is also required to be implemented; which is non-trivial to do even when suitable libraries are available. As a result, interpretive-style hosting code is typically complex, and hence harder to understand and maintain. This latter point is important as maintenance of programs has long been a very common activity (see eg [Win79]), and is even considered by some ([Leh96]) to be a *necessary* activity.

In contrast, when implementing a new (guest) programming language via definitional-style hosting code, nearly all host programming language functionality is implicitly reused rather than needing to be reimplemented. As well-stated in [KG95], "reusability [is] the one sure-fire way

to increase programmer productivity and program quality", and this is certainly borne out when comparing interpretive and definitional means of language-extension.

One should note that reimplementation or reuse of language features when doing language-extension is well-recognised in the literature. Pfenning and Lee in [PL89], for example, refer to features of the host programming language not needing to be reimplemented for the guest programming language as being 'inherited'; and give a number of examples including type-inference, unification (for the logical programming paradigm), and evaluation-order. Also, in [LLMP89] (based on [PE88]), the specific example is given of the well-known technique of hosting a program so that variables in the guest are variables in the host, and name bindings (eg lambdas) in the guest are name bindings in the host. As variables and bindings are inherited, this leads to some type checking also being inherited. Further examples of inheritance of features when using definitions can be found in [CGKF01]; these are explicitly stated as being occurrences of reuse. Finally, similar inheritance of host programming language features can also be seen to be occurring in [Wei01], where programming languages with powerful type operators are implemented via definitional-style language-extension: we draw particular attention to this as hosting typing operators is somewhat unusual.

Finally, one should note that these insights can be applied to situations other than those where hosting code and guest code are clearly present. Consider a function, one of whose parameters is data used to modify the function's behaviour. One can view the function, with the host programming language, as doing language-extension: the data given as the parameter can be viewed as guest code. If one wants to be able to vary the function's behaviour over a wide range, one needs to build significant complexity into the function. In contrast, if the function took first-class semantics as input, the host programming language's ability to parse and produce semantics is reused instead. For example, one may have a function called 'membershiptest' which takes an item of data representing a set; and whose behaviour (ie the mapping from supplied element to test to Booleans) depends on that item of data. If one wanted to be able to represent arbitrary computable sets, 'membershiptest' would typically be quite complex. This is especially true if the set-representations are desired to be as good reuse-wise as one desires for one's code generally, ie able to be easily modified, combined, etc to represent other desired sets. Note that such also implies ease of maintenance, for maintenance can be viewed as creating a new construct while having the old one available to reuse (as a whole or by parts). In contrast to representing sets as data and needing a complex membership testing function, one can use

Characteristic Predicates instead. They are much simpler to implement, and 'membershiptest' can be just the identity function: this indicates the extent of reuse of host programming language functionality. An additional positive is that as today's programming languages continue to evolve towards letting programmers write more reusable code, Characteristic Predicate set-representations that are increasingly good reuse-wise will automatically become available as well.

6.3.4 Result of the differences: the programmer needs to make trade-offs

The differences between the two styles of language-extension leads to the programmer needing to make trade-offs

As intimated above, the choice of which style of language-extension to use involves trade-offs. Perhaps the most important arises from the fact that interpretive language-extension has great expressivity (the Turing-computable functions) in terms of mappings from guest code to overall computational semantics; but is inexpressive in regard to other outputs (ie of '⟦ ⟧'). For an example of trading-off the various outputs, consider host programming languages which don't support parallelism. Typically, definitional-style hosting will not be able to yield the desired mapping from guest code to overall computational semantics (specifically, for guest code using parallel constructs). However one can instead host the guest code interpretively, using the well-known technique of 'dovetailing'. The desired overall computational semantics can then be achieved, at the cost of other outputs such as what one would want in terms of static type checking of the guest code. For another example, when first-class semantics is used in the guest code and such isn't available in the host programming language, interpretive language-extension can produce the desired computational semantics when definition cannot.

We now consider an additional trade-off: having to modify the guest code to permit the desired language-extension style. Rather than using an interpreter, it can be preferable to permit the guest code to be modified in minor ways, so that the host programming language's parser will permit a definitional hosting. Similarly, it is sometimes the case that even for an interpretive-style hosting to be possible, some change to the guest code is required. For example, in many programming languages, string-literals in guest code can't wrap over multiple lines and modifications are required. Seemingly irrelevant changes to guest code can vary when a

definitional-style hosting versus an interpretive-style hosting is possible. For instance, the following guest programming language cannot be hosted in many host programming languages using definitional-style hosting code:

Guest code	Observable output
zero	0
succ zero	0
succ (succ zero)	1
succ (succ (succ zero))	2
etc..	

This guest programming language maps from guest code to a value of one less than what the guest code nominally denotes. Now, the first row implies, in many host programming languages, that a definitional hosting must have ‘zero’ defined to have the value ‘0’. This means that in the second row ‘succ’ must be defined such that it maps ‘0’ to ‘0’. Putting these together implies that the third row should have ‘0’ rather than ‘1’ as the output: a contradiction. However, if we change the guest code to:

```
dummy zero
dummy (succ zero)
dummy (succ (succ zero))
dummy (succ (succ (succ zero)))
etc
```

ie introduce a distinguished head (or ‘root’) constructor, ‘dummy’, then definitional hosting code is now possible: one which binds ‘zero’ to ‘0’, ‘succ’ to an increment operator, and ‘dummy’ to a decrement operator (one which returns ‘0’ on being supplied ‘0’). The reason comes down to ‘fold’-expressibility, and the above trick can be generally applied.

Changing the guest code does however involve some costs. Importantly, the more modifications are done, the more removed the guest code is from the Design from which it would have been originally constructed. Properties explicit in the Design (ie statically-determinable from some good representation of the Design) may no longer be statically-determinable from the guest code: see the unpublished work [McA98] for an interesting discussion of these sort of issues. There are a number of other interesting aspects to Designs and guest code, such as the benefits of having guest code ‘aligned’ with the Design, and the rising costs and lost reuse opportunities for

different forms of misalignments. To avoid straying too far afield, we give a single example. Say we have a Design, and we write a program based on it. When we change the Design, we will generally desire a new program. It is usually easiest to produce the new program by modifying (reusing) the old one. Effort is minimised if only localised changes to the guest code are required (for a concrete example, see eg [KG95]). To have this always be the case, one can argue that the guest code and the Design need to be in alignment; ie distinct objects in the Design should be self-contained and contiguous in the guest code.

Finally, one should note that tradeoffs may have to be revisited as circumstances change. Definitional-style hosting code is typically less complex and hence easier to modify (maintain) when required. However, the range of possible modifications is usually rather limited. Starting with interpretive-style hosting code might in the end minimise total costs as it can be reused at each step; rather than one being required to switch from definitional to interpretive style hosting code at some stage. Interestingly, due to this and the variety of other potential reuse considerations for a given project (eg what is already available for reuse and what one may want to build in the future that could draw from current work), it seems highly unlikely that any particular hosting code or guest code is a poor one for *all* circumstances. One can likely be able to imagine scenarios in which any particular hosting code or guest code is actually optimal. Hence even heavily-interpretive code might be the best choice, if eg the ease of its production is considered to outweigh other reuse considerations. Writing interpreters is not necessarily always bad, it can, perhaps rarely, be the best choice.

6.3.5 Bridging the differences

There are intermediates between the definitional and interpretive styles of language-extension

Under our conceptualisation, the interpretive and definitional styles of language-extension differ in three regards:

a. The properties the host programming language assigns to the guest code
b. The properties the hosting code assigns (or modifies)
c. The resultant outputs which are produced.

Above, we have considered various consequences of these differences. However, are these differences fundamental?

On the first point of difference, why must the programmer be forced to choose between the guest code being read in as 'real code' (only), versus data (only)? Couldn't the guest code be read in as 'real code', but then be able to be investigated and modified by interpretive-style hosting code?

On the second and third points, note that with interpretive-style hosting it is often the case that the programmer knows certain information to be derivable, but has no mechanism available by which to instruct the host programming language to do the derivation and then generate appropriate outputs. For example, the programmer may know that particular optimisations can be done to data representing code; or that certain static analysis eg type checking could be applied, but cannot make that happen. We can imagine programming languages in which hosting code can do that sort of analysis, and then flag the results to be outputted as an overriding of the usual outputs so that the outputs now refer solely to the guest code, rather than the entire program.

It seems that none of the three differences appear to be fundamental; ie one cannot draw a hard line between definition and interpretation. Rather, it is apparent that limitations of today's common programming languages give the appearance of there only being two alternatives, rather than a continuum. We suggest that these limitations in today's programming languages are akin to an historical accident. Current programming languages tend to have a schism between reading in the guest code as 'real code', giving it the full set of properties, and reading it in as data. Added to this is a lack of support for hosting code to affect outputs pertaining to the guest code, other than those related to overall computational semantics.

We discuss below some specific means by which the apparent differences between interpretation and definition can be bridged, and the relative costs of interpretation reduced.

6.3.5.1 Improved static analysis

Many interpreters are amenable to static analysis, hence could theoretically give the same outputs as when definitions are used

In theory, there is no reason why languages could not give the same outputs (ie those of '⟦ ⟧') for hosting code utilising a single standard interpretive loop with fixed tests (eg 'fold') as for definitional hosting code. The host programming language could detect such hosting code, and engage in static analysis of it (as in eg [Hug96]); resulting in it being treating it as if it were instead a set of definitions. While more generally interpreters can pose an insurmountable challenge for static analysis due to undecidablility, powerful but theoretically-limited analysis techniques may perhaps often prove adequate in practice (especially when coupled with programmer annotations, eg of invariants).

It is now clear that one certainly should not try to characterise interpretation based on differences in the output of '⟦ ⟧', ie whether the guest code is apparently treated as 'real code' or as data. Such differences can arise solely due to difficulties of static analysis of program code.

6.3.5.2 Exposing for reuse

Programming languages could expose additional primitives to make interpretive-style language-extension easier

As discussed earlier, the larger size of, and effort in producing, interpreters can be put down to the programmer often having to reimplement host programming language functionality because such is not applied when the guest code is being read in as data. Reimplementation can be very costly, as typically only small, general-purpose constructs are exposed for the program's use. However, libraries have already been mentioned as a means of reducing this cost. Further, note that the host programming language could expose (more-properly, 'reify') significant constructs from its own construction, such as its lexical and static analyser. One specific possibility concerns the fact that in many cases, an interpreter is simply a pattern-matching loop which translates from data representing guest code to terms in the host programming language. Rather than having to reimplement this loop from scratch for each interpreter, the host programming

language could expose as a primitive a built-in extendible interpretive loop. As a concrete example, consider the following hosting code:

```
EvalUsing   (
            AddBinaryConstruct "_ pow _"  (λx,y. iterate y (mul x) 1) BaseInterp
            ) "[hole]"
```

Three host programming language primitives are used here: the first, 'BaseInterp', refers to an interpreter for the host programming language exposed by it, implemented for example as:

```
BaseInterp =
    (λx, recurse.
        if (match x  "_ add _") (λlhs, rhs. add
                                    (recurse lhs recurse) (recurse rhs recurse))
        else if ..
    )
```

The other two primitives could be implemented as:

```
AddBinaryConstruct construct semantics oldinterp =
    (λx, recurse.
        if (match x construct)  (λlhs,rhs. semantics
                                    (recurse lhs recurse) (recurse rhs recurse)
                                )
        else (oldinterp x recurse)
    )

EvalUsing interp string = interp string interp
```

The size of the hosting code compares favourably to that of an equivalent definition, and the two are rather equal in terms of reusability.

This example, and so-called 'reflective' programming generally, certainly blurs the distinction between what one would consider definitional versus interpretive.

6.3.5.3 Interleaving hosting and guest code

The cost of interpretive-style hosting code can often be reduced by interleaving it with the guest code

Interpretive hosting code typically has numerous costs associated with it, as we've mentioned. However, these costs can be minimised by interleaving the hosting code with the guest code, resulting in something of a hybrid between the two styles of language-extension. The interleaving is accomplished by semantics-preserving transformations (partial-evaluation, most-simply) that aim to 'push' the interpreter into the guest code. This can lead to a result where fewer constructs require interpretation, and the interpreters are simple. Consider:

interpret "if x+4=5*3 then @ x 9 else @ 4 x"

If '@' is a new construct, and all the rest are constructs which are to have the same semantics as if they appeared outside of the string-quotes, then we can typically transform the above to:

if x+4=5*3 then (interpret "@ x 9") else (interpret "@ 4 x")

Hence 'interpret' no longer needs to reimplement '+' ,'if', '=', etc. Further, another transform can remove the need even to parse:

if x+4=5*3 then (interpret x 9) else (interpret 4 x)

Despite this, uninterleaved code may often be preferable as the hole the guest code fills is contiguous. Intermingled hosting and guest code becomes specific to a particular guest program and is not very maintainable or otherwise reusable. But for basic language-extension to add new constructs, this approach is often quick and easy, compared to the alternative of reimplementing the entire programming language.

Two additional examples further illustrate the general technique. Firstly, Characteristic Predicate functions, discussed previously, can be seen to be the result of 'pushing' an interpreter into the terms representing the sets. For example:

membershiptest "..(Empty ∪ AboveZero).." x

transforms to:

..(membershiptest "(Empty ∪ AboveZero)" x)..

which can then be simplified by partial-evaluation to:

..(membershiptest "Empty" x) or (membershiptest "AboveZero" x)..

and then to:

..false or (x>0)..

The second example is a verbose means of carrying out 'defunctionalization' (citations in Chapter 5), the removal of first-class functions from code. One starts with code containing first-class functions, and then wraps that code with string-quotes and prefixes an interpreter for it. Transformations then 'push' the interpreter into the code. If 'F' is a first-class function, then:

interpret "..F.."

can, with the right transforms, be transformed to:

..(interpret "F")..

The result of the transformations is the production of the 'defunctionalized' result: a program equivalent to the original but with first-class functions (only) replaced by strings.

One might consider the existence of the above transforms as evidence for TFP's assertion that *all* interpreters, no matter how simple, really do extend the host programming language. One can take a program which uses an interpreter and use transforms to 'push out' the interpreter to form explicit, separated, hosting code which is applied to guest code representing an entire program. Whether it is always *meaningful* to do that is questionable however, especially when the input to the interpreter includes variables or dependencies on the state.

6.3.5.4 Use of representations other than data

There are intermediates between the interpretive and definitional styles of language-extension, between treating the guest code as data, and defining its constructs

As discussed, some of the differences between interpretation and definition arise in today's common programming languages due to the guest code being read in as data. However, as discussed in earlier Chapters, interpreters can also act on functional representations such as Church Objects. The guest code can be read by the host programming language as 'real code' not data, but an interpreter is still applied which produces completely different semantics. In such situations the hosting is in some ways definitional as well as interpretive in style. We now offer a related concrete example.

One can implement a construct to do parallel evaluation by using dovetailing and representing the two sub-programs it takes as input as primitive data. Alternatively, the two sub-programs

can be represented as a (Church, perhaps) structure containing first-class functions or procedures, with only the 'connections' between them being data. For example, rather than:

dovetail "call $proc_1$; call $proc_2$" "call $proc_3$; call $proc_4$"

one could have:

dovetail2

[$proc_1$, proc2]

[$proc_3$, $proc_4$]

Here, each element of the lists is a reference to a procedure. The function 'dovetail2' simply executes the first procedure from each list, then the next, etc. The individual constructs (the procedures) are not represented as data, yet the *static control-flow structure* in which they appear *is* represented by data. Is the 'dovetail2' hosting therefore interpretive in style? Or definitional?

In light of the above, we propose that whether language-extension is considered interpretive or definitional in style should be based solely on whether representations are being used. A suggested outline of this is as follows. Take the hosting code, guest code, and a Conceptual Model of the guest code. Then, check to see if in the first and second stages of our conceptualisation of the language-extension process the host programming language and hosting code assign properties to the guest code which are consistent with the Conceptual Model. If so, then representations are not being used and one can call the hosting code 'definitional'. If instead the host programming language or hosting code assigns any properties to the guest code that are incompatible with the Conceptual Model, then representations can be said to have been used and one can call the hosting code 'interpretive'. Note that one could consider not incompatibility *per se* but instead a continuum of 'how far away' in some sense the assigned properties are from those in the Conceptual Model. The further they are away, the more 'interpretive' the hosting is.

The inherent subjectivity of detecting representations in the above cannot be avoided. What is being asked is a question of *intent*, as captured in the Conceptual Model. One needs to distinguish when the program actually really is just processing data (or other), rather than such data being representations. Are those numbers, or Gödel-numbers? Is that a function being used innocently, or is it a Church Object being used to represent something else? Such concerns are inherently subjective. Hence, while one could consider characterising 'interpretation' itself based on the above, as the *presence* or *use* of representations, or as the *process* of going from a

representation to something closer to what is represented, it would be intrinsically subjective to detect.

Finally, it is important to recognise the primary reason *why* representations are used: because using instead what is represented would be too unsatisfactory in terms of the resultant observables, the outputs of '⟦ ⟧'. Often, the lack of satisfaction is in regards to the paramount observable, overall computational semantics. Returning to dovetailing as an example, typically in programming languages lacking parallelism one cannot implement an operator that takes two sub-programs and executes them in parallel. No attempted implementation of the operator can extract sufficient information from its arguments: the function is incomputable and hence the desired overall computational semantics for the program cannot be achieved. However, different representations have different sets of functions computable on them (recall the discussion on computability from Chapter 3). By using dataful (or eg Church Object) representations of the sub-programs, as illustrated above, a function with suitable semantics becomes computable and we can now implement the desired operator.

6.4 Implications for language design

Good programming languages let the programmer denote what they want, how they want

Our work in this Chapter has highlighted some apparent deficiencies in today's common programming languages with regards to language-extension. We discuss these now in general terms.

A good general-purpose programming language should permit the largest number of guest programming languages to be able to be denoted, via the writing of hosting code. We specifically include those guest programming languages whose outputs (ie of '⟦ ⟧' in its PLM) are not just overall computational semantics. Note that there is a wide variability in the classes of outputs one may want (eg type checking: yes or no), and also within each class (eg which type-system to use). Hence it is desirable to have the ability to add or modify static analysis at will, and to be able to make observable the results of such analysis.

As a general statement, a good programming language should also let 'best practice' programs be written, ie code that is constructed as effortlessly as possible and reusable in whole or part. Such code will generally not feature interleaving of hosting and guest code. Having hosting code and guest code separated in the source-code gives a separation of concerns between what is the hosting and what is the guest. Practically, it results in easier maintenance.

Given all of the above, and the variety of concerns relating to reuse in differing situations, one is drawn to the conclusion that good programming languages should let their users (ie programmers) denote *what* they want, *how* they want. This matches current thought (see eg [IRA06] for a summary) on quality generally. With such programming languages, one would never be forced to tradeoffs, such as to use representations, or interleave guest and hosting code; both with associated complexity and obscurity.

The next question is how to construct such expressive programming languages, programming languages that allow full control of their observables by small and reusable (ie well-written) hosting code. This is a difficult and involved topic we do not have a complete answer for. There are of course various works in the literature on specific problems that can be seen to be related to this general goal. Most closely related are works on 'programming language extensibility' such as [Sta75], and modern approaches such as 'mixins' ([CL03]), 'reflection' ([DM95]), and 'abstraction' generally ([Gua78]). The work [Aik91] is also relevant, as it covers explicit modifications to hosting code while leaving guest code untouched.

6.5 Conclusion

Whether interpretive-style or definitional-style language-extension is used in a program cannot be determined both objectively and without reference to the programming language's implementation; and even on weakening this one has the two styles as being more of extremes of a continuum than two distinct classes

In TFP is observed that an interpretive-style language-extension is worse than one done using definitions, but little explanation is offered. Supplying a general framework we discussed this in detail, taking particular care to distinguish what is objective and independent of programming language implementations from what is subjective or implementation-dependent.

In our framework, definitional and interpretive styles of language-extension are very similar in their gross characteristics. They both involve hosting code combined with guest code. Interestingly, objectively and independently of the host programming language's implementation, the actual type of the language-extension done cannot be determined. There is an essential inability to know how programs execute, what its semantics is, without either knowing details of the programming language's implementation, or being subjective.

Conceptualisations of how the two styles of language-extension operate, which are consistent with the above observations, were then outlined. We discussed primary and secondary conceptual differences between the two styles. The primary conclusion reached is that interpretive language-extension is in an important respect *less-powerful*, as fewer observable outputs pertaining to the guest code are controllable.

Subsequently, we proposed a new characterisation of the two styles of language-extension, based on whether there is straight-line progress towards the properties of a Conceptual Model representing the guest code, or whether a detour into a representation is made. This could be considered as the basis for a possible characterisation of 'interpretation' generally, however such a characterisation as discussed would be highly subjective in nature. Regardless, the use of representations is certainly to be avoided, due to the extra obscurity and complexity they involve.

Overall, we tend to the conclusion that the various differences between interpretation and definition considered in this Chapter do not reflect anything regarding the intrinsic nature of interpreters, or symbols and symbol tests specifically, only the effects of their use for language-extension. This Chapter indicates the probable ultimate fruitlessness of trying to find an *objective* characterisation of interpreters, in terms of a contrast with definition.

Finally, we reconsider the 'programming equals language-extension' theme in TFP. Our work makes it clear that *any* part of a program can indeed be considered as a hosting for the remainder. This is an implicit assumption in the 'proto-TFP' work of [BCP94]. However, as the split between what is hosting code and what is guest code can be made freely, the suggestion in TFP that programs should be evaluated with respect to language-quality criteria would in practice require some subjective determinations as to *which* programming should be seen as doing language-extension. An arbitrary choice as to what is hosting and what is guest code will

typically result in a rather poor (quality-wise) guest programming language: consider for example the guest programming language formed by taking the first 20 characters of a program to be hosting code. We suggest that the view in TFP that programming is a form of a language-extension should be replaced with the view that only a *selected part* of the program's source-code denotes a language-extension (namely, the hosting code), and that programs should be written with this part of the source-code distinguished.

Chapter 7

The nature of interpretation

An interpreter is something that does different computational steps depending on its input

Having made significant progress on related topics, we now tackle the central lacuna of the current formulation of TFP, a characterisation of interpretation. Without a suitable characterisation of interpretation, none of the outcomes of TFP, such as TFP languages and tools, can be realised. At first glance, interpretation might be expected to have a simple characterisation in terms of the symbol tests by which it commonly presents. However asymbolic interpretation does exist, which broadens the problem significantly. For example, from Chapter 5 we know that programs written using symbols and tests thereon can be transformed into programs that only use functions. There is no readily-apparent indication of the presence of interpretation in lambda calculus programs; one lambda-term looks much like another.

We begin by discussing our exact requirements. This is followed by details of a series of investigations we have carried out regarding interpretation, making use of results from earlier Chapters. These investigations, by elimination, act to narrow down what the nature of interpretation is. After investigating in detail whether interpretation can be considered to be testing, we give our proposed characterisation of interpretation: input-varying computational steps. Simply, a program is interpretive if and only if the computational steps it carries out depend on its input. For example, tests (on symbols or otherwise) that have one or more of their branches being code rather than data are interpretive as they do different computations for different inputs. Such tests can be found in:

add a b ≙ IF (iszero a) b (add (pred a) (succ a))

and:

fold list op b ≙ if list == nil then b else if list == cons e r then op e (fold r op b)

However, apart from the above common form of testing, there are tests that take data and return data. These include Boolean-valued equality tests such as appear in Characteristic Predicates

and Combinator Parsers. We cannot show that these tests are interpretive, or that they are not; but either answer as we show has very interesting implications.

Consequently, our characterisation of interpretation is consistent with, but goes beyond, the informal characterisation of interpretation in TFP as being symbol testing (with the possible exception of those all-data tests). As part of our analysis of the implications of our characterisation, we discuss testing for the presence of interpretation, including when the computations of the constructs involved are not known. We also consider varying degrees of interpretation, the extent to which computational processes of a function vary with its input. We show that TFP-style programming is indeed less interpretive than the usual symbol- and test-based style; but that it and first-class functional programming generally is fundamentally interpretive. Following this Chapter, we present a summary our research and detail some avenues for future work.

One should note that in this Chapter whenever details of a program's operation are described, as per Chapter 6 we are implicitly assuming a computational model. For functional languages, it is one where functions on execution first bind supplied (ie actual) parameters to named parameters, and then execute their body. This is consistent with the various canonical β-reduction -based computational models of the lambda calculus.

7.1 Requirements

An objective characterisation of interpretation, consist with interpreters being things that 'ascribe meaning', is required

We require a characterisation for interpretation that is consistent with the informal uses of the word. The consistent meaning of the word 'interpret', across disciplines, is 'giving a meaning to' ie 'ascribe meaning'. The word was of course used first in non-computing contexts. An 'interpreter' for a spoken language is of course a translator, but can also be seen as one who gives meaning (with respect to the audience) to words in a foreign language. Similarly in Model Theory, a branch of mathematical logic, we find statements such as "we interpret the binary-relation symbol *R* as a binary relation over this domain" [EFT84, p25]. In computing, the word

is used in a specific sense, to mean the assignation of meaning to representations (ie primitive data), especially symbols representing a program.

We also require a characterisation be found which is objective in nature: either a certain program or computational system is interpretive, or it is not, without any subjectivity. The requirement that the identified nature of interpretation be objective is quite important, and we will be referring to it as the 'Objectivity Criterion'. One would also prefer to be able to tell when a program is interpretive or not without dependence on the programming language's implementation. Hence decisions as to whether a program is interpretive would be independent of whether the programming language is executed via compilation or interpretation; and could be made even if the programming language is abstract, without concrete implementation. However, as discussed in Chapter 6, how programs execute cannot be determined objectively and independently of the programming language's implementation. Without implementation-dependence one cannot tell if what looks like an interpreter will execute as one. The best one can hope to do is to be able to make objective judgments as to interpretation based on classes of programming language implementations, or under the assumption of a computational model.

7.2 Topics that when investigated that may lead to identifying the nature of interpretation

A number of topics are potentially able to be investigated to give clues as to the nature of interpretation, however on analysis most can be dismissed

We discuss here topics that are hypothesised to be worth investigating in the hope of gleaning clues to the nature of interpretation.

7.2.1 The nature of interpretation can be discovered by investigating language-extension

Studying interpretive-style language-extension hasn't resulted in discovery of an objective characterisation of interpretation

Our investigations of interpretive-style language-extension in Chapter 6 did not result in discovery of an objective characterisation of interpretation. Indeed, the very concept of interpreters for programming languages was shown to be a very amorphous concept.

7.2.2 The nature of interpretation can be discovered by investigating sub-recursive programming and type-systems

The sub-recursive programming languages we have seen are seemingly unrelated to interpretation

As discussed in Chapter 4, sub-recursive programming that comes about via type-system restrictions does not seem to offer any clues as to the nature of interpretation. Sub-recursive programming more generally appears to be a result of the programming languages being limited with regards to general expressiveness, rather than anything specific to interpretation. We know of no practical language that prevents tests or more-obvious interpreters from being written.

7.2.3 The nature of interpretation can be discovered by investigating the derivation of TFP-style programs

The derivation of TFP-style programs seems unrelated to interpretation

As discussed in Chapter 5, TFP-style programs can be derived from data-centric programs via transformations and simplifications. Neither of these seem related to the nature of interpretation: the simplifications and transformations which are done are general and not eg limited to interpreters.

7.2.4 The nature of interpretation can be discovered by investigating undecidability

Investigating undecidability will likely not lead to discoveries as to the nature of interpretation

Because interpreters such as Universal Turing Machines are used in proofs of the undecidability of the 'halting problem' ([Tur36]), it may be suggested that looking at decidability itself may lead to insights as to the nature of interpretation. However, there are various reasons that we have identified for why it may be impossible for any implementation to be able to 'decide', none of which offer much for determining the nature of interpretation.

Firstly, the system may not be sufficiently powerful. For example, the numbers involved can 'grow too rapidly' for the logic (see [Sim87] etc). These are known as 'natural independence results' and do not appear to be related to interpretation rather than simply limitations of the logic in question.

Secondly, in the given system no implementation may be able to extract the required information from its inputs in finite time. For example, no implementation exists which can state when two infinite 'lazy' lists are equal. This also does not appear to be related to interpretation.

Thirdly, the question asked may be paradoxical. A simple example of this is the requirement that a implementation be found that gives a correct answer to the question "what is a number which is greater than 5 and less than 3?".

Fourthly and finally, the question asked may not be paradoxical *per se*, but with additional assertions regarding the implementation a paradox arises: contradictory demands are made. This can be seen to be the case with the famous disproof that any Turing Machine can solve the 'halting problem'. Proceeding via the usual process, consider first the specification for a Turing Machine that solves the 'halting problem':

doeshalt x y ≜

true	if Turing Machine number x halts when given input number y
false	otherwise

This specification is fine; it is well-defined and not paradoxical in itself: Turing Machines do either eventually halt, or not. Now, as per usual, assume there is a Turing Machine which implements this specification; call it 'haltsq', and consider a Turing Machine specified as:

(λn. if (haltsq n n) then donthalt else halt)

where 'donthalt' and 'halt' are self-explanatory. Let the Gödel Number of this Turing Machine be 'g'. Now we have the well-known paradox: for input of 'g' and 'g' 'doeshalt' must return both (and neither) 'true' and 'false'. While 'doeshalt' isn't paradoxical by itself, if one adds the additional assumption that an algorithm exists which solves the 'halting problem', then a paradox is formed. One also finds a similar situation elsewhere, for example with Kolmogorov Complexity and even in conceptually very simple systems such as those involving only complex arithmetic ([BY07]). Note that none of the above involves much in the way of details of computational processes, hence there is little to investigate regarding interpretation.

7.2.5 The nature of interpretation can be discovered by investigating the logical programming paradigm

The logical paradigm is based on tests and is therefore unlikely to offer clues as to the nature of interpretation

The essence of the logical programming paradigm, unification, appears to be based on tests on data: it is hard to imagine how unification could be implemented without use of pattern-matching. Hence investigating the logical programming paradigm seems unlikely to yield clues as to the nature of interpretation not able to be found by considering the use of data in the functional paradigm

7.2.6 The nature of interpretation can be more-readily established in the realm of logic

The world of proofs doesn't seem of offer any advantages over the world of programming

As discussed in earlier Chapters, there is a well-known correspondence between programs, types and proofs. This correspondence may permit questions pertaining to interpretation be recast and

answered in the realm of logic, where the answers might be easier found. The answers could then be brought back to the realm of programming. However, from the materials that we have read on the topic such as [Lei83], this approach would not seem to offer any advantages in practice.

7.2.7 Analog systems are not interpretive, hence by contrasting them to other systems the nature of interpretation may be able to be determined

Some analog systems are interpretive

Analog systems are finding specialised use even with today's availability of fast computing hardware. For example, one can set up equipment to near-instantaneously calculate the computationally-expensive two-dimensional Fourier Transform and its inverse, see eg [CK99]. It is suggested in TFP that analog systems don't engage in the interpretation common in digital computational systems. However, analog systems are actually capable of interpretation. As shown in [Bra95], using Ordinary Differential Equations one can simulate a Universal Turing Machine, the canonical interpreter and something which itself can step-wise simulate any other system one may consider interpretive.

7.2.8 Church Objects being the sole members of their polymorphic type is related to interpretation

Church Objects being the sole members of their polymorphic type is unrelated to interpretation

It had been noticed during the development of TFP that some of the examples of higher-order functional-programming are the sole members of particular polymorphic types. For example, the only functions of type

$$\forall\alpha.\ \alpha \multimap \alpha \multimap \alpha$$

are the Church Booleans; and the only functions of type

$$\forall\alpha.\ (\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$$

are the Church Naturals; similarly for Church Lists. In TFP, there is no explanation for this relationship between type and Church Object; but it is postulated that it might relate to interpretation.

As covered in Chapter 5 however, these types arise due to Church Object Elements being the result of parameterising constructors from data or other terms. No link to the nature of interpretation seems able to be made.

7.3 Investigated possible characterisations of interpretation

A number of hypotheses regarding the nature of interpretation are demonstrably false

Below, we discuss various hypothesised characterisations of interpretation, dismissing each.

7.3.1 Interpretation is 'enlivening' of the inert

Interpretation is not 'enlivening' of the inert

It is noted within TFP that interpretation typically turns inert data into behaviours; the simplest example of this is the humble 'if..then..else' construct which turns inert Booleans into the semantics of the branches. However, 'enlivening' the inert cannot be the essence of interpretation, as for example lambda calculus programs consists only of behaviours (ie everything is a function) and yet interpreters, specifically 'universal interpreters' eg the Partial Recursive Functions can still be written ([Bar84]).

7.3.2 Interpretation is redefinition of the application (meta-)operator

Interpretation is not a redefinition of the application operator

Apart from this work, and the previously-cited TFP papers, the only other published investigation into TFP is the doctoral dissertation of M. Gong ([Gon93]). The central contention of the work is that interpretation occurs when the 'application' operator (commonly denoted by juxtaposition) in functional languages is redefined in order to host some desired language. One could view this more weakly as a claim that interpretation is required when definitions involving *constructs* (but not application) are insufficient to implement the desired language-extension.

Having the nature of interpretation actually being a redefinition of application, as is claimed, in our opinion cannot be supported as:

1. Definition bodies can contain interpreters, which will be ignored

2. The guest code can contain interpreters, which will be ignored

3. It requires splitting the source-code into guest and hosting code, which is a subjective process if a program without such an explicit separation is supplied

4. It is restricted to functional programming, however other paradigms exist in which interpreters can be written

5. Code written one way may be interpretive by this characterisation, but nearly identical code (perhaps transformed mechanically) in eg the Continuation-Passing Style may not

6. One can mechanically introduce a new construct called 'Apply' before every application of a given program. For example:

F (G H)

rewrites to:

Apply F (Apply G H)

This new construct can then be defined as necessary instead of redefining application. Hence one is asked to believe that all interpreters can be made non-interpretive by such a trivial transformation.

7. Even redefinition of application may be *insufficient* to implement the desired hosting; ie the purported characterisation of interpretation does not cover all cases. For example, consider programs written using a zero-arity construct 'Z' and a unary construct called 'S'; eg:

S (S (S Z))

Say we wanted to have these programs denote a Natural number, specifically, the number of 'S' constructs they contain, minus one. Definitions for 'S' and 'Z' to achieve this are not possible, as the requirements equate to the 'predecessor' operation being 'fold'-expressible, which it is well-known not to be. No solution will exist *even if* one also

permits redefinition of application, as can be seen by introducing before each application a new construct ‘Apply’ which will take on the redefinition instead:

Apply S (Apply S (Apply S Z))

Now, writing ‘AS’ for ‘Apply S’ gives:

AS (AS (AS Z))

As no suitable definitions for ‘S’ and ‘Z’ exist no suitable definitions for ‘AS’ and ‘Z’ can exist, and therefore no suitable definitions for ‘Apply’, ‘S’, and ‘Z’ can exist. To host this language we need to add a new construct or introduce non-functional features to the language.

7.3.3 Application of function-valued functions is related to the nature of interpretation

Application of function-valued functions need not occur in interpreters, hence is unrelated to the nature of interpretation

An interesting observation can be made regarding lambda calculus functions: the functions that do tests and seem to be interpretive contain applications of the results of functions. Given that in the lambda calculus all terms are functions, one has to make a determination as to when functions return. We take a function to return when it has been supplied with as many arguments as it takes in de-curried Head-Normal Form, hence:

(λx. (λy. M))

will return after being given two arguments, not just one.

As an example, consider addition on Church Naturals defined as:

add a b ≜ (λs,z. a s (b s z))

In this function, there are only two applications. The first, of ‘a’, does not have its result applied to. The second, of ‘b’, gives a result that may not be ever applied (depending on ‘s’). In contrast we have the definition:

add a b ≜ (iszero b) a (add (succ a) (pred b))

where:

iszero n c d ≜ n (λx. d) c

This definition for most ‘b’ applies the result of ‘pred b’, in ‘iszero’.

We now show that terms that involve applications of results of functions can be transformed to terms that do not (albeit with a change of input and output representation). As such transforms are applicable to all lambda calculus terms, even those one may consider interpreters, there is nothing in this form of application that relates to the nature of interpretation.

We now introduce a transform, denoted by ‘⇒’, that takes yet-to-be-transformed lambda-terms (designated by underlining) into new lambda-terms whose evaluation will not involve application to functions returned from other functions. The transform is given by the following three rewrite rules:

T1: $\underline{V} \Rightarrow V$
where ‘V’ is any variable

T2: $\underline{A\ B} \Rightarrow (\lambda y,o.\ \underline{A}\ \underline{B}\ (\lambda j.\ j\ y\ o))$
where ‘o’ and ‘y’ are fresh variables

T3: $\underline{(\lambda V.M)} \Rightarrow (\lambda V,t.\ t\ \underline{M})$
where ‘t’ is a fresh variable

To understand the transform, it is useful to first rewrite it in terms of tuples: we recognise Church Tuples in the above, note that ‘<a,b> M’ equals ‘M a b’, and re-express the transformation as:

$\underline{V} \Rightarrow V$

$\underline{A\ B} \Rightarrow (\lambda y,o.\ \underline{A}\ \underline{B}\ \langle y, o\rangle)$

$\underline{(\lambda V.M)} \Rightarrow (\lambda V,\langle y,o\rangle.\ \underline{M}\ y\ o)$

One can note that terms such as ‘$\underline{(F\ X)\ Y}$’ etc will contain nested tuples in the above formulation. This suggests a list-based reformulation:

$\underline{V} \Rightarrow V$

$\underline{A\ B} \Rightarrow (\lambda y,o.\ \underline{A}\ \underline{B}\ (cons\ y\ o))$

$\underline{(\lambda V.M)} \Rightarrow (\lambda V,t.\ \underline{M}\ (head\ t)\ (tail\ t))$

As an explanation of the transform: rather than functions taking an unboundedly-large number of parameters, ie 'M N_1 N_2 N_3..', which will eventually exceed the function's arity and result in an application to its result; instead terms take only two parameters, ie 'M N_1 [N_2, N_3..]'. And, rather than a lambda-abstraction returning a result which is then applied, instead its nominal return result is applied *inside* the lambda-abstraction, and functions 'never' return. Consider for example (we use the list reformulation):

(λx.x) A B

⇒ (T2)

(λy,o. (λx.x) A B (cons y o))

⇒ (T2)

(λy,o. (λz, p. (λx.x) A (cons z p)) B (cons y o))

→*

(λy,o. (λx.x) A (cons B (cons y o)))

Now, we transform '(λx.x)':

(λx.x)

⇒ (T3, T1)

(λx,p. x (head p) (tail p))

Note that this takes two arguments, which in this case will be 'A' and 'cons B (cons y r)'. The execution proceeds:

(λy,o. (λx,p. x (head p) (tail p)) A (cons B (cons y o)))

→* (λy,o. A (head (cons B (cons y o))) (tail (cons B (cons y o))))

→* (λy,o. A B (cons y o))

etc

Hence, similarly to the Continuation-Passing Style, when the result of transforming '(λx.x)' is executed, the execution will involve all the rest of the program. There is no application of the value returned from the function '(λx.x)' transformed to, as if a value were returned, it would be the value of the entire program.

To prove that function-results are indeed never applied, we proceed as follows. We introduce three types: 'Σ' for the type of transformed terms, 'Φ' for the type of (Church) tuples, and 'Ω' for the return-value of a 'Σ' term. Specifically:

$\Sigma \triangleq \Sigma \multimap \Phi \multimap \Omega$

$\Phi \triangleq \Sigma \multimap \Omega$

The typed transform is as follows:

$\underline{V} \Rightarrow V{:}\Sigma$

$\underline{A\ B} \Rightarrow (\lambda y{:}\Sigma, o{:}\Phi.\ (\underline{A}{:}\Sigma\ \underline{B}{:}\Sigma\ (\lambda n{:}\Sigma.\ n\ y\ o)){:}\Omega\ ){:}\Sigma$

$\underline{(\lambda V.M)} \Rightarrow (\lambda V{:}\Sigma, t{:}\Phi.\ (t\ \underline{M}{:}\Sigma){:}\Omega){:}\Sigma$

The transformed '$\underline{A\ B}$' is type-correct, as '$(\lambda n.\ n\ y\ o)$' is verifiably of type 'Φ' ie '$\Sigma \multimap \Omega$', given that 'Σ' is '$\Sigma \multimap \Phi \multimap \Omega$'; and since '$\underline{A}$: Σ' is the same as '$\underline{A}{:}\Sigma \multimap \Phi \multimap \Omega$'. The transformed '$\underline{(\lambda V.\ M)}$' is also clearly type-correct, as 't: Φ' is the same as 't:$\Sigma \multimap \Omega$'. Finally, it can be seen that the result of transforming any term 'X' is a function of type 'Σ'. Note that functions of type 'Σ' take two parameters, ie all transformed terms take two parameters. That this arity is not exceeded is easily seen by the fact that 'Ω' is left undefined. As 'Ω' is the type of the return-value of a transformed term, and 'Ω' is undefined (ie needs not be a function-type) then the return-value of transformed-terms is provably never applied.

As evidence for the semantic correctness of the transform, we prove now that if two terms are related via a beta-reduction, then after transformation they are still equal, ie:

$\forall A,B.\ A \rightarrow B\ \Rightarrow\ \underline{A} = \underline{B}$

Now, note that it must be true that:

$A = ..((\lambda x.M)\ N)..$

$B = ..M[x\backslash N]..$

hence it is necessary to show only that:

$\underline{..\ (\lambda x.M)\ N\ ..} = \underline{..\ R\ ..}$

where:

$R = M[x\backslash N]$

To formalise and prove this, we require more-powerful notation. To this end, we re-present the transform as a function, named 'Transform', from a data-structure representing untransformed terms to lambda-terms. The untransformed terms are constructed using 'app' and 'lam' and 'var' data constructors to represent application, abstraction, and variables (whose names are given as strings), as follows:

Untransformedterm ::= Variable | Application | Abstraction

where

Variable ::= var "a" | var "b" | ..

Application ::= app Untransformedterm Untransformedterm

Abstraction ::= lam Variable Untransformedterm

The ‘Transform’ function has the following semantics (‘Λ’ is the set of lambda-terms):

Transform : Untransformedterm ─> Λ

T1:

Transform (var “a”) = a

Transform (var “b”) = b

..

T2: Transform (app A B) = (λy,o. (Transform A) (Transform B) (λn. n y o))

where ‘y’ and ‘o’ are fresh variables.

T3: Transform (lam V M) = (λ(Transform V),t. t (Transform M))

where ‘t’ is a fresh variable.

We introduce a function which implements capture-avoiding substitution (similar to ‘[\ ]’), ‘Sub’, which has the standard semantics:

Sub:: Untransformedterm ─> Variable ─> Untransformedterm ─> Untransformedterm

Sub (var W) V N = if (var W) == V then N else (var W)

Sub (app A B) V N = app (Sub A V N) (Sub B V N)

When ‘W’ matches no free variable of ‘N’:

Sub (lam W M) V N = if W == V then (lam W M) else (lam W (Sub M V N))

When ‘W’ matches a free variable of ‘N’:

Sub (lam W M) V N = Sub (lam X (Sub M W X)) V N

where ‘X’ is fresh.

Our requirement can now be rewritten as:

∀C ∈ ζ. Transform (C[app (lam V M) N]) = Transform (C[Sub M V N])

where:

ζ ::= □ | lam Variable ζ | app Untransformedterm ζ | app ζ Untransformedterm

Here, ‘C’ is what is known as a ‘context’, in this case an arbitrary (finite) data-structure containing a ‘hole’, denoted by ‘□’. The structure ‘C’ with its hole filled by some ‘d’ is denoted by ‘C[d]’.

We first prove a lemma, that:

Transform (app (lam (var “x”) M) N) = Transform (Sub M (var “x”) N)

proceeding:

Transform (app (lam (var “x”) M) N)

= (semantics of ‘Transform’)

(λy,o. (Transform (lam (var “x”) M)) (Transform N) (λj. j y o))

= (semantics of ‘Transform’)

(λy,o. (λx,t. t (Transform M)) (Transform N) (λj. j y o))

= (reductions)

(λy,o. (λx. (Transform M) y o) (Transform N))

= (‘y’ and ‘o’ being fresh)

(λy,o. (Transform M)[x\(Transform N)] y o)

= (eta; as ‘y’ and ‘o’ are fresh, ‘Transform’ won’t return a term containing them free)

(Transform M)[x\(Transform N)]

NB: ‘[\ ]’ substitutions happen on lambda-terms, ie on the *result* of ‘Transform M’.

We now show by induction the remainder, that:

(Transform M)[x\(Transform N)] = Transform (Sub M (var “x”) N)

Base-case: ‘M’ is a Variable, ‘var “x”’

(Transform (var “x”))[x\(Transform N)]

= (semantics of ‘Transform’)

x[x\(Transform N)]

= (property of substitution)

Transform N

= (property of ‘Sub’)

Transform (Sub (var “x”) (var “x”) N)

as required.

Second base-case: ‘M’ is a Variable, but not ‘var “x”’:

(Transform (var “y”))[x\(Transform N)]

= (semantics of ‘Transform’)

y[x\(Transform N)]

= (property of substitution)

y

= (property of ‘Sub’)

Transform (Sub (var “y”) (var “x”) N)

as required.

Inductive case: ‘M’ is a representation of a lambda-abstraction, not of ‘x’:

(Transform (lam (var “y”) F)[x\(Transform N)]

= (semantics of ‘Transform’)

(λy,t. t (Transform F)) [x\(Transform N)]

= (property of substitution)

(λy,t. t (Transform F)[x\(Transform N)])

= (inductive step)

(λy,t. t (Transform (Sub F (var “x”) N)))

= (semantics of ‘Transform’)

Transform (lam (var “y”) (Sub F (var “x”) N))

= (‘y’ fresh)

Transform (Sub (lam (var “y”) F) (var “x”) N)

= (definition of ‘M’)

Transform (Sub M (var “x”) N)

as required.

Second inductive case: ‘M’ is a representation of a lambda-abstraction of ‘x’:

(Transform (lam (var “x”) F))[x\(Transform N)]

= (semantics of ‘Transform’)

(λx,t. t (Transform F) [x\(Transform N)]

= (no free ‘x’)

(λx,t. t (Transform F)

= (definition of ‘Transform’)

Transform (lam (var “x”) F)

= (no free ‘x’)

Transform (Sub (lam (var “x”) F) (var “x”) N)

= (definition of ‘M’)

Transform (Sub M (var “x”) N)

as required.

Third inductive case: ‘M’ is a representation of an application

(Transform (app A B))[x\(Transform N)]

= (semantics of ‘Transform’)

(λy,o. (Transform A) (Transform B) (λj. j y o)) [x\(Transform N)]

= (freshness of ‘y’ and ‘o’)

(λy,o. (Transform A)[x\(Transform N)] (Transform B)[x\(Transform N)] (λj. j y o))

= (inductive step, twice)

(λy,o. (Transform (Sub A (var “x”) N)) (Transform (Sub B (var “x”) N)) (λj. j y o))

= (freshness of ‘y’ and ‘o’)

(λy,o. (Transform (Sub A (var “x”) N)) (Transform (Sub B (var “x”) N)) (λj. j y o))

= (semantics of ‘Transform’)

Transform (app (Sub A (var “x”) N) (Sub B (var “x”) N))

= (property of ‘Sub’)

Transform (Sub (app A B) (var “x”) N)

as required.

QED, lemma.

With the lemma, we can now show that:

$\forall C \in \zeta$. Transform (C[app (lam V M) N]) = Transform (C[Sub M V N])

by induction on the structure of the context, ie on the four cases of:

$\zeta ::= \square$ | lam Variable ζ | app Untransformedterm ζ | app ζ Untransformedterm

Case 1, the base-case:

Transform (C[app (lam V M) N])

= (case 1, ‘C = □’)

Transform (app (lam V M) N)

= (lemma)

Transform (Sub M V N)

= (case 1)

Transform (C[Sub M V N])

as required.

Case 2, an inductive case:

Transform (C[app (lam V M) N])

= (case 2, ‘C = lam W $C^{\dagger}$’, for some ‘W’ and sub-context ‘$C^{\dagger}$’)

Transform (lam (W $C^{\dagger}$[app (lam V M) N]))

= (semantics of ‘Transform’, abusing notation: ‘w’ is such that ‘var “w” = W’)

(λw,t. t (Transform $C^{\dagger}$[app (lam V M) N])

= (inductive step)

(λw,t. t (Transform $C^{\dagger}$[Sub M V N])

= (semantics of ‘Transform’)

Transform (lam (W $C^{\dagger}$[Sub M V N]))

= (case 2)

Transform (C[Sub M V N])

as required.

Case 3, an inductive case:

Transform (C[app (lam V M) N])

= (case 3, ‘C = app O $C^{\dagger}$’, for some ‘O’ and sub-context ‘$C^{\dagger}$’)

Transform (app O $C^{\dagger}$[app (lam V M) N])

= (semantics of ‘Transform’)

(λy,o. (Transform O) (Transform ($C^{\dagger}$[app (lam V M) N])) (λj. j y o))

= (inductive step)

(λy,o. (Transform O) (Transform ($C^{\dagger}$[Sub M V N])) (λj. j y o))

= (semantics of ‘Transform’)

Transform (app O $C^{\dagger}$[Sub M V N])

= (case 3)

Transform (C[Sub M V N])

as required.

Case 4, an inductive case:

Transform (C[app (lam V M) N])

= (case 4, 'C = app $C^{\dagger}$ O', for some 'O' and sub-context '$C^{\dagger}$')

Transform (app $C^{\dagger}$[app (lam V M) N] O)

= (semantics of 'Transform')

(λy,o. (Transform ($C^{\dagger}$[app (lam V M) N])) (Transform O) (λj. j y o))

= (inductive step)

(λy,o. (Transform ($C^{\dagger}$[Sub M V N])) (Transform O) (λj. j y o))

= (semantics of 'Transform')

Transform (app $C^{\dagger}$[Sub M V N] O)

= (case 4)

Transform (C[Sub M V N])

as required.

QED.

7.4 Investigated hypotheses regarding the requirements for interpretation and testing

A number of hypotheses regarding what is required for one to be able to interpret or test are considered

We now consider various hypotheses regarding what may be required to test, and also to interpret. While the informal characterisation of interpretation is testing, in the below we don't assume the two concepts are synonymous, or for that matter distinct.

7.4.1 To be able to implement many tests on Naturals, tuples are required

While commonly used, tuples are not essential for implementing most tests on Naturals

An observation which can be made about implementations of tests on Church Naturals is that they often seem to involve internal use of 'container types', such as Church Tuples. Indeed, the implementations of most tests (equality, greater-than, etc) that we have seen have all made use

of ‘container types’. Only for a few tests have we seen ‘container type’-free implementations; such as ‘iszero’. However, we show below that many tests on recursive types can actually be implemented without use of ‘container types’.

The underlying reason for the apparent need for ‘container types’ is that the tests in question do not always have Structural Recursive but instead Primitive Recursive mappings. As discussed in Chapter 5, one can implement Primitive Recursion using Structural Recursion and tuples. Commonly, a non- Structural Recursive test on Naturals is defined in terms of a sole Primitive Recursive operation, typically ‘predecessor’, and it is in the implementation of that operation only that ‘container types’ appear. For example:

greaterthan a b = not (iszero (sub a b))

equals a b = and (iszero (sub a b)) (iszero (sub b a))

where ‘sub x y = y pred x’, ‘iszero’ and ‘and’ have standard definitions, and:

pred n ≜ (λ<a, b>. a) (n (λ<a, b>. <b, succ b>) <zero, zero>)

or, via an application of the ‘BVE’ transformation:

pred’ n s z ≜ (λ<a, b>. a) (n (λ<a, b>. <b, s b>) <z, z>)

Now, recall from Chapter 5 the standard scheme for doing Primitive Recursion using Structural Recursion, which forms the basis of the definition of ‘pred’, above:

$F\ a_1..a_n\ x \triangleq (\lambda<r, y>.\ r)\ (J\ a_1..a_n\ x)$

where:

$J\ a_1..a_n\ (succ\ x) \triangleq j_1\ a_1..a_n\ (J\ a_1..a_n\ x)$

$J\ a_1..a_n\ zero \triangleq j_2\ a_1..a_n$

and:

$j_1\ a_1..a_n <r, x> \triangleq <h_1\ a_1..a_n\ r\ x, succ\ x>$

$j_2\ a_1..a_n \triangleq <h_2\ a_1..a_n, zero>$

then ‘J’ is Structural Recursive (by definition), and ‘F’ satisfies the Primitive Recursion equations:

$F\ a_1..a_n\ zero = h_2\ a_1..a_n$

$F\ a_1..a_n\ (succ\ x) = h_1\ a_1..a_n\ (F\ a_1..a_n\ x)\ x$

To produce the standard definition of 'pred', one simply sets:

$h_1\ a_1..a_n = (\lambda r,x.\ x)$

$h_2\ a_1..a_n = \text{zero}$

We have discovered a differing scheme for doing Primitive Recursion using Structural Recursion, which doesn't involve tuples. The alternative definition-bodies are:

$F\ a_1..a_n\ x \triangleq (G\ a_1..a_n\ x)\ (h_2\ a_1..a_n)\ \text{zero}$

where:

$G\ a_1..a_n\ (\text{succ}\ x) \triangleq g_1\ a_1..a_n\ (G\ a_1..a_n\ x)$

$G\ a_1..a_n\ \text{zero} \triangleq g_2\ a_1..a_n$

and:

$g_1\ a_1..a_n\ k\ \triangleq\ (\lambda r,y.\ k\ (h_1\ a_1..a_n\ r\ y)\ (\text{succ}\ y))$

$g_2\ a_1..a_n \triangleq (\lambda r,y.\ r)$

Here, our new 'G' is also a Structural Recursion; and we prove that the new definition for 'F' still satisfies the two requirements for being a Primitive Recursion:

$F\ a_1..a_n\ \text{zero} = h_2\ a_1..a_n$

$F\ a_1..a_n\ (\text{succ}\ x) = h_1\ a_1..a_n\ (F\ a_1..a_n\ x)\ x$

as follows.

For the first requirement:

$F\ a_1..a_n\ \text{zero}$

(evaluating 'F')

$= (G\ a_1..a_n\ \text{zero})\ (h_2\ a_1..a_n)\ \text{zero}$

(evaluating 'G')

$= g_2\ a_1..a_n\ (h_2\ a_1..a_n)\ \text{zero}$

(evaluating 'g_2')

$= (\lambda r,y.\ r)\ (h_2\ a_1..a_n)\ \text{zero}$

(simplifying)

$= h_2\ a_1..a_n$

as required.

For the second requirement, we begin by proving the following lemma, where '+' denotes infix addition:

$\forall z$:Nat. G $a_1..a_n$ (succ z) r y = h_1 $a_1..a_n$ (G $a_1..a_n$ z r y) y+z

The proof is by induction.

Base-case ('z' is 'zero'):

G $a_1..a_n$ (succ zero) r y

(evaluating 'G')

= g_1 $a_1..a_n$ (G $a_1..a_n$ zero) r y

(evaluating 'G')

= g_1 $a_1..a_n$ (g_2 $a_1..a_n$) r y

(evaluating 'g2')

= g_1 $a_1..a_n$ (λr,y. r) r y

(evaluating 'g1')

= (λk,r,y. k (h_1 $a_1..a_n$ r y) (succ y)) (λr,y. r) r y

(simplifying)

= (λr,y. r) (h_1 $a_1..a_n$ r y) (succ y)

(simplifying)

= h_1 $a_1..a_n$ r y

(rewriting so as to be in terms of the form of the definition-body for 'g2')

= h_1 $a_1..a_n$ ((λr,y. r) r y) y

(recognising 'g2')

= h_1 $a_1..a_n$ (g2 $a_1..a_n$ r y) y

(recognising 'G', the 'zero' case)

= h_1 $a_1..a_n$ (G $a_1..a_n$ zero r y) y+zero

as required.

Step-case ('z' is 'succ x', for some 'x'):

G $a_1..a_n$ (succ (succ x)) r y

(evaluating 'G')

= g_1 $a_1..a_n$ (G $a_1..a_n$ (succ x)) r y

(evaluating 'g1')

= (λk,r,y. k (h_1 $a_1..a_n$ r y)) (succ y)) (G $a_1..a_n$ (succ x)) r y

(simplifying)

= (λr,y. G $a_1..a_n$ (succ x) (h_1 $a_1..a_n$ r y)) (succ y)) r y

(simplifying)

= G $a_1..a_n$ (succ x) (h_1 $a_1..a_n$ r y) (succ y)

inductive step:

= h_1 $a_1..a_n$ (G $a_1..a_n$ x (h_1 $a_1..a_n$ r y) (succ y)) (succ y)+x

(rearranging)

= h_1 $a_1..a_n$ ((G $a_1..a_n$ x) (h_1 $a_1..a_n$ r y) (succ y)) y+(succ x)

(rewriting to give in terms of the definition-body for 'g_1')

= h_1 $a_1..a_n$ ((λk,r,y. k (h_1 $a_1..a_n$ r y) (succ y)) (G $a_1..a_n$ x) r y) y+(succ x)

(recognising 'g_1')

= h_1 $a_1..a_n$ (g_1 $a_1..a_n$ (G $a_1..a_n$ x) r y) y+(succ x)

(recognising 'G', the 'succ' case)

= h_1 $a_1..a_n$ (G $a_1..a_n$ (succ x) r y) y+(succ x)

as required.

We can now proceed with the proof of the second equation:

F $a_1..a_n$ (succ x)

(evaluating 'F')

= (G $a_1..a_n$ (succ x)) (h_2 $a_1..a_n$) zero

(via the above lemma)

= (λr,y. h_1 $a_1..a_n$ (G $a_1..a_n$ x r y) y+x) (h_2 $a_1..a_n$) zero

(simplifying)

= h_1 $a_1..a_n$ (G $a_1..a_n$ x (h_2 $a_1..a_n$) zero) zero+x)

(recognising 'F')

= h_1 $a_1..a_n$ (F $a_1..a_n$ x) zero+x

(simplifying)

= h_1 $a_1..a_n$ (F $a_1..a_n$ x) x

QED.

The corresponding new definition of 'pred' with this scheme is:

$pred_2$ n ≙ n (λk,r,y. k (λr,y. y) r y (succ y)) (λr,y. r) zero zero

ie

$pred_2$ n ≙ n (λk,r,y. k y (succ y)) (λr,y. r) zero zero

We have not seen this definition elsewhere in the literature. Re-arrangement and renaming gives the syntactic form we most-often use:

$pred_2\ n \triangleq n\ (\lambda r,a,b.\ r\ (succ\ a)\ a)\ (\lambda a,b.\ b)\ zero\ zero$

It can also be further rewritten (via the transformation techniques of Chapter 5) to:

$pred_2'\ n\ s\ z \triangleq n\ (\lambda r,a,b.\ r\ (s\ a)\ a)\ (\lambda a,b.\ b)\ z\ z$

Similarly, the below two versions of 'iseven' can be seen to be related in the same manner as 'pred' is to '$pred_2$', but with the tuple-less version being the canonical definition:

$iseven\ n\ x\ y = (\lambda<a,b>.\ a)\ (n\ (\lambda<a,b>.\ <b,a>)\ <x,y>)$

versus

$iseven_2\ n\ x\ y = n\ (\lambda r,a,b.\ r\ b\ a)\ (\lambda a,b.\ a)\ x\ y = n\ not\ true\ x\ y$

ie

$iseven_2\ n = n\ not\ true$

It is interesting to consider the operation of '$pred_2$', and how it differs from 'pred'. The core of '$pred_2$' can be seen to be written in the Continuation-Passing Style; and in some sense is 'pred' turned 'inside-out': computation occurs top-down, rather than usual bottom-up. As a result of this, one can see that what used to be members of tuples are now members of argument-lists. This leads us in the direction of believing that tuples of known-cardinality are merely a convenience in programming which can always be done without. This would imply that tuples aren't necessary to write interpreters.

To finish our discussion, we note that an even simpler version of predecessor is possible, especially in the Continuation-Passing Style (CPS). The simplest we have constructed is:

$pred\ n\ f\ x\ k = n\ (\lambda a,g.\ g\ (a\ f))\ (\lambda g.\ x)\ k$

ie: (eta-contract)

$pred\ n\ f\ x = n\ (\lambda a,g.\ g\ (a\ f))\ (\lambda g.\ x)$

The non-CPS equivalent is of course:

$pred\ n\ f\ x = n\ (\lambda a,g.\ g\ (a\ f))\ (\lambda g.\ x)\ (\lambda k.\ k)$

This was derived via a different representation for Naturals, specifically:

$gzero \triangleq (\lambda g,f,x.\ x)$

$gsucc\ n \triangleq (\lambda g,f,x.\ g\ (n\ f\ f\ x))$

This representation of Naturals is the same as the Church Naturals, but with the first 'f' distinguished, ie replaced by 'g'. For example:

gsucc (gsucc (gsucc gzero)) = (λg,f,x. g (f (f x)))

Conversion from Church Naturals is readily achieved; from Chapter 5 we know that application of a Data-Alternative's Constructors to a Church Object (of the same data-type) gives the corresponding Element of the Data-Alternative. Hence define:

fromChurchNat n ≜ n gsucc gzero

An inverse to this particular conversion exists, and can be denoted simply by:

toChurchNat n f x ≜ n f f x

One can easily combine this second conversion with the predecessor operation, ie map to the Church Natural which is the corresponding predecessor, by simply binding 'g' to the identity function:

toPredChurchNat n f x ≜ n (λk.k) f x

ie:

toPredChurchNat n ≜ n (λk.k)

Hence predecessor on the Church Naturals can be defined as:

pred n ≜ toPredChurchNat (fromChurchNat n)

ie:

pred n f x ≜ n (λa,g,f,x. g (a f f x)) (λg,f,x. x) (λk. k)

One can now apply the 'BVE' transform introduced in Chapter 5 to reach:

pred n f x ≜ n (λa,g. g (a f)) (λg. x) (λk. k)

Subsequently to constructing this definition, we discovered we were not the first to find it: we have elsewhere seen it (without an associated derivation or citation) credited to one J. Velmans.

7.4.2 'Throwing away' arguments is required to permit 'if' tests

One can implement 'if' tests without needing to discard one or the other branch

Interpreters tend to involve tests which act as if they selectively discard one argument and selectively execute another. The canonical such test is the 'if..then..else' test. However, selective discarding of an argument is merely a common technique, not a necessity, of

implementing these tests. That this is true can be seen to follow from there being a variant of the lambda calculus, called the λ-I calculus (see eg [Bar84, p37]), in which no value ever gets discarded yet is Turing-complete ([Bar84, p192]). We now discuss how this is possible, how one can get the *effect* of discarding, ie 'ignoring', without *actually* discarding.

Consider when a function takes a known argument, which is to be ignored without being discarded. Terms can be effectively ignored by passing them arguments that result in the identity function ('id') being returned: the term is still applied, and computations within it done, yet the end-result is known in advance to the identity function, which can then be reduced away. For example, consider the following function:

$$H \triangleq (\lambda f,g,y.\ f\ (g\ y))$$

Say this function will only be applied to a certain function as the 'f' parameter, which we want to ignore: call this function 'F'. Then, if for this function 'F' one can find an input 'X' such that 'F X = id', then we can rewrite 'H' as:

$$H \triangleq (\lambda f,g,y.\ (f\ X)\ (g\ y))$$

On execution, one has:

$$(\lambda f,g,y.\ (f\ X)\ (g\ y))\ F$$
$$\rightarrow (\lambda g,y.\ (F\ X)\ (g\ y))$$
$$\rightarrow (\lambda g,y.\ id\ (g\ y))$$
$$\rightarrow (\lambda g,y.\ g\ y)$$

Hence if the argument to a function is known, it is at least sometimes possible to pass it arguments that cause it to 'self-destruct', so to speak. This technique was used in [Kle35b] to, for example, select only one element from a Church Tuple of Church Naturals by applying the other Church Naturals to 'id'.

For precisely when a function's argument of known value can be made to 'self-destruct', we turn to the literature. In [Bar84, pp41-42], the concept of 'solvability' of a function is introduced. If a function is 'solvable' then arguments exist that cause the function to return the identity function. It is noted that a term is solvable if and only if it has a Head-Normal Form (ie isn't '⊥'), including in the λ-I -calculus. Hence any argument whose value is known, and whose value isn't '⊥', can be made to 'self-destruct'.

We now consider ignoring an argument when its value isn't known, but is known to have been drawn from some set. As reported in [Bar84, p261], if the set is finite, and contains functions with Normal Forms, then it can be 'separated' using λ-I -calculus terms. Here, 'separated' means that an 'if' test can be written which branches on each of the elements of the set. By simply requiring that each of the branches be 'id' (which note is a λ-I -calculus term), we derive that if the set of values the argument is drawn from is finite and contains only terms with Normal Forms, then the argument can be 'self-destructed'. Sets with terms without Normal Forms have to be dealt with on a case-by-case basis.

The next circumstance is when an argument is drawn from an infinite set. In many cases one can indeed 'self-destruct' it; however we are not aware however of any general results. For example, following [Kle35b] one can apply an argument which is a Church Natural to 'id' and 'id', to produce 'id'. Similar results hold for all other Church Objects.

Finally, consider the case of primary concern, when a function takes an argument and depending on other arguments either ignores it, or not. In this case, we only want to sometimes 'self-destruct' the argument. An example is execution of an 'if' test: one branch is to be ignored, but which branch depends on the input. In [Bar84, p191] there is a concrete illustration, credited to Kleene, of two ways by which this can be done. They both involve the branches being supplied as simple parts and, depending on an input-value, the computation either reconstructing the branches from those parts, or making the parts 'self-destruct'.

7.4.3 Interpretation requires recursion or self-application

As one can define a fixpoint function that doesn't use self-application, at least some interpreters can be written without self-application and recursion

There are well-known means ([Bar84]) by which any program that uses recursion (or loops) can be rewritten to use fixpoint functions instead. The fixpoint functions that we have seen are self-applicative. However, we now show that they need not be. Consequently, interpreters can be written without recursion, loops, or self-application. One can also recall from Chapter 5 that one can use (non- self-applicative) terms involving Church Naturals to *approximate* fixpoint functions for all practical purposes.

We begin by showing that one can have an infinite-reduction sequence without self-application, before building on those results to define a fixpoint function that does not use self-application.

Firstly, we desire terms:

A B ..

which after some (non-zero in number) reductions reduce to the same expression, ie

A B ..

At no stage in any possible reduction-sequence should there be self-application. By this we mean terms of the form

F .. F ..

for some 'F'; ie an operator one of whose arguments (or arguments to its return-value, or a return-value from a return-value, etc) is the same ('↔') lambda-term as itself. This is perhaps a somewhat stricter condition than may be required in practice, as usually the chosen evaluation strategy will result in only a single reduction-sequence occurring at runtime.

We exhibit the following solution:

(λa. a (λk.k a)) (λk. k (λa. a (λk.k a)))

Using:

A ≜ (λa. a (λk.k a))

we can re-express the above as:

A (λk. k A)

That this has the property we require can be seen from the following reduction-sequence:

A (λk. k A)

=

(λa. a (λk.k a)) (λk. k A)

→

(λk. k A) (λk. k (λk. k A))

→

(λk. k (λk. k A)) A

→

A (λk. k A)

This reduction-sequence does not involve any self-application, and a quick check reveals it as in fact the only reduction-sequence possible. Hence we have found suitable terms for which no reduction-sequence involves self-application.

An alternate representation exists, exposing the terms as being tuple-oriented. Recognising Church Tuples in the above, ie '(λk. k X)' as '<X>', and noting that '<X> F' reduces to 'F X', we can rewrite the above definition of 'A' as:

$$A \triangleq (\lambda a.\ a\ \langle a \rangle)$$

With this new definition, one has that:

$$A\ \langle A \rangle$$
$$=$$
$$(\lambda a.\ a\ \langle a \rangle)\ \langle A \rangle$$
$$\rightarrow$$
$$\langle A \rangle\ \langle\langle A \rangle\rangle$$
$$\rightarrow$$
$$\langle\langle A \rangle\rangle\ A$$
$$\rightarrow$$
$$A\ \langle A \rangle$$

Based on these definitions, we can generate a fixpoint function based on the canonical 'Y' function, call it 'myY'. The well-known property required of this combinator is that, for any 'F':

$$myY\ F = F\ (myY\ F)$$

For reference, the 'Y' function is defined as:

$$Y \triangleq (\lambda a,f.\ f\ (a\ a\ f))\ (\lambda a,f.\ f\ (a\ a\ f))$$

Using this as a template, we define:

$$myY \triangleq (\lambda a,f.\ f\ (a\ (\lambda k.k\ a)\ f))\ (\lambda k.\ k\ (\lambda a,f.\ f\ (a\ (\lambda k.k\ a)\ f)))$$

To show its correctness, we first let:

$$B = (\lambda a,f.\ f\ (a\ (\lambda k.k\ a)\ f))$$

recall:

$$\langle B \rangle = (\lambda k.\ k\ B)$$

then:

$$myY\ f$$

=

B <B> f

=

(λa,f. f (a (λk.k a) f)) <B> f

→

f (<B> (λk.k <B>) f)

→

f ((λk.k <B>) B f)

→

f (B <B> f)

=

f (myY f)

which is as required. Note that 'f' is never provided as an argument to 'f'. Hence, a non- self-applicative fixpoint function does indeed exist, which programs can use instead of recursion.

7.4.4 Recursions or loops, however implemented, are required by 'universal interpreters'

'Universal interpreters' do not all require recursions or loops

While plausible, it does not seem to be the case that recursions or loops, however implemented (eg by fixpoint functions or otherwise), are required by so-called 'universal interpreters'. We characterise the universal interpreters broadly, as the computational systems whose range is all the Turing-computable functions, and whose domain is (possibly, a representation of) data. The following function 'U' is a universal interpreter, and it lacks recursion, self-application, or anything similar:

U ≜ (λf. f S K (λy,z. y z))

Here, 'S' and 'K' are the classical combinators (see eg [Bar84]) with the usual semantics: '(λf,g,x. (f x) (g x))' and '(λf,x. f)' respectively. The input to 'U' is an arbitrary SK combinatory term which has had 'S', 'K', and the application meta-operator parameterised, ie a term of the form:

(λs,k,a. M)

where 'M' consists only of the variables 's', 'k', 'a', and brackets. For example:

S (S K) K

yields:

(λs,k,a. a (a s (a s k)) k)

That ‘U’ is a universal interpreter is a consequence of the well-known fact that the SK combinatory calculus is Turing-complete; as long as its inputs are a representation of data. This last can be argued to be true from the existence of an injective function from those inputs to some data-type, such as:

ToString t ≙ t “S” “K” (λa,b. “(” ++ a ++ “ ” ++ b ++ “)”)

Above, ‘++’ denotes infix string concatenation. That ‘ToString’ is indeed injective should be obvious: the ‘s’ variables in the input to ‘U’ will be bound to “S”, the ‘k’ variables to “K”, and the ‘a’ variables to a function which places its arguments in brackets with a space between them. A more compact mapping to strings of binary digits can be found in [Tro02] (a universal interpreter on a Data-Alternative is also defined with it, but uses explicit tests and recursion). It is:

ToString t ≙ t “00” “01” (λa,b. “1” ++ a ++ b)

Reference to recursion-free implementations of universal interpreters using lambda (rather than SK) terms can be found in [Bar97].

This example appears to indicate that loops (and, explicit tests) are not required for writing a universal interpreter. Alternatively, one could take this example as indicating that not all ‘universal interpreters’ are actually that interpretive. Perhaps functions that turn data (or representations thereof) into the full range of behaviours are in fact not interpretive if those behaviours are ‘sufficiently-close’ in some sense to the behaviours or structures of the corresponding inputs. As an alternative explanation, perhaps it is not the ‘universal interpreters’ that do interpretation, but their inputs. After all, the ‘interpreter’ simply gives the input arguments, and the input then does all the computational work.

7.4.5 Something lacking in quantum computation is required for tests

Quantum computation does not prohibit tests

Quantum computers, as currently envisioned (see eg [Ber97]), involve setting up a physical system so it has some initial state and then applying a sequence of 'unitary' (implying reversible) operations to that system. In order to prevent collapse of the wave-function, which would result in the end of quantum computation, one cannot engage in a measurement (ie a test) of the system, in order to, for example, determine what operation to apply next. Hence perhaps there is something lacking in quantum computers, something that is required for tests and perhaps interpretation generally. However, this inability to do tests is limited only to classical tests on the system: it is quite possible to do so-called 'conditional operations' on the system without collapsing the wave-function, ie within the quantum world as it were. A pair of 'conditional operations' can be used to implement an 'if..then..' test ([Öme05]). Hence tests can indeed be done in quantum computation.

7.4.6 Data is required for interpretation but not for programming generally

Data is required for essentially all programming

In TFP is the suggestion that data is essentially only used by interpreters, and that it might be possible to prevent interpreters (only) from being written by removing data from programming languages. In this section by 'data' we mean both primitive data and functions representing data.

The analysis presented below indicates that terms that can act as data are used throughout programming, not just in explicit interpreters, and hence are impractical to avoid or remove. We also discuss whether one can keep terms which can act as data but prevent their *use* as data, by not permitting tests.

7.4.6.1 The nature of data

Any term can act as data as long as certain operations can be implemented

To the programmer, primitive data and primitive operations thereon form an 'abstract data type' (described in eg [Rey83]). In an 'abstract data type' the data is completely undefined; it can't be investigated by any means but by being passed to one of the operations. The operations are also opaque, but satisfy certain (eg axiomatic) semantics.

What therefore can act as data? One is drawn to the conclusion that it is any set on which the operations associated with data can be implemented. We term such a set 'dataful'. As for what are the exact operations, we follow the lead of languages with primitive data. Primitive data, being general-purpose, typically is provided with only basic operations; eg constructors, destructors, and simple tests: most-commonly a pattern-matcher with a branch for each constructor. The simplest data-type is a 'flat' (ie non-recursive) data-type; these have no destructors and only one canonical test, a constructor pattern-matcher. Hence any set which can have that pattern-matcher implemented for it can be said to be able to act as data.

7.4.6.2 Removing terms that can act as data

Programming without terms that can act as data is impractical

We can now consider whether terms that can act as data can be removed from programming languages. Firstly, primitive data can be readily removed from programming languages. We then have to remove anything else that can act as data. Particularly, we must remove all first-class functions that can form 'dataful' sets. Consider now only finite 'dataful' sets, whose elements act as 'zero-arity' constructors of some non-recursive data-type. As per above, for some set to be able to act as data, a constructor pattern-matcher needs to be implementable on it. As the constructors are 'zero-arity' they correspond to the elements of the sets. Hence, a constructor pattern-matcher for one of those sets can be simply an 'if' test that branches on each element of the set. As shown in [BDPR79], any finite set of lambda-terms with (distinct) Normal Forms can have an 'if' test written for them (ie the set is 'separable' or 'discriminable'). Therefore, any finite set of functions with Normal Forms is 'dataful'. This means that to ban data, one would have to at least ban the use of all (but one, at most) functions with a Normal

Form. Otherwise two of them could be used to represent eg ‘true’ and ‘false’. A very large number of functions used in lambda calculus programs have a Normal Form. Hence banning all that is ‘dataful’ would seem to be completely impractical, at least for the lambda calculus, as there would be nothing left to program with. As for other programming languages, note that the ability to define ‘if’ tests is also going to be impractical to avoid.

One might suggest that there could exist some transform that when applied to a lambda calculus program gives an equivalent program in which no function used has a Normal Form. However, such a transform would also transform any implemented ‘if’ tests, so separable sets would persist.

Below we present a differing approach to preventing ‘if’ tests from being implementable, by using types rather than restricting the allowable values to a set that isn’t separable.

7.4.6.3 Preventing tests

Tests can be prevented using universally-quantified types, but this approach is likely to be impractical

Rather than trying remove functions that can form separable sets from programming, one could consider if it is possible to prevent ‘if’ (and other) tests instead.

If we require our functions to be suitably (‘parametrically’) polymorphic or ‘generic’ then we know certain of their arguments cannot be tested on, without having to look at the implementation. Consider the type:

$$\forall\alpha.\ (\alpha \text{—>} \alpha) \text{—>} \alpha \text{—>} \alpha$$

All a function of this type can do with its second argument is pass it to some other parameter which takes an ‘α’; as provable from the only functions of this type being the Church Naturals (recall from Chapter 5), which do just that. Similarly, functions of this type can’t test their first argument, only apply it to their second (or the result of such an application etc). The ability to reason about functions based on their polymorphic types is well-known, [Wad89] contains examples of theorems derived from such.

The underlying reason for this as we see it is that not all sets of lambda-terms are in practice 'testable'. What we mean by that is that no test of any kind (not just 'separation') can in practice be done on some sets. This is due to the variety of terms they include. As any attempt by a function to test an argument must involve applying it to something, if the argument can be lambda-terms of eg unboundedly-large arity this will lead to the function not having the required type. Alternatively if the function is of known arity (eg 'α —> α') but of unknown input or output type, a similar argument applies.

The general idea here turns out to be not new: [GLT89, p90] among others expresses the sentiment that the 'uniformity' of functions with certain polymorphic types is at odds with them doing case-based computation. The use of abstract types to prevent case-analysis in [WW03] is another example. In addition, note that parametricity for higher-order functions has been formalised; see [Rey83], with its 'abstraction theorem', and also the latter work [PA93].

Preventing tests on all arguments to all functions using universally-quantified types will likely be impractical however. One cannot practically program if one can't assume anything about the semantics of the functions given as inputs. However, perhaps other approaches to preventing testing might be suitable.

7.4.7 Any simulation of a Universal Turing Machine or equivalent interpreter must be a step-wise simulation

An 'enumerate-and-test' strategy can replace step-wise simulation

In a simulation of some Turing-complete computational system, one may think that undecidability would require the computations carried out to parallel those done by the system being simulated. However this is not the case. We give three examples, based around the idea that one can enumerate a set of strings (or tuples), testing each to find one corresponding to the desired 'trace' of the system's execution.

Note that being Turing-complete means that one can engage in what certainly would be considered interpretation. For example execution of a 'Pascal' program on some input can be simulated, if that program's source-code and input are supplied, encoded.

7.4.7.1 Post Correspondence

One can simulate a Turing Machine on a given input by enumerating candidate 'traces' of its execution and looking for the desired one

Post's Correspondence Problem can be simply presented as follows: given a finite set of dominos, can one arrange them so that the same string appears along top and bottom? More formally, given two list of strings of equal length, '$[w_1,..w_k]$' and '$[x_1,..x_k]$', one asks if there is a function 'F' and a positive integer 'n' such that '$w_{F\ 1}$ ++ .. $w_{F\ n}$' equals '$x_{F\ 1}$ ++ .. $x_{F\ n}$', where '++' is string concatenation.

One can compute any computable function via the Post Correspondence Problem via producing an appropriate pair of lists, enumerating in size-order, and equality-testing strings to see if a solution has been found. When it has the result of the function can be extracted. For simulating a Turing Machine one constructs the pair of lists such that the concatenated string 'solution' to the Post Correspondence Problem is a 'trace' (ie state plus 'tape' contents, for each step) of the chosen Turing Machine's execution on the given input. For more details, one can consult [HU79].

How the Turing Machine is simulated is somewhat subtle: the machine isn't step-wise simulated but instead treated as a set of constraints; solutions to which correspond to executions. Rather than directly simulating a machine step-by-step, one instead enumerates-and-tests. One could generate all possible strings, and check to see which one actually corresponds to a valid execution and starts from the chosen input. The *optimisations* which can be seen to be employed when using Post Correspondence are:

1. Only a relatively small set of strings (and via simple rules, at that) are generated, so fewer have to be checked

2. A second related string is generated concurrently, such that checking whether the first string is a valid execution 'trace' from the desired input can be done simply, by testing the two strings for equality

7.4.7.2 Grammars

One can simulate a Turing Machine by constructing a pair of grammars whose intersection is the set of 'traces' of terminating executions of the Machine

Somewhat similarly to using Post Correspondence, one can for any given Turing Machine produce a pair of (context-free) grammars. The intersection of the languages (ie strings) generated by the grammars is the set of terminating 'traces' of that Turing Machine's execution on every possible initial 'tape'; these can be tested for the desired one. Details can be found in [HU79].

Similarly to Post Correspondence, computing with grammars can be seen as an optimisation of the 'produce every string then see which has each execution step correct' process. Rather than checking every possible string to see which are step-wise correct 'traces', instead a limited number of strings are generated from each grammar, fewer of which will be incorrect. Again, equality-testing suffices to pick the ones which are correct.

7.4.7.3 Diophantine equations

Turing Machines can be simulated using simple mathematical equations

Diophantine equations are of the form of a simple polynomial function on variables equated with, typically, zero. The polynomial uses only addition, subtraction, multiplication and exponentiation. For more details, see eg [Mat96]. While one might think any 'computation' one may be able to do with them would be trivial, it has been shown that any recursively enumerable set can have a corresponding Diophantine equation constructed; and specifically that Register Machines can be mechanically converted to Diophantine equations: see [JM84] for details. Interestingly, there have also been proved to exist, analogously to Universal Turing Machines, 'universal' Diophantine equations. Solving any given Diophantine equation can be reduced to solving a 'universal' Diophantine equation, which is supplied with a numeric representation of that given Diophantine equation. For details, one can consult eg [Jon82].

One can compute with Diophantine equations as follows. To semi-decide whether an element is in a set, one simply tries to find solutions to the equation, most-simply by simply enumerating every possible combination of values for the variables. Building on this basic procedure, one can compute Boolean-valued functions; for example whether a given number is prime or composite. One simply takes the appropriate Diophantine equations for primes and composites (such are given in [Mat96]), binds a certain variable in each equation to the number in question, and then searches through values for the other variables (ie in order) until one or the other equation holds. More generally, to simulate a given function 'F' using Diophantine equations, one can construct a Diophantine equation whose solutions feature a particular variable ranging over precisely the elements of the set '{t | ∃x. t = encode x (F x)}'. Here, 'encode' encodes a pair of eg Naturals as a single Natural, for which there are standard schemes. The values of that variable of found solutions can be checked until the one whose first element is the desired input is reached; the other element then being returned as the result.

7.4.8 Unusual computational systems show that one can interpret without use of tests

In some Turing-complete computational systems tests may not be immediately apparent however they are still intuitively required

As we present below, certain (classes of) unusual computational systems do not at first glance appear to internally *do* tests, despite being able to *compute* tests. All of the systems here are Turing-complete, and are hence interpretive as discussed earlier. We show that computing with any of these systems actually does require tests.

7.4.8.1 Cellular automata

Cellular automata involve explicit tests

Cellular Automata operate by taking a one or two (usually) dimensional array of cells, and updating each cell based on the values held by a finite number of (typically) neighbouring cells. The update process intuitively involves a test on the values of the neighbouring cells to see what

value to place in the current cell of interest at the next time-step. Some very simple Cellular Automata can step-wise simulate Universal Turing Machines, as shown in [Coo04].

Note that some 'stop' mechanism is required to compute with Cellular Automata, else computation will never terminate. One could take a particular cell having some value as a trigger to terminate. We further discuss 'stop' mechanisms later.

7.4.8.2 Grammars, Post Correspondence, and Diophantine equations

Computing using grammars, Post Correspondence, and Diophantine equations requires explicit tests

Computation using Diophantine equations certainly involves explicit tests, even though the equations themselves do not. Specifically, at a minimum, one has to test to see whether the equation is satisfied, such as whether the polynomial equals zero. Based on this equality test, the system either continues execution, or terminates. One could also question whether enumeration of variable-values also requires testing. The same is true with Post Correspondence and grammars: while their formulations do not contain tests, in using them computationally one does string equality testing as described earlier. It seems that equality tests on strings suffice in some sense to replace all the symbol tests a Turing Machine would normally do.

7.4.8.3 Partial Recursive Functions

The Partial Recursive Functions are constructed using tests

Many different sets of primitives are used in the literature to form the 'Partial Recursive Functions'. Following [HU79], the Partial Recursive Functions are those definable using:

- Zero
- Successor
- Composition (n-ary)
- Projections (ie functions which return one of their arguments)
- An operator for unbounded minimisation

- Recursion of the form:

$$f\ a_1..a_n\ zero = g\ a_1..a_n$$

$$f\ a_1..a_n\ (succ\ n) = h\ a_1..a_n\ (succ\ n)\ (f\ a_1..a_n\ n)$$

Other presentations have the argument to 'h' being 'n' rather than 'succ n' in the recursion.

Note that in the formulation of recursion pretty-syntax is being used, pretty-syntax that acts to disguise explicit tests for 'zero' and 'succ'.

7.4.8.4 Logic

Proving a statement of logic corresponding to a Register Machine requires explicit testing

As presented in [EFT84, pp159-162] one can for any Register Machine mechanically construct a statement of first-order logic which is true if and only if the Register Machine halts. Note that Register Machines can be mechanically converted to Turing Machines, and can compute anything Turing Machines can. The statement in question is in effect a list of rules about how if the Machine is in a particular state, then after the next execution step, it will be in a related state. A proof of the statement involves taking the initial state of the Machine, testing to see which rule matches, and deducing the next state. This is then repeated. At the end of the proof the output of the Machine can be read off. Note that the same tests the Register Machine does on its state the theorem prover would have to do on its representation of the state.

7.4.8.5 Rewrite systems

Rewrite systems intuitively require tests

Some rewrite systems (see eg [O'D85]) have Turing-complete computational power. While they perhaps may not be thought of as containing tests, execution of them is canonically done by a process of symbol-matching (ie testing) terms against the left-hand side of productions, and rewriting the term as appropriate.

The lambda calculus and SK combinatory calculus are examples of rewrite systems. The SK combinatory calculus can be presented as the following very simple rewrite system:

$(\forall f,x)\ K\ f\ x \rightarrow f$

$(\forall f,g,x)\ S\ f\ g\ x \rightarrow (f\ x)\ (g\ x)$

7.4.8.6 Object-Oriented languages

Object-Oriented languages implicitly include support for tests

We show how Object-Oriented languages can host, without apparent use of symbols and tests, a Turing-complete rewrite system. Interestingly, this shows that arbitrary user-defined tests are not required for Turing-complete computational ability, in Object-Oriented languages. We then discuss how tests are actually indeed present, just implicit.

The system we choose to implement is the well-known SK combinatory calculus. The full list of classes used in our implementation is:

'Apply', instances of it have two items of state

'K', instances of it have no items of state

'K_1', instances of it have one item of state

'S', instances of it have no items of state

'S_1', instances of it have one item of state

'S_2', instances of it have two items of state

The program to be executed is represented by a tree of 'S' and 'K' objects, with each node of the tree being an 'Apply' object.

Every object supports the following methods:

Bind

DoOutermostApplicationIfAny

ReduceToNormalForm

Only 'Bind' takes a parameter, a single one.

The semantics of these methods are as follows. We use 'Q[a,b,c..]' to denote an instance of class 'Q' with the first component of its state being 'a', second being 'b', etc. Occurrences of this on the right-hand side different from the left-hand side refer to fresh instances; if not different then the same instance is indicated. We will write 'Q' for 'Q[ ]'. The notation

‘Q[a,b..].M(x,y..)’ denotes objects of class ‘Q’ having method ‘M’ being invoked, whose arguments are ‘x,y..’. Finally, we write ‘—⟩’ for the operational semantics rewrite relation, with the properties that ‘T —⟩ T’ and ‘T —⟩ U ∧ U —⟩ V ⇒ T —⟩ V’.

K.DoOutermostApplicationIfAny —⟩ K
K.ReduceToNormalForm —⟩ K
K.Bind(f) —⟩ K_1[f]

K_1[f].DoOutermostApplicationIfAny —⟩ K_1[f]
K_1[f].ReduceToNormalForm —⟩ K_1[f]
K_1[f].Bind(x) —⟩ f

S.DoOutermostApplicationIfAny —⟩ S
S.ReduceToNormalForm —⟩ S
S.Bind(f) —⟩ S_1[f]

S_1[f].DoOutermostApplicationIfAny —⟩ S_1[f]
S_1[f].ReduceToNormalForm —⟩ S_1[f]
S_1[f].Bind(g) —⟩ S_2[f,g]

S_2[f,g].DoOutermostApplicationIfAny —⟩ S_2[f,g]
S_2[f,g].ReduceToNormalForm —⟩ S_2[f,g]
S_2[f,g].Bind(x) —⟩ Apply[Apply[f,x],Apply[g,x]]

Apply[m,n].DoOutermostApplicationIfAny —⟩ m.Bind(n)
Apply[m,n].ReduceToNormalForm —⟩
    Apply[m,n].DoOutermostApplicationIfAny.ReduceToNormalForm
Apply[m,n].Bind(arg) —⟩ Apply[m,n].DoOutermostApplicationIfAny.Bind(arg)

Note that there is no primitive data in the above. There are also, surprisingly, no explicit tests.

We now verify that the semantics of 'ReduceToNormalForm' are correct. The particular (Head) Normal Form we wish to show 'ReduceToNormalForm' produces is simply 'NF' in:

NF ::= S | K | S_1[Term] | S_2[Term,Term] | K_1[Term]

where:

Term ::= NF | Apply[Term,Term]

Firstly, it can be shown that 'Bind' is progress-making and semantics-preserving, ie does reduction. To do this, we prove that rewrites occur in the system paralleling the combinatory-reductions 'K f x → f' and 'S f g x → (f x) (g x)'. By writing 'T1 ► T2' if for all 'o' it is true that 'T1.Bind(o) —⟩ T2.Bind(o)', we can state the proposition:

Apply[Apply[K,f],x] ►f

Apply[Apply[Apply[S,f],g],x] ►Apply[Apply[f,x],Apply[g,x]]

To prove the first of these, note first that:

K.Bind(f).Bind(x)

—⟩ K_1[f].Bind(x)

—⟩ f

Now, as:

A —⟩ B

implies

∀method,args. A.method(args) —⟩ B.method(args)

then:

∀o. K.Bind(f).Bind(x).Bind(o) —⟩ f.Bind(o)

⇔ (as 'Apply[m,n].Bind(o) —⟩ m.Bind(n).Bind(o)', and transitivity of '—⟩')

∀o. Apply[K,f].Bind(x).Bind(o) —⟩ f.Bind(o)

⇔ (again)

∀o. Apply[Apply[K,f],x].Bind(o) —⟩ f.Bind(o)

hence, by definition, we get the required expression:

Apply[Apply[K,f],x] ►f

Similarly, for the 'S' case, note that:

S.Bind(f).Bind(g).Bind(x)

—〉 S_1[f].Bind(g).Bind(x)

—〉 S_2[f,g].Bind(x)

—〉 Apply[Apply[f,x],Apply[g,x]]

Now, as:

∀o. S.Bind(f).Bind(g).Bind(x).Bind(o) —〉 Apply[Apply[f,x],Apply[g,x]].Bind(o)

⇔ (as ‘Apply[m,n].Bind(o) —〉 m.Bind(n).Bind(o)’, and transitivity of ‘—〉’)

∀o. Apply[S,f].Bind(g).Bind(x).Bind(o) —〉 Apply[Apply[f,x],Apply[g,x]].Bind(o)

⇔ (again)

∀o. Apply[Apply[S,f],g].Bind(x).Bind(o) —〉 Apply[Apply[f,x],Apply[g,x]].Bind(o)

⇔ (again)

∀o. Apply[Apply[Apply[S,f],g],x].Bind(o) —〉 Apply[Apply[f,x],Apply[g,x]].Bind(o)

then, by definition, we get the required expression:

Apply[Apply[Apply[S,f],g],x] ►Apply[Apply[f,x],Apply[g,x]]

Secondly, it can be easily verified that if a term is already in the Normal Form, then calling ‘ReduceToNormalForm’ returns that term, unchanged:

K.ReduceToNormalForm —〉 K

K_1[f].ReduceToNormalForm —〉 K_1[f]

S.ReduceToNormalForm —〉 S

S_1[f].ReduceToNormalForm —〉 S_1[f]

S_2[f,g].ReduceToNormalForm —〉 S_2[f,g]

Third and finally, we need to show that if a term isn’t in the Normal Form, then calling ‘ReduceToNormalForm’ doesn’t change its semantics, and returns a term in the Normal Form. To begin, note that if a term isn’t in the Normal Form, then it is ‘Apply[m,n]’, for some ‘m’ and ‘n’. Consider:

Apply[m,n].ReduceToNormalForm —〉 m.Bind(n).ReduceToNormalForm

If the result of ‘m.Bind(n)’ is in the Normal Form, then as per above that term will be returned. The only other option is that the result of ‘m.Bind(n)’ is ‘Apply[a,b]’ for some ‘a’ and ‘b’. In this case the above rewrite rule will be invoked again. Hence the only operation carried out by ‘ReduceToNormalForm’ is to rewrite ‘Apply[x,y]’ for some ‘x’ and ‘y’ as ‘x.Bind(y)’. This is semantically-correct since, as can be verified:

Apply[x,y].Bind(o) —⟩ x.Bind(y).Bind(o)

ie:

Apply[x,y] ► x.Bind(y)

It should be noted that 'ReduceToNormalForm' will not always terminate. That it makes progress can be seen from the fact that 'Bind' makes progress; and that it terminates as often as possible follows from the reduction-strategy being, as can be seen, the well-known 'Normal Order'.

While the above system contains no explicit tests, it can be seen that implicit tests in the 'polymorphism' of the Object-Oriented language take their place, to ensure different rewrites occur for different terms. The 'polymorphism' of Object-Orientation refers to the code executed for a method-call depending on the class of the object. This intuitively involves a test on the object's class. The special language support in Object-Oriented languages for this testing can be seen as part of the appeal of those languages, and a validation of TFP. Object-Oriented programs are 'nicer' because a very common test one wants performed is already present in the language, with special syntax (the 'object.method' notation). Further, most Object-Oriented languages limit the power of this test, specifically by not letting new classes or methods be produced at runtime. Without this limitation, one can imagine some particularly unreadable and complex programs being able to be written.

7.4.8.7 Self-modifying imperative code

Certain imperative languages are Turing-complete but do not offer primitives that do tests; however tests are still required for their execution

An interesting example of a system with an apparent ability to *implement* tests but that executes without *doing* tests is given in [Roj96]. Therein it is demonstrated that conditional branching can be replaced by self-modifying code. It is shown that one can produce a computational system of Turing-complete power using only 'LOAD', 'STORE', 'INC' (increment) and 'GOTO' instructions, represented by (fixed-size, via use of relative-addressing) numbers in memory. There is also an implicit fifth instruction, meaning 'stop'. None of these primitives would be considered to be, or do, tests. However, as the instructions are stored as numbers, executing a load, store, increment, or go-to as appropriate requires the language to test numbers to see which instructions they represent.

7.5 Testing and interpretation

Interpretation is not testing

In this section, we consider whether interpretation can be characterised as testing. As a first step, we show that some of the already-discussed Turing-complete systems can have their primitives implemented abstractly without using tests. We also identify an analog, and test-free, computational system.

As far as *physical* implementations of the above systems are concerned, one needs to ask how test-free fundamental physics is. It may well be that all physical computing systems are based on tests.

Subsequently, we detail how the above results can be leveraged to show that arbitrary programs written in arbitrary languages can be implemented (abstractly) without using tests.

Our conclusion is that testing is an *emergent* property: one can build tests from building-blocks that don't involve tests. Due to the subjectiveness of identifying tests, testing cannot be our desired characterisation of interpretation.

7.5.1 Test-free implementations of Turing-complete systems

There are at least three abstract Turing-complete computational systems that are test-free

Previously, we have discussed how Object-Oriented polymorphism involves doing an implicit test on the class of an object, to determine which method-implementation to execute. However, consider if all method-names occurring in a program were numbered, from one up to some number 'n'. Also, let each object include as part of itself (and, in a consistent location) a Church Tuple of size 'n', with element number 'i' being the object's implementation of method-name number 'i'. This implementation could be error-rasing code, if the method-name is notionally unimplemented by the object. Then, all occurrences of 'o.m' (for any object 'o' and method-name 'm') in the program could be replaced by '(getlist o) ($\lambda a_1..a_n.\ a_x$)', where 'x' is the number

of the method-name ‘m’, and ‘getlist’ extracts the tuple of the supplied object. Hence Object-Oriented polymorphism can apparently be test-free.

This leads one to suspect that Turing-complete systems could be implemented without using tests. We now give a pair of brief and informal presentations that act to demonstrate that this is indeed possible; that it is possible to implement the primitives of (at least some) Turing-complete computational systems without using tests. Our chosen abstract implementations are such that they can be recast as physical implementations, which would be as test-free as the underlying universe (at least, the fundamental laws they make use of). Note that any requirement for infinite space in the implementations can be approximated, as is usual practice. We also identify a Turing-complete system that is already test-free in its canonical presentation.

7.5.1.1 Universal Turing Machines

The primitives of Universal Turing Machines can be implemented without using tests

Turing Machines are canonically presented (eg [HU79]) as containing an infinite ‘tape’ on which symbols have been placed, and a Finite State Machine called the ‘head’. Each execution step involves the head testing the symbol at the current location on the tape then changing state, writing a new symbol at the current tape location, and either stopping or moving the tape left or right one position.

These primitives (tape, head, and symbols) can be implemented abstractly without tests, using a chosen abstract ‘universe’ that parallels the physical universe. Firstly, implement the head as a two-dimensional array of individual machines (we call them ‘hs-machines’), whose semantics will be described shortly. The array permits any vertical column of hs-machines to be brought forwards then back in line. Secondly, implement the tape as an infinitely-long one-dimensional array of ‘boxes’ each of which holds a machine, with the array being on wheels or similar so it can be moved left and right in front of the head. The machines which are held in the ‘boxes’ are implementations of symbols, and we term them ‘symbol-machines’. All machines representing the same symbol-value are to have identical semantics, which will be described shortly. Finally, let there be for each sort of symbol-machine a ‘spring-loaded’ queue holding an infinite number of those symbol-machines; the use of these queues will soon be made clear.

Initially, the tape is aligned so that a 'boxed' symbol-machine is directly in front of the middle of the head, with the tape extending to the left and to the right. Also, one vertical column of hs-machines has been brought forwards. Computation begins when the symbol-machine in the middle of the head is activated (eg plugged into a power supply) by an external agent.

The semantics of hs-machines are as follows. We use the word 'particular' to mean something that is 'hard-wired' into a given hs-machine and is a constant for it, rather than something which is the result of a runtime choice. When activated (eg when power is supplied), a hs-machine:

1. Picks up and discards the symbol-machine on the tape in front of it, replacing it with a new symbol-machine from a particular queue

2. Moves the tape in a particular direction, left or right or not at all (corresponding to reaching a 'stop' state).

3. Moves the vertical column it is part of back and moves a particular vertical column (possibly the same one) into the forwards position

4. Activates the symbol-machine it had placed on the tape

5. Deactivates itself (eg unplugs itself from the power supply)

We now turn to the semantics of symbol-machines. When activated, symbol-machines corresponding to the n-th symbol activate the n-th element of the currently-forward vertical column of the state machine; then deactivate themselves.

Note that the semantics of each and every machine is simple, and does not involve searching for other machines or testing to see what type of machine they are: each machine simply does particular things at particular locations without testing what it is interacting with. We assert that for any Turing Machine we can find appropriate hs-machines and symbol-machines that satisfy the above semantics and result in the same mapping being computed as the Turing Machine.

The only obscure part of the above is that which corresponds to the test the head would normally do on the current tape symbol, to work out what transition to undertake. Vertical columns of hs-

machines correspond to a single state of the Finite State Machine. Each hs-machine element of a column corresponds to a transition triggered by a tape symbol, and appears in order. This permits the current tape symbol when activated to itself pick the appropriate transition, rather than a test being done. The same general technique can be seen in using:

(λx. x A B)

where 'x' is a Church Boolean, instead of

(λx. if x == true then A else if x == false then B)

where 'x' is a Boolean. It is also the same general technique used in test-free Object-Oriented polymorphism as described previously, and below for test-free copying of SK terms.

7.5.1.2 SK combinatory calculus

The S and K terms of the SK combinator calculus can be implemented test-free

Any implementation of the SK combinatory calculus one might think is required to do pattern-matching, to determine which rewrite rule to apply at each execution-step. However, consider the following implementation of the SK combinatory calculus, using abstract machines. We call these machines 'SK machines', and have detailed a software analogue of them previously. The types of machines used are:

1. 'Apply': a machine of this type, when connected to two other machines, activates (eg the connections complete an electrical circuit) and replaces itself with the first machine after connecting the second machine to it

2. 'K': a machine of this type when connected to another machine, call it 'P', activates and transforms itself into a 'K_1' machine holding 'P', then deactivates

3. 'K_1': a machine of this type, holding an item 'P', activates when connected to another machine, discards or destroys that other machine, then replaces itself with 'P' and deactivates

4. 'S': a machine of this type when connected to another machine activates and transforms itself into an 'S_1' machine holding that other machine

5. ‘S_1’: a machine of this type, holding an item ‘P’, activates when connected to another machine ‘Q’ and transforms itself into an ‘S_2’ machine, holding ‘P’ and ‘Q’

6. ‘S_2’: a machine of his type, holding items ‘P’ and ‘Q’, when connected to another machine activates and immediately replaces itself with an ‘Apply’ machine connected to:
    a. An ‘Apply’ machine connected to ‘P’ and the other machine
    b. An ‘Apply’ machine connected to ‘Q’ and a copy of the other machine

Initially, one starts with a binary tree whose nodes are ‘Apply’ machines and whose leaves are individually ‘S’ or ‘K’ machines. One then eg supplies a common source of electromotive force to power activated machines, and then waits until the machines finish rearranging themselves. These machines are equivalent in spirit to Programmed Graph Reduction, introduced in Chapter 2.

Note that at no stage does a machine investigate its surroundings, ie no testing is done. However, one can ask if the copying done by ‘S_2’ machines requires testing: testing what was supplied piece-wise in order to select what to add to the result-in-progress. We suggest that each machine could come in two parts, one of which has semantics as per above. The other part, when activated, is able to pick up a machine from one of a number of queues each containing an infinite supply of one of ‘S’, ‘S_1’, ‘S_2’, ‘K’, ‘K_1’ or ‘Apply’ -type machines. It then activates the same part of any connected or held machines. Hence, rather than the copy process testing to see which new machines are required, instead each old machine picks out an appropriate new machine then ensures machines below it in the tree do the same.

One may have to add some form of (test-free) synchronisation or timing mechanism to the above formulation, to prevent simultaneous machines activations and copy-processes from interfering with each other.

7.5.1.3 Certain sets of differential equations

There exist analog computational systems which do not require tests

In [Bra95], particular Ordinary Differential Equations are shown to have the power to be able to simulate Universal Turing Machines, based on quite reasonable definitions of ‘simulation’. This

form of computational system is *analog* rather than discrete in nature, both in time and values (ie quantities are continuously-variable). It is also test-free, in its presentation as equations regarding rates of change. One might however argue that use of Ordinary Differential Equations will require a test to be used to determine whether the equivalent of a 'stop' state of a Turing Machine has been reached. However, determining when the computation has completed has to be done with *any* computational system. Inspection of a system by the user or another process we don't consider to be a test done *by* the system during its computation.

It is interesting to consider physical realisations of these Ordinary Differential Equations. Due to their simplicity and relative fundamentality, we believe it likely that if any Turing-complete system can be physically realised test-free, then Ordinary Differential Equations also can. Physical realisations of Ordinary Differential Equations seem likely to be the easiest to investigate, and most likely to be test-free.

One should note that these Ordinary Differential Equations are far from the only analog Turing-complete computation systems: similarly-expressive analog systems are surveyed in [Orp97]. We also point out [CMC00] as being of some interest, with its linkages to familiar concepts in discrete computation.

7.5.2 Executing arbitrary programs in arbitrary languages without testing

Using any test-free Turing-complete system, arbitrary languages can be hosted so that their programs' executions are test-free

All of the abstract test-free Turing-complete computational systems outlined above are able to be used to execute arbitrary programs written for arbitrary programming languages, via interpretive hosting and the Church-Turing Thesis. Hence all (computable) programs are able to be executed without the use of tests. Note that one is also often able host definitionally. First, if possible, define the program's constructs as lambda-terms. For example, definitions for Partial Recursive Functions can be found in [Bar84], and for Universal Turing Machines in [Mai92]. Then, use the well-known transformation from lambda calculus terms to SK terms to produce test-free SK machines.

7.5.3 Analysis

Testing is an 'emergent' property

Chapter 5 contains numerous examples of how one can take a program-fragment, remove all tests by evaluating them, yet end up with the program-fragment still doing a test. For example, one can transform

iszero n ≜ iterate n (λx. false) true

where 'iterate' does pattern-matching tests on its Natural number input, to:

iszero n ≜ n (λx. false) true

where 'n' is now a Church Natural. Note that this 'iszero' doesn't use any testing constructs, but does a test. Similarly,

(λx,A,B. if x == true then A else if x == false then B)

where 'x' is a Boolean, can be transformed to:

(λx,A,B. x A B)

where 'x' is a Church Boolean. Rather than 'x' being tested to determine which of 'A' and 'B' to choose, instead each value of 'x', which knows what itself is, so to speak, chooses between the two.

One is therefore lead to the conclusion that testing is an 'emergent' property, in the general sense of the word. One can apparently implement tests from building-blocks that don't do tests.

However, one may validly suggest that in the above examples perhaps tests are always still present, implicitly, in the programming language itself. For example, while beta-reductions may not at first glance appear to involve tests, maybe to implement them tests are always required. The suggestion can be discounted. As we've outlined above, a number of Turing-complete computational systems can be implemented abstractly without tests. One can host any programming language, hence any program that does tests or is itself a test, using test-free Ordinary Differential Equations, test-free SK machines, or our test-free formulation of Universal Turing Machines. Hence, testing is truly an 'emergent' property.

7.5.4 Relation to interpretation

Testing is subjective, and hence cannot be our desired characterisation of interpretation

If one accepts the informal characterisation of interpretation as being testing, then one is forced by the 'emergent' nature of testing to the position that interpretation is also 'emergent'. In other words, just because a particular program does interpretation, doesn't mean that any particular primitive used is interpretive, ie does tests. As a result, to determine whether something is interpretive or not, one would have to be able to determine when the primitive building-blocks have been combined to form a test. For example, one would have to be able to recognise when particular Differential Equations implement a test. Some specific examples in the lambda calculus are also illuminating: one would need to be able to identify that

(λx. x M N)

is a test for certain members of its input-domain, specifically a test on Church Boolean 'x'; and that

(λx,y,z. x (λw. z) y)

is a test on Church Natural 'x', the 'iszero' test. Further, nearly any lambda-term can be viewed as an 'if' test. One need merely pick two inputs which give different (with respect to '↔') outputs, call those inputs representations of Booleans, and call the two corresponding outputs the branches of the 'if' test.

Simply, whether something is a test or not is subjective, and not able to be meaningfully formalised. The inherent subjectivity of identifying tests exposes the informal characterisation of interpretation (as being testing) to be incompatible with the Objectivity Criterion. However, what then *is* interpretation? What is it that canonical test-and-loop interpreters do or have, even when written in the lambda calculus using Church Objects, or implemented as SK machines? What is common to Universal Turing Machines and self-modifying imperative code?

7.6 A characterisation of interpretation

The intrinsic nature of interpretation is input-varying computational steps

Based on our investigations in this and prior Chapters, we now propose an objective characterisation of interpretation. We describe it in detail, then briefly discuss the benefits of it over characterising interpretation to be testing. We also give a detailed justification of the characterisation based on our desiderata. Interestingly, it can be seen that the characterisation exposes (much, at least) testing as being interpretive, hence can be seen to *extend* the informal characterisation of interpretation, beyond testing. Subsequently we move on to discussing the practical application of our characterisation.

In the following section we discuss a related and somewhat complementary property, 'inherent interpretiveness', before focusing on the costs and benefits of interpretation, and how programming with reduced interpretation is both feasible and desirable. We also validate TFP-style programming as being less-interpretive programming.

7.6.1 Proposed characterisation

We characterise something to be interpretive if and only if the computational steps it carries out during execution vary with its input

We believe the commonality we are after between interpreters written in various ways is that what is done, computationally, during execution varies with the input supplied. Interpretation is not testing, but what testing is usually *used for*. Simply, we propose that the characteristic nature of interpreters is that the computational steps they carry out during execution can vary depending on the inputs. From this point forwards, we will take 'interpretive' to mean 'has input-varying computational steps', and (abusing grammar), 'interpretation' to mean 'input-varying computational steps'. Often, the computational steps vary depending on only one part of the input. We term such the 'interpreted' part of the input, which the interpreter 'interprets on' or simply 'interprets'.

This characterisation has surprisingly strong consequences: as we discuss, nearly all programming is exposed as interpretive. This does not however void TFP: even though it turns out to be impractical to fully eliminate interpretation, minimising the use of it is still worthwhile, as is choosing interpreters with the least-undesirable properties. Programming in an interpretation-aware manner is indeed beneficial, as we detail later.

Interestingly, our characterisation does not immediately place data in a negative light, only one thing it can be used for: to determine what semantics to execute. Indeed, opposite what was expected based on TFP, it can be seen that it is first-class semantics which is intuitively undesirable rather than primitive data.

7.6.1.1 Non-extensionality

Computational steps are important, not extensional semantics

It is important to be aware that it is internal computational steps which are of interest, not extensional semantics (ie the overall mapping from input to output). For example a function on Naturals, executing as:

(λx. if (iseven x) then x else (div (mul x two) two))

is certainly interpretive by our characterisation as the computations carried out vary with the input, even though its input-output mapping is equivalent to that computed by '(λx. x)'.

Note that the need for non-extensional information is completely consistent with our informal judgements as to interpretation: two programs might have the same mapping from inputs to outputs, yet one may execute via eg simulating a Universal Turing Machine. Two programs might have the same overall semantics, but for questions of interpretation one cares about *how* they execute.

7.6.1.2 Range of inputs

A program which uses an interpreter is not always itself interpretive

We turn now to elaborating what is meant by 'input'. Consider a program which performs consistent computational steps for a certain subset of its input, but for other inputs different

computational steps will be carried out. If the program is only used on that subset of its inputs, is it still interpretive? We have to say that it is. All possible inputs should be considered, as a restriction to what executions (if any!) are done in practice would reduce the objectivity of the definition. We want to be able to classify things as being interpretive in and of themselves, independently from their uses.

A related question concerns when a program contains individual constructs whose computational steps vary with their input, but for no input to the program is any other than a single consistent computational-sequence carried out. Individually the constructs are interpretive, but the program itself is not interpretive under our characterisation. Consequently, one can build a non-interpreter from interpretive building-blocks; ie interpretation-freeness is an 'emergent' property. One could consider similar but different characterisations for interpretation, for example stating that something is an interpreter if any individual part or aggregates thereof exhibits input-varying computational steps over its potential (rather than actually-used) range of inputs. However as the situation in question almost never occurs in practice we are satisfied to leave our characterisation as is, leaving any tinkering for future work.

7.6.1.3 Distinctness of computational steps

Should computational steps be distinguished by their actions alone?

One situation of interest is when a program carries out differing *instances* of computational steps depending on the input, but they are the same steps. For example consider the following function, which takes a Boolean as input:

```
F b x = IF b (S x) (S x)
```

Here, either the first 'S x' is executed, or the second. Is the fact that the two branches entail the same computational steps sufficient to classify this apparently interpretive program as non-interpretive? Based on the characterisation as it stands, the answer is in the affirmative.

Usually the program will be able to be considered interpretive based on other code it contains. Hence in practice this situation, rarely arising, is even more rarely needs to be considered. Being of low practical import we don't discuss it further, leaving resolution of this issue as future work.

7.6.2 Benefits of the proposed characterisation

Input-varying computational steps is a more practical characterisation of interpretation than testing

Our characterisation of interpretation is objective, and avoids the difficulties found with characterising interpretation to be symbol-based computation, specifically testing. These difficulties include the 'emergent' nature of testing, and the subjectivity required to identify when something one considers a test has been formed.

Rather than focusing on the common (but as we've seen not always-present) means by which an interpreter *decides* what do to, ie tests, our characterisation is based on what the interpreter *does*, its varying computations. Our characterisation does not rely on detecting tests, or detecting symbols or representations thereof. We take the view that there are many ways in which a function can have varying computations, and not all of these need to involve some explicit test on data or a representation thereof.

7.6.3 Comparison to desiderata

This characterisation of interpretation meets all our requirements

Our characterisation of interpretation can be seen to meet the desiderata, as we present now.

Firstly, the characterisation meets our Objectivity Criterion, as it is objective: either the computations of the entity in question vary with its input, or they do not.

Secondly, our characterisation is consistent with the general meaning of 'interpretation' as 'ascribed meaning', when the 'meaning' is 'computations'. Under our characterisation, interpreters ascribe computations to their inputs. The only point of departure here is that we require that not all inputs be ascribed the same computations.

Thirdly, our characterisation is consistent with specific strong uses of the word 'interpretation' to describe Universal Turing Machines and other systems of Turing-complete computational

power. Turing-complete systems are able to 'act like' (hence have computations which vary with) whatever machine or function encoded in their input. Moreover, as our characterisation doesn't have interpretation being an 'emergent' property, we should be able to identify in each of the Turing-complete systems exactly what part of the computation varies. This is indeed generally the case. Universal Turing Machines have computational steps (ie moves of the head left or right) that vary with their inputs. SK machines engage in differing sequences of construction based on their inputs. Turing-complete self-modifying imperative code engages in differing sequences of 'LOAD' 'STORE' and 'GOTO' computations based on its input. Lambda calculus terms execute computations supplied as inputs, and hence have input-varying computations. When computing with Diophantine equations, grammars, Post Correspondence, or logic, the sequence of enumerate and test computations carried out varies with the supplied input. In Partial Recursive Functions when 'succ' is supplied as input a recursion will be done, unlike when 'zero' is supplied: different computations. What actually constitutes computational steps in Ordinary Differential Equations is not immediately obvious. Perhaps the steps themselves are continuous, and it is the ability for one quantity to vary another, or that other's rate of change. A refinement of our characterisation of interpretation does appear to be required to cover such systems. We discuss Cellular Automata later. Overall, it is apparent that there is a very close fit between most examples of known interpreters, and our characterisation of interpretation. Indeed, it is hard to see what else the 'LOAD' 'STORE' 'GOTO' imperative language, SK machines, the lambda calculus etc have in common other than the ability for their programs to vary computations with inputs. Our characterisation of interpretation hence appears to be the correct one, even if one was to relax the Objectivity Criterion.

Fourthly, our characterisation is also consistent with 'interpretation' as contrasted to 'compilation'. On execution of interpreted code, each construct (representation) is 'looked up' and the appropriate computation executed. On execution of compiled code, this runtime selection of computation for constructs of guest code does not occur, such selections happen statically during compilation. Interpretive and definitional styles of hosting code differ in the same way.

Finally, testing and interpretation have a very close relationship, indeed we considered previously the possibility that it was one of synonymity. One would want any characterisation of interpretation to be able to explain the common co-occurrence of the two. Our characterisation is so able. Simply, most tests execute one of their branches depending on their

input: these tests hence have input-varying computational steps. For example if one passes 'IF' a Boolean and two items of code, one or the other of those items, depending on the Boolean, will be executed. Similarly other pattern-matchers which selectively execute branches depending on input have input-varying computational steps. Also, note that tests whose branches are data rather than code, such as a Boolean-valued equality tests, are often associated with interpretation: often the results of the test are used to determine what computation to execute next, such as whether to continue a recursion or loop.

7.6.4 Relation to informal characterisation of interpretation

Our characterisation of interpretation is in essence an extension of the informal characterisation of interpretation

As discussed above, our characterisation of interpretation exposes as interpretive tests that selectively execute their branches. Hence it extends the informal characterisation of interpretation as being symbolic-style computation, specifically testing, which has been used previously. The sole possible point of difference is in regards to tests whose inputs and outputs are data, such as equality tests returning Booleans. Under the intuitive characterisation of interpretation as being symbol-based computation these tests are interpretive, but under our characterisation it is unclear whether they are actually interpretive or not as we discuss later. These contentious tests have appeared only sparingly earlier in this dissertation; instead most tests which appeared selectively execute their branches. For example, 'fold' defined as:

fold list op b ≜ if list == nil then b else if list == cons e r then op e (fold r op b)

is interpretive by the new characterisation, as the 'if' test results in either the recursion continuing, or not.

7.6.5 Applying the characterisation

Our characterisation is applicable to any system for which computational steps and inputs can be identified; most such non-trivial systems are interpretive

Our characterisation is applicable not just to programs, but to physical and abstract computational systems generally. To apply our characterisation of interpretation to determine whether some given computational system is interpretive or not, one needs to identify:

1. The inputs to the computational system
2. The computations (ie the 'trace' of computational steps) done by the system during execution

Typically only identification of computations may be problematic, and unfortunately, usually is. We cover three common scenarios, and suggest approaches.

7.6.5.1 Abstract computing systems

For abstract computing systems, one asserts what constitutes a computational step

When dealing with abstract machines, what constitutes a computational step is undefined. We suggest that this is a free choice, as the system is after all abstract. One would simply assert that certain individual primitives denote non- input-varying computational steps, and make similar assertions that any 'ambient' semantics is carried out as certain sequences of simple, atomic computational steps. Interestingly, despite this freedom, non-trivial abstract systems will usually be interpretive: most algorithms engage in different calls to primitives for different inputs. A simple example of this is sorting. Typically one would assert that comparing elements involves a distinct computational step, and as the number of comparisons done vary for different-length inputs, the system is interpretive.

For abstract mechanical computational systems, one would usually consider individual movements to be computational steps. Hence our formulation of mechanical Turing Machines and SK machines are interpretive: different inputs result in different mechanical motions being carried out.

What then is an interpretation-free abstract computational system? Simply, one whose computations do not vary with input. Such systems can only consist of a fixed sequence of computations. For program-like systems, they cannot have (non-degenerative, at least) use of guarded loops or recursion, conditional jumps, etc. For systems with constructs that only take and return data, each construct would typically be asserted to involve non- input-varying computations (unless they eg intuitively required different computational time for different inputs). To be non-interpretive, such systems must not modify, rewrite, rearrange etc their own components during execution. In other words, to be interpretation-free, the system must be of the style of simple analog computers, in that they must do the same computation regardless of input. This is a perhaps-surprisingly consistency with the positive view in TFP of analog-style computation. Such interpretation-free systems can be contrasted to the von Neumann architecture (again, this confirms thought in TFP). Abstract systems of that architecture are interpretive: one views the memory of the computer as its input, into which values are placed by some means. Then, the computations carried out on execution depend on those memory-contents, those inputs: this is the essence of the stored-program computer. Analog computers instead have to be 'reprogrammed' via rewiring: rather than the computations being varied by the computer itself *during* execution instead they are varied by the user *between* executions.

Another interesting question one could ask, regarding a particular abstract computational system, is whether *there exists* a choice of computational steps for its primitives which makes the system non-interpretive. We discuss this further later.

7.6.5.2 Physical computing systems

For physical computing systems, what constitutes a 'computational step' requires determination

When dealing with physical computational systems (eg programs running on PCs), the computational steps carried out during execution are those of the underlying universe. One requires information about the fundamental physics, about the individual computational steps during the execution. What constitutes such a computational step is far from clear, hence making a certain determination as to whether any given physical computing system is interpretive or not is impossible with our current knowledge. However, one can make assumptions about particular sub-processes; specifically that certain of them involve different computational steps from others. Depending on the semantics of the components in question,

these assumptions can appear to be quite reasonable. For example, if for some inputs the physical computing system activates one sub-process of itself and for other inputs it activates another; if the two have substantially different semantics, one could reasonably call the system interpretive. Using such assumptions, one can conclude that nearly any program running on today's digital hardware is interpretive: depending on the program's input different instructions will be executed by the CPU (due to 'conditional jumps'), and also individual instructions themselves seem to require differing computational steps; for example, the arity-two 'XOR' versus arity-one 'INCREMENT' instructions. Even if what the *actual* underlying computational steps are during execution cannot be determined, whether they are *different* for different inputs does seem able to be informally argued in many cases. Note that the same technique can be used for abstract computational systems, as well. Later on we introduce another means of reaching conclusions regarding interpretation in the absence of information about individual computational steps, 'inherent interpretiveness'.

Perhaps a modified characterisation of interpretation will be required to formally discuss physical computing systems. We believe that expanding 'computational step' out to mean 'change' generally may be successful. After all, what is a 'computation' but a modification of some things based on other things? Generalising 'computational step' to 'change' exposes the universe as being interpretive. Consider SK machines. The computations carried out, ie the sequence of changes, of moving, duplication and destruction of machines, depends on the input supplied. Similarly, physical (approximations to) Turing Machines engage in differing moves of their head depending on their input. Additionally, electrical computing devices (eg computer CPUs) would clearly be interpretive, due to changing electrical flows and patterns of charge. If the characterisation of interpretation was modified in a way that exposes interpretation in physics, it would one would think also permit analysis of abstract continuous computational systems with no well-defined computational steps, such as Ordinary Differential Equations.

7.6.5.3 Programs, specifically

The computations done during execution of programs must be determined or assumed

With regards to programs, abstract or physical, it should be noted that, following Chapter 6, the computational steps a program engages in will depend on the particular programming language implementation chosen. If the computational steps are unknown, they can be assumed via a

Conceptual Model, however the results are correspondingly weakened. For example, one may assume, as is done commonly, that a particular program executes *as if* certain constructs carry out certain computations. If those computations vary with the input to the program, then all one can say is that if the program executes as assumed, then it is interpretive. Further, abstract programs need to have assertions made as to what constructs have input-varying computational steps; while programs executing on physical hardware need to have underlying computational steps identified. An alternative, applicable to both and discussed above, is to make assumptions about how the program executes, in terms of which constructs involve differing computational steps.

7.7 Inherent interpretiveness

Often it is not whether something is actually interpretive, but whether it needs to be, that is of key interest

There are two infelicities of applying our characterisation of interpretation, arising from the need for computational steps to be identified in order to see if they are input-varying.

The first infelicity is that from the source-code of programs and knowledge of their input-output mappings alone, we cannot determine what computations are actually carried out. Even also knowing the input-output semantics of individual language primitives is insufficient. One requires knowledge of the language's implementation, of the underlying computational steps.

The second infelicity is that even if this knowledge is available, it is undesirable to require it: one would like the maximum possible language-implementation-independence. For example, nearly all programs executing on today's PCs may be interpretive under our characterisation, but could this be an artefact of our computing hardware: are the programs able to be executed non-interpretively on different hardware? One would like to somehow factor-out the underlying hardware or execution-platform, ie ignore as much as possible implementation-details.

We have found an (incomplete) means of avoiding these infelicities; a means which doesn't require either low-level information or assumptions regarding computation. Essentially, one asks whether something *could* execute without interpretation or whether its execution is

necessarily interpretive. In other words, one asks if the input-output mapping is 'inherently interpretive', ie whether all implementations of it are interpretive. Whether something necessarily has to be implemented interpretively is usually a better question to ask than whether its actual implementation is interpretive, and requires only extensional information. An affirmative answer tells us that the implementation is interpretive without needing to know what that implementation actually is, ie without needing (or needing to assume) details of the language's implementation. Conversely, if the answer is in the negative, then often it is justifiable to not care whether it is actually interpretive: any interpretation is an accident of the implementation, not the fault of the program. Unfortunately we are unable to completely characterise input-output mappings; we simply do not know if certain mappings are inherently interpretive or not.

We give more details below.

7.7.1 Forms of mappings

Mappings are opaque functions

Mappings are opaque functions, presentable as sets of input-output pairs (multiple inputs are notionally combined into a tuple). We leave extension to nondeterminism and adding a temporal component (as is sometimes necessary, see [Rus89]) for future work. With regards to the inputs and outputs of mappings, we leave this general. Certainly primitive data and semantics (as denoted by eg lambda-terms) can be supplied as input or produced as output. Also, '⊥' can appear as an input or output, denoting 'no information' such as from a non-terminating (and side-effect -free) computation. Hence mappings can include inputs for which no output is produced, by having such inputs map to '⊥'.

To validly link a mapping to a particular construct, it must agree with the construct on the number of inputs (arity). The mappings must also be defined on at least as many inputs as the construct.

We will permit implementations to be used to denote mappings, for example:

(λx. x)

can be stated to be the identity mapping. We permit the set of inputs such a mapping is defined on to be elided if clear from context (eg actual or implied types); if elided otherwise the set of inputs is assumed to contain first-class semantics, data, and '⊥'.

7.7.2 Implementations of mappings

A mapping usually has multiple implementations, some of which are interpretive

A given mapping can usually be implemented in a number of ways; ie there is usually a number of implementations whose sets of input-output pairs are a superset of the mapping. We do not place any particular limitations on what constitutes an implementation.

As an example, the implementation below, a lambda calculus term:

(λx. x)

implements the mapping whose domain is the Natural numbers, and which takes each Natural to itself, ie '0' to '0', '1' to '1' etc. This mapping has other implementations, for example one which step-wise simulates a Universal Turing Machine on the input, then throws away the simulation result and returns the input. One can (nearly) always add needless interpretation to an implementation, ie add some redundant calculation whose computational steps varies with the program's input. The 'nearly' in the above arises in practice due to the following two reasons.

The first is the possibility that the programming language chosen could be quite restrictive in the programs it allows. In all general-purpose programming languages however interpretation can always be added to programs without changing their mapping; hence at least one interpretive implementation will exist.

The second exception we have identified concerns mappings that return a non-'⊥' value when all inputs are '⊥'. In many programming languages no implementation of these mappings can investigate their inputs, as such an investigation would lead to '⊥' rather than a non-'⊥' value being returned. If inputs can't be investigated then input-varying computation is not possible; hence all implementations must be non-interpretive.

Finally, note that not all mappings have implementation. The mapping that maps numbers representing Universal Turing Machines and their inputs to a Boolean indicating whether the represented Machine would halt or not is a famous non-implementable mapping.

7.7.3 Mappings with only interpretive implementations

Many common mappings have only interpretive implementations

We come now to a key observation: some mappings have *only* interpretive implementations, recall that we term these mappings 'inherently interpretive'. Formally:

(inherentlyinterpretive M)

⇔

(¬∃ F: interpretationfreeimplementations . ∀<i,o>:M. (F i) = o)

Here 'interpretationfreeimplementations' denotes the hypothetical set of all interpretation-free implementations.

Hence, one can sometimes tell that an implementation is interpretive solely from its mapping, without having to determine or assume its computational steps. Conversely, if a given mapping is inherently interpretive, then we know it would be a waste of time to try to find an interpretation-free implementation of it, in *any* programming language. This is a benefit of keeping 'implementations' as inclusive as possible; another is that we maximise the chances of a mapping having a non-interpretive implementation.

We have identified a number of classes of inherently interpretive mappings, which interestingly cover functional programming and 'universal interpreters'.

The first class of inherently interpretive mappings are those whose outputs vary when different semantics are supplied as input, in such a way that we know that those inputs are applied. Application of semantics provided as input makes the implementation interpretive as the computations carried out during its execution will depend on its input. The difficulty in determining whether inputs are applied lies in excluding those mappings whose implementations could simply take the input semantics and place them into the output, without applying them.

Consider mappings that return '⊥' when given '⊥'. Where did the output come from, in a given implementation? Either the input was executed, or:

a. No execution of the input was done, as the output is just a copy of the input

b. No execution of the input was done, the output is independent of the input

We can rule out option 'a' if the specification is not an identity-mapping; and we can rule out option 'b' if the mapping is not a constant mapping to '⊥'. Formally, we claim that (for some set 'Semantics'):

$(\exists n, i_1..i_n, k.$

$<<i_1..i_{k-1}, \bot, i_{k+1}..i_n>, \bot>:M$

$\wedge \exists v:Semantics, w. <<i_1..i_{k-1}, v, i_{k+1}..i_n>, w>:M \wedge v \neq \bot \wedge w \neq \bot$

$\wedge \exists v:Semantics, w. <<i_1..i_{k-1}, v, i_{k+1}..i_n>, w>:M \wedge w \neq v$

$) \Rightarrow$ (inherentlyinterpretive M)

For example, consider the 'iszero' mapping (for Church Natural 'n'):

iszero n t f ≜ n (λx. f) t

This specification returns '⊥' when 'n' is '⊥', and is not an identity-mapping or a constant-mapping on 'n'; hence 'n' is applied and the computations done during execution of any implementation of 'iszero' will vary with 'n'. It should be noted that nearly all mappings which take semantics as input are detected as being inherently interpretive by the above. Execution of input-supplied semantics is interpretive, and appropriate variation of output with input is evidence of execution. Hence we can conclude that essentially all lambda calculus programs are interpretive.

The second class of inherently interpretive mappings which we have been able to identify consists of the mappings which, when given ground data (but not '⊥') , sometimes terminate and sometimes don't. Formally (for some set 'Data'):

$(\exists n, i_1..i_n, k, d:Data, e:Data, o.$

$<<i_1..i_{k-1}, d, i_{k+1}..i_n>, o>:M \wedge <<i_1..i_{k-1}, e, i_{k+1}..i_n>, \bot>:M \wedge o \neq \bot$

$) \Rightarrow$ (inherentlyinterpretive M)

The justification for this is that a non-terminating computation must engage in a different sequence of computational steps than a terminating computation, as the first is necessarily infinite and the second necessarily finite in duration. Hence we know that anything which can take an encoding of a machine or program as input and simulate its execution is interpretive, as long as sometimes the execution won't terminate. This includes the 'universal interpreters',

such as Universal Turing Machines. As another example, a 'Prolog' program that sometimes succeeds and sometimes doesn't terminate, depending on the input, is also interpretive.

More generally, if one had runtime information as well as the input-output mapping, one can argue that differing runtimes on differing inputs implies inherently interpretiveness, assuming each computational-step takes a fixed amount of time regardless of input. In 'Θ'-notation it would seem to be true that:

$\neg$ (inherentlyinterpretive M) $\Rightarrow$ M is $\Theta(1)$

M is $\Theta(>1)$ $\Rightarrow$ (inherentlyinterpretive M)

where '$\Theta(>1)$' means everything other than '$\Theta(1)$'. Note that determining the specific complexity is not required; determining that it is non-constant is all that is required.

A related argument (which we do not attempt to formalise) is as follows. Consider a mapping whose input is unboundedly-large (ie is a tree, list, or any other recursive type, including the Natural numbers). Any implementation of the mapping we assert must investigate its input by parts. Taking investigations as entailing computation exposes the mapping as inherently interpretive, if all parts of the input are looked at: inputs require a different number of investigations depending on their length. While it may be difficult in general to know whether all implementations of a given mapping must necessarily carry out a differing number of investigations depending on their input, in some cases it is clear that such must be true. For example, a mapping which sums the elements of an arbitrary-sized list is certainly inherently interpretive: different investigations (ie computations) occur for differently-sized inputs. In contrast, we cannot say with this reasoning that a mapping which sums the first (or last) ten elements of an arbitrarily-long list given as input is inherently interpretive.

The third class of inherently interpretive mapping discovered pertains to strictness. We consider now if a mapping has what we call the 'AIIS' ('all implementations inconsistently strict') property. By saying that an implementation is inconsistently strict, we mean that there is some part of the input which the implementation sometimes but not always investigates, and we argue that such implementations are interpretive. We propose that we can test for mappings that are AIIS using the following:

$$(\text{AIIS M}) \Leftarrow (\exists n, i_1..i_n, k, h_1..h_n, w, v.$$
$$\langle\langle i_1..i_n\rangle, w\rangle : M \wedge w \neq \bot$$

$\wedge$ <<i_1..i_{k-1}, $\perp$, i_{k+1}..i_n>, $\perp$>:M

$\wedge$ <<h_1..h_n>, v>:M $\wedge$ v $\neq \perp \wedge h_k = \perp$)

The first part of this finds a parameter implementations of the mapping will appear to be strict on (ie investigates), the 'k'th parameter. This is evidenced by a change of that parameter to '$\perp$' making the output change from a non-'$\perp$' value to '$\perp$'. However in the next line when the parameters are changed to the 'h', an implementation of the mapping will not appear to be strict on (ie investigate) the 'k'th parameter, as a non-'$\perp$' value is returned when '$\perp$' is supplied for that parameter. For implementations of the mapping that investigate their parameters sequentially, the above *appearance* of inconsistent strictness does imply *actual* inconsistent strictness. Taking investigations to entail computations, the fact that the parameter investigations done varies with input values implies input-varying computation, ie interpretation. For implementations that do not examine their arguments sequentially, implementations that appear by the above to be inconsistently strict may not actually be so: possibly all parameters are always investigated. We argue that input-varying computations must still be occurring anyway as follows. In such a non-sequential ('parallel') implementation, having:

<<i_1..i_n>, w>:M $\wedge$ w $\neq \perp$

$\wedge$ <<i_1..i_{k-1}, $\perp$, i_{k+1}..i_n>, $\perp$>:M

means that the 'k'th parameter is certainly investigated, as the output varies with it and all other parameters are kept at the same values. Further, the 'k'th parameter must be investigated forever when its value is '$\perp$': monotonicity implies one cannot 'time-out' an investigation, only abort it based on a discovered value of another parameter (but here all other parameters have the same values in both lines). Now, having

<<h_1..h_n>, v>:M $\wedge$ v $\neq \perp \wedge h_k = \perp$

means that the implementation doesn't investigate the 'k'th parameter fully (ie forever) even when it is '$\perp$', as the computations do terminate (ie a non-'$\perp$' value is returned). Instead the investigation is aborted based on the value of one of the other 'h's. Alternatively, the investigation of the 'k'th parameter is never begun. Hence, sometimes the 'k'th argument is investigated fully, and sometimes the investigation begins but is aborted, or never begins. Such is a difference of computation with input, ie interpretation. Hence, we claim that:

(AIIS M) $\Rightarrow$ (inherentlyinterpretive M)

For a concrete example of such an inherently interpretive 'parallel' mapping, consider the 'parallel-OR' operation. It returns 'true' if either of its parameters is 'true', 'false' if both are 'false', and '$\perp$' otherwise.

7.7.4 A useful property regarding inherently interpretive mappings

A mapping which includes an inherently interpretive sub-mapping must itself be inherently interpretive

Two useful properties of inherently interpretiveness can be noted. Firstly, if a mapping is a superset of an inherently interpretive mapping, then the mapping must also be inherently interpretive. This follows from the fact that any non-interpretive implementation of the larger mapping is also an implementation of the smaller. Formally:

$N \subseteq M \Rightarrow ((\text{inherentlyinterpretive } N) \Rightarrow (\text{inherentlyinterpretive } M))$

Secondly, if some mapping with an arity of two or greater is restricted to a mapping on fewer parameters, as indicated below, then the original mapping is inherently interpretive if the new mapping is. This is because any non-interpretive implementation of the original mapping, when passed a constant-value as the appropriate argument, is also an implementation of the new mapping. Formally:

$(\text{inherentlyinterpretive } \{<<i_1..i_{j-1}, i_{j+1}..i_n>, o> \mid \exists c. <<i_1..i_{j-1}, c, i_{j+1}..i_n>, o> : M\})$

$\Rightarrow (\text{inherentlyinterpretive } M)$

Hence we have a particular interest in *small* inherently interpretive mappings, which other mappings can be checked against in the above two ways.

7.7.5 Is any total mapping on data inherently interpretive?

We are currently unable to determine whether an important set of mappings are inherently interpretive

Some mappings are not inherently interpretive, for example identity mappings on data, and 'constant' functions (ie those that ignore their arguments). We find it hard with any confidence to state any others, because there is one important class of mappings whose status regarding inherently interpretiveness in unclear. Specifically, is *any* mapping whose inputs and outputs are primitive data (excluding '⊥') and finite in number, and is inherently interpretive?

7.7.5.1 Reason to think there are such mappings

Intuitively, one would think that some total mappings between data and data are inherently interpretive

It does seem likely that at least some test- and test-like mappings are inherently interpretive. Consider for example the mapping denoted by the following program-fragment:

IF b add divide

Here, if 'b', a Boolean, is 'true' then 'add' is returned, else 'divide' is returned; for execution by the surrounding program. Consider now replacing the above by a construct with the following semantics:

(λx,y. IF b (add x y) (divide x y))

This change won't affect the program; but does it permit a reduction in interpretation? Will eta-converting specifications so that they take and return primitive data (excluding '⊥') always give non- inherently interpretive specifications? We would like to believe not. However, note that the above program could be implemented so that it always executes *both* branches, and selects between the results. Hence there may well be a fundamental change in the computations possible.

7.7.5.2 Tests

Tests are a common sub-class of such mappings, and we identify one simple yet important test

Of interest to us is the particular sub-class of these mappings which corresponds to tests. Tests whose branches can be arbitrary code have inherently interpretive mappings, as noted previously. However, how about when the branches are primitive data? We focus here on a simple test, call it 'T', which simply takes three Booleans (or representations thereof) and whose input-output mapping is given by:

T true x y = x

T false x y = y

We consider the implications of the mapping 'T' being inherently interpretive, and of it not being inherently interpretive. In either case, the implications are profound, and suffice to answer the motivating question of this sub-section.

7.7.5.3 If it is inherently interpretive

If the mapping is inherently interpretive, many other common mappings can be shown to be inherently interpretive

If the mapping 'T' is inherently interpretive, then (by definition) all implementations of it are interpretive. Consider some of them now:

a. Lookup tables can be used to implement 'T'

b. The expression '(λc,a,b. c × a + (1 − c) × b)' implements 'T', where 'true' and 'false' are represented by '0' and '1'. Note that the addition could be replaced by a function that returns the maximum of its inputs.

c. The function '(λc,a,b. max (a + c − 1) (b − c))', where 'max' returns the maximum of its inputs and 'true' and 'false' are represented by '0' and '1', implements 'T'. Another is '(λc,a,b. min (a + c) (b − c + 1))', 'min' being a function which returns the minimum of its inputs.

d. Using the empty set to represent 'false' and a single-element set 's' to represent 'true', one can implement 'T' as '(λc,a,b. ((s − c) ∩ b) ∪ (c ∩ a))', where '−' is element-wise set-subtraction.

e. One way of implementing 'T' using Boolean operators is '(λc,a,b. (c ∧ a) ∨ ((¬ c) ∧ b)'.

Interestingly, assuming that 'T' is inherently interpretive, each of the above implementations of 'T' must also be interpretive, ie exhibit varying computation with input. The varying computation must be occurring in at least one of the constructs, in each implementation. Hence, assuming that we are correct in saying that 'T' is inherently interpretive:

a. Certain lookup tables are inherently interpretive.

b. One or more of multiplication, subtraction and addition, and one or more of multiplication, subtraction and maximum, must be inherently interpretive.

c. One or more of addition, subtraction, and maximum must be inherently interpretive, as must one or more of addition, subtraction, and minimum.

d. One or more of the set operations union, intersection and subtraction (ie set difference) must be inherently interpretive.

e. One or more of '$\wedge$', '$\vee$' and '$\neg$' must be inherently interpretive, as more generally must any complete 'basis' for Boolean logic

Also, any mapping which includes 'T' (in the senses detailed previously) we know will also be inherently interpretive. There are a *very* large number of such mappings; indeed we would go as far to say that most mappings between data one would meet in practice would contain 'T', and hence would be inherently interpretive if 'T' was.

As a point of interest, one may note that many of these implementations of 'T' have a commonality. This includes the implementation using Boolean operations. The commonality is that they involve selectively turning one branch into some form of 'zero' value, while the other branch unchanged. The two are then combined using a symmetric binary operation, which returns whichever input isn't 'zero'. As 'T' is (assumed to be) inherently interpretive, one or both of the 'zeroing' operation and 'combining' operation must be inherently interpretive.

7.7.5.4 If it is not inherently interpretive

If the mapping is not inherently interpretive, programming does not (theoretically) ever require interpretation

If the mapping of 'T' is *not* inherently interpretive, then, by definition, there exists a non-interpretive implementation of it. We now argue that this means that *every* program whose inputs and outputs are data, or can be represented as such, can be implemented (to a suitable level of approximation) without interpretation. The implementations are theoretically possible, but not generally practical.

Firstly, note that, in practice, we do not let our programs run for unboundedly-long periods of time. Hence one can impose some (sufficiently-large, say 100 years) cap on their runtime, making every program total. Similarly, the amount of input looked at and output produced by the program when running on computing hardware is certainly finite if the runtime is finite. Hence we can find an approximation to every program, an approximation which is total, takes finite input, and produces finite output. Now, note that every program so approximated can be implemented by a lookup table. The final step is to recognise that the lookup table in turn can be implemented by nested instances of the 'T' test, after representing the program's inputs and outputs as tuples of bits. That this can be done follows from 'T' being able to be used to implement the 'NOR' Boolean operator, which is a 'basis' for Boolean logic:

NOR x y = T x false (T y false true)

As 'T' is assumed not to be inherently interpretive there must be a non-interpretive implementation of it, hence any program can be implemented (to any desired level of approximation) without interpretation.

Note that even if such an implementation is theoretically possible, one may wish to disregard it as infeasible. One may desire some differing, subjective perhaps, notion of inherently interpretive, one which asks not whether there is some non-interpretive implementation, but whether there is some *feasible* non-interpretive implementation. We do not progress this thought further, but raise it for future work.

7.7.5.5 Additional importance of the test

The status of the mapping is the status of the main question

If 'T' is not inherently interpretive, then no total mapping between data is inherently interpretive (as, similarly to the above, 'T' can be used to implement all such mappings). On the other hand, if 'T' is inherently interpretive, then it is a witness that total mappings between data are sometimes (perhaps even mostly, as discussed previously) inherently interpretive. Hence considering 'T' alone will suffice to answer the motivating question of this sub-section, namely whether any total mappings between data (which include arithmetic, set, and Boolean operators) are inherently interpretive.

Overall, based on the above discussions, we believe that 'T' is likely to be inherently interpretive. This is backed up by consideration of Cellular Automata. Conceptually each 'cell' in a Cellular Automaton is a function from the values of neighbouring cells to the value of that cell. In the absence of clear computational steps, our characterisation of interpretation is somewhat difficult to apply. However as certain Cellular Automata are Turing-complete ([Wol02]), these functions are expected in general to be inherently interpretive. Further, as each cell can be seen to be able to be implemented as nested instances of 'T', this would imply that 'T' is inherently interpretive.

7.8 Applying inherently interpretiveness

The concept of inherent interpretiveness can be used to make judgements regarding interpretation not just for programs, but for fragments thereof and individual constructs as well

If the mapping of a program is inherently interpretive, then we know immediately that the program is interpretive, without having to consider its internals. However if the mapping is *not* inherently interpretive, it may still be implemented interpretively. And even if the mapping is inherently interpretive, one may still be interested in *localising* where in the program the interpreters are. In both cases, as we discuss below, one can apply the concept of inherent interpretiveness to the mappings of user-defined constructs, as well as the language's primitives (including 'ambient' semantics).

7.8.1 Analysing within programs

In order to analyse within programs without requiring details of the language's implementation, Conceptual Models (or equivalent assumptions) are required

Recall from Chapter 6 that little can be said, objectively and independently of the programming language's implementation, about a program. One can however make assumptions or Conceptually Model the program; for example assume that programs 'execute as written' (as per say their canonical operational semantics). Based on those assumptions or the Conceptual Model, one can identify 'ambient' semantics and individual constructs and their mappings used

in the program. Rather than needing to then determine or assume the computational-steps done by each in order to test for interpretiveness, one can turn to inherent interpretiveness.

Before checking constructs for inherent interpretiveness, the range of inputs for each needs to be determined. This should be precisely the set of inputs which are possible to ever be supplied to the construct during executions of the program. If we just asked whether some primitive's full mapping is inherently interpretive or not, and the answer is that it is, this would by itself be insufficient to say that the overall program is interpretive: for example it could be the case that the primitive is only ever passed one value as input. Hence we need to consider 'restricted' mappings, restricted to inputs which can occur during the program's execution. If at least one of the constructs does have an inherently interpretive 'restricted' mapping, then its computations vary with the inputs given to the program. Consequently one can validly say that the construct is interpretive in the program, and that the program as a whole is interpretive; even if the program's mapping is not inherently interpretive. Recall however that one also has the issue, raised earlier, of different instances of the same computational steps being possible; in practice this rarely occurs in such a way to render a program non-interpretive.

Conversely, if none of the constructs (and 'ambient' semantics if any) have inherently interpretive 'restricted' mappings, then one knows that each construct, as far as its semantics as used in the program are concerned, could be implemented non-interpretively. Hence, the program as a whole could be implemented non-interpretively, using its current source-code.

7.8.2 Choice of assumptions

Different assumptions can lead to different judgements as to where the interpretation is in a program

Unfortunately, different assumptions regarding what constructs have what semantics can result in significantly-different results about what part of a program is interpretive, and often no set of assumptions is clearly a more reasonable choice than all the others. As discussed previously, one cannot objectively and language-implementation-independently determine the computational steps of program constructs. For an interesting illustration of this, consider the program text:

```
F b (add x y) (mul x y)
```

Say we may know that if ‘b’ is ‘true’ the result is the same as ‘add x y’; and if ‘b’ is ‘false’ the result is the same as ‘mul x y’; for Natural numbers ‘x’ and ‘y’. Does the execution of ‘F’ involve looking at ‘b’, then picking the appropriate branch and executing it? Or does ‘F’ evaluate both of its branches, then pick between the results based on ‘b’? Or, does ‘F’ simply return one of its arguments for another construct to execute? Perhaps ‘F’ is supplied with its arguments already fully evaluated. This ambiguity is reflected in the type of ‘F’. It could be:

Boolean —> Natural —> Natural —> Natural

or

Boolean —> Semantics —> Semantics —> Semantics

or

Boolean —> Semantics —> Semantics —> Natural

Each choice results in (possibly program-input -varying) computational steps occurring in different constructs (and/or ‘ambient’ semantics). One should also note that the above indicates the need to be careful when assuming mappings for constructs: the assumptions need to be consistent with each other, one cannot simply treat each construct individually, picking for it some view of it which maximises the chances of its mapping being non-inherently interpretive.

However, sometimes there is minimal ambiguity. For example, it is sometimes the case than an ‘if’ test could not possibly have both its branches executed. Use of ‘if’ tests to guard a loop or recursion is a good illustration of this. If both branches of such tests were executed, the program would theoretically never terminate. Hence the ‘if’ test must not be supplied with the recursive branch (at least) evaluated, and must not always evaluate that branch itself. As the recursive branch is selectively executed, such occurrences of ‘if’ tests are always inherently interpretive.

7.8.3 Application to functional programming languages

First-class functional programming is interpretive

Non-constant lambda calculus functions, assuming they execute as functions, ie ‘as written’, are typically interpretive. The same goes for terms in other functional languages; the problem is support for first-class semantics. We know that (nearly) all specifications whose inputs are first-class semantics are inherently interpretive. As should be the case, the canonical Conceptual Models are consistent with this. A lambda-term ‘(λx. M)’ is viewed in these models as taking

first-class semantics as input. If 'x' is ever applied within 'M' (note that in the particular example of the λ-I calculus, this is *always* the case), then the semantics of 'x' will be executed. Hence the computations done in '(λx. M)' will depend on its input, 'x'. Application of functions supplied as arguments makes the program interpretive if those supplied arguments vary with the program's input. In any class of language implementation or Conceptual Model where what is written as functions act like functions, programs written using first-class functions are interpretive. One might consider an alternative class of language implementation, or a different Conceptual Model of lambda calculus execution, one in which 'ambiently' 'x' is bound to its value, *before* invoking the function-body. Then, the function-body doesn't actually have any inputs as such, and therefore can't have input-varying computations. Similarly, one could pick a Conceptual Model in which lambda calculus programs are executed via syntactic transforms done by the 'system'; ie so all semantics are 'ambient'. Where has the interpretation gone in these cases? The answer is that the entire system is still interpretive: the input-varying computation has been moved to 'ambient' semantics, in these cases it is not the program but the language which is interpretive, in some sense.

7.8.4 Application to other than functional programming languages

Many key constructs used in non-functional programs are interpretive

Occurrences of 'if' tests, conditional jumps, guarded loops, etc in a (Conceptual Model of a) program will typically make it interpretive. Whenever constructs have inherently interpretive 'restricted' mappings, their implementations will engage in computations that vary with the input to the program. Nearly all non-trivial programs will contain (non-degenerative) uses of these constructs. However, some simple programs will not contain any of these constructs, and may contain only constructs with mappings whose inherent interpretiveness we do not know. For example, a program may take a number as input and do only simple arithmetic on it. Recall that it is uncertain whether some or all of the specifications of total mappings between data are inherently interpretive or not.

7.9 Removing and using less interpretation, and its positives and negatives

Interpretation in programs can be reduced by using TFP-style programming

In this sub-section we discuss the positives and negatives of interpretation, before moving on to how to compare computational-systems with regards to interpretation, and making them less interpretive. We also detail how TFP-style programming is less interpretive than the standard data- and test- based programming style.

7.9.1 Positives of interpretation

The prevalence of interpretation is explained by the fact that the computations of interpretive code are input-dependent, such being not only useful but indeed required for nearly all non-trivial programming

Interpretation can quite rightly be viewed positively. The advent of von Neumann -style computing, where a computer's computations can be varied by its input (ie its memory-contents), is considered to be a major advance. Input-varying computation is an integral part of programming: almost always, we *want* our programs to do different computations depending on the input given. Interpretation is prevalent for good reason.

For individual constructs, to be able to do different computations depending on the input supplied (which may itself vary with the program's input) is advantageous with regards to reuse. For a simple example, note that a typical implementation that inserts an element into a sorted list or heap typically does different things depending on whether the sorted list or heap is empty or not. Being able to call a single routine to do the insert is preferable than the user needing to determine whether the data-structure is empty or not, and call the appropriate routine. Similarly, one often programs not a (possibly very large: see eg [KEH91]) set of related procedures, but instead a single procedure with a 'flag' argument which modifies its behaviour. Related to this is 'ad hoc' polymorphism, where a choice between eg a floating-point 'square-root' operation or a 32-bit integer 'square-root' operation is made based on a data-tag indicating the type of the input.

Finally, consider language implementations, which are important artefacts in computing. Language implementations are certainly interpretive: their computations are those of the program whose source-code is given as input, and hence vary with that source-code. For language-extension done via hosting code, interpretation is again important. Definitional-style hosting code acts as assertions that particular guest constructs always have particular computations; compilers can use this information to replace the constructs by those computations. In contrast, interpretive-style hosting code is ('as written') a function which takes a representation of the guest code as input, and does varying computations depending on it. For an example of this, consider interpretive-style hosting of parallelism using a sequential host programming language. One represents the guest code as data, and then uses a function to do 'dovetail' execution.

7.9.2 Negatives of interpretation

Analysis of code which features input-dependent computations can be difficult; making the input look (and act) like a program minimises the difficulty

Based on our characterisation, interpretation is to be avoided if and only if input-varying computation is to be avoided. Having some part of the system engage in different computations makes many analyses dependent on being able to determine what computations are done for what inputs. Such analyses are also less useful in their conclusions than those for non- input-varying computations, for:

a. To know what computations are done, one can need to know the value of the input, or at least which of some sets it is drawn from

b. If one wants to prove a property is true for all executions ie all inputs, one needs to show individually that it is true for each computation, the number of which can compound exponentially. Alternatively, one can show that all possible computations are satisfactory

Many factors affect how difficult interpreters are to analyse. Firstly, if the computations done by the program vary 'smoothly' or 'continuously' with change of input, analysis is typically easier than if there are 'discontinuities', especially when at each input. Similarly, the wider the range

of computations which can occur in the program, the generally harder it is for the program to be analysed. Type-systems can play an important role in easing the determination of the range (and, 'continuity') of the computations, for example if in:

(λa,b. a succ b)

the variables 'a' and 'b' are typed as Church Naturals then we know the term will only engage is some number of applications of 'succ' to 'b'. Secondly, while the variance of computation with input in any single part of the program may be minor, these can be connected together in such a way that analysis becomes difficult or even impossible. The simple computations of a Universal Turing Machine and its resultant undecidability come to mind as an example. Finally, we note that while syntactically a program may superficially appear able to express a wide range of computations (eg many general-purpose interpreters are used); only some or one of them may actually be executed, and only on a limited range of inputs: syntactic complexity but semantic simplicity. As an example of this, one can consider a program '(λa,b. F 44 a b)' which uses a complex interpreter 'F' whose range of return-values is the set of all lambda-terms.

Generally, difficulties of analysis impact on software quality (as argued in eg [HS93]) and result in complex interpreters being hard to maintain. We contend that it is both the difficulty of analysis, and the harder-to-use results of such analysis, which are the main negatives of interpretive systems.

Interestingly, note that the easiest programs to analyse are those in which constructs have distinct semantics, and where the interaction between constructs is simple: hence small changes to the source-code lead to small changes of semantics. Programming languages have evolved to maximise this 'continuity' of semantics/computation with input, with compositionality playing a key part. They have evolved so that the input-output semantics, and computations, are as easily-determined as possible from the input. This is exactly what one wants for ones' interpreters. Hence to minimise the costs of interpretation, interpreters should be implemented so that the interpreted part of the inputs looks like programs written for high-quality programming languages. Additionally, as pointed out to us by P.A. Bailes, that part of the input, as a 'program', should itself also be as little in the interpretive style as possible: the same considerations of this subsection, and Chapter 6, apply to it. For example, depending on the situation, it should have part of its source-code be syntactically-contiguous hosting code, in the definitional style, and it should avoid use of representations.

7.9.3 Comparing systems with regards to interpretation

To compare systems with regards to interpretation both positives and negatives should be taken into account, but it is more practical to only consider the negatives, approximately

To properly compare two computational systems (eg programs) with regard to interpretation, one must take into account both the positives and the negatives of the interpretation in each. This can be quite difficult in practice, as the costs and especially the benefits depend heavily on the particular circumstances, and are generally hard to determine.

There are also simpler and more practical approaches. For example, one can ignore any positives and focus only on the negatives and simply ask: which system is the easier to analyse? Similarly, one can compare computational systems, approximately, based on the number of changes of computations that occur with change of input. Say one had a computational system which engages in a different sequence of computations for each input. If one took this system and modified it so that some of the inputs now shared the same computational-sequence, then the new system could validly be considered 'less interpretive' than the old. Another example of an approximation is that one can consider a computational system 'less interpretive' than another if strictly fewer of its parts contain computations that vary with the input; this requires matching up parts between the systems. Consider:

(λx. F (G (H x)))

This is 'more interpretive' than

(λx. F x)

if 'G' is an inverse of 'H', and 'G' or 'H' have computations that vary with 'x'. Note that the computations done by 'F' (ie whether they vary with 'x' or not) are the same in both systems, hence are irrelevant. Similarly, one can consider a computational system 'less interpretive' than another if it features a lower *total number* of computational steps that are involved in variation of computation with program input. This permits a system to in some places have more program-input varying computations than another, and in some places less, but still be able to be compared to it with regards to interpretiveness. Due to excellent applicability to the problem of using less interpretation in programs, and independence of anything other than the systems being compared, these approximate methods are one ones used for the remainder of this section.

7.9.4 Removing all interpretation

If a mapping isn't inherently interpretive, theoretically one can remove all interpretation from any implementation of it

Any system that is interpretive but whose mapping is not inherently interpretive we term 'needlessly interpretive'. Often such systems engage in different but semantically-equivalent computations depending on the input provided. For example, one can implement the identity mapping on Naturals as

(λx. if x == 0 then x × 1 else if x == 1 then x × 2 - 1 else if x == 3 then 3 else x)

which does different computations depending on its input. As a program it is interpretive, needlessly so.

Conversely, if the mapping of an implementation *is* inherently interpretive, interpretation will not be able to be completely removed. However it might be able to be lessened, as described below.

7.9.5 Using less interpretation

The techniques of Chapter 5 can be used to reduce the amount of interpretation in programs

As mentioned above, a program can be considered less interpretive than another if it exhibits fewer computational steps involved in program-input -varying computation; ie computational steps with the property that they may or may not be executed, or be executed in a different sequence, depending on the input to the program. Hence if a program is transformed so that it computes the same mapping but has fewer of those computational steps, the amount of interpretation done by the program has been reduced.

Firstly, consider programs that call particular functions (not necessarily with input-varying computations) in a sequence which varies with the program's input. One has program-input -varying computation as each of the functions entails computation. If any of those functions can be simplified so as to do less computation, interpretation has been reduced in the program. Similarly, applications can be simplified to reduce interpretation. This includes applications that

are only formed at runtime: one may recall from Chapter 5 that there are techniques by which programs can be transformed so that applications that are formed at runtime instead appear statically, and hence can be simplified. We discuss 'BVE' and 'AAE' specifically below. Apart from localised simplifications, such as of functions and of applications, one can also consider whole-of-program techniques. For example, one can remove repeated computations. Rather than repeatedly calling a function 'F' on a value, one could call 'F' on the value once (this application could possibly be simplified) and pass around the instead. Other techniques have been discussed in Chapter 5.

Secondly, consider a function in a program which has program-input varying computation, ie executes different computational steps depending on its input which in turn depends on the input to the program. If one can simplify this function so that it does fewer computations, even just on some inputs, then one has reduced the amount of interpretation in the program. Similarly if the function is used in a repeated computation, then removing the repetition as per above will also reduce the amount of interpretation in the program, even without simplifying the function. This 'pre-interpretation' has been discussed in Chapter 5.

Finally, one should note that the techniques described below can preserve the positives of interpretation while reducing the negatives. Constructs that we want do different things on different inputs can be kept unchanged extensionally, while involving fewer computations (especially fewer input-varying computations) makes them easier to analyse and maintain.

Below, after revisiting 'AAE' and 'BVE', we consider specifically if use of Data-Alternatives (ie TFP-style programming) can make programs less interpretive. We conclude that it can justifiably be said that TFP-style programming is a less-interpretive style of programming.

7.9.5.1 Using the 'AAE' and 'BVE' transforms

The 'AAE' and 'BVE' transformations both simplify programs and permit further program simplifications, hence can be used to remove computations that are program-input -varying

Consider the 'BVE' and 'AAE' transformations from Chapter 5. They permit applications that would otherwise only be formed at runtime, ie dynamically, to be present statically. They can then be simplified, which may involve removal of program-input -varying computations via eg

evaluating them. For an example, consider the following program-fragment, whose input is typed to be a Church Boolean:

(λb. b add sub 6 4)

We assume that differing program-inputs will result in the fragment being executed on differing values of 'b'. Depending on the value of 'b', either 'add' will be applied to '6' and '4'; or 'sub' will be. We assume that 'add' and 'sub' entail different computational steps. Hence, this program-fragment is interpretive. Via 'AAE', it can be transformed to:

(λb. b (add 6 4) (sub 6 4))

Simplifications can then take place, to give:

(λb. b 10 2)

Here, 'add' and 'sub' have been evaluated-away: their computations have been fully removed. Commonly however, the evaluation will be only partial. Even in those cases, the simplification will still achieve a decrease in the number of computational steps.

Finally, from the derivation of them, 'BVE' or 'AAE' can be seen to reduce the number of computations done in terms. For a concrete example, consider the following function, which takes two Church Naturals and two other functions as input:

(λa,b. a (λn,f,x. f (n f x)) b)

This is transformable via 'BVE' (after an eta-expansion) to:

(λa,b,f,x. a f (b f x))

This second function implements the same mapping but does strictly fewer computations, as evidenced by the two, on application to a single Church Natural, being related by '→*'.

7.9.5.2 Using Data-Alternatives

Using Data-Alternatives instead of data can make programs less interpretive

We now discuss how replacing data by Data-Alternatives can often make programs less interpretive. Due to the variety of Data-Alternatives possible, the discussion is necessarily general. We give a specific example later, when detailing reducing interpretation by use of TFP-style programming.

In a typical program one has data, constructed by data constructors, and functions that consume that data. If one rewrites the program to use Data-Alternatives instead of data one will have

Data-Alternative Constructors, and Data-Alternative Non-Constructors. The Data-Alternative -based program will be less interpretive than the original when the Data-Alternative Non-Constructors exhibit fewer program-input -varying computations than the functions that consume the data; unless this is outweighed by the Data-Alternative Constructors having more program-input -varying computations than the data constructors. Generally one would not expect implementations of data constructors to do computations that vary with their input, but this may not be true for the Data-Alternative Constructors. We discuss this further below. Even if so, Data-Alternative Non-Constructors often exhibit less program-input -varying computation than the functions that consume the data. If there is one function that consumes the data, and it does so by a large number of input-varying computations such as recursive pattern-matching on the data constructors, this is likely to be the case. This is especially true when the corresponding Data-Alternative Non-Constructor is the intrinsic operation of the Data-Alternative, implemented trivially (ie as the identity function).

For an example of a change to Data-Alternatives that makes such programs more interpretive, consider representing binary trees as Gödel numbers, Church Naturals. Most operations on the trees would have to be reimplemented using quite complex code, ie potentially quite large amounts of program-input varying computation. However, note that the operation which returns the successor of the Gödel number of a given tree will become much simpler to implement. Similarly 'pred' on Church Naturals requires more computation than when certain other functional representations of Naturals are used, such as 'succ n ≜ (λs,z. s n), zero ≜ (λs,z. z)'. However 'iterate' on Church Naturals is trivially implementable, and operations that can be defined in terms of it such as 'add' can be implemented fairly simply as we've seen. Even when switching to a different data representation rather than using Data-Alternatives, certain operations can become require more computational steps to implement. Consider representing Naturals as lists of digits. A unary representation permits subtraction to be performed via simply taking the tail of the list; unlike with a binary representation. Similarly division by two with any remainder being discarded can be done via removing the head of the list in binary, but in unary is more complex. It is apparent that depending on the choice of value (data or Data-Alternative), there is a differing 'envelope' of operations that can be implemented at no or little computational cost.

The choice of value, data or Data-Alternative, is hence crucial for attempts to reduce the amount of interpretation done by a program. A poor choice will mean more computations, computations that are often program-input varying. Generally, one would want to pick a Data-Alternative which minimised the total number of occurrences of program-input varying computations. One should note that as many operations are Structurally Recursive (ie 'fold'-expressible), Church Objects can be seen to be a good choice of Data-Alternative to have less-interpretive programs. Recall that 'fold' involves large amounts of input-varying computation (ie by being a recursive pattern-matcher), but is trivially implementable on Church Objects. Hence 'pure' (ie data-free) TFP-style programming, programming with Church Objects, is often substantially less-interpretive programming. In addition TFP-style programming in the general sense, ie programming with Data-Alternatives, can be substantially less-interpretive programming.

7.9.5.3 Details regarding constructors

Church Object Constructors like data constructors are apparently not inherently interpretive

Consider building a value using data constructors versus building one using Data-Alternative Constructors, with regards to program-input -varying computation. Note that 'zero-arity' constructors are not of interest here, as they don't take input.

To begin with, consider two programs one of which uses data constructors and the other Data-Alternative Constructors but are otherwise equal. If the sequence of constructor-calls varies with the program's input then it will vary equally in both programs. In these cases if Data-Alternative Constructors do more computational steps than data constructors, which is often the case, the Data-Alternative Constructors will contribute more program-input -varying computation to the program than the data constructors.

Secondly, consider data constructors against Data-Alternative Constructors with regard to input-varying computations. One would think that data constructors do not necessarily involve input-varying computation, ie are not inherently interpretive. Generally they are conceived as placing their arguments or references thereto into a (eg 'linked') structure without looking at them, ie they do the same computations regardless of input. In contrast Data-Alternative Constructors take first-class semantics (ie functions) as input. Given the variety of Data-Alternatives, some Data-Alternative Constructors would be inherently interpretive. Church Objects are of particular

interest to us in this work, so we consider them specifically now. Consider the Church Natural Constructor for 'successor' (other Church Object Constructors follow correspondingly), defined canonically as:

succ ≜ (λn,f,x. f (n f x))

Execution of this on an 'n' would usually involve execution of 'n'. However, if appropriate typing is done (Chapter 4 shows that suitable type-systems exist) a smart compiler could avoid this execution and jump straight to the end result. For example, if it knows that 'n' is a Church Natural, it could simply prefix another instance of (or reference to) the first variable of 'n' to the body of 'n'. This can be done without investigating 'n' hence the computation is not input-varying. The simplicity of it indicates that use of Church Object Constructors will not necessarily result in more program-input -varying computation than use of data constructors. Note that a smart compiler is not necessarily needed if suitable constructs are added to the programming language that can express such syntactic changes.

Consequently one can note that if all one is going to do with the data produced by the data constructors is call 'fold' on it (which is quite common), use of Church Objects ie TFP-style programming is a less interpretive alternative. As discussed above the constructors are not inherently different with regards to input-varying computation. The difference comes with 'fold': 'fold' on data has input-varying semantics (as evidenced by a different depth of recursion will be reached on different inputs), while 'fold' is trivially implementable on Church Objects.

7.10 Further theoretical work

Analysis of a class of simple computational systems offers the possibility of future theoretical advances

It may be recalled that we lack the techniques to determine which mappings from finite data to finite data are inherently interpretive; and we identified that simple such mappings were especially important. We present now what we believe is the *simplest possible* class of those mappings. This class of mappings take inputs and give an output (or a tuple of outputs; logic gate-like) whose values are drawn from a very peculiar data-type. Perhaps-surprisingly, the standard binary or Boolean data-type is not the simplest possible, there exists a *sub-binary* data-type (indeed, it can be proven that it is the *only* non-trivial sub-binary data-type). The two

values of this data-type are '⊥', meaning as per Domain Theory 'no information (yet)', and a data-value we label 'i'. What makes this data-type sub-binary is that a single variable of this type holds less than one bit's worth of information, ie cannot hold a Boolean value: we can detect when the value is 'i', but as per monotonicity '⊥' cannot be detected. Hence these sub-binary values are the natural values of so-called 'semi-decision' procedures, ie those that will either halt (indicating success), or not. This data-type and associated operations as described that we have discovered we not seen elsewhere. There is however superficial similarities with the system described in [FB96], however in that system the value corresponding to '⊥' is detectable.

To represent values from other data-types in sub-binary, one can be guided by the intuition that 'i' can be used to represent whether the particular value (piece of information) is present or not. For example, one can represent a Boolean variable by two sub-binary variables (or 'channels'), one which is 'i' if and only if the value of the Boolean is 'true', and the other 'i' if and only if the value of the Boolean is 'false'. We thus say that 'true' is represented by 'i⊥', and 'false' by '⊥i', tuples being implicit. Note that 'ii' would be equivalent to the top element, and '⊥⊥' to the bottom element, of the original Boolean Domain. This representation of Booleans can be readily proven to be the simplest possible in sub-binary. Any other representation leaves one or both Boolean values undetectable (eg we can't distinguish 'ii' from 'i⊥') or by removing some of the channels, the above representation (modulo changing the order of the channels) is reached. We will use subscript notation to refer to the individual sub-binary values that make up a tuple representing a value from some other data-type. For example the first sub-binary value of a representation of 'v' is 'v_1' and the second is 'v_2'.

As indicated previously, we can denote mappings which take (finite in number) sub-binary inputs and give one or more (finite in number) sub-binary outputs. We use a simple tabular notation for this. For example, the Boolean '∧' mapping in sub-binary is:

x	y	(x ∧ y)
i⊥	i⊥	i⊥
i⊥	⊥i	⊥i
⊥i	i⊥	⊥i
⊥i	⊥i	⊥i

In order to implement mappings, one requires constructs. There is one natural choice for such; two simple constructs which, it can be proven, suffice with 'wiring' (eg 'T'-splits) to implement *any* (finite) monotonic sub-binary mapping. We call these two constructs '$OR^{i\perp}$' and '$AND^{i\perp}$'. Their mappings are:

x	y	$(OR^{i\perp}$ x y)
$\perp$	$\perp$	$\perp$
$\perp$	i	i
i	$\perp$	i
i	i	i

x	y	$(AND^{i\perp}$ x y)
$\perp$	$\perp$	$\perp$
$\perp$	i	$\perp$
i	$\perp$	$\perp$
i	i	i

Note that one cannot simulate an '$OR^{i\perp}$' with '$AND^{i\perp}$'s and 'wiring' alone or vice-versa: the two are orthogonal. For an example of an implementation of a mapping, the first output-channel of the '$\wedge$'mapping above can be implemented by:

$$x_1 \; AND^{i\perp} \; y_1$$

and the second by:

$$x_2 \; OR^{i\perp} \; y_2$$

ie:

$$(x \wedge y)_1 = x_1 \; AND^{i\perp} \; y_1$$

$$(x \wedge y)_2 = x_2 \; OR^{i\perp} \; y_2$$

For a more significant example, ‘IF’ on Booleans has the sub-binary mapping:

c	a	b	(IF c a b)
i⊥	i⊥	i⊥	i⊥
i⊥	⊥i	i⊥	⊥i
⊥i	i⊥	i⊥	i⊥
⊥i	⊥i	i⊥	i⊥
i⊥	i⊥	⊥i	i⊥
i⊥	⊥i	⊥i	⊥i
⊥i	i⊥	⊥i	⊥i
⊥i	⊥i	⊥i	⊥i

This can be implemented as:

$$(\text{IF c a b})_1 = (c_1 \text{ AND}^{i\perp} a_1) \text{ OR}^{i\perp} (c_2 \text{ AND}^{i\perp} b_1)$$

$$(\text{IF c a b})_2 = (c_1 \text{ AND}^{i\perp} a_2) \text{ OR}^{i\perp} (c_2 \text{ AND}^{i\perp} b_2)$$

Note that some mappings are unimplementable, these are precisely the mappings that are non-monotonic. For example:

a	(¬ a)
i	⊥
⊥	i

is non-monotonic and cannot be implemented.

The initial definitions and examples now over, we move on to details of interest. Finite sub-binary mappings can be readily tested for inherent interpretiveness, via checking for AIIS: such a check can be done mechanically. Now, mappings which are AIIS require ‘$OR^{i\perp}$’s to implement: using ‘$AND^{i\perp}$’s alone, it is impossible to produce an implementation with an AIIS mapping. Consistency would seem to require that as the mappings that are inherently interpretive require ‘$OR^{i\perp}$’ to implement, ‘$OR^{i\perp}$’ must be inherently interpretive. Indeed, this is the case: the mapping of ‘$OR^{i\perp}$’ is AIIS and independently able to be established as inherently interpretive in the same way as done earlier for parallel constructors such as ‘parallel-OR’. We believe, due to its intrinsic simplicity, that ‘$OR^{i\perp}$’ is *the* simplest interpreter. As for ‘$AND^{i\perp}$’, an implementation of that seems to exist that is interpretation-free: simply wait for the first argument to be ‘i’, then wait for the second to be ‘i’. As ‘$AND^{i\perp}$’ and ‘wiring’ suffice to

implement all (monotonic, ie computable) non-AIIS mappings, non-AIIS mappings are *not* inherently interpretive.

Hence mappings whose (finite in number) inputs and outputs are sub-binary 'channels' can be *completely* characterised with regards to inherent interpretiveness, via a mechanical test. Further, implementations themselves can be *completely* characterised into the interpretive (ie containing '$OR^{i\perp}$') or interpretation-free (ie not containing '$OR^{i\perp}$'). One does however have to check in the first case that at least one '$OR^{i\perp}$' is exercised over a sufficient range of its inputs in a given program, else the situation may be one where an interpretive construct is used in a context such that only one of its differing computations is ever carried out.

Interestingly, all non-constant and non-projection (modulo negation) mappings from Booleans to Booleans are readily established as being inherently interpretive; and the same goes for other data-types: they all require '$OR^{i\perp}$' to implement. It should be noted that this is consistent with the 'T' test (introduced earlier in this work) being inherently interpretive. One may additionally note that the 'zeroing' operations identified with that test conceptually align with '$AND^{i\perp}$', and the 'combining' operations with the 'confluence' of '$OR^{i\perp}$' (as described later).

Interestingly, one can 'normalise' implementations. Any implementation can be rewritten as a composition of '$OR^{i\perp}$'s and '$AND^{i\perp}$', ie with inputs being '$AND^{i\perp}$'d together and then those results combined with '$OR^{i\perp}$'s. In such a normal form, assuming no redundancy (such is readily detectable), the number of '$OR^{i\perp}$'s, ie interpretive calls, is minimised.

A question which we are not able to answer with confidence is what all this means for programming with standard data-types. If one takes a mapping and shows that it is inherently interpretive in sub-binary, can one then say that the original mapping is also inherently interpretive? To argue so, one would have to make the case that all information is *fundamentally* presented as atomic pieces of information which are either present or not (yet) present (ie as represented by 'i' and '$\perp$'). Alternatively, one may be able to show that any implementation would be *at least* as interpretive as an interpretation-minimal sub-binary implementation. If one could make this argument all non-trivial programming would be exposed as interpretive. Further, one would also be able to make the case that interpretation can be identified with the presence of '$OR^{i\perp}$'. This raises the interesting possibility that there exists an equivalent but far

more fundamental characterisation of interpretation: that interpretation is 'confluence'. The meaning of 'confluence' is the combining of two inputs into one analogously to two streams of water coming together: whenever either channel has a value, that value is placed on the output. This is precisely, and only, what '$OR^{i\perp}$' does: '$OR^{i\perp}$' *is* 'confluence', of the most fundamental (ie sub-binary) sort. That interpretation could be characterised as being to do with data rather than computation is certainly an interesting thought.

7.11 Conclusion

Interpretation has an objective characterisation with a myriad of practical and theoretical implications

In this Chapter we investigated a number of hypotheses relating to interpretation, before presenting our proposed characterisation of it. Characterising interpretation as input-varying computation not only exposes symbol-based computation (eg testing symbols) as interpretive in line with intuition, but also doesn't require any identification of symbols or functions that can act as such to be made. The characterisation is very consistent with common uses of the word 'interpretation' and we would even go so far as to suggest that it is *clearly the correct* one. Based on this characterisation, most programming was exposed as interpretive. We also developed a somewhat-complementary property, 'inherent interpretiveness'. This property is quite useful in that it helps bridge the gap between how programs *execute* and how they are *written*.

The costs and benefits of interpretation (as characterised) were discussed, together with means of comparing interpretive programs and lessening the amount of interpretation in programs. Some novel theoretical work involved with fundamental aspects of computation was also presented. While interpretation-free programming may be infeasible, we showed that TFP-style programming can lead to less-interpretive programs.

Chapter 8

Summary

Theoretical and conceptual underpinnings exist for a practical style of programming substantially liberated from interpretation

We began this work by introducing interpretation, and then a methodology for eschewing it, TFP. A number of open questions and lacunae were noted for TFP, some of which later Chapters resolved. Below, each Chapter is summarised. We conclude with details of how our research has advanced specific topics within TFP.

8.1 Interpretation

Interpreters are prevalent, often implicit, and significantly complicate programming

In Chapter 1, we introduced interpretation. We discussed symbol-based computation, and explicit and implicit interpreters. Also covered was how interpretation is prevalent, and can lead to a difficulties relating to static analysis etc. These costs of interpretation, combined with its prevalence and the existence of underused alternatives to it, indicate its status as a problem requiring redress.

8.2 Totally Functional Programming

TFP is a methodology that attempts to minimise interpretation in programming

In Chapter 2 we introduced Totally Functional Programming, TFP. We began by detailing its development, including observations regarding Platonic Combinators and Characteristic Methods, that one can view programming as language-design, and that there are particular program fragments that apparently illustrate how to program with less than the usual amounts of interpretation. TFP was then presented as the synthesis of those observations. Finally, the various expected outcomes from TFP were described.

8.3 Key outstanding issues

TFP, while promising, contains a number of lacunae in its formulation to date

In Chapter 3 we detailed a number of lacunae within TFP's current formulation. Difficulties surrounding producing formal semantics for TFP were discussed in detail, as were suggestions for developing TFP languages. Indicated to be addressed in the remainder of this work were the issues of the need to identify a type-system for TFP, to explain how to derive the examples of apparently less-interpretive programming, to distinguish in detail interpretive from definitional language-extension, and, of paramount importance, the requirement for the intrinsic nature of interpretation to be discovered.

8.4 Types for TFP

The 'System F' and above type-systems are suitable for TFP

In Chapter 4, we discussed possible type-systems for TFP languages, which are required for making TFP programming practical. We were able to identify some suitable type-systems. The Chapter began with a discussion of our motivations for wanting a type-system, and the major type-systems for the lambda calculus were then introduced and evaluated against some general and TFP-specific criteria. Using the examples of TFP-style programming we were able to show that many of those type-systems are unsuitable, and that TFP requires 'F_2' or higher polymorphism with partial type-inference. Even then, as we discussed, some TFP-style programs are still not be typable: but this was considered an acceptable loss. We also considered some less common type-systems.

In the Appendix, the unsuitability for TFP of perhaps the most common type-system for functional programming, Hindley-Milner typing, is expounded in detail.

This Chapter validates the hypothesis in TFP that appropriate (albeit not perfectly so) type-systems exist for TFP-style programming, and identifies some.

8.5 Derivation of the TFP-style programs

Formal derivation of TFP-style programs is possible by application of both well-known and novel techniques

In Chapter 5, we found means to derive TFP-style programs, including the examplars of (apparently) less-interpretive programming. Finding such means was a central requirement for making TFP-style programming practical. We began by listing a number of derivation techniques, then showed how many could be unified using the concept of parameterising constructors. Two new transformations ('BVE' and 'AAE') were subsequently introduced, which served to complete our ability to, often-mechanically, derive all of the identified examples of TFP-style programming from Chapter 2.

This Chapter therefore validates the hypothesis that TFP-style programs can be systematically derived; and provides concrete means for doing such derivations.

8.6 Interpretive- versus definitional- style language-extension

Theoretical rationales exist for preferring definition over interpretation; these include the typically-limited expressivity of interpreters with regards to language-extension

In Chapter 6 the use of interpretation as a language-extension technique was considered in detail, with interpretive-style hosting code contrasted with definitional-style hosting code. After first supplying a general framework suitable for the analysis we wished to conduct, we determined the objective differences, independent of details of programming language implementations, between interpretive and definitional styles of language-extension. By introducing a model of the hosting process we were able to identify a number of additional, albeit subjective or dependent on the programming language's implementation, differences between the two styles. We then showed that these differences were in fact a matter of degree and not schism, and the apparent dichotomy arises from peculiarities of common programming languages in use today. Possible characterisations for interpretation were also discussed. Specifically, use of representations could be considered interpretive. Alternatively, interpretation could be characterised as simply 'poor' language-extension, ie a hosting which satisfies only some of our

desires with regards to observable outputs (including when due to the use of representations). Or, one could characterise interpretation as the turning of representations into what they represent. More generally, we now see that there is some scope for argument as to which of the following the central interest of TFP should be:

1. The *use* of representations (most commonly inanimate data or functional representations of data)
2. The *means* by which representations are turned back into what is represented (or a simulation thereof), or
3. What the underlying meaning of the word "interpretation", as consistent with wide-ranging uses of that word, is (we propose: 'input-varying computation')

This Chapter validates the idea in TFP that 'programming is language-extension'. It does however argue against the postulated interpretation-versus-definition dichotomy in TFP. That two distinct styles of language-extension exist is, unexpectedly, not due to any great theoretical schism. The Chapter also suggests that interpretation and poor language-extension are two *distinct* topics, which have been perhaps mistakenly conflated in TFP.

8.7 The nature of interpretation

An interpreter is something that does different computational steps depending on its input

In Chapter 7, we turned our attention to the key outstanding question of TFP, the nature of interpretation. We introduced our criteria, then considered various hypotheses relating to interpretation. Building on these results, and those from previous Chapters, we proposed a novel characterisation of interpretation: that a computational system is interpretive if and only if the computational steps it carries out depends on its input. This proposal, and the implications thereof, were kept separated from other investigations that could be prosecuted independently of its truth. Those investigations were presented prior to it, ie earlier in Chapter 7 and also in earlier Chapters, to make this independence explicit.

Subsequently to introducing our proposed characterisation, we discussed the implications of it, and the practical difficulties of applying it. A related property, 'inherent interpretiveness', was developed which acts to bridge the gap between how programs execute, and how they are

written. If a program (modelled either as a single mapping; or as some composition etc of mappings) is inherently interpretive, then details of its language's implementation are irrelevant. Conversely if a program isn't inherently interpretive, then any interpretation is merely an accident of the implementation and again the language's implementation is not of concern. Unfortunately, we were unsure whether a significant class of mappings are inherently interpretive or not. Even with this gap in our knowledge, we were still able to establish that (nearly) all programming is interpretive, and hence the aim in TFP of interpretation-free programming is infeasible. However, we were successfully able to show that TFP-style programming is (often) *less* interpretive. Programming using Church Objects permits one to avoid a certain class of interpreters, a particularly common form of recursive pattern-matching. Hence TFP-style programming offers a substantial liberation from interpretation.

Finally, some exciting theoretical research on 'sub-binary' computation was also detailed.

8.8 Summary

The complete work is summarised

In this Chapter, Chapter 8, we have summarised our introduction to, and investigations of, open questions in TFP.

8.9 Future work

There are numerous opportunities for future work

In the final Chapter, Chapter 9, we identify some avenues for future work.

8.10 Evaluation of claims in TFP

Most of the TFP methodology has been upheld by our research

By way of further summary, our research has affirmed the following core of TFP:

1. Interpreters are harder to analyse and reason about.

This is because input-varying computations are harder to analyse, and the results of such analysis are weaker.

2. Interpretation should be avoided, or at least minimised in programming.

This is because it makes analysis of programs difficult.

3. TFP-style programming is usually preferable over the standard test, data, and loop/recurse -based style of programming.

This is because it often entails less program-input -varying computation.

4. TFP-style programs can be systematically derived.

Techniques based on ‘fold’ exist to derive TFP-style programs.

5. Definitional-style language-extension is less interpretive than interpretive-style language-extension.

Replacing tests on data-constructors by definitions avoids some input-varying computations.

6. All, even small, interpreters are language-extensions.

Yes, as systems whose computations vary with input have the fundamental nature of programming languages.

7. Alternatives exist to testing symbols.

In the case of symbol-testing used to systematically convert constructs represented as symbols to chosen semantics (such as done by 'fold'), two alternatives have been covered. The first is to statically bind the constructs to their semantics via definitions. The second is to parse the constructs as variables and use beta-reduction to bind them to their semantics (ie Church Objects).

Our research has however cast doubt on the following key claims in TFP:

1. TFP-style programming minimises interpretation in programming.

This appears to be incorrect, as introduction and use of first-class functions will make any otherwise non-interpretive program interpretive. However, very few programs are non-interpretive.

2. Interpretation can often be completely removed from programs; equivalently, one can often find a non-interpretive program with the required semantics.

This does not appear to be true; nearly all non-trivial mappings encountered in practice are inherently-interpretive.

3. Definition can be used systematically in place of interpretation.

One can only replace interpretation by definition when the interpreter is 'fold'-expressible. Hence, one does sometimes have to use interpreters for language-extension (in today's common languages), even if one isn't hosting a Universal Turing Machine or similar.

4. There is a dichotomy between interpretation and definition.

We suggest that two distinct topics have been conflated in TFP, and that the use of ‘representations’ should be considered as a separate issue from interpretation.

5. Interpretation should always be avoided.

Input-varying computation can be useful, and can be the best solution.

Finally, our research has expanded the following key claims in TFP:

1. Interpretation is symbol-based programming.

Interpretation is done by the most-common form of test on data (ie ‘fold’); but interpretation is more than tests.

2. Interpretation is prevalent.

Interpretation is more prevalent than originally expected: nearly all mappings are inherently-interpretive.

Chapter 9

Future work

There are numerous avenues for future work

Below we discuss major avenues for future work. These primarily concern better language-extension techniques, the nature of interpretation, and comparing interpretive programs (and finding the best such).

9.1 Future work in TFP, generally

TFP could be recast in terms of language-extension, rather than interpretation

While interpretation-free programming may be impossible, programming with as little interpretation as possible is worthwhile, and there is still significant research able to be done in this area. For example, minimally-interpretive solutions to various common problems could be identified.

In addition, we have identified that there is opportunity to add to TFP work on avoiding poor means of language-extension; such is within the spirit of TFP. The use of 'representations' and not being able to achieve desired observable outputs are important topics with scope for improving programming independently of considerations of interpretation.

9.2 Future work on the nature of interpretation

Further work can be done on the intrinsic nature of interpretation

With regards to our characterisation of interpretation, we have identified a few significant avenues for further work. The status of programs which use constructs with input-varying computations only on inputs for which their computations don't vary needs to be further clarified, and if necessary the characterisation of interpretation modified appropriately. As well,

the nature of 'computational steps' would be interesting to consider formally, especially in the physical world and analog systems, eg in Ordinary Differential Equations. We are unsure whether one could find a single satisfactory system-independent formalisation. Also, note that our characterisation of interpretation is not directly applicable to analog systems, or Cellular Automata. However, one notes that in our 'sub-binary' computational system, input-varying computations and 'confluence' coincide. Input-varying computation is about *computations*, while 'confluence' is about *values*, the constituents of analog systems and Cellular Automata. Further, 'confluence' makes an appearance in other possibly-related disciplines: Smale's 'horseshoe' operator of Chaos Theory involves it and also interestingly has been directly linked with computation (see [SI00]). Hence perhaps interpretation really is 'confluence', and characterising it as such would permit detection of interpretation in analog systems and Cellular Automata. Regardless of any ties to interpretation, it would also be interesting to investigate 'confluence' in its own right.

One may also note that we have been assuming that the computations of constructs are deterministic. Investigating interpretation when there is nondeterminism could be done. Perhaps one may need to generalise interpretation from being 'input-varying' computations to computations whose probabilities of execution vary across inputs.

The issue of whether different *instances* of the same computational-steps should be distinguished also requires resolution. We feel that functions which use such are not automatically inherently-interpretive as the instances could in fact be *shared* sub-expressions; ie the *one* computation called and returned from multiple locations.

Finally, on related matters, applying the concept of input-varying computation to give a foundation to Characteristic Methods and Platonic Combinators may well be possible.

9.3 Future work on transforming-away interpretation

Additional techniques could be developed to remove interpretation via program transformation

In Chapter 5 we covered a number of techniques that are able to remove some interpretation from programs. We were motivated by the desire to be able to derive all the examples from

Chapter 2 of TFP-style programming. No doubt many other techniques could be developed or identified. For example in [OS07] is given a transformation that acts to combine two recursions into one.

The precise limits of applicability of the existing techniques also need to be determined, including regarding 'non-regular' data-types.

9.4 Future work on inherent interpretiveness

Discoveries await in the topic of inherent interpretiveness

There are opportunities for future work with regards to inherent interpretiveness. The status of mappings to and from finite data needs to be identified, ie which (if any) of are inherently-interpretive is to be determined. There is also scope for investigating variations on what is considered inherently interpretive. We have indicated that we consider a given specification not to be inherently interpretive if it has at least one non-interpretive implementation. However, in a given programming language, such an implementation may not be possible in practice due to limitations of the language. Similarly, a non-interpretive implementation might be possible but undesirable due to its difficulty of construction, maintenance or reuse. However, note that considering limitations on implementations can only make specifications *more likely* to be inherently interpretive (in practice), and programming typically always involves mappings which are already inherently-interpretive. Hence such limitations will have little practical import.

9.5 Future work on comparing interpretive systems

More work can be done on measuring the costs of interpretation

We turn now to comparing interpretive systems. We have discussed generally the positives and negatives of interpretation, but some more formal measures for ranking programs or comparing them would be useful. Hopefully, one might be able to determine whether there is some form of natural partition between rather innocuous variation of computation with input, and the degree of variation which leads to a Universal Turing Machine being Turing-complete. It would be

interesting to determine what it is about some interpretation that we appear to find acceptable for use throughout our programs without second thought, while interpreting eg whole languages is considered extremely bad form.

The exact circumstances TFP-style programming (ie Data-Alternatives) should be employed, and which particular representation of data to use to give the best-possible program, could be investigated in detail. Note that when determining which interpretive solutions are better than others, one should take into account efficiency concerns. As discussed in [KEH91], runtime code generation permits one to avoid (usually repeated) execution of 'if' statements via use of what is in effect first-class semantics, resulting in faster execution. Also as discussed in that work, one can gain speed increases by moving as many tests as possible out of inner loops. No doubt there is much fruitful research able to be done on the tradeoffs and techniques of writing high-efficiency highly-maintainable minimally-interpretive code.

A related research direction is to see if one can identify a measure of the costs for a given interpreter, or the interpreters in a given program, so as to permit automatic identification of code for which improvement is likely to be possible.

9.6 Future work with regards to programming languages

TFP overlaps with other active areas of research in Computer Science regarding programming language design

Two areas in which today's programming languages could be improved with regards to interpretation can be identified.

Firstly, when interpretation is needed, its costs should be minimised. Compilers can optimise-away needlessly input-varying computation; or simplify programs so that they do less interpretation when executing. This can make programs easier to analyse, and hence reduce the costs of interpretation. Similarly an improved ability for programming languages to do static analysis will also reduce the costs of interpretation. Static analysis (including type-systems) is an active research area in Computer Science, with many promising results (see eg [BL07] for one recent interesting example).

Secondly, sometimes programmers choose to add interpretation due to various concerns. For example, it is quite infeasible to use individual routines rather than the single interpretive 'bitblt' construct, as discussed in detail in [KEH91]. By incorporating a layer of input-varying computation, a single 'bitblt' routine can be reused in a very large number of situations, with its computations changing as required. Program analysis then unfortunately becomes more difficult. However, better programming languages can make the need for trading-off the negatives of interpretation with its succinctness less likely. The use of 'generic programming' techniques and 'templates' is a good step in this direction. As stated in [CEGVV00], the main goal of 'generic programming' is to improve reusability via providing the ability to do (powerful) static instantiations; this can be seen to eliminate some of the need for hard-to-analyse interpretive solutions.

9.7 Application to related areas

Input-varying computation is of concern in other research areas

Our work relates to other research topics in Computer Science, and hence there are interrelationships to be explored. We have already discussed some, and a further example should serve to illustrate the breadth of related topics. We refer to the work on testing using 'honest domain partitioning' in [SC96]. Here, 'domain' here does not refer to Domain Theory *a la* Scott, but in the sense of the 'domain' and 'range' of functions. In domain partitioning, one splits up the input domain into subsets to try to ensure branch coverage: to test a program, one attempts to test as many as possible of the execution-paths through it. These subsets are such that the program, even though it has input-varying computations, is expected to execute the same computation for every member of a given subset.

Our work however indicates that not uncommonly the only subsets over which programs have such 'constant' computations are subsets with only one element: the specification of many operations exhibits AIIS for any pair of inputs. However, 'honest domain partitioning' appears to work well in practice: no doubt it is the issue of 'acceptable' interpretation again. Work on 'domain partitioning' therefore may have direct relevance for TFP: the number of 'domain

partitioning' subsets necessary to 'reasonably' test an interpreter might be able to be taken as a measure of how interpretive the program is.

List of References

[Abr90] S. Abramsky, The lazy lambda calculus, in Research Topics in Functional Programming, D. Turner Ed., Addison-Wesley, 1990, 65-116

[Ack28] W. Ackermann, Zum hilbertschen aufbau der reellen zahlen, Mathematische Annalen 99, 1928, 118-133

[Aik91] A. Aiken and J. Williams and E. Wimmers, Programming a Language, Research Report, IBM Almaden. Research Center, 1991

[ACC93] M. Abadi and L. Cardelli and P. Curien, Formal parametric polymorphism, Theor. Comput. Sci. 121 1-2, 1993, 9-58

[ACPR95] M. Abadi and L. Cardelli and B. Pierce and D. Rémy, Dynamic typing in polymorphic languages, Journal of Functional Programming, 5(1), 1995, 111-130

[AJ95] S. Abramsky and A. Jung, Domain Theory, in Handbook of Computer Science and Logic 3, Clarendon Press, 1995

[AM91] H. Alblas and B. Melichar Eds., Attribute Grammars, Applications and Systems, LNCS 545, Springer-Verlag, 1991

[AMM05] T. Altenkirch and C. McBride and J. McKinna, Why Dependent Types Matter, unpublished manuscript, version of 2005/4/14. Available online at www.dcs.st-and.ac.uk/~james/RESEARCH/ydtm-submitted.pdf; downloaded 28th November 2007

[AWL94] A. Aiken and E. Wimmers and T. Lakshman, Soft typing with conditional types, in Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, 1994, 163-173

[Bai86] P. Bailes, The Programmer as Language Designer (Towards a Unified Theory of Programming and Language Design), in Proc. 1986 Austrln. Software Eng. Conf., 1986, 14-18

[Bai01] P. Bailes, Programming Without Data - Towards a Totally Functional Programming Style, in Proc 2001 Thai National Computer Science and Engineering Conference, invited paper, 2001, i5-i11

[Bar84] H. Barendregt, The Lambda Calculus - Its Syntax and Semantics, North-Holland, Amsterdam, 1984

[Bar91] H. Barendregt, Introduction to generalized type systems, Journal of Functional Programming 1(2), 1991, 125-154

[Bar97] H. Barendregt, The impact of the lambda calculus in logic and computer science, Bulletin of Symbolic Logic 3(2), 1997, 181-215

[Bay01] I. Bayley, Generic Operations on Nested Datatypes. Ph.D. Dissertation, University of Oxford, 2001

[Ber97] A. Berthiaume, Quantum computation, Complexity Theory Retrospective II, chapter 2, L. Hemaspaandra and A. L. Selman Eds., Springer-Verlag, 1997, 23-50

[Bon89] A. Bondorf, A Self-Applicable Partial Evaluator for Term Rewriting Systems, in Proc. of the Int'l Joint Conf. on Theory and Practice of Software Development, LNCS 352, Springer, 1989, 81-95

[Bra95] M. Branicky, Universal computation and other capabilities of hybrid and continuous dynamical systems, Theoretical Computer Science 138, 1995, 167-100

[BB85] C. Böhm and A. Berarducci, Automatic synthesis of typed Lambda-programs on term algebras, Theoretical Computer Science 39(2/3), 1985, 135-154

[BBH97] J. Bell and F. Bellegarde and J. Hook, Type-driven defunctionalization, in Proceedings of the Second ACM SIGPLAN international Conference on Functional Programming, A. Berman Ed., ACM Press, New York, 1997, 25-37

[BC90] H. Boehm and R. Cartwright, Exact Real Arithmetic: Formulating Real Numbers as Functions, in D.A. Turner Ed., Research Topics in Functional Programming, Addison-Wesley, 1990, 43-64

[BC92] P. Bailes and T. Chorvat, Facet Grammars: Towards Static Semantic Analysis by Context-Free Parsing, Journal of Computer Languages 18(4), 1992, 251-271

[BCG82] E. Berlekamp and J. Conway and R. Guy, Winning Ways four your Mathematical Plays (volume 2), Academic Press, 1982

[BCP94] P. Bailes and T. Chorvat and I. Peake, A Formal Basis for the Perception of Programming as a Language Design Activity, in Proc. 1994 International Conference on Computing and Information, 1994

[BDPR79] C. Böhm and M. Dezani-Ciancaglini and P. Peretti and S. Ronchi Della Rocca, A discrimination algorithm inside λ-β-calculus, Theoretical Computer Science 8(3), 1979, 271-291

[BGM93] P. Bailes and M. Gong and A. Moran, Why Functional Languages Really Need Parallelism, in Proceedings of the Fifth international Conference on Computing and information, O. Abou-Rabia and C. Chang and W. Koczkodaj Eds., IEEE Computer Society, Washington, DC, 423-427

[BH90] H. Barendregt and K. Hemerik, Types in lambda calculi and programming languages, in European Symposium on Programming, Copenhagen, N. Jones Ed., LNCS 432, Springer, 1990, 1-36

[BK03a] P. Bailes and C. Kemp, Formal Methods within a Totally Functional Approach to Programming, in B.K. Aichernig and T. Maibaum Eds., Formal Methods at the Crossroads: from Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, The International Institute for Software Technology of The United Nations University, Springer, 2003, 287-307

[BK03b] P. Bailes and C. Kemp, Integrating Runtime Assertions with Dynamic Types: Structuring Derivation From an Incomputable Specification, in Proceedings of the 27th Anunual International Computer Software & Applications Conference, IEEE, 2003, 520-526

[BK04] P. Bailes and C. Kemp, Obstacles to a Totally Functional Programming Style, in Proceedings of the 2004 Australian Software Engineering Conference, IEEE, 2004, 178-189

[BK05] P. Bailes and C. Kemp, Fusing Folds and Data Structures into Zoetic Data, in Proceedings of the IASTED International Conference on Software Engineering 2005, ACTA Press, 2005, 299-306

[BKPS03] P Bailes and C. Kemp and I. Peake and S. Seefried, Why Functional Programming *Really* Matters, in Proceedings of the 21st IASTED International Multi-Conference on Applied Informatics, Acta Press, 2003, 919-926

[BL03] D. Le Botlan and R. Didier, MLF: raising ML to the power of system F, in Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, ACM Press, New York, 2003, 27-38

[BL07] A. Ben-Amram and C. Lee, Program termination analysis in polynomial time, ACM Trans. Program. Lang. Syst. 29(1), 2007, 5

[BM98] R. Bird and L. Meertens, Nested datatypes, in Proceedings 4th Int. Conf. on Mathematics of Program Construction, J. Jeuring Ed., LNCS 1422, Springer-Verlag, Berlin, 1998, 52-67

[BP99a] R. Bird and R. Paterson, De Bruijn notation as a nested datatype, Journal of Functional Programming, 9(1), 1999, 77-91

[BP99b] R. Bird and R. Paterson, Generalised folds for nested datatypes, Formal aspects of computing 11(2), Springer, 1999, 200-222

[BPL88] G. Burn and S. Peyton Jones and J. Robson, The spineless G-machine, in Proceedings of the 1988 ACM Conference on LISP and Functional Programming, ACM Press, New York, 1988, 244-258

[BPS03] A. Bucciarelli and A. Piperno and I. Salvo, Intersection Types and lambda-Definability, Mathematical Structures in Computer Science 13(1), 2003, 15-53

[BY07] M. Braverman and M. Yampolsky, Constructing Non-Computable Julia Sets, in Proceedings of the 39th ACM Symposium on Theory of Computing, ACM, New York 2007, 709-716

[Chi99] O. Chitil, Typer inference builds a short cut to deforestation, in Proceedings of the Fourth ACM SIGPLAN international Conference on Functional Programming, ACM Press, New York, 1999, 249-260

[Chu33] A. Church, A set of postulates for the foundation of logic (second paper), The Annals of Mathematics Second Series 34(4), 1933, 839-864

[Chu40] A. Church, A formulation of the simple theory of types, The Journal of Symbolic Logic 5(2), 1940, 56-68

[Chu41] A. Church, The Calculi of Lambda Conversion, Annals of Mathematics Studies 6, Princeton University Press, 1941

[Coo04] M. Cook, Universality in Elementary Cellular Automata, Complex Systems 15(1), 2004, 1-40

[Cur34] H. Curry, Functionality in combinatory logic, in Proceedings of the National Academy of Science of the USA 20, 1934, 584-590

[CCM92] J. Cross and J. Chikofsky and C. May, Reverse Engineering, Advances in Computers 35, Academic Press, 1992

[CDP99] L. Correnson and E. Duris and D. Parigot, Equational Semantics, in Proceedings of the 6th International Symposium, LNCS 1694, Springer Berlin / Heidelberg, 1999, 264-283

[CEGVV00] K. Czarnecki and U. Eisenecker and R. Glück and D. Vandevoorde and T. Veldhuizen, Generative Programming and Active Libraries, in Selected Papers From the

international Seminar on Generic Programming, M. Jazayeri and R. Loos and D. Musser Eds., LNCS 1766, Springer-Verlag, London, 2000, 25-39

[CGKF01] J. Clements and P. Graunke and S. Krishnamurthi and M. Felleisen, Little Languages and their Programming Environments, in Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration, 2001, 1-18

[CK99] Z. Chen and M. Karim, Speed limitation of hybrid optical/digital sequential image processing, Optics Communications 168(1-4), 1999, 75-83

[CL90] L. Cardelli and G. Longo, A semantic basis for Quest: (Extended abstract), in ACM Conference on Lisp and Functional Programming, 1990, 30-43. Extended version available as DEC SRC Research Report 55, 1990

[CL03] R. Cardone and C. Lin, Using Mixin Technology to Improve Modularity, in Mehmet Aksit and Siobhan Clarke and Tzilla Elrad and Richard Filman Eds., Aspect-Oriented Software Development, Addison-Wesley, 2003, 219-242

[CMC00] M. Campagnolo and C. Moore and J. Costa, Iteration, inequalities, and differentiability in analog computers, J. Complex. 16(4), 2000, 642-660

[CS68] D. Cudia and W. Singletary, The Post Correspondence Problem, The Journal of Symbolic Logic 33(3), 1968

[CW85] L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, Computing Surveys 17(4), 1985, 472-522

[CW05] S. Carlier and J. Wells, Expansion: the crucial mechanism for type inference with intersection types: Survey and explanation, Technical Report HWMACS -TR-0029, Heriot-Watt Univ., School of Math. & Comput. Sci., 2005

[DL99] N. Danner and D. Leivant, Stratified polymorphism and primitive recursion, Mathematical. Structures in Comp. Sci. 9(4), 1999, 507-522

[DM82] L. Damas and R. Milner, Principal type-schemes for functional programs, in Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1982, 207-212

[DM95] F.-N. Demers and J. Malenfant, Reflection in logic, functional and object-oriented programming: a short comparative study, in Proc. IJCAI'95, Workshop on Reflection and Metalevel Architectures and their Applications in AI, 1995, 29-38

[DMP96] O. Danvy and K. Malmkjaer and J. Palsberg, Eta-Expansion Does the Trick, ACM Transactions on Programming Languages and Systems, 18(6), 1996, 730-751

[DN01] O. Danvy and L. Nielsen, Defunctionalization at work, in Proceedings of the 3rd ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming, ACM Press, New York, 2001, 162-174

[Erw97] M. Erwig, Functional programming with graphs, in Proceedings of the Second ACM SIGPLAN international Conference on Functional Programming, A. M. Berman Ed., ACM Press, New York, 1997, 52-65

[EFT84] H.-D. Ebbinghaus and J. Flum and J. Thomas, Mathematical Logic, Springer-Verlag, New York, 1984

[Far03] W. Farmer, The seven virtues of simple type theory, SQRL Report number 18, McMaster University, 2003 (revised version 2006 also available).

[Fla93] K. Flannery, λ-calculi with decidable ∩-type checking, Computing and Information, in Proceedings ICCI '93, 1993, 13-19

[FB96] K. Fant and S. Brandt, NULL convention logic: a complete and consistent logic for asynchronous digital circuit synthesis, in: International Conference on Application Specific Systems, Architectures, and Processors, 1996, 261-273

[FLO83] S. Fortune and D. Leivant and M. O'Donnell, The expressiveness of simple and second-order type structures, Journal of the ACM, 30(1), 1983, 151-185

[FS96] L. Fegaras and T. Sheard, Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space), in Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, 1996, 284-294

[Ghi07] D. Ghica, Geometry of Synthesis: A structured approach to VLSI design, in Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2007, 263-264

[Gir71] J.-Y. Girard, Une extension de l'interprétation de Gödel a l'analyse, et son application àl'élimination des coupures dans l'analyse et la théorie des types, in Proceedings of the secondScandinavian logic symposium, J.E.Fenstad Ed., North-Holland, 1971, 63-92

[Gir72] J.-Y. Girard, Interpr'etation Fonctionnelle et Elimination des Coupures de l'Arithm'etique d'Ordre Sup'erieur, Th`ese d'Etat, Universit'e Paris 7, 1972

[Gir73] J.-Y. Girard, Quelques résultats sur les interprétations fonctionnelles, Cambridge Summer School in Mathematical Logic, Lecture Notes in Math. 337, Springer, Berlin, 1973, 232-252

[Gir86] J.-Y. Girard, The system F of variable types, fifteen years later, Theoretical Computer Science 45, 159-192

[Gon93] M. Gong, Towards a declaratively-complete basis for functional programming, Thesis (Ph.D.) University of Queensland, 1993

[Gua78] L. Guarino, The Evolution of Abstraction in Programming Languages, CMU-CS-78-120, Department of Computer Science, Carnegie-Mellon University

[GH80] R. Griswold and D. Hanson, An Alternative to the use of Patterns in String Processing, ACM TOPLAS 2(2), 1980, 153-172

[GHA01] G. Gibbons and G. Hutton and T. Altenkirch, When is a function a fold or an unfold?, Coalgebraic Methods in Computer Science, Electronic Notes in Theoretical Computer Science, 44(1), 2001

[GLT89] J.-Y. Girard and Y. Lafont and P. Taylor, Proofs and Types, Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press, 1989

[GR88] P. Giannini and S. Ronchi Della Rocca, Characterization of typings in polymorphic type discipline, in Proceedings of the Third Annual Symposium on Logic in Computer Science, 1988, 61-70

[Hin69] R. Hindley, The principal type scheme of an object in combinatory logic, Transactions of the American Mathematical Society, 146, 1969, 29-60

[Hin99] R. Hinze, Manufacturing Datatypes, in Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages, 1999. Also available as Technical Report IAI-TR-99-5, Institut fur Informatik III, Universitat Bonn

[Hin00] R. Hinze, Efficient Generalized Folds, in Proceedings of the Second Workshop on Generic Programming, 2000

[Hin05] Ralf Hinze, Theoretical Pearl: Church numerals, twice!, Journal of Functional Programming, 15(1), 2005, 1-13

[Hof00] M. Hoffman, Programming languages capturing complexity classes, ACM SIGACT News 31(1), 2000, 31-42

[Hug96] J. Hughes, Type Specialisation for the Lambda-Calculus; or, A New Paradigm for Partial Evaluation Based on Type Inference, in Selected Papers From the international seminar on Partial Evaluation,. O. Danvy R. Glück and P. Thiemann Eds., LNCS 1110, Springer-Verlag, London, 1996, 183-215

[Hut89] G. Hutton, Parsing Using Combinators, Proc. Glasgow Workshop on Functional Programming, Springer, 1989

[Hut98] G. Hutton, Fold and Unfold for Program Semantics, 3rd ACM International Conference on Functional Programming, Baltimore Maryland, 1998

[Hut99] G. Hutton, A Tutorial on the Universality and Expressiveness of Fold, Journal of Functional Programming 9(4), 1999, 355-372

[HK96] G. Hillebrand and P. Kanellakis, On the expressive power of simply typed and let-polymorphic lambda calculi, in Proceedings Logic in Computer Science, IEEE Computer Society Press, 1996, 253-263

[HM91] F. Henglein and H. Mairson, The complexity of type inference for higher-order lambda calculi, Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1991, 119-130

[HP98] M. Hoffman and B. Pierce, Type Destructors, Indiana University CSCI Tehnical Report #502, 1998

[HS93] J. Hopkins and M. Sitaraman, Software quality is inversely proportional to potential local verification effort, Proc. 6th Ann. Workshop on Software Reuse, Owego, NY, 1993

[HU79] J. Hopcroft and J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979

[Imm87] N. Immerman, Languages that capture complexity classes, SIAM Journal of Computing 16, 1987, 760 -778

[Int94] B. Intrigila, Some Results on Numeral Systems in λ-Calculus, Notre Dame Journal of Formal Logic 35(4), 1994, 523-541

[IRA06] G. Issac and C. Rajendran and R. Anantharaman, An instrument for the measurement of customer perceptions of quality management in the software industry: An empirical study in India, Software Quality Journal 14(4), Springer, 2006, 291-308

[Joh05] P. Johann, On Proving the Correctness of Program Transformations Based on Free Theorems for Higher-order Polymorphic Calculi, Mathematical Structures in Computer Science 15(2), 2005, 201-229

[Jim95] T. Jim, Rank-2 type systems and recursive definitions, Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, 1995

[Jon82] J. Jones, Universal Diophantine Equation, The Journal of Symbolic Logic 47(3), 1982, 549-571

[Jon97] M. Jones, First-class Polymorphism with Type Inference, in Proc. Symposium on Principles of Programming Languages, 1997

[Jun96] Edited by A. Jung, with contributions by M. Fiore, A. Jung, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini and I. Stark, Domains and Denotational Semantics: History, Accomplishments and Open Problems, Bulletin of the EATCS, 59, 1996, 227-256

[JG07] P. Johann and N. Ghani, Initial Algebra Semantics is Enough!, in Proceedings, Typed Lambda Calculus and Applications, 2007, 207-222

[JM84] J. Jones and Y. Matijasevic, Register Machine Proof of the Theorem on Exponential Diophantine Representation of Enumerable Sets, The Journal of Symbolic Logic 49(3), 1984, 818-829

[JMT02] Y. Jun and G. Michaelson and P. Trinder, Explaining Polymorphic Types, The Computer Journal 2002 45(4), 436-452

[Ker94] M. Kerber, On the translation of higher-order problems into first-order logic, in Tony Cohn Ed., Proceedings of ECAI-94, John Wiley & Sons, 1994, 145-149

[Kle35a] S Kleene, A theory of positive integers in formal logic, Part I. American journal of mathematics 57(1), 1935, 153-173

[Kle35b] S. Kleene, A theory of positive integers in formal logic, Part II, American journal of mathematics, 57(2), 1935, 219-244

[KEH91] D. Keppel and S. Eggers and R. Henry, A case for runtime code generation, Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, 1991

[KG95] S. Kamin and E. Golin, Report of a workshop on future directions in programming languages and compilers. SIGPLAN Not. 30(7), 1995, 9-28

[KMTW99] A. Kfoury and H. Mairson and F. Turbak and J. Wells, Relating typability and expressibility in finite-rank intersection type systems, in Proc. 1999 Int'l Conf. Functional Programming, ACM Press, 1999, 90-101

[KTU93] A. Kfoury and J. Tiuryn and P. Urzyczyn, Type reconstruction in the presence of polymorphic recursion, ACM Trans. Prog. Lang. Syst. 15(2), 1993, 290-311

[KV05] L. Kristiansen and P. Voda, Programming languages capturing complexity classes, Nordic Journal of Computing 12, 2005, 89-115

[KW94] A. Kfoury and J. Wells, A direct algorithm for type inference in the rank-2 fragment of the second-order lambda-calculus, in Proceedings of the 1994 ACM conference on LISP and functional programming, ACM Press, New York, 2004

[Leh96] M. Lehman, Laws of Software Evolution Revisited, Position Paper, EWSPT96, LNCS 1149, Springer Verlag, 1997, 108-124

[Lei83] D. Leivant, Reasoning About Functional Programs and Complexity Classes Associated with Type Disciplines, 24th Annual Symposium on Foundations of Computer Science, IEEE, 1983, 460-469

[Lei90] D. Leivant, Discrete Polymorphism, in Proc. 1990 ACM Conf. on Lisp and Functional Programming, 1990, 288-297

[Lon99] J. Longley, When is a functional program not a functional program?, in Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, Paris, ACM Press, 1999, 1-7

[Lon02] J. Longley, The sequentially realizable functionals, in Annals of Pure and Applied Logic 117(1), 2002, 1-93

[Lon05] J. Longley, Notions of computability at higher types I, in R. Cori, A. Razborov, S. Todorcevic, and C. Wood Eds., Logic Colloquium 2000, Lecture Notes in Logic 19, ASL, 2005, 32-142

[LLMP89] P. Lee and M. Leone and S. Michaylov and F. Pfenning, Towards a Practical Programming Language Based on the Polymorphic Lambda Calculus. Technical report, School of Computer Science, Carnegie Mellon University, 1989, Ergo Project Report ERGO-89-085

[LN00] G. Lakoff and R. Núñez, Where mathematics comes from: how the embodied mind brings mathematics into being, Basic Books, New York, 2000

[LS95] J. Launchbury and T. Sheard, Warm fusion: deriving build-catas from recursive definitions, in Proceedings of the Seventh international Conference on Functional Programming Languages and Computer Architecture, ACM Press, New York, 1995, 314-323

[Mai91] H. Mairson, Outline of a Proof Theory of Parametricity, in Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, J. Hughes, Ed., LNCS 523, Springer-Verlag, London, 1991, 313-327

[Mai92] H. Mairson, A simple proof of a theorem of Statman, Theoretical Computer Science 103, 1992, 387-394

[Mat96] Y. Matiyasevich, HILBERT'S TENTH PROBLEM: What can we do with Diophantine equations?, 1996. Available online at logic.pdmi.ras.ru/Hilbert10/journal/preprints/H10histe.ps; downloaded November 14, 2006

[McA98] D. McAllester, Object conversion is type-preserving, July 1998. Unpublished paper. Available online at http://ttic.uchicago.edu/~dmcallester/objects.ps; downloaded 13th November 2006

[Mil75] R. Milner, Processes, a mathematical model of computing agents, in Logic Colloquium, Bristol 1973, North Holland, Amsterdam, 1975, 157-174

[Mil78] R. Milner, A theory of type polymorphism in programming, Journal of Comp. Syst. Sci. 17, 348-375

[Mit84] J. Mitchell, Semantic Models for Second-Order Lambda Calculus, Foundations of Computer Science, 1984, 289-299

[Mit88] J. Mitchell, Polymorphic type inference and containment, Information and Computation, 76, 1988, 211-249

[Mit96] J. Mitchell, Foundations for Programming Languages, MIT Press, 1996

[Moo65] G. Moore, Cramming more components onto integrated circuits, Electronics Magazine 19, 1965

[Mur07] C. Murthy, Advanced Programming Language Design in Enterprise Software, Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2007, 263-264

[MFP91] E. Meijer, and M. Fokkinga and R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, J. Hughes Ed., Springer-Verlag, New York, 1991, 124-144

[MH95] E. Meijer and G. Hutton, Bananas in space: extending fold and unfold to exponential types, in Proceedings of the Seventh international Conference on Functional Programming Languages and Computer Architecture, ACM Press, New York, 1995, 324-333

[MJ95] E. Meijer and J. Jeuring, Merging Monads and Folds for Functional Programming, in Advanced Functional Programming, First international Spring School on Advanced Functional Programming Techniques-Tutorial Text, J. Jeuring and E. Meijer Eds., LNCS 925, Springer-Verlag, London, 1995, 228-266

[MMZ97] J. Malolepszy and M. Moczurad and M. Zaionc, Schwichtenberg-style lambda definability is undecidable, LNCS 1210, Springer, 1997, 267-283

[MP88] J. Mitchell and G. Plotkin, Abstract types have existential type, ACM transactions on programmming languages and systems, 10(3), 1988, 470-502

[NM07] H. Nelson and D. Monarchi, Ensuring the quality of conceptual representations, Software Quality Journal 15(2), Springer, 2007, 213-233

[O'D79] M. O'Donnell, A programming language theorem which is independent of Peano arithmetic, 11th Annual ACM Symposium on Theory of Computing, 1979, 176-186

[O'D85] M. O'Donnell, Equational Logic as a Programming Language, MIT Press, 1985

[Oka99] C. Okasaki, From fast exponentiation to square matrices: An adventure in types, in Procedings of the 4th International Conference on Functional Programming, 1999, ACM, 28-35

[Öme05] B. Ömer, Classical Concepts in Quantum Programming, International Journal of Theoretical Physics 44(7), Springer, 2005, 943-955

[Orp97] P. Orponen, A survey of continuous-time computation theory, in D.-Z. Du and K.-I. Ko, Eds., Advances in Algorithms, Languages, and Complexity, Kluwer Academic Publishers, Dordrecht, 1997, 209-224

[OG89] J. O'Toole and D. Gifford, Type reconstruction with first-class polymorphic values, in Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, ACM Press, 1989, 207-217

[OMG03] MDA Guide Version 1.0.1, J. Miller and J. Mukerji Eds., available online at http://www.omg.org/docs/omg/03-06-01.pdf; downloaded 14 November, 2006

[OS07] A. Ohori and I. Sasano, Lightweight Fusion by Fixed Point Promotion, Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2007, 143-154

[Pey87] S. Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall International, Hemel Hempstead, 1987

[Pey03] S. Peyton Jones Ed., Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press, 2003

[Pfe88] F. Pfenning, Partial polymorphic type inference and higher-order unification, in Proceedings of the 1988 ACM Conference on LISP and Functional Programming, New York, 1988, 153-163

[Pfe93] F. Pfenning, On the undecidability of partial polymorphic type reconstruction, Fund. Informa 19(1,2), 1993, 185-199

[Plo77] G. Plotkin, LCF Considered as a Programming Language, Theoretical Computer Science 5, 1977, 223-255

[PA93] G. Plotkin and M. Abadi, A logic for parametric polymorphism, in M. Bezem and J.F. Groote Eds., Typed Lambda Calculi and Applications, LNCS 664, Springer-Verlag, 1993, 361-375

[PDM89] B. Pierce and S. Dietzen and S. Michaylov, Programming in Higher-Order Typed Lambda-Calculi, Technical report CMU-CS-89-111, Carnegie Mellon University, 1989

[PE88] F. Pfenning and C. Elliott, Higher-order abstract syntax, in Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation, 1988, 199-208

[PL89] F. Pfenning and P. Lee, LEAP: a language with eval and polymorphism, in TAPSOFT'89**,** Proc. International Joint Conf. Theory Practice Softw. Develop, Springer-Verlag, 1989

[PVWS07] S. Peyton Jones and D. Vytiniotis and S. Weirich and M. Shields, Practical type inference for arbitrary-rank types, Journal of Functional Programming, J. Funct. Program. 17(1), 2007, 1-82

[Rea89] C. Reade, Elements of Functional Programming, International Computer Science Series, Addison Wesley, 1989

[Rem05] D. Rémy, Simple, partial type-inference for System F based on type-containment, in Proceedings of the Tenth ACM SIGPLAN international Conference on Functional Programming, ACM Press, New York, 2005, 130-143

[Rey74] J. Reynolds, Towards a theory of type structure, in Proc of the Colloque sur la Programmation, LNCS 19, Springer-Verlag, 1974, 408-425

[Rey83] J. Reynolds, Types, abstraction and parametric polymorphism, in Information Processing 83, REA Mason, ed., North-Holland, 1983, 513-523

[Rey85] J. Reynolds, Three approaches to type structure, Mathematical Foundations of Software Development, LNCS 185, Springer-Verlag, 1985

[Roj96] R. Rojas, Conditional Branching is not Necessary for Universal Computation in von Neumann Computers, Journal of Universal Computer Science 2(11), 1996, 756-767

[Rus89] J. Russell, Full abstraction for nondeterministic dataflow networks, Foundations of Computer Science, 30th Annual Symposium on, 1989, 170-175

[SB02] H. Schwichtenberg and S. Bellantoni, Feasible computation with higher types, in H. Schwichtenberg and R. Steinbrüggen Eds., Proof and System-Reliability, Proceedings NATO Advanced Study Institute Marktoberdorf 2001, Kluwer Academic Publisher, 2002, 399-415

[Sch76] H. Schwichtenberg, Definierbare Funktionen im λ-Kalkul mit Typen, Archiv fur Logik und die Grundlagen der Mathematik 17, 1976, 113-114

[Sch86] D. Schmidt, Denotational Semantics - A Methodology for Language Development, Allyn and Bacon, Boston MA, 1986

[Sco72] D. Scott, Lattice theoretic models for various type-free calculi, in Proc 4th International Congress for Logic Methodology and Philosophy of Science, Bucharest, 1972

[Sim87] S. Simpson, Unprovable theorems and fast-growing functions, Contemporary Math. 65 1987, 359-394

[Sta75] T. Standish, Extensibility in programming language design, SIGPLAN Not. 10(7), 1975, 18-21

[Sta79] R. Statman, The typed lambda-calculus is not elementary recursive, Theoret. Comput. Sci., 9(1), 1979, 73-81

[Sto77] J. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977

[SC96] P. Stocks and D. Carrington, A Framework for Specification-Based Testing, IEEE Trans. Softw. Eng. 22(11), 1996, 777-793

[SF93] T. Sheard and L. Fegaras, A fold for all seasons, in Proceedings of the Conference on Functional Programming Languages and Computer Architecture, ACM Press, New York, 1993, 233-242

[SGT96] M. Sperber and R. Glück and P. Thiemann, Bootstrapping higher-order program transformers from interpreters, in Proceedings of the 1996 ACM Symposium on Applied Computing, K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower Eds., ACM Press, New York, 2006, 408-413

[SI00] S. Yuzuru and I. Takashi, Nonlinear Computation with Switching Map Systems, Journal of Universal Computer Science 6(9), 2000, 893-937

[SS00] S. Seres and J. Spivey, Higher-Order Transformation of Logic Programs, in Selected Papers from the 10th international Workshop on Logic Based Program Synthesis and Transformation, K. Lau Ed., LNCS 2042, Springer-Verlag, London, 2000, 57-68

[SU98] M. Sorensen and P. Urzyczyn, Lectures on the Curry-Howard isomorphism, Technical report TOPPS D-368, Univ. of Copenhagen, 1998

[Tro02] J. Tromp, Kolmogorov Complexity in Combinatory Logic, 2002. Available online at http://homepages.cwi.nl/~tromp/cl/LC.ps; downloaded 4th December, 2006

[Tur36] A. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, in Proceedings of the London Mathematical Society Series 2 number 42, 1936-7, 230-265

[Tur04] D. Turner, Total Functional Programming, Journal of Universal Computer Science 10(7), 2004, 751-768

[TM95] A. Takano and E. Meijer, Shortcut deforestation in calculational form, in Proceedings of the Seventh international Conference on Functional Programming Languages and Computer Architecture, ACM Press, New York, 1995, 306-313

[TW01] F. Turbak and J. Wells, Cycle therapy: a prescription for fold and unfold on regular trees, in Proceedings of the 3rd ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming, ACM Press, New York, NY, 2001, 137-149

[Urz97] P. Urzyczyn, Type reconstruction in Fω, Math. Structures Comput. Sci. 7(4), 1997, 329-358

[Wad76] C. Wadsworth, The Relation between Computational and Denotational Properties for Scott's D_∞-models of the Lambda-Calculus, SIM J. Comput. 5(3), Sep 1976

[Wad89] P. Wadler, Theorems for free!, in 4'th International Conference on Functional Programming and Computer Architecture, ACM, 1989, 347-359

[Wad90] P. Wadler, Deforestation: transforming programs to eliminate trees, Theoretical Computer Science 73(2), 1990, 231-248

[Wad97] P. Wadler, How to Declare an Imperative, ACM Computing Surveys 29(3), 240-263

[Wad03] P. Wadler, The Girard-Reynolds isomorphism, Information and Computation 186, 2003, 260-284

[Wan80] M. Wand, Continuation-Based Program Transformation Strategies, J. ACM 27(1), 1980, 164-180

[Wei01] S. Weirich, Encoding Intensional Type Analysis, in Proceedings of the 10th European Symposium on Programming Languages and Systems, D. Sands Ed., LNCS 2028, Springer-Verlag, London, 92-106

[Wel96] J. Wells, Typability and type checking in the second-order lambda-calculus are equivalent and undecidable, in Proc. 9th Ann. IEEE Symp. Logic in Comput. Sci., 1994, 176-185

[Wel99] J. Wells, Typability and type checking in System F are equivalent and undecidable, Ann. Pure Appl. Logic 98(1-3), 1999, 111-156

[Wel02] J. Wells, The essence of principal typings, in Proc. 29th Int'l Coll. Automata, Languages, and Programming, LNCS 2380, Springer-Verlag, 2002, 913-925

[Win79] T. Winograd, Beyond programming languages, Commun. ACM 22(7), 1979, 391-401

[Wit74] L. Wittgenstein, Tractatus Logico Philosophicus, David Pears and Brian McGuinness (trans.), Routledge, London, 1974

[Wol02] S. Wolfram, A new kind of science, Wolfram Media, 2002

[WC04] T. Weber and J. Caldwell, Constructively Characterizing Fold and Unfold, in Logic Based Program Synthesis and Transformation, 13th International Symposium, LNCS 3018, Springer Berlin / Heidelberg, 2004, 110-127

[WW03] G. Washburn and S. Weirich, Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism, in Proceedings of the Eighth ACM SIGPLAN international Conference on Functional Programming, ACM Press, New York, 2003, 249-262

[XP99] H. Xi and F. Pfenning, Dependent types in practical programming, in Conference Record of the 26th Symposium on Principles of Programming Languages, A. Aiken Ed., ACM Press, 1999, 214-227

[YC79] E. Yourdon and L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Prentice-Hall, 1979

[YD95] C. Yap and T. Dubé, The Exact Computation Paradigm, in Computing in Euclidean Geometry, World Scientific, 2nd edn., D.-Z. Du and F. Hwang Eds., 1995

[Zai95] M. Zaionc, Lambda representations of operations between different term algebras, LNCS 933, 1995, 91-105

Appendix 1

Introduction to the lambda calculus

The lambda calculus has simple syntax and semantics

The programming language used throughout this dissertation is the lambda calculus. Note that as communicated in [Bar97], the original lambda calculus of Church was the λ-I calculus, in which each argument of a function must be used, ie appear free in the body. Nowadays 'the lambda calculus' refers to the λ-K calculus, as presented. We direct the reader interested in the formal aspects of the lambda calculus to [Bar84], for one of the original presentations to [Chu41], for an introduction in historical context, [Bar97]. The following is a simplified informal overview of it sufficient for the reading of this work.

A1.1 Syntax

The terms of the lambda calculus are variables, lambda-abstractions, and applications

As a grammar, the set of terms of lambda calculus is:

$$\Lambda ::= \lambda V.\ \Lambda \mid \Lambda\ \Lambda \mid V \mid (\Lambda)$$

where 'V' is a set of variable-names. Nested lambda-abstractions are often written using the 'pretty-syntax' of a single 'λ' and combining the variables into a list, ie:

(λx. (λy. ..))

is often written as:

(λx,y. ..)

Note that in this work, we always bracket lambda-abstractions as we believe this adds clarity. For example, we write:

(λx,y. (λz. add x z) y)

when

λx,y. (λz. add x z) y

is acceptable. Similarly, to aid comprehension we will occasionally use infix notation for applications when such isn't ambiguous.

Note that variables are described as being ‘free’ when they are not enclosed by a lambda-abstraction of the same variable, otherwise they are described as ‘captured’ or ‘bound’. In the above example, ‘x’ and ‘y’ and ‘z’ are captured while ‘add’ is free.

It is also convenient to add to the above syntax a construct for definitions, specifically ‘≜’. For convenience, we set ‘F x ≜ RHS’ to be equal to ‘F ≜ (λx. RHS)’. Hence, the following definitions are equivalent:

F x y z ≜ RHS

F x y ≜ (λz. RHS)

F x ≜ (λy,z. RHS)

F ≜ (λx,y,z. RHS)

For information on the various syntactic ‘Normal Forms’ mentioned in this work, any standard text on the lambda calculus can be consulted eg [Bar84].

A1.2 Semantics

Evaluation is capture-avoiding syntactic replacement

Evaluation of lambda-terms is canonically presented as proceeding via the following syntactic rewrite rule:

(λx.M) N

→

M[x\N]

Here, ‘M[x\N]’ denotes replacing free occurrences of ‘x’ in ‘M’ by ‘N’. The rewriting relation ‘→’ entails ‘reduction’; with this particular rewrite rule the reduction is what is termed a ‘beta’ (or ‘β‘) reduction. If free variables in ‘N’ would become captured, variables in ‘M’ are renamed to avoid that capture as an implicit part of the reduction. For example:

(λx,w.x) (λy. x)

first has ‘x’ renamed to ‘z’ in the operator:

(λz,w.z) (λy. x)

and then the substitution takes place. A change of name from ‘x’ to ‘w’ would be illegal. A change of a bound variable-name, in isolation, is also expressible as a rewrite rule and is known as the ‘alpha’ (‘α’) rule. The other common rule is the ‘eta’ (‘η’) rule, which rewrites ‘(λx. M x)’ to ‘M’ when ‘M’ doesn’t contain ‘x’ free. This rule is also considered to be a reduction, ie a ‘→’. Note that ‘→*’ means a sequence of zero or more reductions. Generally, one does not make explicit uses of the alpha rule, ie it is valid to say:

(λx,w.x) (λy. x)

→*

(λz,b. x)

To illustrate evaluation, we give the following example of a reduction-sequence:

(λf,x. f (f (λf. f x) x)) (λx.x) (λy. g y)

→ (λx. (λx.x) ((λx.x) (λf. f x) x)) (λy. g y)

→ (λx.x) ((λx.x) (λf. f (λy. g y)) (λy. g y))

→ (λx.x) (λf. f (λy. g y)) (λy. g y)

→ (λf. f (λy. g y)) (λy. g y)

→ (λy. g y) (λy. g y)

→ (λy. g y) g

→ g g

If there exists some sequence of the three rewrite rules (each instance can be in the forwards or reverse direction) which can convert one term into another, then those two terms are called ‘convertible’, written ‘↔’. As these rewrite rules don’t change the semantics of functions, convertibility implies semantic equality.

One should note that definitions, which appear frequently in this work, are pretty-syntax, a convenience only, and can be mechanically removed. Most-simply one has:

F ≜ RHS

B

being the same as:

(λF. B) RHS

For recursive definitions, more complex techniques (see [Bar84, p142]) are required. Similarly, we use the following pretty-syntax regarding tuples:

(λ<a,b..>. M)

stands for ('t' is fresh):

(λt. (λa,b.. . M) (firstelement t) (secondelement t)..)

for appropriate implementations of 'firstelement', 'secondelement' etc.

Finally, we will on occasion use data and data-type introductions ('::='), 'if..then', 'if.. then..else', '==' (symbol-equality), etc in otherwise- lambda calculus terms. Such is done informally, the hypothetical language implied is that of the lambda calculus with those features added and '→' appropriately extended.

Appendix 2

TFP-style programming under Hindley-Milner typing

Hindley-Milner typing and the Simple typing can type many, but not all, TFP-style programs

We show that Hindley-Milner typing is nearly suitable for TFP-style programming, and give examples illustrating the problematic aspects. Our examples in this section are written for the 'Haskell' programming language ([Pey03]).

We begin by discussing how function types can be used to guarantee type-correctness of all applications of a function. Subsequently, we explain how when multiple functions are involved, interactions amongst the types can lead to type-errors even if each application is typable in isolation. We then detail how to often-times avoid such situations, using type-conversion functions to somewhat systematise the process.

Recall that the Simple typing and Hindley-Milner typing have equivalent typability, hence this section is pertinent to the Simple typing as well.

A2.1 Functions in isolation

With Hindley-Milner typing, the type of a function directly gives the set of its allowed inputs

Generally, programmers need to determine whether a given function can be type-correctly applied to arbitrary, some, or none of some desired set of input-terms. A pertinent example appears in [BK03a], where it was noted that a given arithmetic function may have an inferrable Hindley-Milner type, but only applications of it to some but not all Church Naturals will type.

In Hindley-Milner typing, such a determination can readily be carried out; ensuring that application of a function to members of some set will never result in a type-error can be guaranteed by simply making sure that the function has an inferred type which covers all members of the set. If it has too-small a type, then for some of the set the application will fail

with a type error. Simply, the type of a function denotes the limits of its applicativity. Note that functions which don't type can be considered to have the empty type, an extreme case of too-small a type. For example, self-exponentiation defined as:

selfexponentiate x = x x

won't type in Hindley-Milner due to the self-application, hence won't work for *any* input in its nominal domain. The reason that self-application of a variable is not valid in Hindley-Milner is because each occurance of the self-applied variable must have a different monotype: but as it's a variable, ie a function parameter, it can only have a single monotype. In the case above, the variable 'x' needs to have both type 'a' and 'a —> a'.

Interestingly, the ability to directly read the set of allowable inputs from the type is not possible for System F and higher due to impredicativity. For example, consider:

(λx. x x) : (∀a. a —> a) —> (∀a. a —> a)

As 'a' can range over polytypes, including '(∀c. c —> c)', one would think that an application with any input of type '(∀c. c —> c) —> (∀c. c —> c)' would type. However, one possible input with such type is '(λx. x x)' itself, and of course:

(λx. x x) (λx. x x)

is not typable in those systems, not being strongly normalizable.

A2.2 Programs containing multiple functions

Even if no function in a program is applied to an input-term outside of its type, the program in question can still be type-incorrect

The example in [BK03a] of a program whose applications to some Church Naturals 'surprisingly' doesn't type also illustates a troublesome aspect of trying to do practical TFP-style programming under Hindley-Milner typing. It is well-known that with Hindley-Milner typing, functions in a program might only be applied to input-terms which the functions' types indicate are acceptable, yet the program can be type-incorrect. The underlying cause of this is a lack of polymorphism: the inputs and outputs of a given function not only have to be able to be given a (mono) type that is compatible with that function, but *also* with any other functions which use or produce them. For a simple example:

cnpow v w = w v

has a typing, namely:

cnpow :: (Church a) —> (Church (a —> a)) —> (Church a)

where:

Church t = (t —> t) —> t —> t

Recall that '∀t. (t—>t) —>t —> t' is the type of Church Naturals; ie if a parameter is typed as being 'Church X' for some 'X', then the function will accept any Church Natural as that parameter.

Now, a call to:

(λx. cnpow x x)

with the supplied 'x' being a Church Natural will fail, as in Hindley-Milner typing 'x' cannot concurrently have both type 'Church a' and type 'Church (a —> a)'.

As an aside, it should be noted that 'Church (a —> a)' is actually identical to '(Church a) —> (Church a)'. Hence 'cnpow' can take as its second parameter a Church Natural, and also iterate it via application of another Church Natural (a nice synergy between Hindley-Milner typing and TFP). For the second parameter to take an arbitrary Church Natural it needs to be of type 'Church X' for some 'X': and it has such a type, one where the 'X' is 'a —> a'. For the second parameter to be iterated, it must be of type 't —> t' for some 't': and it has such a type, one where 't' is 'Church a'.

As a more realistic example of the problems faced with TFP-style programming under Hindley-Milner typing, we present the following. Consider the following definition of multiplication, 'cnmul':

cnmul v w = w (cnadd v) cnzero

where:

cnadd v w = w cnsucc v

cnsucc n f x = f (n f x)

Also, define:

cnzero f x = x

cnone f x = f x

cntwo f x = f (f x)

cnthree f x = f (f (f x))

..

The function ‘cnsucc’ implements successor and ‘cnadd’ addition, on Church Naturals. These two have (inferred) Hindley-Milner types which show that they take and return arbitrary Church Naturals. However applications of ‘cnmul’ to certain Church Naturals will fail to type. The most appropriate Hindley-Milner type ‘cnmul’ can be given is:

```
forall a,b. Church a —> (
        (Church (Church a) —> Church a) —> (Church b) —> (Church a)
                ) —> (Church a)
```

The definition of ‘cnmul’ indicates that it takes a Church Natural as its first argument, and returns a Church Natural. However its second argument is not of type ‘Church X’ for some ‘X’; and only some Church Naturals have that particular type. Hence for some Church Naturals given as a second argument, a type-error will occur. The Church Naturals which are of this type are only ‘cnzero’ and ‘cnone’. The Church Natural ‘cnzero’ has type:

```
(Church (Church a) —> Church a) —> (Church a) —> (Church a)
```

If we compare this to the type of the second parameter of ‘cnmul’:

```
(Church (Church a) —> Church a) —> (Church b) —> (Church a)
```

then we can see that ‘cnzero’ is indeed of this type, as ‘b’ can be unified with ‘a’. Similarly, the Church Natural ‘cnone’ has type:

```
(Church (Church a) —> Church a) —> (Church (Church a)) —> (Church a)
```

which is the type of the second parameter of ‘cnmul’ with ‘b’ unified with ‘Church a’. For ‘cntwo’ (and greater) there is no ‘b’ which suffices, as can be seen by there being no ‘b’ for which the following is type-correct:

```
(λf: (Church (Church a) —> Church a), x:(Church b). f (f x) : (Church a))
```

By considering the overlapping type-constraints, the reason why ‘cnmul’ has this undesirably-small type can be established. Firstly, note the type for ‘cnsucc’ is:

```
(Church a) —> (Church a)
```

and hence that the type desired for ‘cnadd’ is:

```
(Church a) —> (
                ((Church a) —> (Church a)) —> (Church a) —> (Church a)
                ) —> (Church a)
```

ie:

```
(Church a) —> (Church (Church a)) —> (Church a)
```

Recall the definition of ‘cnmul’:

cnmul v w = w (cnadd v) cnzero

The first argument to ‘w’ is ‘cnadd v’, of type

(Church (Church a)) —> (Church a)

We want applications ‘cnmul’ to type, for all Church Natural ‘w’. However, if ‘w’ is a Church Natural then its type is ‘(t —> t) —> t —> t’ for some ‘t’. This can’t unify with the type of the first second argument to ‘w’, ‘(Church (Church a)) —> (Church a)’: ‘t’ can’t be both ‘Church a’ and ‘Church (Church a)’. Hence ‘w’ cannot be typed to be a Church Natural; and type-inference deduces the narrower type given earlier for it.

Before we finish with this example, it is interesting to note that in this particular case the requirement for a variable to concurrently have two different monotypes can be seen to be the result of self-application, occuring when the second-parameter is ‘cntwo’ or greater. Using ‘cnmul v w = w (cnadd v) cnzero’, and ‘cntwo f x = f (f x)’:

(λv. cnmul v cntwo)
= (λv. cntwo (cnadd v) cnzero)
= (λv. (λf,x. f (f x)) (cnadd v) cnzero)
= (λv. (cnadd v) ((cnadd v) cnzero))

Then, as ‘cnadd v w = w cnsucc v’:

(continuing)
= (λv. (λw. w cnsucc v) ((λw. w cnsucc v) cnzero))
= (λv. (cnzero cnsucc v) cnsucc v)
= (λv. ((λf,x. x) cnsucc v) cnsucc v)
= (λv. v cnsucc v)

which is clearly self-applicative.

Finally (we consider the program in [BK03a] a little later), we have an example which demonstrates that not just inputs but also outputs of functions need to be considered with regards to overlapping type-constraints:

cnadd :: (Church a) —> (Church a) —> (Church a)
cnadd v w f x = v f (w f x)

```
cnmul :: (Church (Church a)) —> (Church a) —> (Church a)
cnmul v w = v (cnadd w) cnzero

cnpow :: (Church (Church a)) —> (Church (Church a)) —> (Church a)
cnpow v w  = w (cnmul v) (cnsucc cnzero)

cnaddpow v w = cnadd (cnpow v w) w
```

The function ‘cnaddpow’ certainly seems reasonable, but its type-correctness hinges on whether the operations called (‘cnadd’ and ‘cnpow’) impose non-contradictory conditions on the types of the variables. The function ‘cnadd’ takes two inputs of the same type, hence the type of ‘cnpow v w’ and the type of ‘w’ must be identical. However, the type of ‘cnpow’ indicates that its second argument is of different type to its result. This contradiction results in ‘cnaddpow’ having no Hindley-Milner type, as its ‘w’ needs to have both type ‘Church a’ and ‘Church (Church a)’, for some ‘a’.

A2.3 Semantically-equivalent functions with better types

The problem of a function having a type smaller than desired can sometimes be solved via use of semantically-equivalent but differently-typed sub-functions, or using a different algorithm

If one function-definition fails to have an acceptable type, then it may be the case that some other semantically-identical definition *does* have an acceptable type. However, finding such functions can be non-trivial and in any case having to do so places a large impost on the programmer. We illustrate with some case-studies now.

Consider the example given previously:

```
cnsucc n f x = f (n f x)
cnadd v w = w cnsucc v
cnmul v w = w (cnadd v) cnzero
```

We noted then that the type of ‘cnadd’ is:

```
(Church a) —> (Church (Church a)) —> (Church a)
```

and hence that of ‘cnadd v’ is:

(Church (Church a)) —> (Church a)

This renders ‘cnmul’ too-narrowly typed as the function ‘w’ in it is applied to needs to be of type ‘t —> t’, for some ‘t’. However if we use a different, semantically-identical definition of ‘cnmul’, specifically the one implied by the symmetry of the multiplication operation:

cnmul v w = w (λc. cnadd c v) cnzero

then the first function ‘w’ is applied to is now of type:

(Church a) —> (Church a)

and as the second function it is applied to, ‘cnzero’ is of type:

(Church a)

then the inferred type for ‘w’ is now:

((Church a) —> (Church a)) —> (Church a) —> (Church a)

ie

Church (Church a)

As this is ‘Church X’ for some ‘X’, the ‘w’ given to ‘cnmul’ can now be an arbitrary Church Natural, as desired.

An alternative to replacing sub-functions with semantically-equivalent ones with better types (which will not be possible in all cases) is to fundamentally change the algorithm used in the program. A simple example of this is the self-exponentiation function defined as:

selfexponentiate x = cnpow x x

where:

cnpow :: (Church a) —> (Church (a —> a)) —> (Church a)

cnpow v w = w v

As discussed previously, ‘cnpow’ has an acceptable typing, but ‘selfexponentiate’ does not, due to the multiple type-constraints on ‘x’. If we instead use the semantically-identical but differently-typed function:

cnpow :: (Church (Church a)) —> (Church (Church a)) —> (Church a)

cnpow v w = w (cnmul v) (cnsucc cnzero)

then rather than ‘x’ needing to have both type ‘Church a’ and ‘Church (a —> a)’, it instead just needs to have type ‘Church (Church a)’. Hence ‘selfexponentiate’ now has an acceptable typing, namely:

(Church (Church a)) —> (Church a)

At this point, Hindley-Milner is looking decidedly impractical for TFP. Whenever one needs an operation, one has to try to find an implementation of it which types its input relative to its output appropriately for the circumstances in which it is used: a dire loss of compositionality. With a library of different implementations such choices could perhaps be done by the language via type-dispatching (ie 'overloading'). However, there is likely to be situations in which a rewrite of the program to use different algorithms is required.

A2.4 Type-conversion functions

Semantically-redundant type-conversion functions are an alternative to replacing functions with differently-typed versions

The use of Hindley-Milner typing for TFP-style programs actually turns out to be somewhat more practical than the previous examples suggest; as we have discovered and as is known from the literature, there exist semantically-redundant functions that can act as *type converters*. These can be introduced (often mechanically) to turn untypable programs into typable ones. This is clearly a better solution than requiring the programmer to come up with versions of functions which type together, or if such versions cannot be identified, rewriting the program to try to improve things in this regard.

Recall the previous example, where:

```
cnadd :: (Church a) —> (Church a) —> (Church a)
cnpow :: (Church (Church a)) —> (Church (Church a)) —> (Church a)
cnaddpow v w = cnadd (cnpow v w) w
```

The function 'cnaddpow' has no Hinley-Milner type, as 'w' needs to be both of type 'Church (Church a)' and concurrently of type 'Church a'.

If however, we could come up with an identity function on Church Naturals, call it 'uptype', of type

```
(Church (Church a)) —> (Church a)
```

then we could change the definition of 'cnaddpow' to a semantically-identical one:

```
cnaddpow v w = cnadd (cnpow v w) (uptype w)
```

and hence have it type. Given that in 'cnaddpow' 'v' and 'w' now have the same type, one could even define:

cnaddpowself x = cnaddpow x x

and have it be type-correct.

Such a type-conversion function does exist (it is indeed reasonably well-known), and can be defined as:

uptype n = n cnsucc cnzero

It is semantically correct, ie:

uptype n f x = n f x

as, proceeding by induction on 'n', ie 'n' is either 'cnzero' or 'cnsucc n':

Base-case: 'n' is 'cnzero'

uptype (cnzero) f x

= cnzero cnsucc cnzero f x

= cnzero f x

as required.

Step-case: 'n' is 'cnsucc m' for some 'm'

uptype (cnsucc m) f x

= (cnsucc m) cnsucc zero f x

= (cnsucc m cnsucc zero) f x

= cnsucc (m cnsucc zero) f x

= f (m cnsucc zero f x)

= f (uptype m f x)

Now, via the induction hypothesis, that 'uptype m f x = m f x':

= f (m f x)

= (cnsucc m) f x

as required.

Note that we do not require:

∀a. ∀n:(Church(Church a)), f:(a─>a), x:(a). uptype n f x = n f x

Such is stronger than necessary, as we only require 'uptype' to work on Church Naturals not constant functions such as '(λf,x. cnthree)' that are also of type 'Church(Church a))'.

Along the same lines, we have discovered a second type-conversion function of type '(Church (a─>a)) ─> (Church a)'. This one lets 'cnpow v w = w v' be used freely, even when 'v' and 'w' are from the same function-argument. Intestingly, a semantically-redundant type-conversion function is all that is required to permit near- self-applicative behaviour.

This second type-conversion function can be defined as:

uptype2 :: (Church (a─>a)) ─> (Church a)

uptype2 n f x = n (λr,z. f (r z)) (λz. z) x

We later found this definition in [FLO83], where it is indicated that all programs from a set 'N' can have inserted instances of this function for type-conversion so as to make them type under Hindley-Milner. The set 'N' is:

N ::= (λf,x. x) | (λf,x. f x) | cnadd N N | cnpred N | cnmul N N | cnpow N N

| cniszero N N N

One may also add free variables (to be bound to Church Naturals) to the above. An implementation for predecessor, 'cnpred' above, can be found in [FLO83] or later in this work. The 'cnpow' is as per [FLO83] '(λx,y. y x)', 'cnadd' is '(λx,y,f,x. x f (y f x))', 'cnmul' is '(λx,y,f,x. x (y f) x)', and 'cniszero' is '(λx,q,r,f,x. x (λy. q f x) (r f x))'. Other definitions will not necessarily work, type-wise. While this result of [FLO83] is of interest we of course are interested in more general cases, such as when 'N' can be applied directly as an iterator, and when other definitions of the operations are used.

The above definition of 'uptype2' can be seen to be semantically correct, ie for all Church Natural 'n':

uptype2 n f x = n f x

Using induction:

Base-case, 'n' is 'cnzero'

uptype2 cnzero f x

= cnzero (λr, z. f (r z)) (λz. z) x

= (λz. z) x

= x

= cnzero f x

as required.

Step-case, ‘n’ is ‘cnsucc m’ for some ‘m’

uptype2 (cnsucc m) f x

= cnsucc m (λr,z. f (r z)) (λz. z) x

= (λr,z. f (r z)) (m (λr,z. f (r z)) (λz. z)) x

= f (m (λr,z. f (r z)) (λz. z) x)

= f (uptype2 m)

Now, via the induction hypothesis, that ‘uptype2 m f x = m f x’:

= f (m f x)

= (cnsucc m) f x

as required.

As an example of its usage, consider the following (type-correct) program:

```
cnadd :: (Church a) —> (Church a) —> (Church a)
cnadd v w f x = v f (w f x)

cnpow :: (Church a) —> (Church (a —> a)) —> (Church a)
cnpow v w  = w v

selfexponentiate x = cnpow (uptype2 x) x

cnaddpow v w = cnadd (cnpow (uptype2 v) w) (uptype2 w)
```

Finally, two things about type-conversion functions are worthy of note. Firstly, the need for them could perhaps be detected by the type-system during type-checking and type-inference, and a hidden call to them made as required. Of course, a better solution would be for the type-system to include a new typing-judgement rather than engage in semantically-redundant calculation. The feasibility of such extensions is unclear. Secondly, an alternative to the type-system (or user of a function) doing type-conversion when necessary is to use the type-conversion functions when defining functions to ensure that no later type-conversion is required. For example, one could define:

```
bettercnpow :: (Church (a —> a)) —> (Church (a —> a)) —> (Church a)
bettercnpow v w = w (uptype2 v)
```

This function has both parameters with identical type, hence it can be used in programs in which those parameters to be bound to the same variable, without requiring the type-system (or caller) to do type-conversion. Indeed, if one could use type-conversion functions to turn user-defined functions into semantically-identical versions with each input and output is typed identically, then those new functions could be freely combined.

A2.5 More type-conversion functions

Type-conversion functions for both directions are required, but do not always exist; more specifically in a given program the particular type-conversions one may desire can be impossible to accomplish

As well as 'uptype' and 'uptype2', type-conversion functions in the opposite direction are required for general programming, ie from a simpler to more-complex type. We begin with the situation described in the TFP paper [BK03a]. The definitions used are (renamed slightly to match our conventions, and with types added):

```
cnzero :: (Church a)
cnzero f x = x

cnsucc :: (Church a) —> (Church a)
cnsucc n f x = f (n f x)

cnadd :: (Church (Church a)) —> (Church a) —> (Church a)
cnadd v w = v cnsucc w

cnmul :: (Church (Church a)) —> (Church (Church a)) —> (Church a)
cnmul v w = v (cnadd w) cnzero

cnpow v w = w (cnmul v) (cnsucc cnzero)
```

These definitions result in a type-error occuring when calling 'cnpow' with the arguments 'cnone' and 'cntwo'. This occurs because the function iterated in 'cnpow', namely 'cnmul v', is

of type '(Church (Church a)) —> (Church a)' rather than of type 't —> t', for some 't', and hence cannot be iterated via application of an arbitrary Church Natural. This will be reflected in its inferred type.

The specific type-conversion required in the above case, call it 'downtype', is one which takes a Church Natural of type 'Church a' into one of type 'Church (Church a)'. With such a function, one could write:

```
cnpow v w = w (λc. cnmul v (downtype c)) (cnsucc cnzero)
```

and have the type of this function reflect the fact that it can be used for arbitrary Church Naturals. Unfortunately, no implementation of the type-conversion 'downtype' on Church Naturals can exist, as we show below. Hence in general, use of type-conversion functions cannot solve the typability issues of Hindley-Milner, for TFP-style programming.

A2.5.1 Non-existence of 'downtype'

One specific type-conversion, 'downtype', cannot be implemented

That 'downtype' is impossible to construct is proved now via contradiction. If an implementation of 'downtype' existed, then one could define 'cnpow' as:

```
cnzero f x = x
cnsucc n f x = f (n f x)

cnmul:: (Church a) —> (Church a) —> (Church a)
cnmul v w f x = v (w f) x

cnpow:: (Church a) —> (Church a) —> (Church a)
cnpow v w = (downtype w) (cnmul v) (cnsucc cnzero)
```

Note the type of 'cnpow'. Now, in [Sch76] and [Sta79] has been shown that the only functions of type 'T —> .. —> T' where 'T' is '(Church a)' are the 'extended polynomials', which don't include exponentiation. From [Sch76], the 'extended polynomials' are the functions on variables able to be generated by Church Natural constants 'cnzero', and 'cnone', constant functions (eg '(λx. cnthree)') addition ('(λv,w,f,x. v f (w f x))'), multiplication ('(λv,w,f,x. v (w

f) x)') and the test for zero ('(λn,q,r,f,x. n (λw. r f x) (q f x))'). Slightly different ways of generating the 'extended polynomials' also exist: compare the above with [FLO83], [Bar84, p564] and [MMZ97].

Hence no implementation of 'downtype' can exist.

More strongly, no function with the same type as 'downtype' but semantically-redundant *at least when used in the definition of 'cnpow' as above (but not necessarily otherwise)*, can exist. This rules out the possibility of there existing a set of approximations to 'downtype', each of which being semantically-redundant only in particular situations, but with the set giving complete coverage. Simply, type-conversion from 'Church a' to 'Church (Church a)' is impossible in at least some situations; and this acts as an existence-proof that not all type-conversions one may need to do in a given program are possible to accomplish.

A2.5.2 Approximation to 'downtype'

In many situations, 'downtype' can be successfully approximated by a single function

While 'downtype' is not implementable, we have discovered an approximation to it which is valid in many situations. It is:

$\text{downtype}_{\text{approx}}$ n f x g y = n (f cnzero g) (x g y)

The function '$\text{downtype}_{\text{approx}}$' is an approximation because it works (ie is semantically-redundant) only for certain 'f', 'x', 'g', 'y', as we characterise soon. It is however correct over the range of 'f', 'x', 'g' and 'y' found in its use in our motivating example from [BK03a]; ie the following are type-correct:

```
cnpow :: (Church (Church a)) —> (Church (Church a)) —> (Church a)
cnpow v w = w (λc. cnmul v (downtype_approx c)) (cnsucc cnzero)
```

Or, alternatively:

```
cnpow :: (Church (Church a)) —> (Church (Church (Church a))) —> (Church (Church a))
cnpow v w = w (λf. downtype_approx (cnmul v f)) (cnsucc cnzero)
```

A2.5.2.1 Applicability

The applicability of the approximation is established to be determined by its second parameter

We now determine the limit of applicability of this approximation 'downtype$_{approx}$' via induction, and follow with some examples. It turns out that the approximation is valid when the second argument is the standard successor implementation or partially-applied addition, but not partially-applied multiplication.

Firstly, some notation. We denote repeated application via superscripts, and denote the numeric value a Church Natural 'n' represents by '|n|'.

Inductively, 'n (f cnzero g) (x g y) = n f x g y' if:

Base-case: 'n' is 'cnzero'

cnzero (f cnzero g) (x g y)

= x g y

= cnzero f x g y

as required.

Step-case: 'n' is 'cnsucc m', for some 'm'

(cnsucc m) (f cnzero g) (x g y)

= (f cnzero g) (m (f cnzero g) (x g y))

Now, via the induction hypothesis, that 'm (f cnzero g) (x g y) = m f x g y':

= (f cnzero g) (m f x g y)

Recall that we are trying to show that this equals

(cnsucc m) f x g y

ie: f (m f x) g y

Denote 'm f x' by 'o'; then we desire that:

(f o) g y = (f cnzero g) (o g y)

ie: f o g y = f cnzero g (o g y)

In other words, we need that:

f = (λo,g,y. (f cnzero g) (o g y))

Now, as 'f' is a function from Church Natural to Church Natural, '(λz. f cnzero g z)' equals 'v g', for some Church Natural 'v'; we need to show that:

∃v:ChurchNatural. f = (λo,g,y. (v g) (o g y)) ∧ v = f cnzero

Note that 'f = (λo,g,y. (v g) (o g y))' implies that 'v = f cnzero':

f cnzero

= (λo,g,y. (v g) (o g y)) cnzero

= (λg,y. (v g) (cnzero g y))

= (λg,y. v g y)

= v

and so the above simplifies to:

∃v:ChurchNatural. f = (λo,g,y. (v g) (o g y))

also writable as:

∃v:ChurchNatural. f = (λo,g,y. $(g^{|v|})$ (o g y))

ie:

∃k:Natural. f = (λo,g,y. (g^{k}) (o g y))

In other words, '$\text{downtype}_{\text{approx}}$' will have correct semantics for any 'f' satisfying the above condition.

A2.5.2.2 Positive examples

The applicability of the approximation is illustrated via three examples

We now give three common examples of 'f'-parameters satisfying the above condition for semantically-correct use of '$\text{downtype}_{\text{approx}}$'.

The first example is the identity function. The proof of that it satisfies the condition is that:

(λo.o)

= (λo,g,y. o g y)

= (λo,g,y. g^{0} (o g y))

Our second example is the canonical definition of successor for Church Naturals as '(λn,f,x. f (n f x))'. The proof is:

(λn,f,x. f (n f x))

= (λo,g,y. g^{1} (o g y))

Based on 'cnsucc' satisfying the condition, it immediately follows that '$downtype_{approx}$' in the below definition of 'cnadd' is also semantically-correct:

cnadd :: (Church a)─> (Church a) ─> (Church a)

cnadd v w = ($downtype_{approx}$ v) cnsucc w

This 'cnadd' is better-typed than the standard definition, because the inputs and outputs are the same type. Hence it can be given the same variable twice as input, as well as have its output and one of its inputs also passed to another instance of 'cnadd' etc.

While we are on the topic of such better-typed functions, the work of [Lei90] should be pointed out. It states that if a function on Church Naturals (it also deals with others) can be implemented so that it has the same input as output type in the Simple typing, then it can *always* be implemented so as to be of Hindley-Milner type '∀a. T─>..─>T', where 'T' is '(a ─> a) ─> a ─> a'. Further information about which operations have typable implementations in the Simple typing when inputs and outputs are Church Naturals or other functional data representations can be found in [Zai95] (these results generalise those from [Sch76] covered previously).

Our third and final example is partially-applied addition of Church Naturals, where the addition is '(λv,w. v f (w f x))'. The proofs for the two cases of partial-application are:

(λw. cnadd v w)

= (λw,f,x. v f (w f x))

= (λw,f,x. $f^{|v|}$ (w f x))

= (λo,g,y. $g^{|v|}$ (o g y))

as required.

(λv. cnadd v w)

= (λv,f,x. v f (w f x))

which via a property of Church Naturals:

= (λv,f,x. w f (v f x))

= (λv,f,x. $f^{|w|}$ (v f x))

= (λo,g,y. $g^{|w|}$ (o g y))

as required.

We note that as a consequence of this, it follows that the below nicely-typed definition of 'cnmul' is also semantically-correct:

```
cnadd v w f x = v f (w f x)

cnmul :: (Church a) —> (Church a) —> (Church a)
cnmul v w = (downtype_approx w) (cnadd v) cnzero
```

Likewise, so is 'cnpow' when defined as:

```
cnmul ::  (Church a) —> (Church (Church a)) —> (Church a)
cnmul v w = w (cnadd v) cnzero

cnpow :: (Church a) —> (Church (Church (Church a))) —> (Church a)
cnpow v w = uptype (w (λf. downtype_approx (cnmul v f)) (cnsucc cnzero))
```

A2.5.2.3 Negative examples

Three examples illustrate situations in which the approximation is inapplicable

We now give three examples of 'f'-parameters which don't satisfy the applicability condition, and hence for which the approximation fails.

Firstly, we have constant functions, ie '(λn. z)', where 'z' is any Church Natural. The proof of this is as follows. To begin with, note that 'f' which satisfy the condition, ie equal '(λo,g,y. (g^k) (o g y))' for some 'k', are strict on their first argument for *some* (but not all) values of their second argument. In contrast, '(λn.z)' is *never* strict on its first argument, regardless of the value of the second argument. Hence '(λn.z)' is not one of the 'f' which satisfy the condition, ie there is no 'k' such that '(λn. z)' equals '(λo,g,y. (g^k) (o g y))'.

Secondly, we present the other definition of the successor operation on Church Naturals, 'cnsucc n g y = n g (g y)'. The proof regarding this is similar to the one for '(λn.z)': '(λn,g,y. n g (g y))' is always strict on its first argument, while none of the 'f' which satisfy the condition are always strict on their first argument.

Thirdly and finally we have partially-applied multiplication of Church Naturals, even if the operand is a constant, and *regardless of the definition of multiplication used*. The cause of the failure is that '$\text{downtype}_{\text{approx}}$' will pass that partial-application 'cnzero', and multiplying anything by zero gives zero. As a consequence, the semantics are incorrect, ie:

$\text{downtype}_{\text{approx}}$ n (cnmul w) x g y
= n ((cnmul w) cnzero g) (x g y)
= n (cnzero g) (x g y)

We reach the same point for the other case of partially-applied multiplication:

$\text{downtype}_{\text{approx}}$ n (λv. cnmul v w) x g y
= n ((λv. cnmul v w) cnzero g) (x g y)
= n ((cnmul cnzero w) g) (x g y)
= n (cnzero g) (x g y)

Further simplifying:

n (cnzero g) (x g y)
= n ((λh,z. z) g) (x g y)
= n (λz. z) (x g y)

which, as 'n' is, or at least has the semantics of, a Church Natural:

= x g y

Hence, '$\text{downtype}_{\text{approx}}$ n (cnmul w)' has the semantics of the identity-function, which is clearly incorrect as 'n (cnmul w)' certainly has different semantics. That '$\text{downtype}_{\text{approx}}$' fails with partially-applied multiplication is to be expected, as recall that the proof given previously, regarding the non-existence of an implementation of 'downtype', involved type-conversion with partially-applied multiplication.

A2.6 Conclusions

Hindley-Milner typing is impractical for TFP-style programming

Type-conversion functions make Hindley-Milner typing nearly practical for TFP-style programming. Unfortunately in some situations the appropriate type-conversion function, and even approximations to them, may not exist. One is then reduced to trying to find alternative, differently-typed, implementations of functions; or trying significantly rewriting the program. This makes TFP-style programming impractical under Hindley-Milner typing.