View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Queensland eSpace

THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Denotational Trace: A Category-Theoretic Solution to the Practical Problems of Software Execution Tracing

Leighton Brough

B. E. (Comp. Sys., Hons.), B. Sc. (Comp. Sci.)

A thesis submitted for the degree of Doctor of Philosophy at The University of Queensland in 2015

School of Information Technology and Electrical Engineering

Abstract

Here, the use of category theory, long advocated as applicable for manipulating many abstractions found in computer programming, is extended to serve additionally as a meta-level design tool to address a practical, software engineering problem: the design and use of effective software execution tracing systems. Software execution tracing is a widely used technique arising out of basic debugging practices with diverse implementations found in many software engineering processes, but it lacks a useful, theoretical foundation, resulting in engineering problems at several levels: for the designers, implementers and users of tracing systems all of whom have no recourse to rigorously defined and clearly understood engineering abstractions, when it comes to either implementing or using tracing effectively. Category theory has been used widely and successfully in computer science for decades, and more recently for a variety of applications in software engineering, establishing it as a key mathematical basis for this engineering discipline. Practical execution tracing systems have not used the existing theoretical concept of trace already provided by computer science – the trace monoid – because monoidal traces, while nevertheless useful in the theoretical investigation of concurrent and non-deterministic systems, are the consequence of an operational view of semantics, and therefore lack the complex, compositional structure required to encode much of the source-oriented information needed in practical execution traces. As a consequence of the categorical duality between operational and denotational semantics, the novel, dual notion of 'denotational trace' is introduced, where a 'canonical' denotational trace contains the complete collection of source-oriented and compositional, denotational semantic information required for practical execution tracing activities. To highlight the practical, straightforward nature of both denotational tracing implementation and usage, a canonical, denotational tracer is implemented for a simple language, sufficient to provide some simple examples of nevertheless sophisticated uses for denotational traces, including a specification recovery from execution trace and analyses of space and time complexity using formal reasoning applied to traces (i.e., proof by induction, as implied by categorical reasoning based on the denotational basis, for which induction is the associated proof technique). The novel application of category theory as a meta-level design tool, to the long standing software engineering problem of designing and using effective software execution tracing systems, has resulted in an elegant and systematic solution to those problems, further demonstrating that category theory has much to offer software engineering as a practical toolset based on a solid theoretical grounding. Thus category theory has delivered another useful result in software engineering as further evidence of its general applicability to this field, thereby reinforcing its usefulness for modeling, design and implementation, and suggesting this toolset should become a standard part of the software engineering curriculum.

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. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis.

Publications during candidature

- P. Bailes, L. Brough and C. Kemp, "From Computer Science to Software Engineering - a programming-level perspective", to appear in Proceedings of The 13th International Conference on Intelligent Software Methodologies, Tools, and Techniques, SOMET 14
- L. Brough and P. Bailes, "The Denotational Basis for Software Execution Tracing", Proceedings of the 8th IASTED International Conference on Advances in Computer Science, ACS 2013, Phuket, Thailand, (278-286). 10 - 12 April 2013.
- P. Bailes, L. Brough and C. Kemp, "Higher-Order Catamorphisms as Bases for Program Structuring and Design Recovery", in Proceedings of the 12th IASTED International Conference on Software Enginnering, SE 2013, Innsbruck, Austria, (775-782). 11 - 13 February 2013.
- P. Bailes and L. Brough, "Making Sense of Recursion Patterns", in Proceedings of Formal Methods in Software Engineering: Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, (16-22). 2 June 2012.

Publications included in this thesis

- L. Brough and P. Bailes, "The Denotational Basis for Software Execution Tracing", Proceedings of the 8th IASTED International Conference on Advances in Computer Science, ACS 2013, Phuket, Thailand, (278-286). 10 - 12 April 2013.

Contributor	Statement of contribution
L. Brough (candidate)	Wrote the paper (80%), an increasing proportion with each draft
P. Bailes	Wrote the paper (20%), in particular the example of specification recovery

Contributions by others to the thesis

No contributions by others.

Statement of parts of the thesis submitted to qualify for the award of another degree

None.

Acknowledgements

I will always be grateful to my alma mater, the University of Queensland, being not only a beautiful campus but the forge in which my youth was struck. UQ has provided me not only with my professional education, but also many of my oldest, dearest friends including my wife, intellectual stimulation on-and-off over more than two decades, and also a convenient source of employment from time-to-time. Specifically, I am emdebted to the ITEE School for providing a Confirmation Scholarship which partially funded this project.

Of far more practical and personal significance is the education and on-going conversation in computer science provided to me by my advisor Prof. Paul Bailes. Paul's unfailing patience, tolerance, but nevertheless insistence on clearly enunciatiated concepts derived from the fundamentals is something not only I hope will be with me for life, but something I wish to spread to others. It has been a great pleasure – over almost a decade now – to not only find an intellectual mentor but develop a much-valued, enriching personal friendship.

This project would never have been possible if I had not had sound and stable personal circumstances; the unfailing support of my spouse Rachel has been fundamental. Since this project began, two beautiful children, Caitlin and James, have graced our lives, and life has presented many adventures. Nevertheless, this project has never been a sacrifice to be pursued at the expense of other values, but rather an enriching pleasure to be pursued for its own sake – an enterprise impossible without the support of a like-minded student-in-life.

Keywords

category theory, semantics, execution tracing, trace analysis

Australian and New Zealand Standard Research Classifications (ANZSRC)

ANZSRC code: 080202 Applied Discrete Mathematics, 50%

ANZSRC code: 080309 Software Engineering , 50%

Fields of Research (FoR) Classification

FoR code: 0802, Computation Theory and Mathematics, 50%

FoR code: 0803, Computer Software, 50%

Contents

1 Introduction **17**

1.1 Motivation and Aims . . . 17

1.2 Why Category Theory . . . 18

2 Category Theory for Software Engineering **23**

2.1 Categories . . . 23

2.1.1 Associativity . . . 26

2.1.2 Identity . . . 27

2.2 Exploiting Associativity and Identity . . . 30

2.2.1 Monads . . . 32

2.2.2 Functors . . . 36

2.2.3 Initial Algebras . . . 39

2.3 Duality . . . 42

2.4 Applications . . . 46

3 Tracing in Practice **48**

3.1 Origins . . . 48

3.2 Examples . . . 51

3.2.1 Software Engineering Processes . . . 51

3.2.2 Application Domains . . . 53

3.2.3 Implementation Approaches . . . 54

3.3 Practical Requirements . . . 57

3.3.1 Source Orientation . . . 65

3.3.2 Compositional Structure . . . 66

3.4 Generalised Tracing . . . 67

3.5 Engineering Problems . . . 68

4 Theoretical Foundations for Tracing **70**

4.1 Origins . . . 70

4.2 Operational Semantics . . . 71

4.3 The Trace Monoid . . . 73

4.4 Application Domains . . . 75
4.5 Fit with Tracing in Practice . . . 75

5 Denotational Trace **77**
5.1 Denotational is Dual to Operational . . . 78
5.1.1 Operational Semantics are Final-Coalgebraic . . . 79
5.1.2 Denotational Semantics are Initial Algebraic . . . 80
5.2 Tracing Denotational Semantics . . . 82
5.2.1 Monads for Modular Semantics . . . 82
5.2.2 Canonical Tracing . . . 83
5.2.3 Canonical Traces . . . 84
5.3 Two Sides to the Semantic Coin . . . 85

6 A Denotational Tracer **89**
6.1 Executable Semantic Specifications . . . 90
6.2 A Simple Denotational Interpreter . . . 90
6.3 Trace Generation . . . 94
6.4 A Simple Example Program . . . 96
6.4.1 Specification Recovery . . . 99
6.4.1.1 ‘readable’ Trace Analysis Tool . . . 101
6.4.2 Time Complexity . . . 102
6.4.2.1 ‘timecomp’ Trace Analysis Tool . . . 105
6.4.3 Space Complexity . . . 106
6.4.3.1 ‘spacecomp’ Trace Analysis Tool . . . 106

7 Conclusions **108**
7.1 Summary of Results . . . 108
7.2 Significance . . . 110
7.2.1 Modeling of the Problem . . . 110
7.2.2 Design of the Solution . . . 111
7.2.3 Implementation of an Example . . . 111
7.3 Implications and Future Directions . . . 111
7.3.1 Further Exploration of Tracing via Category Theory . . . 112
7.3.2 Categories as Domain Specific Languages . . . 112
7.3.3 Categorical Programming Language Design . . . 112
7.3.4 Category Theory in Software Engineering Education . . . 113

Preface

The presentation of this thesis follows the *one-sentence* synopsis convention: at each level of the document structure, starting with the entire document itself – then the chapters, sections, subsections and so forth – a précis of the contents is provided, in the form of a single sentence. For the convenience of the reader, the next section provides a summary of all of the summary sentences in this thesis, being a précis of the entire document, organised as per its hierachical structure. This presentation makes explicit the logical and hierachical structure to the argument presented in this thesis, with summary sentences for each part of the thesis being justified by or developed in the sub-sentences for each of the corresponding document substructures.

Denotational Trace:
A Category-Theoretic Solution to the Practical Problems of Software Execution Tracing

A novel application of category theory as a meta-level design tool, to the long standing software engineering problem of designing and using effective software execution tracing systems, has resulted in an elegant and systematic solution to those problems, further demonstrating that category theory has much to offer software engineering as a practical toolset based on a solid theoretical grounding.

1 Introduction **17**

Here, the use of category theory, long advocated as applicable for describing and manipulating many abstractions found in computer programming, is extended to serve additionally as a meta-level design tool to address a practical, software engineering problem: the design and use of effective software execution tracing systems.

1.1 Motivation and Aims 17

The practice of engineering in any discipline rests on the foundation of rigorously defined, clearly understood, engineering abstractions based on sound, justified, mathematical theory; here category theory is explored as just such a basis for the practical problem of software execution tracing, sufficient to support the practical requirements of tracing system designers and users.

1.2 Why Category Theory 18

Category theory was chosen as a meta-level design tool because isomorphism makes it possible to discuss tracing in abstract, categorical terms, independent of language syntax and semantics, and thereby use the fact that every category has a dual to show there exists a unique alternative to the existing notion of trace found in computer science.

2 Category Theory for Software Engineering **23**

Category theory has been used widely and successfully in computer science for decades, and more recently for a variety of applications in software engineering, establishing it as a key mathematical basis for this engineering discipline.

2.1 Categories 23

Consisting of just the basic template of an associative composition operation, categories can be used to describe many composable artefacts found in computing.

2.1.1 Associativity 26

By enforcing the associative law, categories provide a minimal and abstract, yet effective system of composition for a wide variety of possible applications.

2.1.2 Identity 27

With the addition of the identity law, categories support a precise notion of abstract sameness, isomorphism.

2.2 Exploiting Associativity and Identity 30

Categories provide an effective basis for domain specific languages both in mathematics where they originated and in computer programming, and come equipped with a notion of equivalence that gives precise description to a programmer's intuition as to how two program artefacts may be abstractly 'the same', thereby establishing category theory not only as a mathematical basis for programming but as a rigorously specified program design pattern catalogue.

2.2.1 Monads 32

The monad extends the categorical notion of composition to include functions that manipulate state or return exceptions, perform input and output and exhibit a wide variety of other program phenomena including non-determinism, concurrency and parsing.

2.2.2 Functors 36

Functors provide a mechanism to combine the functionality of various categories in a consistent fashion that preserves the category laws; a pattern already familiar to programmers as type-level operators, as well as many other program phenomena.

2.2.3 Initial Algebras 39

Initial algebras can describe not only concrete and abstract data types, but the pattern of recursion naturally associated with these, in generic terms, for any type.

2.3 Duality 42

Category theory is useful not only for describing categories but also for manipulating and reasoning about them at the meta-level; categorical duality ensures that every category has a dual, i.e., every category-theoretic structure has a unique (up to isomorphism) 'opposite'.

2.4 Applications 46

This project adds to the diverse range of applications to which category theory has been successfully applied in both computer science and software engineering, by using several categorical ideas already well established in computing, for the novel application of systematically exploring the semantic foundations for practical software execution tracing.

3 Tracing in Practice 48

Software execution tracing is a widely used technique arising out of basic debugging practices with diverse implementations found in many software engineering processes, but it lacks a useful, theoretical foundation, resulting in engineering problems at several levels: for the designers, implementers and users of tracing systems all of whom have no recourse to rigorously defined and clearly understood engineering abstractions, when it comes to either implementing or using tracing effectively.

3.1 Origins 48

Execution tracing arises naturally as a systematic extension of basic debugging practices.

3.2 Examples 51

Execution tracing is a widely used technique, with diverse implementations found in many software engineering processes and application domains.

3.2.1 Software Engineering Processes 51

Tracing, by providing an aid to program comprehension, is applicable to a range of software engineering processes.

3.2.2 Application Domains 53

Tracing is found in diverse application domains, including the development of object-oriented systems, concurrent and distributed systems, and servers and operating systems.

3.2.3 Implementation Approaches 54

A diverse range of implementations exist for practical tracing systems, the norm being ad hoc solutions, tuned to the problem at hand.

3.3 Practical Requirements 57

Practical traces, useful for software engineering tasks, need to refer to entities in the program source; this source-orientation in turn requires traces to be compositional, and contain events including information of varying types.

3.3.1 Source Orientation 65

Because all activities involving execution tracing involve some aspect of program comprehension, traces must refer to entities in the program source text, i.e., be source-oriented.

3.3.2 Compositional Structure 66

Source-orientation in turn requires a trace to reflect complex, nested structures, corresponding to those found in the source code.

3.4 Generalised Tracing 67

Category theory has not previously been applied to the software engineering problem of designing and using effective execution tracing, and there has otherwise been little attention paid to the abstract, theoretical basis for practical execution tracing systems.

3.5 Engineering Problems 68

The lack of a sound and useful theoretical basis for practical execution tracing results in engineering problems at several levels; for the designers, implementers and users of tracing systems all of whom have no access to sound, useful theoretical guidance or tools.

4 Theoretical Foundations for Tracing 70

Practical execution tracing systems have not used the existing theoretical concept of trace already provided by computer science – the trace monoid – because monoidal traces, while nevertheless useful in the theoretical investigation of concurrent and non-deterministic systems, are the consequence of an operational view of semantics, and therefore lack the complex, compositional structure required to encode much of the source-oriented information needed in practical execution traces.

4.1 Origins 70

The term 'trace' has been linked to an operational model of semantics since the early days of computer science.

4.2 Operational Semantics 71

The central idea of operational semantics is that of the transition system, the operation of which automatically induces a trace.

4.3 The Trace Monoid 73

Execution of a transition system inherently generates a monoid.

4.4 Application Domains 75

The trace monoid is the foundation of process algebrae, and has been useful in the study of concurrency and non-determinism.

4.5 Fit with Tracing in Practice 75

The trace monoid is too simple a structure to encode the compositional, source-oriented details of practical interest for many tracing activities.

5 Denotational Trace **77**

As a consequence of the categorical duality between operational and denotational semantics, the novel, dual notion of 'denotational trace' is introduced, where a 'canonical' denotational trace contains the complete collection of source-oriented and compositional, denotational semantic information required for practical execution tracing activities.

5.1 Denotational is Dual to Operational 78

Denotational semantics is dual to operational semantics.

5.1.1 Operational Semantics are Final-Coalgebraic 79

The basis for the structure of operational semantics is the abstract machine, or in categorical terms, the final coalgebra.

5.1.2 Denotational Semantics are Initial Algebraic 80

Final coalgebras have a dual: initial algebras, that provide the structural basis for denotational semantics, otherwise known as initial algebra semantics.

5.2 Tracing Denotational Semantics 82

The transformation of denotational semantics by the addition of execution tracing is modelled by an algebra homomorphism, more specifically a canonical isomorphism.

5.2.1 Monads for Modular Semantics 82

Monads allow denotational semantics to be structured in a modular fashion.

5.2.2 Canonical Tracing 83

The morphism required to transform an initial algebra describing denotational semantics into semantics augmented with tracing output is a homomorphism, more specifically an isomorphism, which should be canonical with respect to the original semantics.

5.2.3 Canonical Traces 84

We call the maximal set of trace information available from a dual-consequent, denotational perspective, a "canonical trace", when produced by canonical tracing.

5.3 Two Sides to the Semantic Coin 85

Operational and denotational traces are complementary.

6 A Denotational Tracer **89**

To highlight the practical, straightforward nature of both denotational tracing implementation and usage, a canonical, denotational tracer is implemented for a simple language, sufficient to provide some simple examples of nevertheless sophisticated uses for denotational traces, including a specification recovery from execution trace and analyses of space and time complexity using formal reasoning applied to traces (i.e., proof by induction, as implied by categorical reasoning based on the denotational basis, for which induction is the associated proof technique).

6.1 Executable Semantic Specifications 90

The denotational semantics specified here are implemented in the meta-language Haskell; execution of the denotational specification constitutes an interpreter, a useful experimental tool.

6.2 A Simple Denotational Interpreter 90

A simple language with just enough structure to exercise some interesting examples is constructed: the syntax is a simple variant of ML-style languages, the semantics are simple, with a strict/eager execution order, and the value domain contains just a few basic types.

6.3 Trace Generation 94

In a straightforward way, canonical trace structures are defined for the language, and modified semantics with tracing added are generated by application of a suitable monad transformer.

6.4 A Simple Example Program 96

As a demonstration of the practical usefulness of denotational trace to users of tracing systems, a simple example program is executed using the denotational tracer and the resulting execution traces are used to perform several software engineering tasks, including specification recovery and analyses of space and time complexity, with mathematical induction being a natural proof technique.

6.4.1 Specification Recovery 99

Denotational trace is used to recover a specification for the program.

6.4.1.1 ‘readable’ Trace Analysis Tool 101

More specific traces can be derived from canonical traces via structural induction over the trace structures.

6.4.2 Time Complexity 102

The time complexity of the program is analysed using proof by induction.

6.4.2.1 ‘timecomp’ Trace Analysis Tool 105

A measure of algorithmic time complexity can be derived by structural induction over trace structures.

6.4.3 Space Complexity 106

The space complexity of the program is analysed using proof by induction.

6.4.3.1 ‘spacecomp’ Trace Analysis Tool 106
A measure of algorithmic space complexity can be derived by structural induction over trace structures.

7 Conclusions 108
Thus category theory has delivered another useful result in software engineering as further evidence of its general applicability to this field, thereby reinforcing its usefulness for modeling, design and implementation, and suggesting this toolset should become a standard part of the software engineering curriculum.

7.1 Summary of Results 108
Category theory has delivered another useful result for the engineering of software by identifying, justifying and describing precisely a unique, abstract alternative to the existing notion of the trace monoid, that turns out to fit well with the requirements for practical tracing systems, and provides an elegant, integrated solution to the software engineering problems inherent in tracing systems of ad hoc design.

7.2 Significance 110
Category provides a powerful and appropriate mathematical basis for software engineering.

7.2.1 Modeling of the Problem 110
The flexible and powerful notions of the category and categorical notions of sameness, provide an effective basis for modeling software abstractions at many levels.

7.2.2 Design of the Solution 111
Tools such as duality can be used for the design of justified, valid solutions.

7.2.3 Implementation of an Example 111
Category theory provides a design pattern catalogue for program implementation.

7.3 Implications and Future Directions 111
There is interesting work remaining to be done in exploring how operationally and denotationally based traces are complementary, based on the duality and correspondence between operational and denotational semantics, and more broadly in continuing to explore the uses of category theory in software engineering.

7.3.1 Further Exploration of Tracing via Category Theory 112
There is interesting work remaining to be done in exploring how operationally and denotationally based traces are complementary, based on the duality and correspondence between operational and denotational semantics.

7.3.2 Categories as Domain Specific Languages 112
Categories provide an effective template for domain specific languages.

7.3.3 Categorical Programming Language Design 112
While categorical ideas can be ported into any programming language, the usefulness of category theory to software engineering suggest that programming languages should explicitly support categorical notions, to derive the full benefits of automated tool support.
7.3.4 Category Theory in Software Engineering Education 113
Given the useful and appropriate tools that category theory brings to software engineering, and the promise it offers as a mathematical basis for robust software engineering abstractions, category theory should become part of the software engineering curriculum.

List of Tables

1.1 The Argument from Categorical Duality for Denotational Trace 19

2.1 Initial Algebras vs Final Coalgebras 46

5.1 The Argument from Categorical Duality for Denotational Trace, Revisited 78
5.2 Duality Quick Reference 86

6.1 Time Complexity of “f” 102
6.2 Space Complexity of “f” 106

Chapter 1

Introduction

Here, the use of category theory, long advocated as applicable for describing and manipulating many abstractions found in computer programming, is extended to serve additionally as a meta-level design tool to address a practical, software engineering problem: the design and use of effective software execution tracing systems.

1.1 Motivation and Aims

The practice of engineering in any discipline rests on the foundation of rigorously defined, clearly understood, engineering abstractions based on sound, justified, mathematical theory; here category theory is explored as just such a basis for the practical problem of software execution tracing, sufficient to support the practical requirements of tracing system designers and users.

This project springs out of the conviction that software engineering and computer science are both at their best when operating as complementary disciplines: just as older fields of engineering rest on mathematical, theoretical foundations found in the physical sciences, the practical problem of developing and maintaining useful computer programs is tackled most systematically and effectively when it is grounded in relevant computer science. Category theory has long been advocated as useful in both computer science and software engineering (see chapter 2). This thesis provides further evidence of its usefulness by applying it as a strategic design tool at multiple levels to a commonly used technique in software engineering: software execution tracing (see chapter 3).

Execution tracing – a widely-used and effective practical technique presenting non-trivial design, implementation and use issues – has spawned a considerably body of peer-reviewed research documenting the variety of software processes (see section 3.2.1) and applications (see section 3.2.2) where it is found, and the various implementation techniques used (see section 3.2.3). Nevertheless, the topic of the theoretical foundations of tracing as it used in

practice has been neglected (see section 3.4), resulting in most extant tracing systems having the problem of an ad hoc rather than an engineered design (see section 3.5). The ad hoc approaches currently in use produce pragmatic, but nevertheless arbitrary designs for tracing systems, being a product of the accidents of the programmer's experience, the requirements of the specific problem domain being addressed, and the available tools at hand. Furthermore, no guidance is provided as to how the resultant traces can or should be used. Finally, automated tool support for working with ad hoc traces is difficult to provide, due to the absence of a clearly specified understanding of what traces are, what they contain and how these contents relate to program semantics.

Despite the diverse tracing systems in existence, they share some practical requirements in common; they provide source-oriented, compositional information about the execution of the program (see section 3.3). These practical requirements are not supported by the existing notion of trace provided by computer science (see chapter 4, which nevertheless provides a sound and useful basis for the theoretical investigation of concurrency and non-determinism – see section 4.4). An effective theoretical basis for practical tracing has been lacking.

This project uses category theory to address the lack of a theoretical basis for execution tracing, thereby grounding this important practical activity in formal computer science by providing general, abstract principles underlying the design of useful tracing systems – i.e., that they have a denotational-semantic basis – as well as conceptual tools for reasoning about the traces generated (see chapter 5). Some simple examples of the nevertheless sophisticated tasks supported by denotational traces are provided in chapter 6, including a specification recovery from trace and formal reasoning about space and time complexity from traces.

Thus this thesis constitutes a proof-by-example that category theory is ideally suited to the task of describing and manipulating abstract concepts from the world of practical computing. The problem of execution tracing illustrates the fundamental point that category theory illuminates software engineering. In this case, not by producing yet-another-better trace system, but rather by providing a well-justified theoretical basis for all correctly engineered tracing systems and tools in general, adequate to support practical tracing requirements for any given language, independent of the specific details of syntax and semantics.

1.2 Why Category Theory

Category theory was chosen as a meta-level design tool because isomorphism makes it possible to discuss tracing in abstract, categorical terms, independent of language syntax and semantics, and thereby use the fact that every category has a dual to show there exists a unique alternative to the existing notion of trace found in computer science.

No prior knowledge of category theory is assumed on the part of the reader. As categorical concepts are used they are introduced with the necessary explanation.

Two specific benefits are derived from category theory here, the details of which are elaborated in chapters 4 and 5:

i. It order to achieve the aims of this project, it is necessary to discuss abstract notions of tracing (i.e., independently of any specific programming language), while nevertheless retaining mathematical clarity and precision. Without clarity of definition and precise reasoning, any discussion regarding abstract or general matters can quickly become uselessly vague.

   Category theory provides an effective solution to this problem. The fundamental, category-theoretic idea of *isomorphism* provides a useful notion of abstract sameness, which makes it possible to discuss execution trace independently of any specific programming language syntax or semantics. Nevertheless, this abstract treatment of tracing allows a precise specification of what a trace can contain, given a specification of syntax and semantics for a particular language. Isomorphism is explained in detail in section 2.1.2.

ii. Any programmer who might be starting from a point of dissatisfaction with the existing concept of the trace monoid found in computer science as a basis for a practical execution tracing system (the reasons why this happens are explored in details chapters 3 and 4) is apparently faced with an insurmountable task to select a suitable alternative from the multiplicity of potential designs, unconstrained by any guidance as to what constitutes a useful and effective design. Any search for an alternative model of execution trace would be presented with a seemingly infinite array of possibilities, with no suggestion as to which should be chosen.

    The fundamental, category theoretic concept of *duality* is used in this thesis to identify the single alternative to the existing, abstract notion of trace found in computer science (the trace monoid), by identifying a dual to the semantic basis for that abstraction. Duality is explained in detail in section 2.3.

The	(1) trace monoid	(8) denotational trace.	
is induced by	(2) operational semantics	(7) denotational semantics	that induces a
which has the abstract, categorical formalisation as	(3) final coalgebra semantics	(6) initial algebra semantics	the abstract, categorical formalisation of
that is semantics structured as	(4) a final object in the category of F-coalgebras.	(5) An initial object in the category of F-algebras	is the structural basis for

Table 1.1: The Argument from Categorical Duality for Denotational Trace

As an aid to the reader who is already well versed in computer science and the applications of category theory to computer science and software engineering, the argument from categorical duality that is presented in this thesis is summarised in table 1.1. A reader who is not already familiar with the concepts referred to in the table can find an explanation of these in the corresponding sections referenced below. In either case, the table provides a convenient reference as to the relevance of key notions introduced later. Table 1.1 is meant to be read down the two left-hand columns, then back up the two right hand columns, the argument proceeding as follows: The

1. trace monoid (see section 4.3) is induced by
2. operational semantics (see section 4.2) which has the abstract, categorical formalisation as
3. final coalgebra semantics (see section 5.1.1), that is, semantics structured as
4. a final object in the category of F-coalgebras (see section 2.3).

At this point, the argument quite literally hinges on the duality between initiality in the category of F-algebras and finality in the opposite category of F-coalgebras (see section 2.3). Having formally identified operational semantics as final object in the category of F-coalgebras, the categorical tool of duality is used to determine a unique, dual alternative: the initial object in the category of F-algebras (see section 2.2.3). Reversing the logic by which the basis for the trace monoid was deconstructed, an alternative theory of trace from this alternative, dual basis is built:

5. An initial object in the category of F-algebras (see section 2.2.3) is the structural basis for
6. initial algebra semantics (see section 5.1.2) the abstract, categorical formalisation of
7. denotational semantics (see section 5.1.2) that induces a
8. denotational trace (see section 5.2).

The generation of this novel concept of *denotational trace* would not have been possible without the use of category theory. Firstly, via the categorical notion of sameness provided by isomorphism it is possible to precisely characterise the mathematical basis for operational semantics, the semantic basis for the trace monoid. Then, via the categorical notion of duality, a unique and justified alternative basis was found for tracing. In chapter 5, it is shown that this alternative basis for tracing is not only sound and justified from a categorical perspective, but critically, also addresses the practical requirements for execution tracing (see section 3.3) unmet by the existing trace monoid (see section 4.5) — thereby solving the problem that motivated this thesis.

However, the use of category theory is not without challenges. A key difficulty in making use of category theory for software engineering (or any other discipline, for that matter) is inherent in the challenge of grasping any new, abstract notion; usually this is done by reflecting upon one or more familiar, motivating examples that are abstractly 'the same', as examples of the 'pattern' of interest. It is often the realisation that familiar artefacts previously considered distinct, are in fact examples of an abstract pattern shared in common, that provides an 'aha' moment where the abstraction is understood at least partially, if not in its full generality. Thus the examples presented here to illustrate key abstractions from category theory, are deliberately chosen to appeal to the intuitions of a programmer, be they a computer scientist or software engineer. This differs somewhat from the more usual presentation of category theory as it is described to a general, mathematical audience or for the computer scientist.

Mac Lane's seminal *Categories for the Working Mathematician* [Mac98] is considered the foundational textbook in category theory, and both Pierce [Pie91] and Asperti and Longo [AL91] have written books introducing category theory to computer scientists. However, textbooks explaining the relevance of category theory to the software engineer are yet to be written. In their absence a helpful source of category-theoretic examples appealing to the intuitions of programmers is the so-called 'blogosphere'. These Internet articles are written with the practical programmer in mind with a view to popularising category theory, as opposed to the peer-reviewed computer science literature which tends to be more mathematical in nature and speaking to a different audience who face different practical problems. Internet sources such as these will be cited sometimes, when they provide examples particularly well suited to a programmer, in preference to strict formal definitions or examples from mathematical fields normally unrelated to practical programming.

Here the goal when introducing each category theoretic abstraction is to foster a sound intuition as to why that abstraction matters, i.e., to *use* these abstractions to solve a real problem in practical software engineering, rather than provide exhaustive treatment of their definitions and properties. Here Strachey's injunction is followed: "our motto should be 'No axiomatisation without insight'" [Str00]. A mathematically-inclined reader can access a wide array of resources on category theory starting with the textbooks listed above in the previous section.

Category theory has been famously called "abstract nonsense" by enthusiasts and detractors alike. Behind this humorous comment lies a real, potential risk when using category theory: that it can be mis-applied; used where it adds no other value than a new set of obscure terminology, thereby turning an otherwise simple problem into one that is obfuscated by mathematics, resulting in high walls inaccessible to outsiders. This risk was recognised by Goguen, a pioneer in the use of category theory in computer science, who warned against "specious generality" and "categorical overkill" [Gog89]. Here, concepts and abstractions from category theory have been introduced sparingly and specifically to derive the benefits outlined above. Two concepts from category theory are leaned upon particularly heavily here in order to derive these benefits:

isomorphism and duality. But before these two ideas can be introduced, the basic concept of a category and its relevance to software engineering needs to be explained.

Chapter 2

Category Theory for Software Engineering

Category theory has been used widely and successfully in computer science for decades, and more recently for a variety of applications in software engineering, establishing it as a key mathematical basis for this engineering discipline.

2.1 Categories

Consisting of just the basic template of an associative composition operation, categories can be used to describe many composable artefacts found in computing.

On the face of it, from the pragmatic programmer's naïve point of view, it's not obvious there is any deep fit between mathematics and program code. The obvious mathematical model of functions as found in set theory makes a poor substitute for functions in programming. While it is certainly the case that it is possible to model many functions found in programming as a mathematical function between sets, in general set theory does not provide a framework adequate to describe typical program behaviours. For example, consider some Pascal function[1]:

```
function f ( a : Char ) : Char;
  begin
    result := b ( a );
  end;
```

We might attempt to model this in set theory as:

[1]In the Pascal language variant used for examples here and later, `result` is a special identifier for the function result, a convention found in the popular Borland implementations of the language. Otherwise these examples should be intelligible to anyone familiar with any variant of Pascal, or for that matter most programming languages since 1960.

$$f(a) = b(a) \tag{2.1}$$

where

$$f : \mathbf{Char} \rightarrow \mathbf{Char} \tag{2.2}$$

where **Char** is the set of ASCII characters[2].

However objections immediately arise:

1. What would b look like if `b` has either zero or multiple arguments, or doesn't return a result (i.e., is a procedure)?

2. What happens if an exception is thrown by the Pascal function `b`? What then is the type of the mathematical function b?

3. What if `b` performs some input or output? How do we then attempt to write b?

4. What if `b` modifies some internal or global state?

A software engineer is also likely to be considering questions such as:

- What if `b` enters an infinite loop and fails to terminate?

- What can I know about the memory consumption of mathematical function f?

In general, a programmer will have in mind computational considerations such as non-termination, non-determinism, concurrency, algorithmic space and time complexity, etc.

The first objection can be easily dismissed. Procedures can be easily accommodated via functions which return the unit type, often written as "`()`", known as "`void`" in the C programming language family, indicating a type which contains only one value, and therefore no information. Functions which take no arguments are handled similarly. Finally, a function of two (or more) arguments can be described equivalently as a function of a single argument that returns a function that in turn handles the next argument and so forth, i.e., the well known technique known as "Currying" in honour of the famous logician Haskell B. Curry (although the technique itself is due to Schönfinkel) [Rey72].

Nevertheless, basic set theory is not sufficient here; it has no answer for the remaining objections enumerated above. Instead, we are seeking a mathematical framework that is rich enough to describe the abstractions found in programming. In the words of Scott (in [Pie91], p. xi):

[2]Here the symbol ":" indicates a type signature, and can be pronounced, "has type," and "$\rightarrow$" is used in the conventional way to mean that the type on the left is the *domain* or input type, and the type on the right is the output type or *range*.

> What we are probably seeking is a "purer" view of functions: a theory of functions in themselves, not a theory of functions derived from sets. What then, is a pure theory of functions? Answer: category theory.

Asperti and Longo [AL91] explain how category theory answers Scott's question in their summary of the value of category theory to the computer scientist:

> ... a crucial point, though, is that the categorical notion of morphism generalizes the set-theoretical description of function in a very broad sense, which provides a unified understanding of various aspects of the theory of programs

The *morphisms* referred to by Asperti and Longo (sometimes called *arrows*), are fundamental to the structure of categories: categories are defined in terms of such morphisms between *objects*. By design, category theory says nothing about the internal details or structure of the objects, except to define objects solely in terms of the morphisms relating them together. Here we find Scott's "pure theory of functions", the notion of morphism is sufficiently general and abstract that it can accommodate notions found outside of set theory, in the domain of computer programming.

Definition 2.1. Every *category* consists of three components:

1. a collection of *objects*,
2. a collection of *morphisms* (also known as *arrows*) relating those objects, with each morphism having specific and in general different objects for their range and domain, and
3. a binary operation called *composition*, written as $\circ$ in infix manner, which composes morphisms (assuming the objects that are the range and domain of the morphism correspond).

   Thus a category $\mathbf{C}$ with morphisms:

$$f : A \to B \tag{2.3}$$

$$g : B \to C \tag{2.4}$$

$$h : C \to D \tag{2.5}$$

   over objects A, B, C and D has by definition a binary composition operation $\circ$ such that

$$g \circ f : A \to C \tag{2.6}$$

and

$$h \circ g : B \to D. \tag{2.7}$$

In addition a category must obey two laws:

1. The *associative law* requires that composition is an associative operation. For f, g and h in the category $\mathbf{C}$ above, this means:

$$h \circ (g \circ f) = (h \circ g) \circ f \tag{2.8}$$

   i.e., composition of multiple morphisms produces the same results regardless of the order in which sub-compositions are applied.

2. The *identity law* requires that categories have an identity morphism for every object, X:

$$1_X : X \to X \tag{2.9}$$

   such that for every morphism, e.g. f above with domain object A and range object B

$$1_B \circ f = f \circ 1_A = f. \tag{2.10}$$

   i.e., for every morphism there are both left and right identity morphisms under composition.

2.1.1 Associativity

By enforcing the associative law, categories provide a minimal and abstract, yet effective system of composition for a wide variety of possible applications.

Any (so-called composition) operator conforming to these laws above provides the basis for a category. In effect, these laws provide a system where morphisms can be 'composed' together to combine them, that simply 'works' as would reasonably be expected: the composition operator for any category is associative, i.e., three or more morphisms composed together in the same order, have the same meaning, regardless as to the order in which the composition operator is applied, or the expression is constructed.

Without this requirement that composition is associative, the semantics of morphisms when composed would be dependent on the order of definition of their sub-components. Historical issues due to the one-pass compilers of the past aside, computer programmers in general expect that the meaning of their programs or contents of their data structures are not affected by the precise order in which their various sub-components are declared or constructed, e.g., in the case

of the ubiquitous string of characters found in computer programming, a programmer certainly expects that `("a" + "b") + "c"` results in the same string as `"a" + ("b" + "c")` (where `+` performs string concatenation). Because it is associative, `+` provides a suitable composition operator as the basis for a category, where the empty string `""` acts as both left and right identity (in section 4.3 this category is identified as a monoid). The programmer's expectation is that the same string results for either example above, regardless as to the order in which the concatenation operators are applied. Conversely, if a composition operator were defined that was not associative, then the resulting system for the construction of strings would be difficult to use and reason about, because the order in which substrings are concatenated would matter, violating the programmer's expectation above.

The programmer's intuitive expectation of associativity extends to many other situations when programming, for example in a C/Java-like language where calls are made to some functions or methods `a`, `b` and `c`:

```
{
    a(...);
    b(...);
}
c(...);
```

is assumed to be effectively the same as:

```
a(...);
{
  b(...);
  c(...);
}
```

Here again, if we imagine the "`;`" syntax element as an infix statement separator (and ignore the trailing "`;`"s at the end of the blocks), then once again an associative assumption is at play: `{a ; {b ; c}}` is equivalent to `{{a ; b} ; c}`.

2.1.2 Identity

With the addition of the identity law, categories support a precise notion of abstract sameness, isomorphism.

Thus far the discussion above has focused on how the associative category law enforces a system where composition behaves reasonably, raising the question as to what purpose the identity law serves. For the first of the examples given in the previous section, it is clear that character strings would be more difficult and in some cases impossible to work with in programs if there

was no way to specify an empty string. More generally however, identity is an essential property for a category, because the existence of identity morphisms in a category makes it possible to define the fundamental, categorical notion of isomorphism. This concept of isomorphism is of key relevance to programmers because isomorphism provides a precise and formal definition of their intuition as to the 'abstract sameness' between two (categorical, software) artefacts. Isomorphism provides a looser, but nevertheless precise, notion of sameness than the familiar mathematical idea of equality.

Mazur's *When is one thing equal to some other thing?* [Maz08] provides a brief but excellent explanation of how isomorphism as a notion of abstract sameness is central to the basic aims, capabilities and concepts of category theory[3]. Mazur's observation that,

> The heart and soul of much mathematics consists of the fact that the "same" object can be presented to us in different ways.

could be said equally well of the task of computer programming, where patterns of abstraction arise frequently, in diverse situations, nevertheless recognised as "the same" by the programmer. It is an essential instinct for an experienced programmer to identify and 'factor out' common code, or to capture a repeated pattern or design in a type, function, object, module etc (as provided by the features of their programming language). Similarly, Mazur's further observation that,

> Few mathematical concepts enter our repertoire in a manner other than ambiguously a *single object* and at the same time an *equivalence class of objects*. [author's emphasis]

could be said as easily of the abstract ideas found in computer programming.

Thus, one of the great virtues of category theory for computer programmers is that it provides a precise and formally-defined notion of an *abstract object* or *abstraction*, that is entirely compatible with their intuitive notions here.

Definition 2.2. An isomorphism is a morphism which admits an *inverse*, i.e., for some morphism f:

$$f : X \to Y \tag{2.11}$$

if there exists another morphism, by convention called an inverse and written f^{-1}:

$$f^{-1} : Y \to X \tag{2.12}$$

such that

[3] While targeted at the general mathematical reader, Mazur's paper should be quite readable to a wide audience including most programmers, except the latter sections which require a deeper mathematical background.

$$f^{-1} \circ f = id_X \tag{2.13}$$

and

$$f \circ f^{-1} = id_Y \tag{2.14}$$

then f (and for that matter f^{-1}) is called an isomorphism, and X and Y are by definition *isomorphic.*

Equation 2.13 and equation 2.14 state that the 'inverse' morphism works as would be expected, i.e., that identity is preserved and the application of a morphism followed by its inverse gives back the original object as we would anticipate. These two equations also show the importance of the identity law: the existence of the identity morphisms makes it possible to define isomorphism, the foundational concept of sameness provided by category theory.

Given that it is possible to undo or reverse an isomorphism via its inverse, an isomorphism must by definition provide a faithful translation of one object to another, with no loss of information. This means that any two categories (or objects) that are related by an isomorphism can be used interchangeably and are in this sense effectively 'the same.' A programmer or computer scientist might say they are, "the same, just renamed," or "the same pattern" where f would be the "renaming" function, and f^{-1} an 'undo' function which reverts back to the original naming. All objects that are isomorphic are said to form an *isomorphism class* – the collection of objects which share the abstraction in common. In this way, by providing a formal notion of abstraction derived from isomorphism, category theory facilitates statement of abstract, general ideas while retaining mathematical precision.

Isomorphism is such a fundamental notion in category theory that when declaring the uniqueness of an abstraction, the unstated qualifier "up to isomorphism" is rarely explicitly stated. It is common to refer to a specific example of a category or abstraction, when in fact the entire isomorphism class is implicitly under discussion. For example, the expression "the monad" might be used when what is actually meant more precisely is "this monad or any other monad up to isomorphism."

Mazur goes on to say,

> One of the templates of modern mathematics, category theory, offers it's own formulation of *equivalence* as opposed to *equality*; the spirit of category theory allows us to be content to determine a mathematical object, as one says in the language of that theory, *up to canonical isomorphism.*
>
> ...
>
> A uniquely specified isomorphism from some object X to an object Y characterized by a list of explicitly formulated properties—this list being sometimes, the truth be told, only implicitly understood—is usually dubbed a "canonical isomorphism."

> The "canonicality" here depends, of course, on the list. It is this brand of equivalence, then, that in category theory replaces *equality*: we wish to determine objects, as people say, *"up to canonical isomorphism."* [author's emphasis]

Thus we can say not only that two artefacts are abstractly the same, but that in addition, they are equivalent: a looser notion than equality, but stronger than isomorphism. The additional constraint that there must be no choice involved in the isomorphism — that it is unique — makes the isomorphism by definition canonical. There may be many example artefacts that are isomorphic to a particular abstraction, but only one canonical isomorphism is admitted[4].

Conversely the concept of *equivalence of categories* is a slightly looser notion of sameness than isomorphism, that is consequently useful in a wider range of situations. In this case, a morphism between two categories, composed with its 'inverse', does not necessarily result in the same object as the original source object, as would be required by strict isomorphism. Instead all that is required for categories to be equivalent is that the target of the morphism composed with its 'inverse' is isomorphic to the identity morphism on the original source object. Equivalence is effectively isomorphism up to isomorphism.

The essential point here is that category theory provides various formal concepts of sameness, looser than strict equality, but in alignment with the intuitive notions programmers have regarding objects that are 'essentially' or 'abstractly' the same. This is of great practical significance, because in large part computer programming is concerned with the identification and use of repeated abstractions at many levels.

2.2 Exploiting Associativity and Identity

> *Categories provide an effective basis for domain specific languages both in mathematics where they originated and in computer programming, and come equipped with a notion of equivalence that gives precise description to a programmer's intuition as to how two program artefacts may be abstractly 'the same', thereby establishing category theory not only as a mathematical basis for programming but as a rigorously specified program design pattern catalogue.*

To put all this in terms more familiar to programmers, an important motivating example of what a category provides – via the associative law – is a set of basic requirements for an effective, minimal, domain specific language (DSL)[5]. The notation presented in the example of string concatenation above provides a tiny but sufficient DSL for constructing character strings, and

[4]In this thesis most examples of isomorphism are of the former kind (examples of abstract sameness), although critically, the notion of canonical isomorphism is used in chapter 5 in defining a key result of this thesis: *canonical trace*.

[5]From this perspective essentially all of mathematics can be seen as various categorical DSLs.

the second example above is a DSL for constructing sequences of statements, i.e., the familiar notion of block structured sequences of statements as found in most languages since Algol 60.

The purpose behind any DSL is to make it convenient to combine elements of interest into some larger expression, data structure, computation, etc. Any such language must include, either implicitly or explicitly, a composition operation to achieve this combination of elements into a larger whole. What the associative law requires is that this DSL should behave reasonably from the point of view of a programmer, such that the meaning of an expression in that language is independent of the order in which the components of that expression are defined or constructed. This understanding of a category as a minimal but sufficient system for effective composition has been called the "composition design pattern" [Gon12a]. It is this very basic and highly generic system of composition that 'works', provided by the category — requiring that a category fulfill just two simple laws — that underlies the widespread existence of categories throughout many fields of mathematics as well as in the domains of computer science and practical programming.

The relevance and usefulness of category theory to computer programming is not limited to the fact that category theory provides a descriptive framework suitable for discussing artefacts found in computing in abstract terms. In recent decades, computer scientists have discovered many structures already documented in category theory which are isomorphic to those found in computing. The realisation that a structure in computer science is already known in category theory, gives a programmer immediate recourse to whatever formal properties and governing laws have been already established in precise, mathematical terms. It should perhaps not be surprising that computer programmers and mathematicians may face some similar intellectual challenges for which similar solutions have been found.

From the point of view of a software engineer, isomorphism is the category-theoretic notion which precisely captures the concept of abstract sameness inherent in the notion of *design pattern*. In this sense category theory can be used as a descriptive framework for program design patterns[6]. It has already been observed that the basic structure of the category itself provides a design pattern for compositional domain specific languages, however a number of other, more specific patterns exist within category theory, each serving various useful purposes in programming.

This perspective on category theory as a design pattern catalogue has been used particularly fruitfully in the development of the Haskell Typeclassopedia [Yor09] that provides a design pattern catalogue, in the tradition of the famous *Design Patterns* book [GHJV93], but specifically for functional programming in the Haskell language. Another, similar catalogue of standard categorical constructions has been implemented in the ML language [RB88]. The primary weakness of the *Design Patterns* book and "pattern" meme as it stands is that ad hoc methods are used to describe and categorise the patterns which are themselves selected in an

[6]Isomorphism provides a more abstract concept than that of just program design pattern, although this provides an intuitive and fairly general example, familiar to programmers.

ad hoc fashion, based solely on the intuition and experience of the authors. Certainly many useful patterns have been identified and catalogued. But the significance of category theory as a rigorous framework for describing program design patterns and the corresponding laws which govern them, is that it is both more precise and powerful because it is mathematically – and therefore programmatically – tractable. Among other benefits discussed later in this chapter, category theory offers the software engineer a collection of well defined, composable (program) abstractions governed by clearly understood laws, for use when formulating a design, as is taken for granted in other engineering disciplines.

2.2.1 Monads

> *The monad extends the categorical notion of composition to include functions that manipulate state or return exceptions, perform input and output and exhibit a wide variety of other program phenomena including non-determinism, concurrency and parsing.*

Having identified that the category provides a design pattern for compositional systems such as DSLs, we can return to the remaining objections raised but not yet addressed in section 2.1: i.e., how to model exceptions, I/O and mutable state? Wadler [Wad90] was the first to show how *monads* can be used to model many program phenomena including mutable state, exceptions, parse text and invoke continuations. Monads also subsequently provided an elegant solution for I/O, concurrency, exceptions and foreign-language calls – the so-called "awkward squad" [Pey01] – being practical necessities that had previously proven difficult to accommodate in pure, functional languages.

The consequence of these practical successes for the monad is that it has become the standard structuring notion used in Haskell – by effectively providing embedded DSLs – for all of the various applications already mentioned and a great many others. The Haskell language has been extended with special syntactic sugar for specifying monadic computations, the "do" notation, making monads particularly easy to use in practice. However the usefulness of monads is not limited to either the Haskell language specifically, or the domain of functional programming more generally. In fact monads arise naturally in many places when programming, by no means limited to the functional programming paradigm; they are also ubiquitous in imperative programming [Pip06]. The broad applicability of the monad to describing many commonly found and interesting notions in programming is due to its very abstractness.

Because the monad captures precisely some notions underlying a wide variety of artefacts that are normally considered distinct, is a topic that is famous for having spawned a large number of tutorials [Has14], each providing one or more different software artefacts already familiar to programmers as intuitive examples. Here the tradition is continued by providing a simple motivating example, in the hope that by identifying explicitly the monad that arises in

a familiar context, the same abstraction can be seen elsewhere, especially with the aid of the many other tutorials.

The essential insight behind the notion of the monad, and thus the usefulness of category theory here, is that computer program functions that cannot be described in basic set theory are still 'function-like' in the mathematical sense, if the range of the function is modified appropriately. For example, consider functions that might sometimes return nothing, versus some other value(s) of interest, such as the lookup of a database. Here we can model the lookup of a key in an index or map, with an intentionally naïve and incorrect function[7] (i.e., by deliberately ignoring that fact that a lookup might fail):

```
lookup :: key -> Map key val -> val
```

where the `lookup` function takes a key of some type, a `Map`[8] from keys to values, and returns the value corresponding to the given key. This `lookup` function can then be used to interact with a particular database of information stored in maps, such as[9]:

```
type AutoDB = Map String
                  ( Map String
                        (Map String Integer) )


autos :: AutoDB
autos = fromList
      [ ("Ford"   , fromList [("Falcon", fromList [ ("XE", 1986)
                                                  , ("XA", 1973) ]]
      , ("Porsche", fromList [("928"   , fromList [ ("S2", 1986) ]]
      , ("Volvo"  , fromList [("850"   , fromList [ ("T-5", 1997) ]]]
```

Now it is possible to construct a compound lookup for some information of interest, e.g.:

```
lookup "S2" (lookup "928" (lookup "Porsche" autos))
```

which would return the result `1986`. This specific lookup can alternatively be expressed explicitly as function composition:

```
(lookup "S2" . lookup "928" . lookup "Porsche") autos
```

[7]Here we use the Haskell language for the example, because it is the language in which the notion of the monad is most commonly and explicitly expressed. It is therefore the most convenient language in which to express a monad explicitly.

[8]The `Map` type used here is essentially the same as the standard Haskell `Data.Map` type of the same name.

[9]Here the `fromList` function creates a `Map` from a list of key, value tuples, exactly like the Haskell `Data.Map` version.

However, as was noted earlier, the database can fail to find information in response to a lookup, so we need to define a new type to represent an enhanced range for a lookup morphism:

```
data Maybe x = Nothing | Just x
```

The constructor `Nothing` represents lookup failure, or otherwise the value of interest, some `x`, is returned as `Just x`[10]. Now the type signature of our new, improved lookup function is:

```
lookup' :: Map key value -> key -> Maybe value
```

Now it is clear how to handle lookups that find nothing, but it is no longer possible to compose the lookups as was done above with `lookup`, and the cross-cutting concern of handling lookup failure and short-cutting with a `Nothing` result is intermixed throughout the function:

```
f :: String -- manufacturer
  -> String -- model
  -> String -- type
  -> AutoDB
  -> Maybe Integer
f man mod typ
  = case lookup man autos of
         Nothing     -> Nothing
         Just models -> case lookup mod models of
                             Nothing    -> Nothing
                             Just types -> case lookup typ types of
                                                Nothing        -> Nothing
                                                Just (_, year) -> year
```

Composition no longer works, because the range of the morphism has been changed, i.e., the output type of the lookup function has changed, so the types no longer allow the functions to compose. Whereas previously these types allowed easy composition, with each subsequent call to `lookup` producing the result needed by the next:

```
(lookup "Porsche") :: Map ... -> Map ...
(lookup "928")     :: Map ... -> Map ...
(lookup "S2")      :: Map ... -> ...
```

Now the following types are at play:

```
(lookup' "Porsche") :: Map ... -> Maybe (Map ...)
(lookup' "928")     :: Map ... -> Maybe (Map ...)
(lookup' "S2")      :: Map ... -> Maybe (...)
```

[10] This type `Maybe` is already defined in the Haskell standard prelude.

Following the programmers' instinct to factor out the repeated, housekeeping code associated with `Maybe` as found in the function `f` above, we define a new composition operator, conventionally called `bind`:

```
bind f g = case f of
                Nothing -> Nothing
                Just x  -> g x
```

It is once again possible to compose lookup functions:

```
f' man mod typ = bind ( bind (lookup man autos)
                             (lookup mod) )
                      (lookup typ)
```

However what we really want here is an infix operator, as this is the usual form in which we encounter a function composition operation. Haskell already has a synonym for the infix version of `bind`:

```
(>>=) = bind
```

This provides an application operator, which we can in turn use to define a true composition operator[11]:

```
g >=> h = \x -> g x >>= h
```

Now it is once again possible to use an infix composition operator to define a query, with the bookkeeping details of the `Maybe` type now hidden under the hood of the new composition operator:

```
f'' :: String -- manufacturer
    -> String -- model
    -> String -- type
    -> AutoDB
    -> Maybe Integer
f'' man mod typ = lookup man
          >=> lookup mod
          >=> lookup typ
```

At this point, we have potentially defined a new category where `>=>` is the composition operator. All that remains is to prove that the category laws (as defined in the previous section) are met. The identity law can be met by identifying a function, conventionally called `return`:

[11] Already defined in the `Control.Monad` Haskell module.

```
return >=> g = g -- left identity
h >=> return = h -- right identity
```

Here, the constructor `Just` provides a suitable function. `>=>` as the composition operator needs to meet the associative law, i.e.:

$$\texttt{k >=> (h >=> g)} \equiv \texttt{(k >=> h) >=> g} \tag{2.15}$$

This technique whereby a new composition operator is defined in order to allow functions with a modified range to compose and thereby form a category is due to Kleisli [Mac98]. The operator `>=>` is often called Kleisli composition.

Since the type `Maybe` – introduced here so as to illustrate the motivation for monads – already exists in the Haskell standard prelude, this abstraction can be used as-is as a software engineering abstraction governed by well understood laws. `Maybe` is already known to fulfill not only the category laws, but the monad laws as well, the additional stricture being that a single function (usually called `return`) acts as both left and right identity. This additional restriction with respect to the structure of a category, makes the monad an example of a monoid, an abstraction explored further in chapter 4.

There is no engineering basis here to re-invent the wheel and prove that `Maybe` obeys the category and monad laws. Nevertheless, the programmer should be able to convince themselves easily enough that `Just` when composed with any other lookup will simply pass along the result unmodified, so it will indeed act as left and right identity. Similarly, it should be relatively easily apparent that composing multiple lookups with `>=>` will result in a compound query where lookups occur in the same order, regardless as to the order in which the components of the lookup were constructed. By complying with the category laws, a well-behaved DSL for database lookups has been produced.

2.2.2 Functors

Functors provide a mechanism to combine the functionality of various categories in a consistent fashion that preserves the category laws; a pattern already familiar to programmers as type-level operators, as well as many other program phenomena.

It has been shown in the previous section how the functions, procedures and operators in a language or program can be given a categorical description as morphisms with objects corresponding to the various program types over which they range, even when those functions etc. exhibit a wide variety of program phenomena. Together the morphisms (functions etc.) and objects (types) form a category, with the composition operation being the familiar notion of (program) function composition in the simple case of pure mathematical functions, or Kleisli composition in the case of various other program behaviours in the context of a monad. By enforcing the associative law for composition, category theory provides a framework in which

these program functions can be composed consistently. One such example category is **Hask**, having the types in the Haskell language as its objects, with the morphisms being the various functions between them.

However the categorical notion of morphism is, as was observed in section 2.1, a very general one indeed, and not limited to the description of program functions, procedures etc. Morphisms can equally well be used to describe type-level operations, which rather than manipulating program data at run time, instead manipulate program types statically, to derive new, compound or derived types. To extend the example of **Hask**, a type operator can be applied to any Haskell type, i.e., the entire category of types (objects), such that it constitutes an operation where both the source and target are categories (in this case both are **Hask**). Such mappings between categories are known as *functors*. More specifically, the example of a type operation is an *endofunctor*, meaning that the source and target categories for the functor are the same (i.e., whatever category contains the types of the language in question as objects). The "endo" prefix indicates that the domain and range are the same. In the case of **Hask** we want the category of Haskell types to be closed under any conceivable type operation, so these operations must be endofunctors. Nevertheless, in general, functors do not necessarily have the same source and target category.

Functors are morphisms in the category of categories[12]. Functors provide a mapping between categories, and must preserve composition. Morphisms that previously composed (their domains and ranges permitting) in the source category must still compose when mapped into the target category, and the identity morphisms from the source category must continue to work as identity morphisms when mapped into the target category. Functors map objects and morphisms in the source category to corresponding objects and morphisms in the target category in this 'structure-preserving' way.

To continue with the example of (type) operations applied to **Hask**, we can define the well-known polymorphic cons-list as a type operation that takes an existing type and produces a new type being a cons-list of such elements:

```
data List a = Cons a (List a)
            | Nil
```

where `a` can be any type. We can now say `List Integer` to define a `List` of `Integer`.

Here the functor `List` provides the required mapping from the object `Integer` in **Hask** to the object `List Integer` (also in the category **Hask**). However, a mapping between morphisms

[12]Readers familiar with Russell's paradox will note that the same problem lurks within this category of categories, which apparently is required to contain itself. To solve this, a distinction is drawn between small and large categories, where *small* categories contain morphisms between objects, versus the other *large* categories (i.e., non-small categories). This allows the non-paradoxical definition of the large category **Cat**, which is implicitly in use in the discussion in this section, being the large category that contains all of the small categories and functors between them (but not itself).

is still required to define a complete functor. A mapping from morphisms with the domain of `Integer` and morphisms with the domain of `List Integer` is needed, and more generally, similar handling for any type. This functorial mapping is conventionally called `fmap`:

```
fmap :: (a -> b) -> f a -> f b
```

where `f` is some functor. For the `List` functor specifically what we have is:

```
fmap :: (a -> b) -> List a -> List b
fmap f Nil = Nil
fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

This is a mapping which, when applied to any function that operates on an element contained in a `List`, promotes that function to operate over the entire list (historically, `fmap` is known in functional programming as `map` when applied to a cons-list type as it has been done above). The significance of this is not trivial. Instead of having to write a raft of specialised functions for operating on `List`s, we can reuse any function which operates on a single element of the list. Here the traversal of the compound data structure of a list is encapsulated in the single function `fmap`.

This functorial mapping of morphisms provided by `fmap` is sometimes referred to as 'lifting' a morphism from one category into another. This terminology is usually applied where the target category is a monad, e.g., from the standard Haskell module `Control.Monad`:

```
liftM :: Monad m => (a -> b) -> m a -> m b
```

This function `liftM` takes a function from the category composed using `(.)` with identity function `id` and 'lifts' it into a monad, being the category using Kleisli composition with identity provided by `return`. This function `liftM` means that monads are also functorial, because any regular function can be applied within a monadic context by lifting it, using `liftM`.

Functors provide a consistent and effective way to reuse the functionality provided by one category, within another. Instead of having to construct a single category containing all of the operations, structure and features required, an existing category can be used in the context of another category, as needed, by simply implementing a suitable functor (e.g., the relevant `fmap` function above). The effectiveness of this approach rests on the fact that a categorical approach was used consistently throughout; the functor is structure-preserving, in that it preserves the category laws from the existing category in the context of another category. It must be emphasised that the role played by functors above as a type system operation (in the cases above implementing lists and monads), is just one example of where functors can be found. Gonzalez gives several other examples of the diverse places in programming where functors exist [Gon12b]. Nevertheless, in every case, the usefulness of functors in programming stems from the fact that they allow the functionality of categories to be combined easily and safely (with composition and identity preserved).

2.2.3 Initial Algebras

Initial algebras can describe not only concrete and abstract data types, but the pattern of recursion naturally associated with these, in generic terms, for any type.

Perhaps the most famous, early example of isomorphism as it applies to computing was the recognition by Goguen, Thatcher and Wagner that abstract data types *are* so-called *initial algebras* [GTW78]. The term 'abstract' used with respect to data types in this context refers to machine independence, where the precise encoding of primitive types such as Booleans, characters in various encodings, integers, floating point numbers, etc., is not the focus of attention. Instead the data types of interest are portable, compound types, that combine existing types to form new derived types. Datatypes as known by programmers – that happen to be abstract with respect to the precise representation of the data – are structurally identical to what mathematicians already understand to be an initial algebra. They are abstractly the same (i.e., isomorphic, or the same up to isomorphism), in that they both capture precisely the same abstract notion, while having different terminology and referents for the specific objects in each case. But the sense in which a programmer considers all abstract data types to be the same, is identical to the mathematician's view of all initial algebras as the same.

The insight that initial algebras are structurally the same as abstract data types starts first with the realisation that a concrete data type is a many-sorted algebra, an observation first made by Goguen [GTW78]. The mathematical notion of an *algebra* consists of a set together with a collection of operations on that set. The set associated with the algebra is the (concrete) data type of interest (called the *carrier type*), and the operations on that set are the constructors for that data type. So the observation that concrete data types are algebras is tantamount to saying that concrete data types are defined by their type name and constructors. For example, given a concrete data type, say a list of integers, the following algebra could be constructed to describe this:

$$< IntList, Cons, Nil > \tag{2.16}$$

where $IntList$ is the type and $Cons$ and Nil are the constructors for it.

Effectively, the algebra packages up the type and associated constructors of interest into a single object. The algebra is 'many-sorted' in that it can accommodate not just a single type, but a collection of mutually recursive types. The mathematical concept of an algebra introduces an addition requirement that constrains the types of the associated operations: the operations must operate on the type of interest. Algebras like the example above are traditionally called Ω-algebras (or sometimes Σ-algebras) and provide a *signature* characterising the concrete data type of interest.

The connection between algebras and concrete data types can be extended to cover abstract data types, that are parameterised with respect to other types. In the previous section, it

was shown how functors provide just such type operators, when applied to the category of the types in a programming language of interest. Type operators allow other data types to be composed together into more complex types. Typically, in a language which allows user-defined, compound types, there is some concept of both *sum* and *product* types (named after the corresponding set theoretic operations). The sum of two types is a data type which can contain either of those types but not both; examples include unions in the C programming language family. The product of two types is a data type which contains both of those types, examples include structs in the C programming language family and records in many others. Other, familiar examples of functorial, compound types include `array of ...` in Pascal-style languages, generics and templates in C++ and Java. All of these operations, whether user-definable as they are in C++, Java and functional programming languages like Haskell, or inbuilt syntax used to access a pre-determined and limited selection as in the case of Pascal or C, can be understood categorically as functors. Just as Ω-algebras can model concrete data types, F-algebras — defined in terms of some functor, F — can model abstract data types:

Definition 2.3. For an endofunctor, F,

$$F : \mathbf{C} \to \mathbf{C} \tag{2.17}$$

an F*-algebra* consists of:

$$< A, \alpha > \tag{2.18}$$

where A, the *carrier type*, is an object in $\mathbf{C}$, and α is a morphism in $\mathbf{C}$, of the type:

$$\alpha : FA \to A \tag{2.19}$$

So far the identification of algebras with data types, including abstract data types, has achieved very little from the point of view of a computer programmer. An alternative terminology adds very little value in and of itself. In addition, the mathematical concept of an algebra provides a description of some features of data types as understood in programming, but not all. In particular, a great many Ω-algebras or F-algebras can be defined which do not describe data types — the operations associated with the type need not be its constructors. Data types are in some sense 'special' algebras, with additional constraints.

In particular, programmers are aware of the fact that there is a unique, natural way to recurse over abstract (and concrete) data structures. Functional programmers know this pattern of recursion as 'fold', which has been explored extensively in its application to cons-lists, as well as many other structured data types [Hut99]. Object-oriented programmers are familiar with the analogous, 'visitor' design pattern, being a standard pattern for traversing arbitrary recursive data types [GHJV93]. A second, related feature of data types as understood in

programming is that for any give type, there is a single way to construct each value — each value in the type is distinct. This is a corollary of the unique pattern of recursion associated with each type: each value can be constructed (recursively) in a single way.

The great practical usefulness of the algebraic and categorical approach introduced by Goguen is that it provides a precise definition of these special features of data types: they are *initial* algebras (in the category of all algebras for a given functor, or type constructor). This mathematical definition captures precisely and formally the programmer's intuition that for any arbitrary data type, there is a unique way to recurse over it, and a unique construction for each (consequently distinct) value. Initiality is formalised as follows.

Definition 2.4. An object, typically written $\mathbf{0}$ is said to be *initial* if for every object A, there is precisely one morphism from $\mathbf{0}$ to A.

In order to specify what makes an F-algebra initial, we consider the category of F-algebras (for any given functor/data type constructor, F), F-**Alg**, containing objects such as

$$< A, \alpha > \tag{2.20}$$

and

$$< B, \beta > \tag{2.21}$$

etc. The morphisms of F-**Alg** between these algebras are called *homomorphisms.* A homomorphism is *structure preserving* in the functorial sense, in that the relationship between the carrier type and operations must be retained.

Definition 2.5. An F-algebra is an *initial algebra* in the category F-**Alg** when it is unique (up to isomorphism) and there is a single homomorphism from it to every other F-algebra.

For an algebra to have this universal property of initiality it must contain the largest set of information, with distinct values. It is in this sense that the initial algebra is special: homomorphisms can lose distinctness, but the initial algebra is the once from which any other algebra (in the same category) can be constructed.

The homomorphisms from an initial algebra to the others algebras in the same category use the unique pattern of recursion naturally associated with a data type. While this has been defined in mathematical terms, it nevertheless describes the operation already know to programmers as constructor replacement. In this light, the objects of the category of algebras, F-**Alg**, consists of all of the possible replacements for the constructors of that data type. The abstract data type itself is the (initial) algebra from which all other algebras can be reached. Returning to the example 2.16 above of the (initial) algebra associated with $IntList$, one such constructor replacement might be:

$$< Int, (+), 1 > \tag{2.22}$$

i.e., the length function for $IntList$s.

In categorical terms the unique pattern of recursion associated with each data type is known as a *catamorphism*. Here category theory has been able to formalise this familiar notion, and generalise it to arbitrary data types. In effect, from the point of view of a programmer, initiality gives arbitrary data types consistent semantics, i.e., it defines how to 'fold' over them in the general case. Thus the intuition of an experienced programmer, developed through experience, is formalised precisely, in general terms. An extended example of an initial algebra and the associated fold operation is given in chapter 6.

Based on Goguen's insight, patterns of recursion as they relate in general to arbitrary, abstract data types have been a particular focus of research using category theory, and produced results of practical relevance to computer programming. Meijer, Fokkinga and Patterson have presented various well-known patterns of recursion categorically, as composable recursion combinators, that can be used to calculate programs from their specifications [MFP91]. Augusteijn has used these recursion patterns to classify sorting algorithms and shown that the choice of recursion pattern is the key design decision for these algorithms, as few remaining decisions remain for the programmer once the pattern of recursion is chosen [Aug98]. In a similar, vein Bailes and Brough have shown how these recursion patterns can be used as the basis for a simplified, sub-recursive style of programming [BB13], and with Kemp, that this can therefore provide a useful basis for program structuring as well as recovery of designs from source code [BBK12]. Because category theory applies well to describing abstract type system structures, it has therefore also been used as a basis for generic (polytypic) programming [BJJM99, Hin12]. Hagino has taken this categorical approach to programming to its logical conclusion, and developed a research programming language based explicitly on category theory [Hag87].

2.3 Duality

> *Category theory is useful not only for describing categories but also for manipulating and reasoning about them at the meta-level; categorical duality ensures that every category has a dual, i.e., every category-theoretic structure has a unique (up to isomorphism) 'opposite'.*

Not only does category theory provide a powerful descriptive framework for abstract constructs via isomorphism, and an effective basis for DSLs, equally importantly it allows mathematically precise manipulation of the (categorical) program artifacts of interest it describes. In this way, category theory can be used as a strategic design tool, to manipulate program artefacts at the meta-level. One such categorical tool, central to the arguments presented in this thesis, is duality – the fact that for every category there is a corresponding dual, 'opposite' category.

Definition 2.6. Each category **C** has a *dual* or *opposite* category $\mathbf{C^{op}}$, which has the same objects as **C**, and morphisms in a 1:1 correspondence, such that if **C** has the morphisms:

$$f : A \to B \tag{2.23}$$

$$g : B \to C \tag{2.24}$$

$$h : C \to D \tag{2.25}$$

where

$$h \circ (g \circ f) = (h \circ g) \circ f \tag{2.26}$$

i.e., composition is associative, then $\mathbf{C^{op}}$ has the corresponding opposite morphisms, for each of which the domain and range types are swapped:

$$f^{op} : B \to A \tag{2.27}$$

$$g^{op} : C \to B \tag{2.28}$$

$$h^{op} : D \to C \tag{2.29}$$

where

$$f^{op} \circ (g^{op} \circ h^{op}) = (f^{op} \circ g^{op}) \circ h^{op} \tag{2.30}$$

i.e., composition remains associative in the opposite category, as is required.

Not only do duals exist for the morphisms, but also for any statement made about the category. We see this above in the relationship between the equations 2.26 and 2.30, being the statement of the associative law in the original category **C** and the opposite statement of associativity for the opposite category $\mathbf{C^{op}}$. In general for some statement about a category, lets call it S, we know that opposite law with statement S^{op} is also true of the opposite category. This is known as the *duality principle.* Thus, not only can a dual be constructed from any category, but also a dual exists for any statement made about that category. Pierce points out that in this way, the necessary existence of duals is, "a convenient source of "free theorems" about categories: once a theorem is proved, its dual follows immediately" [Pie91].

Duality is used in this thesis to justify and then develop a new theory of trace, dual to that of the existing abstraction used in current theoretical approaches to tracing. By identifying the existing category-theoretic structures underlying the existing notion of trace in computer science (the trace monoid, presented in chapter 4), the dual structures are thus also known to

exist, 'for free' (explored in chapter 5). By using duality this way, this thesis is most closely related methodologically to the work of Meijer and Bierman [MB11], who use category theory to explore the relationship between SQL and NoSQL, finding that these two models of data storage are in fact unique alternatives to each other, being each other's duals. A number of other interesting duals have been identified in computer science, including the duality between operational and denotational semantics [Ong95]. It is this fact that is ultimately used to identify the alternative to the existing notion of trace in computer science (see chapter 5).

The duality between operational and denotational semantics (explored further in section 5.1) is based on the duality between initial algebras and *final coalgebras*[13]. Coalgebras are constructed similarly to algebras, consisting of a carrier set and associated operations. However unlike the requirement for algebras that the operations operate on the carrier set, in the case of coalgebras, the operations produce the carrier type.

Definition 2.7. For an endofunctor, F,

$$F : \mathbf{C} \to \mathbf{C} \tag{2.31}$$

an *F-coalgebra* consists of:

$$< A, \alpha > \tag{2.32}$$

where A, the *carrier type*, is an object in $\mathbf{C}$, and α is a morphism in $\mathbf{C}$, of the type:

$$\alpha : A \to FA \tag{2.33}$$

The F-coalgebra is the dual of the F-algebra. Similarly, just as an algebras may have the universal property of initiality, a coalgebra may be *final*[14]:

Definition 2.8. An object, typically written **1**, is said to be *final* (or sometimes *terminal*) if for every object A, there is precisely one morphism from A to **1**.

While initiality requires that any algebra can be reached from the initial algebra, finality says the reverse: that final coalgebra is the target of a homomorphism from every other coalgebra (in the category of F-coalgebras, F-**CoAlg**, for some functor F):

Definition 2.9. An F-coalgebra is a *final coalgebra* in the category of F-coalgebras, F-**CoAlg**, when it is unique (up to isomorphism) and there is a single homomorphism to it from every other F-coalgebra.

[13]The "co-" prefix, conventionally means "dual of".

[14] Note that it is also possible for an F-algebra to be final, or an F-coalgebra to be initial, however these structures have not proven to be of as much interest as the initial algebra and final coalgebra for computing applications.

The particular relevance of final coalgebras to computer programming is that they capture a programmer's intuition as to an abstract machine. Consider the familiar example of a stream: of elements that can potentially be infinite (i.e., they might be produced by a non-terminating process, but need not necessarily be so):

```
data Stream a = Another a (Stream a)
              | Finished
```

With some functions to manipulate it:

```
head :: Stream a -> a
tail :: Stream a -> Stream a
```

and the corresponding coalgebra:

$$< \mathit{Stream}, \mathit{head}, \mathit{tail} > \tag{2.34}$$

Here the coalgebra consists of the carrier type, but instead of constructors, the associated operations are destructors.

Note that the `Stream` data structure is deliberately chosen to be isomorphic to:

```
data List a = Cons a (List a)
            | Nil
```

This is to emphasise that this is just the familiar list structure in disguise, which when given lazy constructor semantics as indeed it does have in Haskell, operates as a potentially infinite stream.

Just as there is a unique pattern of recursion associated with each data type, there is a unique pattern of corecursion for the `Stream` data type:

```
unfoldS      :: (b -> Maybe (a, b)) -> b -> Stream a
unfoldS f b  =
  case f b of
     Just (a,new_b) -> a : unfold f new_b
     Nothing        -> []
```

Functional programmers already know this design pattern as 'unfold,' and will recognise this function as `unfoldr`, the 'unfold' for the basic list type, in disguise [HHJ13]. Looking at the signature of `unfoldS`, the first argument to `unfoldS` is a generator function which takes a seed (type `b`), and produces either a result (type `a`) and another seed, or nothing if it is finished. Thus `unfoldS` takes a generator function and a seed, and repeatedly invokes the generator function as long as it continues to return a result and a new seed, using the new seed for each subsequent invocation.

Here `unfoldS` captures a quite general and abstract idea (with respect to `Stream`): it describes an abstract machine – the exact behaviour is parameterised – which is repeatedly invoked using a seed to produce a list value and another seed, to be used in the next invocation.

If we generalise the 'unfold' to any type, `x`:

```
unfold :: (b -> Maybe (a, b)) -> b -> x a
```

then with a suitable choice of `a`, `b` and `x` types, an arbitrary selection of transition systems can be implemented. This pattern of corecursion associated with final coalgebras, generalised to arbitrary types is known as an *anamorphism*, the dual of the catamorphism presented in the previous section.

The dual concepts presented in this section are summarised in table 2.1.

object:	initial algebra	final coalgebra
familiar example:	data	codata
recursion strategy:	fold	unfold
morphism:	catamorphism	anamorphism
intuition:	abstract data type	abstract machine

Table 2.1: Initial Algebras vs Final Coalgebras

2.4 Applications

This project adds to the diverse range of applications to which category theory has been successfully applied in both computer science and software engineering, by using several categorical ideas already well established in computing, for the novel application of systematically exploring the semantic foundations for practical software execution tracing.

Given the ubiquitous nature of the category, due to its basic and therefore widely applicable structure – providing just associative composition and a notion of abstract equivalence – it should be no surprise that category theory has proved useful in computer science since the late 1970's, with Goguen's identification of the isomorphism between abstract data types and initial algebras being the earliest work to have a major impact on the field [GTW78]. This result is used further in chapters 5 and 6. The use of category theory in computer science is now well established having generated a huge body of literature, with textbooks by Pierce [Pie91] and Asperti and Longo [AL91] giving an overview of the wide range of applications for which it has been useful. Ehrig also provides an overview of the applications of category theory to computer science [EGW98].

Category theory has also been useful in a diverse range of applications within software engineering. It has been used to model multi-agent systems and communications [JMP05, Sob08, Hua11, OD10] and to support model driven engineering [DM12]. Industrial applications include "specification, synthesis and maintenance of industrial strength software" [HW00], the syntactic problem of merging program sources in a commercial software system [NES05], industrial applications of software synthesis [WHB01], tool support for complex software systems, using category theory as modeling language [KOK10], and for configuration of complex software systems [Hil93]. Category theory has also been used in software engineering as both a meta-ontology [JD01] and abstraction mechanism [CDJ01] for information systems, for database engineering [Tot08] and as a unifying concept for information fusion systems [DK99].

Although Wadler was the first to explain the widespread usefulness of monads in computing (as explained in section 2.1.1), the concept was popularised by Moggi who empasised their ubiquity. Moggi used monads specifically for the study of programming language semantics because they allow for a modular description of semantics [Mog89], in contrast to domain theory as first introduced by Scott [Sco70], that despite having a long history of use in the description of semantic domains, has the shortcoming that it is not suited to such modular descriptions of semantics. In this thesis, category theory is used in exactly this way (among others) to make it possible to speak in abstract, unified terms about the nature of execution tracing in *relation* to program semantics (see chapter 5).

More generally, Goguen's "A Categorical Manifesto" [Gog89] outlines several possible uses of category theory in computer science:

1. formulating definitions and theories,
2. carrying out proofs,
3. discovering and exploiting relations with other fields,
4. formalising conjectures and research directions, and
5. dealing with abstraction and representation independence.

Category theory is used in this thesis for all of Goguen's purposes. It is used to formulate and define the problem being explored, by providing a formal characterisation of the semantic basis for the trace monoid (see chapter 4). In chapter 5 duality is used to identify (and thereby implicitly prove) the existence of a unique, well justified alternative semantic basis to the existing one underlying the trace monoid. This use of duality exploits the relation between concepts found in computer programming and in the mathematical field of universal algebra — specifically the duality between initial algebras and final coalgebras, and the correspondence these have to abstract data types and abstract machines respectively. Finally, the results derived here are presented in abstract, categorical terms, independently of any specific representation, or the specifics of any particular language syntax or semantics.

Chapter 3

Tracing in Practice

Software execution tracing is a widely used technique arising out of basic debugging practices with diverse implementations found in many software engineering processes, but it lacks a useful, theoretical foundation, resulting in engineering problems at several levels: for the designers, implementers and users of tracing systems all of whom have no recourse to rigorously defined and clearly understood engineering abstractions, when it comes to either implementing or using tracing effectively.

Software execution tracing is a practical technique whereby an automatic record is made of the execution of a program, for either interactive or later analysis by a programmer. By providing a history of the execution of a program, a trace aids the programmer in comprehending the relationship between the program source code and the corresponding execution behaviour. Unlike interactive debuggers which only show the state of the program at a single point in time, traces contain a record of the history of the program which is useful as an aid to program comprehension for a range of activities, including but not limited to, debugging.

3.1 Origins

Execution tracing arises naturally as a systematic extension of basic debugging practices.

One of the most basic, widespread, and fundamental techniques for debugging is to add a statement or function call to the program, to output a value of interest. This technique allows the programmer an insight into various values that might not otherwise be known, or to understand the control flow of the program at run-time, e.g., determine in which order some functions are executed, or perhaps which branch of an 'if' statement is executed. In any case, this technique is intended as an aid to the comprehension of the program (and the problem) under investigation. For example, consider some Pascal functions:

```
function A ( i : Integer ) : Integer;
```

```
  begin
    ...
  end;

function B ( j : Integer ) : Integer;
  begin
    ...
    result := A ( j );
    ...
  end;

function C ( k : Integer ) : Integer;
  begin
    ...
    result := B ( k );
    ...
  end;
```

where "..." indicates some arbitrary sequence of Pascal statements.

If, in order to better understand the behaviour of the program, the programmer wishes to know the value of `i` when the function `A` is called, then a very standard procedure would be to modify the function:

```
function A ( i : Integer ) : Integer;
  begin
    WriteLn ( i );
    ...
  end;
```

In a similar fashion, the programmer may wish to inspect simultaneously the result returned by the function `B`, for example, in order to debug an incorrect result being returned from the `C` function. This might be done as follows:

```
function B ( j : Integer ) : Integer;
  begin
    ...
    result := A ( i );
    ...
    WriteLn ( result );
  end;
```

These modifications might in turn produce program output like this:

```
...
5
7
12
17
32
58
...
```

At this point, the program is now writing out the two integer values of interest, but it will quickly become confusing as to which integer in the program output corresponds to the argument `i` of function `A` and which to the result of function `B`. So it is natural to add some more output, e.g.:

```
function A ( i : Integer ) : Integer;
  begin
    Write ( 'A: i = ' );
    WriteLn ( i );
    ...
  end;

function B ( j : Integer ) : Integer;
  begin
    ...
    result := A ( i );
    ...
    Write ( 'B = ' );
    WriteLn ( result );
  end;
```

Rather than the confusing output above, this would instead produce:

```
...
A: i = 5
B = 7
A: i = 12
B = 17
A: i = 32
B = 58
...
```

Now it is clear which numbers in the output correspond to the program values of interest: the result of `B` and the argument `i` to function `A`.

It is a relatively small step from this extremely commonplace debugging and program comprehension technique, to two fairly obvious extensions:

1. systematic application of these modifications to every function argument and result, for every function of interest, and

2. refactoring these output statements into generic function entry and exit trace functions.

At this point, the programmer has progressed from debugging by inserting ad hoc `WriteLn` statements, to having developed a primitive tracing system, for the systematic, albeit manual application of execution tracing to a program in their language of interest. In this way, systematic execution tracing arises naturally from the practice of adding output for debugging and program comprehension purposes.

3.2 Examples

> *Execution tracing is a widely used technique, with diverse implementations found in many software engineering processes and application domains.*

Except where explicitly noted to the contrary, all of the academic case studies of tracing systems presented here involve tracing systems developed with some other primary research or practical goal in mind, and hence have an application-driven, ad hoc design. While an extensive survey of the industrial practice of tracing is yet to be done, it seems likely that a great many extant tracing systems have ultimately developed along similar, pragmatic lines. In addition to the evidence of the widespread use of tracing presented here as found in the academic literature, anecdotal evidence suggests that bespoke tracing systems are ubiquitous, and a part of many industrial software engineering projects.

3.2.1 Software Engineering Processes

> *Tracing, by providing an aid to program comprehension, is applicable to a range of software engineering processes.*

There are many case studies of practical execution tracing systems, indicating that tracing is both widely used and suitable for a range of applications, being commonly used in software engineering processes including debugging, testing, reverse-engineering, program comprehension, profiling and dynamic visualisation — examples found in various application domains are presented in the next section. Tracing has been found to be useful for “design, development, tuning and sustaining of hardware, libraries and applications” [Cur94].

Tracing is commonly used to support testing, since it provides insight into the operation of routines which are not closely involved in user interaction – in effect, it provides a test harness. Traces of execution can be used for either manual unit testing (where the programmer inspects the behaviour of routines on an ad-hoc basis) or as input to a more sophisticated, automatic tracing framework. Examples include: automatic testing against formal specifications [Ezu95]; testing whether legacy behaviour is preserved correctly when re-engineering existing object-oriented systems [DGW06]; and comparing execution with and without a woven aspect in an aspect-oriented language [SKB03].

Algorithmic debugging, a semi-automatic procedure for debugging, proceeds by interactively prompting the user to check whether successive values the program are correct or not. Some form of tracing is therefore a necessary component of algorithmic debugging systems in order to record and present source-oriented, run-time values to the user for checking. Execution tracing has therefore been a particular focus of interest in the algorithmic debugging community. The 1990s was a busy period for research into algorithmic and declarative debugging, particularly for functional and logic languages which were an increasing focus of interest at that time, but also for imperative languages. The proceedings of the First Workshop on Algorithmic Debugging (AADEBUG '93) include tracing related research, including Ducassé [Duc93], Nilsson [NF93], Ball [BH93] and Reiss [Rei93].

Ducassè's focus appears to have been automatic debugging of logic programming languages [Duc93], and declarative debugging using logic programming languages [Duc92]. The two ideas are combined in her primary project, Opium, an extensible trace analyser for Prolog [Duc99]. Ducassè also developed the COCA debugger for an imperative language: C [Duc98].

Nilsson developed Freja for his master's thesis. It is a subset of Miranda, with support for algorithmic debugging added [NF93]. He later collaborated with Sparud to develop a tracing structure called the Evaluation Dependence Tree (EDT) designed for lazy functional languages [NS96]. Sparud's dissertation combines the EDT with a source-to-source transformation to produce an algorithmic debugging system for Haskell. Sparud later collaborated with Runciman to develop Redex Trails for Haskell, also a source-to-source transforming system [SR97].

Naish and Barbour developed a declarative debugging system for NUE-Prolog (a language including both logic and functional programming elements), based on Sparud and Nilsson's EDT. This work does not include a formal analysis of their system [NB95]. Naish later collaborated with Pope to develop Buddha, a declarative debugging system for Haskell [Pop98]. This work also builds on the EDT developed by Sparud and Nilsson. Later work is focused on the practicalities of making declarative debugging for Haskell work effectively [PN03].

Watson and Salzman did further work on the tracing of lazy function evaluation. Source code is instrumented so that the application returns a pair including the unmodified output of the program and the trace of execution. Their work is based on a formal statement of lazy function evaluation [WS97b]. They have also developed a 'browser' for the trace, which allows the history of the execution of a program to be traversed [WS97a].

Chitil, Runciman and Wallace at the University of York did an informal comparative evaluation of the usefulness of three different systems for tracing Haskell programs that use lazy function argument evaluation [CRW00]. Chitil, Runciman and Wallace describe Hat, a tracing system for Haskell [CRW03]. Further work by this group at the University of York produced an improved version of Hat based on an augmented version of Sparud and Nilsson's Redex Trails [WCBR01].

3.2.2 Application Domains

> *Tracing is found in diverse application domains, including the development of object-oriented systems, concurrent and distributed systems, and servers and operating systems.*

Although tracing has wide usefulness and applicability in software engineering, there are several application domains in which execution trace is particularly helpful:

Object-Oriented Systems introduce additional run-time complexities (in particular late binding and polymorphism), which are not readily understood via static analysis of source-code. This encourages the use of tracing in order to understand and analyse dynamic, run-time relationships of which there are many examples [LN97, RR99, PHKV93, JS94, Sys98, GM01, JSB97, BJW+01, RD99]. Maintenance of legacy object-oriented systems, including reverse- and re-engineering, program comprehension and dynamic analysis, is also therefore a common application for tracing [GOA05, GO03, FOGG05]. It has also been noted that tracing is very helpful when extending object-oriented frameworks because effective comprehension of their behaviour is required [RDW98]. Hamou-Lhadj and Lethbridge performed a survey of tools and techniques for the visualisation and analysis of execution traces for object-oriented systems. They note that the analysis of large traces is effectively impossible without suitable tools. They observe that the maintenance and reverse-engineering of legacy object oriented systems requires dynamic as well as static analysis tools (again as a consequence of polymorphism and late binding) [HL04].

Concurrent and Distributed Processes are more easily understood by exploring a trace that shows the interactions of components over time than by inspection of individual process states at particular points in time. Traditional, interactive debuggers tend not to be effective in distributed and parallel systems where an inspection of individual program states is often less helpful than an insight into the interaction of software components over time. Traces provide both an alternative and a complement to debuggers, which provide a snapshot of program state at a particular moment. Consequently, a common application for tracing is the analysis, debugging, modeling and visualisation of parallel, threaded or message-passing programs, in distributed, super-computing or grid environments [Sta95, TSS95, ZS95, TFM+01, PY93,

FHL98, DHK$^+$92, MC02]. At the hardware level, tracing has also been used for multiprocessor design and performance evaluation [EKKL90].

Servers and Operating Systems are often difficult or impossible to stop or pause for interactive debugging, and server-side systems often lack a human-focused, interactive interface to provide insight into the values or states inside the program. For these reasons, tracing has been used in academic case studies of kernel debugging [Kue95], profiling [TM99] and measurement [CSL04], A great many examples of servers exist which produce execution trace for the same reasons identified above, although these have not been the focus of academic study. For example, many standard Unix servers will produce traces when maximally verbose logging output is specified. Traces have an additional practical benefit in this context in that they can be analysed off-line, in addition to being collected automatically without interrupting the running system.

3.2.3 Implementation Approaches

> *A diverse range of implementations exist for practical tracing systems, the norm being ad hoc solutions, tuned to the problem at hand.*

Many of the case studies of practical tracing systems are concerned with the specific implementation details, with a view to addressing a particular practical concern. Reiss and Renieris offer a list of implementation considerations for tracing systems that is useful for categorising these studies [RR03] with regard to their focus. These practical considerations include:

Low usage overhead - the system should minimise the effort required by the user to implement the tracing. In other words, the tracing should occur as automatically as possible. Reiss and Renieris suggest that ideally tracing should be implemented in the execution platform (i.e., as an automatic feature of the compiler or interpreter). There are three classes of automatic instrumentation implementations:

1. Instrumentation of executables or object-code

   Various practical techniques exist for implementing instrumentation by re-writing object or executable files [LB94, GOA05]. Instrumentation of executables can potentially have less impact on the behaviour of a program than instrumentation of source, because the inserted code can be tuned at the machine-instruction level.

   Dynamic instrumentation allows tracing to be added and removed during execution of the program under examination, and is a popular technique. It has no effect on the program under examination except for those routines which are currently being traced. Applications therefore include profiling of kernels and realistic (i.e. large) applications for

compiler research, which would otherwise be difficult to examine [CSL04, TM99, TH02, EKKL90, HMG+97, MCC+95].

2. Instrumentation of the execution platform or environment

   Wrapper functions interposed between applications and libraries have been used to collect load profiles and function arguments [Cur94]. Tracing has also been inserted automatically into CORBA interfaces [MC02]. The Java language provides the Java Virtual Machine Profiling Interface (JVMPI), which has been used for tracing purposes [GM01, RR00]. However this technique is limited to interpreted Java programs as the JVMPI is not available during the execution of compiled code.

   Python [Pyt] and Freja (a subset of Miranda - [NF93]) are rare examples of languages where the execution platform includes automatic support for tracing.

3. Instrumentation of source code

   Source-to-source transformation of programs to include tracing instrumentation is a widely used technique [PY93, SR97, JZTB98, PY93, TFM+01, PHKV93, TJ98]. It has the advantage of portability, since a source-to-source instrumentation tool can potentially be used with any compiler or interpreter for the language of interest.

It is feasible to implement source instrumentation for tracing manually, and it seems likely that this is a widely used technique along the lines of what was described in section 3.1. In the simplest, least systematic form, this may consist of ad-hoc print statements added to a program – a technique which has probably been used by every programmer at some time. However there are distinct disadvantages to this approach for the following reasons:

- programmer error may introduce bugs into the instrumentation code;
- manual maintenance of tracing systematically applied to source is time consuming; and
- instrumentation code clutters the source thereby interfering with the readability of the program.

Automatic tools eliminate these problems associated with manual instrumentation.

Aspect oriented languages also provide a mechanism that has also been widely used to implement automatic trace instrumentation [SKB03, DF02, KHH+01, GO03, Har02]. In languages such as Java that provide no source pre-processor or macros, aspect oriented programming (AOP) provides a valuable source-to-source transformation tool. Tracing is arguably one of the key motivations for AOP, and examples of the tracing 'development aspect' are common on the Internet. Goldsmith, O'Callahan and Aitken have a comparison of their source-to-source instrumentation system and AspectJ [GOA05].

Low execution overhead - the system should minimise the resource requirements at run-time. Both the storage/memory requirements and CPU usage required need to be such that traces are not prohibitively large, and do not slow the execution of the program to an unacceptable degree. Reiss and Renieris have considered various techniques for improving the storage requirements for traces, with various compaction and selection techniques for Java and C/C++ [RR01]. Ball and Larus provide two algorithms that can be used to significantly decrease the cost of tracing, such that a smaller set of traces is collected, from which the full trace can be easily reconstructed [BL94]. Eggers, Keppel, Koldinger and Levy use compiler data-flow analysis to reduce both the amount of data collected and the impact on the instrumented program [EKKL90]. Tikir and Hollingsworth also use static analysis of the program to decrease the number of points in the program that must be instrumented, thereby decreasing the run-time overheads for the data collection [TH02]. Zhang and Gupta have developed techniques for storage of complete traces for profiling of 'realistic' applications, used for compiler architecture research [ZG04].

Where traces are stored can be as important a practical consideration as how they are generated. Speed of access, data volume and availability of suitable tools for analysis all need to be considered. A very common technique is to write traces to text files, as there are many standard tools that can be used to analyse and manipulate plain text. However Fischer, Oberleitner, Gall and Gschwind have used standard database technologies for trace storage, and have found that the tables can be kept entirely in memory with sufficient computing power and memory, resulting in usable query times [FOGG05].

Static and temporal selectivity - the user should be able to specify what they want traced, at as fine a level as possible. While it might seem that ideally all trace information should be collected and stored, in practice this is likely to result in excessively large traces, containing information of no interest to the problem at hand, and an unnecessary impact on the execution time of the instrumented program. There is normally no need to collect or see execution traces for frequently-called, well-tested routines. Similarly, the user should be able to select traces between particular times. This is of particular importance when debugging production systems. Often traces must be collected for a relatively long time until an error occurs which is otherwise impossible to replicate. When this happens, the bulk of the trace is of no relevance to debugging the error, so the user needs to be able to exclude this information.

Richner, Ducasse and Wuyts show how declarative queries can be used [RDW98]. Goldsmith et al. note that many of the optimisations available for SQL could also be successfully applied to their Program Trace Query Language (PTQL) [GOA05]. Ducassé has developed a Prolog-based query language for traces [Duc92]. The DTrace system provides a domain specific language for trace instrumentation and queries [CSL04]. Note also that tools as simple as the ubiquitous Unix 'grep' and other text searching tools can and often are used for querying and filtering traces stored in text files.

Frumkin, Hood and Lopez compare three different implementation approaches: source-to-source, compiler-inserted and wrappers on message passing library functions in the context of debugging of parallel, message passing programs [FHL98]. They conclude that "a compiler-debugger interface could reduce the instrumentation effort and the performance overhead, while increasing event resolution, portability, and transparency to the user". Automatic systems for instrumentation that have access to the source code for a program can potentially provide the richest set of trace information.

3.3 Practical Requirements

> *Practical traces, useful for software engineering tasks, need to refer to entities in the program source; this source-orientation in turn requires traces to be compositional, and contain events including information of varying types.*

Practical traces share various features in common, despite a wide variety of specific implementations for a range of different applications, and these features can be found throughout the case studies presented in section 3.2.2. In order to illustrate the features necessary in an execution trace to solve many typical practical problems, we develop a simple but sufficient hypothetical example of a program that produces an incorrect result, showing how a trace of the actual execution behaviour of the program can augment a static analysis of the source code when debugging. Consider the mathematical specification for a program:

$$fib_0 = 0 \tag{3.1}$$

$$fib_1 = 1 \tag{3.2}$$

$$fib_n = fib_{n-1} + fib_{n-2} \tag{3.3}$$

This is simply the well known Fibonacci sequence, where fib_n is the nth number in the sequence[1]:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \tag{3.4}$$

We might write a simple pascal program to implement this mathematical specification (but with an error)[2]:

---

[1]Here we use a zero-based numbering scheme where fib_0 is the 0th value, a standard practice in computing, further justified in this case because the Fibonacci sequence is defined as starting from fib_0 in this instance.

[2]NB: Here and elsewhere, two quite different notations are used: numbered equations in an italic font for specification and reasoning, and a fixed-width font for program code and trace outputs. This is to highlight the interaction between mathematical, equational reasoning by the programmer, and machine-readable source and machine-written traces and their derivatives.

```
program Test1;

function fib ( i : integer ): integer;
  begin
    if i = 0 then
      result := 1
    else if i = 1 then
      result := 1
    else
      result := fib ( i - 1 ) + fib ( i - 2 );
  end;

begin
  WriteLn ( fib ( 3 ));
end.
```

We would expect the output of the program (ignoring the error) to be:

```
2
```

because the 3rd[3] element of the sequence 3.4 is 2. But instead we see:

```
3
```

It is clear that the implementation of `fib` does not match the specification of fib, as we know $fib_3 = 2$. At this point, rather than inspecting the source code for the program to identify the source of the bug, we may instead investigate the trace, which might look like this:

```
2013/05/11 11:21:53.581 ========|EXECUTION TRACE START
2013/05/11 11:21:53.666 main    |Enter()
2013/05/11 11:21:53.668 fib     |  Enter(i=3)
2013/05/11 11:21:53.670 (=)     |    Enter(3,0)
2013/05/11 11:21:53.672 (=)     |    Return(False)
2013/05/11 11:21:53.675 (=)     |    Enter(3,1)
2013/05/11 11:21:53.677 (=)     |    Return(False)
2013/05/11 11:21:53.680 (:=)    |    Enter(result,<expression>)
2013/05/11 11:21:53.685 (-)     |      Enter(3,1)
2013/05/11 11:21:53.688 (-)     |      Return(2)
2013/05/11 11:21:53.691 fib     |      Enter(i=2)
2013/05/11 11:21:53.693 (=)     |        Enter(2,0)
```

[3]Again, using zero-based numbering.

```
2013/05/11 11:21:53.696 (=)     |         Return(False)
2013/05/11 11:21:53.698 (=)     |         Enter(2,1)
2013/05/11 11:21:53.701 (=)     |         Return(False)
2013/05/11 11:21:53.704 (:=)    |         Enter(result,<expression>)
2013/05/11 11:21:53.707 (-)     |           Enter(2,1)
2013/05/11 11:21:53.710 (-)     |           Return(1)
2013/05/11 11:21:53.714 fib     |           Enter(i=1)
2013/05/11 11:21:53.717 (=)     |             Enter(1,0)
2013/05/11 11:21:53.720 (=)     |             Return(False)
2013/05/11 11:21:53.724 (=)     |             Enter(1,1)
2013/05/11 11:21:53.727 (=)     |             Return(True)
2013/05/11 11:21:53.731 (:=)    |             Enter(result,1)
2013/05/11 11:21:53.735 (:=)    |             Return(1)
2013/05/11 11:21:53.739 fib     |           Return(1)
2013/05/11 11:21:53.742 (-)     |           Enter(2,2)
2013/05/11 11:21:53.746 (-)     |           Return(0)
2013/05/11 11:21:53.750 fib     |           Enter(i=0)
2013/05/11 11:21:53.753 (=)     |             Enter(0,0)
2013/05/11 11:21:53.756 (=)     |             Return(True)
2013/05/11 11:21:53.760 (:=)    |             Enter(result,1)
2013/05/11 11:21:53.764 (:=)    |             Return(1)
2013/05/11 11:21:53.768 fib     |           Return(1)
2013/05/11 11:21:53.772 (+)     |           Enter(1,1)
2013/05/11 11:21:53.775 (+)     |           Return(2)
2013/05/11 11:21:53.778 (:=)    |         Return(2)
2013/05/11 11:21:53.782 fib     |       Return(2)
2013/05/11 11:21:53.787 (-)     |       Enter(3,2)
2013/05/11 11:21:53.791 (-)     |       Return(1)
2013/05/11 11:21:53.795 fib     |       Enter(i=1)
2013/05/11 11:21:53.798 (=)     |         Enter(1,0)
2013/05/11 11:21:53.802 (=)     |         Return(False)
2013/05/11 11:21:53.806 (=)     |         Enter(1,1)
2013/05/11 11:21:53.809 (=)     |         Return(True)
2013/05/11 11:21:53.819 (:=)    |         Enter(result,1)
2013/05/11 11:21:53.824 (:=)    |         Return(1)
2013/05/11 11:21:53.829 fib     |       Return(1)
2013/05/11 11:21:53.833 (+)     |       Enter(2,1)
2013/05/11 11:21:53.839 (+)     |       Return(3)
2013/05/11 11:21:53.843 (:=)    |     Return(3)
```

```
2013/05/11 11:21:53.848 fib      |  Return(3)
2013/05/11 11:21:53.852 WriteLn |  Enter(arg=3)
2013/05/11 11:21:53.856 WriteLn |  Return()
2013/05/11 11:21:53.860 main    |Return()
2013/05/11 11:21:53.865 ========|EXECUTION TRACE END (OK)
```

This particular presentation and layout of trace is chosen somewhat arbitrarily, as many other possibilities exist[4]. However, the columns of the trace contain the following information:

1. a time stamp for when the trace event occurred,
2. the name of the function currently being executed – infix operations like assignment (`:=`) and equality (`=`) are shown enclosed in brackets, and the function `main` is the system entry point,
3. indentation, to reflect the nested nature of function applications at run-time – note that this is simply a convenient presentation device, as this information can be calculated easily from the function entry/exit trace events,
4. the string `Enter` or `Return` to indicate the entry and exit of functions, and
5. the zero or more arguments to the function (in the case of `Entry`), or the result of the function (in the case of `Return`).

Returning to the problem of the erroneous program output, we can work inwards in the trace, starting with the incorrect output from the call to `WriteLn` in the program[5]. The argument provided to `WriteLn` is the result of `fib(3)`:

```
2013/05/11 11:21:53.581 ========|EXECUTION TRACE START
2013/05/11 11:21:53.666 main    |Enter()
2013/05/11 11:21:53.668 fib      |  Enter(i=3)
...
2013/05/11 11:21:53.848 fib      |  Return(3)
2013/05/11 11:21:53.852 WriteLn |  Enter(arg=3)
2013/05/11 11:21:53.856 WriteLn |  Return()
2013/05/11 11:21:53.860 main    |Return()
2013/05/11 11:21:53.865 ========|EXECUTION TRACE END (OK)
```

The incorrect value of `fib(3)` is in turn derived from the values of `fib(2)` and `fib(1)`:

[4] In fact this trace was generated by an experimental tool developed early in this research project, being a small Pascal interpreter with inbuilt, automatic, function-level tracing, which was intentionally designed to include many ‘typical’ features found in practical traces.

[5] Not to be confused with the use of `WriteLn` described earlier, where it is used to trace out information of interest about the program as a debugging technique.

```
2013/05/11 11:21:53.668 fib     |  Enter(i=3)
...
2013/05/11 11:21:53.691 fib     |      Enter(i=2)
...
2013/05/11 11:21:53.782 fib     |      Return(2)
...
2013/05/11 11:21:53.795 fib     |      Enter(i=1)
...
2013/05/11 11:21:53.829 fib     |      Return(1)
...
2013/05/11 11:21:53.848 fib     |  Return(3)
```

The value of `fib(1)` is correct, being `1`, however the result of `fib(2)` is not correct, so we investigate this further:

```
2013/05/11 11:21:53.691 fib     |      Enter(i=2)
...
2013/05/11 11:21:53.714 fib     |          Enter(i=1)
...
2013/05/11 11:21:53.739 fib     |          Return(1)
...
2013/05/11 11:21:53.750 fib     |          Enter(i=0)
...
2013/05/11 11:21:53.768 fib     |          Return(1)
...
2013/05/11 11:21:53.782 fib     |      Return(2)
```

As before, the value of `fib(1)` is correct, but we can see that `fib` returns `1` when provided with an argument of `0`, instead of the correct result `0`, from the following trace lines:

```
2013/05/11 11:21:53.750 fib     |          Enter(i=0)
...
2013/05/11 11:21:53.768 fib     |          Return(1)
```

Specifically, these trace lines tell us that:

```
fib(0) = 1
```

If we wish to further investigate how this incorrect result arises, we can look at the intervening trace lines, corresponding to the body of the function call:

```
2013/05/11 11:21:53.750 fib     |          Enter(i=0)
2013/05/11 11:21:53.753 (=)     |            Enter(0,0)
2013/05/11 11:21:53.756 (=)     |            Return(True)
2013/05/11 11:21:53.760 (:=)    |            Enter(result,1)
2013/05/11 11:21:53.764 (:=)    |            Return(1)
2013/05/11 11:21:53.768 fib     |          Return(1)
```

Here we can see the argument to the function `i` is compared to `0` and found to be equal; the function result is thus assigned the value of `1`, whereas the specification of fib says it should be `0`. Herein lies the error. The significance of this example is that it shows how execution trace provides a useful adjunct to static analysis of the program text, by showing the actual behaviour which resulted from execution, as opposed to the behaviour the programmer infers – quite possibly incorrectly – from inspection of the code alone.

Aside from programs which produce incorrect output, another very common scenario is the debugging of a program which exhibits a run-time crash, error or exception (i.e., where the program does not terminate normally), via analysis of a stack back-trace, a technique supported by empirical evidence for its effectiveness [SBP10]. The GNU debugger, GDB [GNU14], provides a convenient tool for inspecting crash dump 'core' files in order to generate a stack back-trace on Unix and Unix-like systems where it is in widespread use. The Microsoft Windows platform provides similar capabilities, as part of the Microsoft Visual Studio development environment for their various languages, also allowing for the extraction of stack back-traces from crash dumps [Mic14]. Even in the Haskell language, where graph reduction better describes the function execution order than the function call stack used by applicative-order languages, a call stack is nevertheless considered desirable for debugging of abnormal termination [APE09].

Stack back-traces can be generated by an interactive debugger by either reading a 'core' or process memory dump file, or by attaching to the excepting process and inspecting the state of the system call stack[6]. One great virtue of execution traces like the one above is that they implicitly contain a stack back-trace, when entry and exit tracing is applied systematically to all functions and procedures. For example, suppose there is a program:

```
program Test2;

function A ( i : integer ) : integer;
  begin
    result := 10 div i;
  end;
```

[6]This usage of a so-called "interactive" debugger requires none of the interactive features, instead it provides a static snapshot of the state of the execution stack at run-time, at the time of the error. In fact this is a simple trace extraction feature.

```
function B ( i : integer ) : integer;
  begin
    result := A ( i ) * 2;
  end;

begin
  B ( 0 );
end.
```

When executed, it terminates prematurely with a run-time error:

```
exception: divide by zero
```

With a moment of thought it is clear that there is only one instance of division in the short program Test2 thereby making the isolation of the source of the error quite trivial – it can be found by a simple visual search of the short program text above. But if we assume instead that we are looking at a much more typical, lengthy and complex program in which division is a frequent operation, then a very common approach to debugging would be to analyse a stack back-trace. We find the stack back-trace is available from a simple inspection of the execution trace:

```
2013/05/30 02:45:40.097 ============|EXECUTION TRACE START
2013/05/30 02:45:40.167 main        |Enter()
2013/05/30 02:45:40.174 B           |  Enter(i=0)
2013/05/30 02:45:40.178 (:=)        |    Enter(result,<expression>)
2013/05/30 02:45:40.196 A           |      Enter(i=0)
2013/05/30 02:45:40.201 (:=)        |        Enter(result,<expression>)
2013/05/30 02:45:40.223 (div)       |          Enter(10,0)
```

Unlike the trace shown above for Test1, this trace does not show the execution terminating normally. The stack back-trace can be read off from the third column of the trace, based on the Enter trace events for which there is no corresponding Return trace event (for this very simple example, there are no Return events at all):

```
main
B
(:=)
A
(:=)
(div)
```

Assuming a typical situation where assignment and basic arithmetic operations are inlined as machine instructions in a compiled program, instead of being implemented via an unnecessarily inefficient function calls, we know the state of the call stack was:

```
main
B
A
```

Having isolated the function `A` as the inner-most function at the time of the division by zero error, the programmer is armed with very useful knowledge as to where in the program source to start looking for the cause of the error. In this case, inspection of the source for function `A` shows the only possible statement which could have caused a division by zero error:

```
    result := 10 div i;
```

The programmer has located where the run-time error has occurred in the source. Not only is the sequence of function calls known, but also the arguments to them – this information is also present in the trace:

```
main()
B(0)
A(0)
```

Now the programmer also knows the specific arguments which caused the error and can answer the question: "where did the zero value for the parameter `i` in function `A` come from?"

Software engineers who have experience of one tracing system will naturally tend to refine and improve upon that design in the development of subsequent tracing systems they may build for future projects. Such pragmatic, but nevertheless arbitrary designs are a product of the accidents of the programmer's experience, the requirements of the specific problem domain being addressed, and the available tools at hand, many examples of which have already been presented in section 3.2. The two examples above show how traces such as those just presented can support debugging. Both code coverage analysis and execution profiling are software engineering activities which are also enabled by execution traces such as the ones just presented as examples. Time-stamps for trace events like the ones above can be used to generate profiling information. Full code coverage information can be derived from a trace which includes the operations of all control-flow structures (e.g., if-then statements, looping constructs etc), thereby showing which paths are taken through the source code and the branches executed.

Since a great many of the tracing systems in use today are implemented as a library or module within the programming language in use – rather than being a feature of the language implementation itself – tracing of the mechanics of in-built control-flow structures is not a common phenomenon. This is a predictable consequence of there being few languages in common

use with any inbuilt tracing features whatsoever. Python is a rare example of a mainstream language which does provide automatic, inbuilt tracing via the "trace" module and "`--trace`" option; this includes tracing of primitive control structures such as "`if`" [Pyt]. (This thesis also presents an example of an automatic tracing system where all control structures are included in chapter 6, with an example trace provided in section 6.4.)

While timing information may or may not be needed in a particular trace (e.g., this does not matter when inspecting the stack back-trace used in the example debugging session above), practical traces as found in the examples chosen here and the case studies presented earlier have several features in common.

3.3.1 Source Orientation

> *Because all activities involving execution tracing involve some aspect of program comprehension, traces must refer to entities in the program source text, i.e., be source-oriented.*

Most fundamentally, every application of execution tracing is as an aid to program comprehension (e.g., the case studies already cited in section 3.2.2). Consequently all existing execution tracing systems are effectively source-oriented, in as much as the details recorded in a trace directly relate to source-level constructs in the program. Aside from the prima facie evidence that existing academic case studies are universally source-oriented, those who have reviewed tracing systems in general also agree they should be source oriented [Rei93, WS97b, CRW00].

Source-oriented tracing makes the relationship between the semantics and syntax of a program explicit: the trace directly relates the behaviour of the program to the source code. If our aim is to help relate semantics to syntax (which is fundamental to program comprehension and therefore many software engineering tasks including development, debugging, testing and maintenance – in fact any activity which involves both source code and an executable program), then source-orientation for tracing seems unavoidable. Correctly understanding the relationship between the syntax and semantics of a program is arguably the most fundamental task for a computer programmer – one that is often difficult, given the complex emergent behaviours which can arise from even simple programs.

While the overwhelming majority of case studies of execution tracing systems are clearly source-oriented (including all those cited earlier in section 3.2.2) there are a few examples which at first appearances are apparently not. Richner and Ducasse use tracing to recover design-level information from object-oriented application implementations [RD99, RD02]. Walker, Murphy, Steinbok and Robillard are also concerned with recovering high-level architectural information from the execution of a system [WMSR00]. Quick has implemented model- rather than source-oriented tracing for performance evaluation of parallel programs, where the trace is at the semantic level of a functional model of the parallel program, rather than at the level of the source programming language [Qui93], and Klar, Quick and Sötz have also worked on

this system, with particular emphasis on model-driven instrumentation [KQS92]. However in all cases the argument can be made that the actual 'source' of interest is that of the model or design, and that these traces are also therefore source-oriented in that they speak directly to the semantic model presumed to be in the mind of the programmer.

3.3.2 Compositional Structure

Source-orientation in turn requires a trace to reflect complex, nested structures, corresponding to those found in the source code.

An unavoidable consequence of source orientation is that trace structures must reflect the compositional, nested structure of the program; the trace must show how the meaning of each part of the program depends on the meaning of its constituent sub-components. The two examples of debugging using trace in section 3.3 – in one case to find the cause of an incorrect result, the other to find the cause of a run-time error – required analysis of the function entry and exit points in relation to each other, i.e., the nested call structure; a reflection of the nested structure of the source code as its semantics are expressed at run-time. This information on the order of function calls is also required in many of the case studies presented in section 3.2.2. In order to be useful generally, practical tracing must provide this information. For the example trace in section 3.3, indentation was used as a presentation device to highlight the compositional structure of the semantic information in the trace, and to allow easy visual correlation of the entry and exit events in the trace (which line up in the same column, for a particular function call instance).

Most programming languages allow more-or-less unrestricted nesting of function or procedure calls, or recursion, where the run-time nesting of calls can be arbitrarily deep[7]. Traces of such arbitrarily-nested function calls must therefore include arbitrarily nested traces: the trace of each function includes the trace of each of the functions it calls, and so forth. This information on the nesting of function calls is implicit in a "stack back-trace" and more generally is necessary to answer many questions regarding correctness, such as why a function returns the wrong value.

Users of practical tracing systems are often concerned with argument values on entry to a function (or procedure) and/or the return value on exit. This information was also necessary in the examples of debugging presented in section 3.3. In the general case, functions may be of any arity, i.e., have any number of arguments and may also return an arbitrary number of results (e.g., in the case of non-determinism, in the logic programming paradigm). This means that traces must potentially record information of varying type and structure. Note that it is very common for traces to be encoded as ASCII or Unicode text, and there are many other

[7]Although there will be some physical or operating system limit such as finite stack or heap space for the process which prevents infinite call depth in practice.

possibilities, but regardless as to the encoding, this information of varying type and structure must be recorded faithfully within.

3.4 Generalised Tracing

> *Category theory has not previously been applied to the software engineering problem of designing and using effective execution tracing, and there has otherwise been little attention paid to the abstract, theoretical basis for practical execution tracing systems.*

Despite the many case studies of specific, practical tracing systems presented in this chapter, there has been surprisingly little work giving abstract or general treatment to practical execution tracing, and none which uses category theory to do so. In 1991, Kishon, Hudak and Consel [KHC91] quoted a survey from 1981:

> program execution monitoring has been neglected as a research topic; the available literature contains mainly case studies; without adequate discussion of the fundamental concepts, goals and limitations [PN81]

The work of Kishon et al. goes on to provide the first example of an execution tracing system with an explicit semantic basis. Executable, "monitoring semantics" are automatically derived from denotational semantics of a specific type (continuation semantics), for a functional language. However this work, which culminated in Kishon's doctoral thesis [Kis92], is only concerned with execution monitoring in a very general sense, and no consideration is given to the nature, structure or content of traces, and how these relate to the semantics of the corresponding program. Further, the choice of the specific basis of continuation-style semantics is quite arbitrary.

Kishon was a student of Hudak, and some of the features of his execution monitoring were included in the work on modular interpreters by Liang, Hudak and Jones [LHJ95]. Their system is arranged along similar lines to Kishon's: denotational semantic specifications in a functional programming language are used to automatically derive interpreters with various features, including an example of a very simple tracer. These features are structured as modules, which are implemented as monad transformers, a practice followed here. However again, no consideration was given to the nature or structure of traces and their relationship to semantics.

In the late 1990's Jahier, Ducassé and Ridoux gave the first example of an explicit, semantic basis for tracing logic programs. They developed a framework for specifying trace models for logic programming languages [JDR00]. Their method derives tracing semantics automatically from a specification of operational semantics for Prolog. Later work by Jahier and Ducassé also demonstrates that several tracing applications (including profiling, visualization and test coverage measurements) can be supported by a single, generic, formally-derived tracing framework

[JD02]. Jahier and Ducassè cite the work of Kishon, but do not build on this, because of the difficulty of obtaining continuation semantics for their language of interest (Mercury, a variant of Prolog). They also note that Kishon's framework only supports interpreted programs, whereas in theirs, compilation is possible.

3.5 Engineering Problems

> *The lack of a sound and useful theoretical basis for practical execution tracing results in engineering problems at several levels; for the designers, implementers and users of tracing systems all of whom have no access to sound, useful theoretical guidance or tools.*

As was stated earlier and can be seen in the case studies already mentioned, practical tracing systems are typically designed and implemented in an entirely ad hoc fashion. Nevertheless, computer science provides an existing notion of trace, known as the *trace monoid*. Chapter 4 explains how this conventional notion of trace used in theoretical computer science is induced automatically by an operational view of semantics. In section 4.5 the trace monoid is shown to be lacking sufficient structure to encode the source-oriented semantic information of interest needed for many practical tracing activities. It is consequently unsurprising that this existing theoretical basis has rarely if every been used in the development of practical tracing systems, certainly not in any of the case studies cited above.

This lack of thorough and effective, theoretical grounding results in engineering problems at several levels:

- Designers of programming languages have no guidance as to how to include a tracing system as part of a mature tool-chain for their language – in particular what information traces can and should include and how they should be structured.

- Language implementers and tracing system designers have no guidance as to how to build sound, practical tracing implementations, or the ability to know (i.e., prove) that what they have built is correct; instead they are guided by experience and intuition only.

- Users of ad-hoc, practical tracing systems have no guarantees (i.e., proofs) that the system they are using is correct or accurately reflects the semantics of the program.

- Users of practical tracing systems have at best ad-hoc (and possibly erroneous) techniques and tools for reasoning about the contents of traces with ad-hoc structure and organisation, again relying only on experience and intuition.

- Lacking a solid theoretical basis, it is difficult to implement generic, correct, useful tools for the specification or implementation of tracing systems, or to support reasoning about traces.

In short, existing practical approaches to software execution tracing are not engineered, but rather ad hoc, albeit pragmatic solutions.

Chapter 4

Theoretical Foundations for Tracing

Practical execution tracing systems have not used the existing theoretical concept of trace already provided by computer science – the trace monoid – because monoidal traces, while nevertheless useful in the theoretical investigation of concurrent and non-deterministic systems, are the consequence of an operational view of semantics, and therefore lack the complex, compositional structure required to encode much of the source-oriented information needed in practical execution traces.

4.1 Origins

The term 'trace' has been linked to an operational model of semantics since the early days of computer science.

The use of the term 'trace' to mean a record of the history of a computation may be as old as the field of computing; the earliest to provide a definition of the term 'trace' in computer science is Scott, writing in 1964 [Sco70]:

> the operational meaning will generally provide a trace of the history of the computation . . . step-by-step evolved sequences of operations on representations

Here Scott uses the term "operational" to refer explicitly to the understanding of semantics as a transition system, as distinct from the "mathematical" semantics he was developing, which eventually became known as denotational semantics, discussed further in chapter 5. He defines "operational" semantics as (any) semantics that produces a "trace," specifically, "evolved sequences of operations of representations." As is explained in section 4.2, the execution of any transition system automatically induces a trace. This particular notion of trace, being linked directly to operational semantics, has become the conventional one in computer science, and in section 4.3 we show how this has been formalised as the *trace monoid*, the object of study in *trace theory*.

4.2 Operational Semantics

> *The central idea of operational semantics is that of the transition system, the operation of which automatically induces a trace.*

The exact origins of the term *operational semantics* are unknown, but as Plotkin observes of the quotation from Scott above [Plo04], it is:

> an early use in a paper of Dana's written in the context of discussions with Christopher Strachey where they came up with the denotational/operational distinction

Scott's definition of operational semantics as involving "step-by-step evolved sequences of operations on representations" describes what is conventionally known in computer science as a *transition system*. This notion of transition system is arguably one of the most fundamental abstract ideas in computer science, being central to the seminal work of both Turing and Church, as is explained below. Operational semantics provides the model of semantics for such "step-by-step" transition systems, by describing how the system transitions, step-wise, from one configuration to another, i.e., by defining a *transition relation*.

Turing's paper from 1936 has been cited as the foundation of the science of computing [Tur36]. His famous proof that there is no solution to the halting problem — the question as to whether it is possible, in general, to know in advance whether any particular computation will terminate or not — involves modeling the computations possible using a simple, hypothetical, mechanical machine. This machine operates exactly as Scott describes, by "step-by-step ... operations on representations": at each step, a Turing machine transitions into a new state, with the tape and symbols in a different configuration, and as Turing proved, the machine might or might not terminate, there is no way to know in advance.

Church independently developed an alternative proof that there is no solution to the halting problem using his *lambda calculus* [Bar81]. The lambda calculus operates "step-by-step", by the successive re-writing of λ-terms, by applying *reduction rules*. Once again, as Church proved, this process might or might not reach a λ-term for which no further reductions are possible (known as *normal form*), and terminate.

Despite the superficial differences between a mechanical machine on the one hand and a mathematical calculus on the other, the "operational", "step-by-step" nature of both Turing and Church's systems are clear. Both systems define the rules for how transitions occur, i.e., a transition relation, and in both cases the sequence of transitions might not necessarily terminate. Turing's machine is not just an example of a transition system, but more specifically what is now known as a *state machine* [HMU03]. The operational semantics of the machine are the rules that govern the machine interaction with the tape, i.e., the rules for moving from one state to the next. The lambda calculus, while also an example of a transition system like the Turing machine, is more specifically a *term re-writing system* [Klo92]. In this case the operational

semantics for the lambda calculus is provided by the reduction rules. A precise, category-theoretic definition of the structure in common between all transition systems is that they have *final co-algebra semantics*, discussed further in section 5.1.1. For now it is sufficient to observe that these two seminal examples of transition systems are inherently operational-semantic in their structure and description.

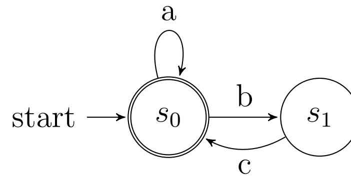
Turing, once he became aware of Church's work, added an appendix to his 1936 paper in which he showed that his machine and Church's calculus were logically equivalent [Tur36]. Given this result, and the fact that Turing subsequently completed his PhD thesis under Church, it is inconceivable that these two thinkers were not both fully aware of the abstract similarity and equivalence between these two foundational examples of transition systems. The fact that not just these two systems, but any others that provide 'universal' computation are equivalent (i.e., can be used to simulated each other) is a central result in computer science, now known as the Church-Turing thesis. Through both the work of Turing and of Church an implicit, operational-semantic perspective was at play from the beginnings of computer science, with origins as deep as the field itself. It took until the mid-1960s when Scott and Strachey began developing an alternative "mathematical" (later known as denotational) semantic perspective for it to even make sense to define the existing state of affairs as "operational". It is not surprising therefore, that Scott would define "trace" as he does in operational-semantic terms, given that this was the universal perspective on semantics at the time, despite going on to develop an alternative perspective on semantics with Strachey (which, as will be shown in the next chapter has an equally natural, associated idea of 'trace').

In order to illustrate Scott's statement above that a record of the transitions of any such "operational" understanding of behaviour is a "trace," and to show how this differs in structure from the example execution traces show in chapter 3, an example is presented along the lines of Turing's machine, rather than a term re-writing system like the λ-calculus. Turing's work has the significant advantage of providing a much more intuitively reasonable definition of effective computability – via this metaphor of a machine – than the more mathematical notion of λ-definability used by Church[1]. Now that digital computing is commonplace, in no small part due to the insights and efforts of Turing as one of the pioneers of computing machinery, this state-machine metaphor for computation is especially familiar to computer programmers, who can recognise an abstract similarity between Turing's machine and digital computers[2]. So rather than using Turing's original mechanical machine with tape-based storage as an example to illustrate how trace arises in the fashion Scott describes above, we instead use an example

[1]This advantage was acknowledged by both Church and also Gödel who was only persuaded by Church's thesis, after accepting it first in the form presented by Turing [Cop04].

[2]The connection between the lambda calculus and generalised computation is less obvious. Nevertheless, software engineers unfamiliar with the lambda calculus will already be familiar with at least one term re-writing system: (high-school) algebra, where it is possible to solve for a variable by successively re-writing an equation, following the rules of algebra at each step, in exactly the same way that λ-terms are reduced according to the reduction rules.

of a state machine even more familiar to programmers of modern digital computers. Consider the following simple automaton:



It has two states, s_0 and s_1, with s_0 being the state in which the automaton starts[3]. Starting from the start state, s_0, the automaton can accept either an a or a b. If the automaton encounters an a, then it returns to state s_0, which being an accepting state, means if no further characters are available, then the automaton has accepted the string. If on the other hand, the automaton encounters a b, then it will transition to state s_1. Since s_1 is not an accepting state, if no more characters are available in the string, then the string will not be accepted by the automaton. From this state, the only character which can be accepted is c, which will return the automaton to s_0. Thus this automaton can accept strings such as:

$$abcbcaaaabcaaa \tag{4.1}$$

An infinite number of other possible examples exist, made up of runs of zero or more a's and bc's, in any order. Since the start state is also an accepting state, the automation will accept an empty string too.

Note however, that the string accepted by the automaton above, like the example 4.1, is in an obvious sense a 'trace' – it records the sequence of transitions taken by the automaton. This operational-semantic notion of trace of the kind defined by Scott is unsurprisingly pervasive, e.g., Jacobs and Rutten also give an example of how this sort of trace arises automatically from a transition system [JR97].

4.3 The Trace Monoid

Execution of a transition system inherently generates a monoid.

In line with the operational perspective on semantics, it has become customary when using formal, mathematical approaches, to model traces of transition systems as monoids [DR95]: abstract, algebraic structures which support a single, associative binary operation, i.e., a sequencing operation which allows abstract trace events to be ordered. In order to understand the trace monoid, it first helps to understand the free monoid (where "free" is used in the conventional mathematical sense here, where it means 'unencumbered by any properties inessential

[3]Here the double circles around state s_0 have the conventional meaning, indicating that this is an 'accepting' state, where the automaton can stop successfully.

to the definition,' i.e., the simplest and most general structure supporting the laws that define it).

The free monoid is typically described as the set of of all finite strings $\Sigma^\star$ over some finite alphabet Σ, including the empty string ϵ. If Σ is a set of symbols, then $\Sigma^\star$ is a set of all words, composed of those symbols (where the unary, postfix operator $\star$ used here is known as the Kleene star). For example, if the alphabet Σ contains:

$$\Sigma = \{a, b, c, d\} \tag{4.2}$$

and there is a binary operation written $\diamond$ (we choose to make it infix), then various strings can be defined including the empty string ϵ, a, $a \diamond b \diamond c$ etc.

This operator $\diamond$ must be associative, such that:

$$\forall f, g, h \in A : (f \diamond g) \diamond h = f \diamond (g \diamond h) \tag{4.3}$$

And the empty string is required to act as a left and right identity:

$$\forall f \in A : f \diamond \epsilon = \epsilon \diamond f = f \tag{4.4}$$

At least one example of the free monoid is already familiar to all programmers: the character string (e.g., [Moe11]). In this interpretation of the free monoid for the example strings above, i.e., ϵ, a and $a \diamond b \diamond c$, the corresponding program objects would be `""`, `"a"` and `"abc"`. Here the operation $\diamond$ is interpreted as string concatenation. More generally, the free monoid is best understood by programmers as the abstract notion of a list. Differing implementation structures, e.g., arrays and linked lists, can be used to represent lists, but they both share in common an abstract notion of list – they are isomorphic to the free monoid.

The identity rule 4.4 and associativity rule 4.3 are also the category laws in disguise. Equation 4.3 is the category associative law (equation 2.8) where the symbol $\circ$ is renamed to $\diamond$, and equation 4.4 is the category identity law (equation 2.10). Thus, the monoid is also a category. The free monoid is, however, more restrictive in structure than a free category, in that ϵ serves as both left and right identity, in equation 4.4 above. This in turn tells us that the type of $\diamond$:

$$\diamond : A^\star \to A^\star \tag{4.5}$$

Thus a monoid is simply a category with only one object, or from the perspective of a programmer, a free monoid is an abstract list containing only one type.

Here we see that in the operational-semantic tradition, computer science chose the most obvious, simple, abstract, mathematical model that can describe the list of transitions taken by a transition system, as a basis for the associated 'trace'. In section 4.5 it will be shown how the fact that a monoid is inhabited by a single type is a distinct disadvantage when working with practical execution traces like those described in section 3.2.2. However first it is important to understand what the trace monoid is useful for.

4.4 Application Domains

The trace monoid is the foundation of process algebrae, and has been useful in the study of concurrency and non-determinism.

In the preceding sections in this chapter, it has been shown how the notions of operational semantics, transition system, and traces structured as a monoid involves a nexus of ideas, defined in terms of each other (originally by Scott), with its genesis found in the early days of computer science. Thus far only the free monoid has been described; here we explore the additional feature present in the trace monoid, and the applications to which it is therefore useful. The trace monoid is by definition a *partially-commutative* free monoid [DR95]. It extends the free monoid by adding the ability to commute (or re-order) sub-strings and letters. This makes the trace monoid useful for modeling concurrent and non-deterministic computations, that may have sub-computations which may appear in any order. As a result the trace monoid has a long history of being used in computer science in the study of concurrent systems and non-determinism.

In recent decades operational semantics has most commonly be expressed in the form of structural operational semantics [Plo81], which consist of inference rules in predicate logic defined via a set of transition relations, expressed as inference rules. It is "structural" in the sense of being syntax oriented, which is not in general a necessary feature of an operational semantics. Structural operational semantics has become the de facto standard for process algebrae [Plo04]. By providing partial commutativity, the trace monoid provides a unifying mathematical basis for process algebrae, such as Milner's CSS and π-calculus [Mil99], and Hoare's CSP [Hoa78].

4.5 Fit with Tracing in Practice

The trace monoid is too simple a structure to encode the compositional, source-oriented details of practical interest for many tracing activities.

As was observed in chapter 3, practical tracing systems must be source-oriented, requiring that for an operational semantic basis to be practically useful for tracing, it must also be source-oriented – a so-called 'big-step' semantics, the most well known of which is structural operational semantics [Plo81]. While not all operational semantics are source-oriented and therefore compatible with the requirements of practical tracing, nevertheless, with a suitable choice of operational semantics, source-oriented trace can be produced. For example, the trace of a Pascal program describing the execution of machine-level instructions will be unhelpful to a Pascal programmer; instead what is needed is a record of the transitions taken by some virtual Pascal machine, in terms of the Pascal source structures of which the program consists.

This operational but source-oriented trace is the record of what a programmer would see when stepping through the execution of a program, statement by statement, using a source-level interactive debugger. In effect, this type of trace is a history of the transitions taken by a virtual (for the example above, Pascal) machine – a snapshot of the machine state at each step of execution.

However, as has been observed in section 4.3, the monoid induced by any operational semantics is a structure that is inhabited by a single type. The monoid is therefore unable to support the practical requirement that tracing be source oriented *and compositional* — it cannot describe the various nested relationships between the source code entities — it is typical for the abstract syntax of a program to be structured as a tree, rather than as the simpler, flatter, list-like monoid. The trace monoid is therefore too simple a structure to support many practical tracing tasks, specifically those requiring the generation and analysis of arbitrarily nested sub-structure, e.g., anything involving the (often complex) run-time relationship between the various function and procedure calls in the program, reflecting the nested relationships between the function and procedure calls found in the source code. Many practical uses for tracing require function (and/or procedure) argument values on entry and return values on exit (see chapter 3 for many examples) — there may be varying numbers and types of arguments to functions. Object oriented programming adds additional semantic artifacts of interest, once more with hierarchical structure.

A reader familiar with algebraic data types might raise the objection here that it is possible to specify a sum type as the single type inhabiting a monoid, thereby effectively allowing multiple types within a monoid. Similarly, a C programmer might propose the type `(void *)` as the single object for a monoid, thereby effectively allowing it to contain 'anything', or an object-oriented programmer might suggest some polymorphic object type. While it is certainly the case that the single object within a monoid can accommodate a complex, structured type, the fact remains that the knowledge that the structure in use is a monoid tells us nothing about the structure of its contents and how these are related to each other (except, of course, that they are of the same type and ordered). Instead the relational information of interest is to be found inside the type suggested as the object for the monoid.

In a similar fashion it is certainly possible to *encode* arbitrarily structured information in a monoid and this is completely routine practice in computing: it is very common to encode complex information as a character string, perhaps the most obvious example is the source code for a program. But knowing nothing more about a program than it is a string of characters, we know very little about it that is of any usefulness. It is only when such a string is decoded via the grammar of the language into an abstract syntax tree (either by machine parsing or by the programmer as they read the text of the program) that it becomes possible to reason about the program and its structure. Once again, because it contains just a single type, the monoid is too simple a structure to express such compositional information.

Chapter 5

Denotational Trace

As a consequence of the categorical duality between operational and denotational semantics, the novel, dual notion of 'denotational trace' is introduced, where a 'canonical' denotational trace contains the complete collection of source-oriented and compositional, denotational semantic information required for practical execution tracing activities.

In chapter 4, the existing theoretical notion of trace already found in computer science – the trace monoid – was found to have an unsatisfactory fit with the source-oriented, compositional requirements for practical execution tracing as found in chapter 3. Faced with this inadequacy, an alternative theoretical basis for practical tracing must be sought. In this chapter, the existing semantic basis for the trace monoid is explored, with a view to finding an alternative semantic basis for tracing; one that does support the requirements of practical tracing outlined in chapter 3.

Duality provides a powerful meta-level design tool here for the task of discovering or designing a novel, yet effective theory of tracing, alternative to that provided by the trace monoid. One of the few basic facts known about every category is that it has a unique dual (up to isomorphism). By characterising the (operational) semantic basis for the trace monoid in categorical terms, the simple fact that it must have a categorical dual can be exploited; here it is used to generate the well justified and unique, alternative basis for tracing — the denotational semantic basis. This argument from duality for what is called here 'denotational trace', was foreshadowed at the beginning of this thesis in table 1.1. Revisiting this here in table 5.1, many of the components of this argument from duality have now been established (ideas already introduced have a grey background). The remaining (italicised) concepts are explained in this chapter.

The relationship between the trace monoid and operational semantics was identified in chapter 4. Chapter 2 introduced initial algebras and final coalgebras and the duality between them. What remains is to establish that:

- final coalgebras provide a categorical, abstract basis for operational semantics (section 5.1.1

The	trace monoid	*denotational trace.*	
is induced by	operational semantics	*denotational semantics*	*that induces a*
which has the abstract, categorical formalisation as	*final coalgebra semantics*	*initial algebra semantics*	*the abstract, categorical formalisation of*
that is semantics structured as	a final object in the category of F-coalgebras.	An initial object in the category of F-algebras	*is the structural basis for*

Table 5.1: The Argument from Categorical Duality for Denotational Trace, Revisited

below), and that dually,

- initial algebras provide a categorical, abstract basis for denotational semantics (section 5.1.2 below).

Having identified the alternative, denotational basis for trace it is finally possible to explore how this fits with execution tracing in practice: below we find that the essential denotational properties of source orientation and compositionality correspond exactly to the requirements for practical traces as found in section 3.3, i.e., source-orientation and compositionality. Thus, based on this argument from duality as summarised in table 5.1, in section 5.2 below the novel idea of *denotational trace* is introduced, being trace structures derived from a denotational semantic basis, rather than the operational semantic basis presented in chapter 4. Denotational trace addresses exactly the inadequacy of the trace monoid for practical purposes identified in section 4.5, i.e., that it has insufficient structure to describe the compositional, source-oriented, semantic information required for the myriad of tracing applications presented in chapter 3.

5.1 Denotational is Dual to Operational

Denotational semantics is dual to operational semantics.

As explained above, an existing category theoretic duality known to computer science is exploited here: operational and denotational semantics are each-other's dual. This result was first formalised categorically by Ong [Ong95], and has been summarised most explicitly and clearly by Hutton [Hut98], as is elaborated in the following subsections. As Hutton explains, it is an insight, the distant origins of which are unclear, that developed gradually over a long period of time.

In the following, the abstract semantic basis for the trace monoid is presented as *final co-algebra semantics*. As was observed in section 2.3, this category-theoretic structure necessarily has a dual, namely *initial algebra semantics*. Just as operational semantics is formalised abstractly as final co-algebra semantics, then, so too is initial algebra semantics the abstract basis for denotational semantics.

5.1.1 Operational Semantics are Final-Coalgebraic

The basis for the structure of operational semantics is the abstract machine, or in categorical terms, the final coalgebra.

Thus far, the trace monoid has been shown to have an explicitly operational semantic basis, and operational semantics has been explained loosely in terms of a transition system or 'machine' which transitions through a series of states, possibly producing a final result. Thus far three examples of transition systems have been presented:

1. Church's lambda calculus as a rewriting system under reduction/conversion,
2. the Turing machine and
3. a simple automaton (i.e., the example presented in section 4.2).

All three of these examples are 'operational' in the classical sense being examples of Scott's, "step-by-step evolved sequences of operations on representations" as discussed in section 4.1. Here Scott captures the programmer's intuition that these are three examples of 'machines' sharing some abstract similarity.

As was explained in chapter 4, operational semantics has been a part of computer science since the dawn of the field in the 1930's. Despite the fact that the practice of defining operational semantics developed well before there was any formalised, mathematical basis established for it, over time the realisation formed that it could be abstractly but formally captured by the mathematical, algebraic perspective. Jacob and Rutten describe how, "the insight gradually grew that ... state-based systems should not be described as algebras, but as so-called coalgebras" [JR97]. Turi and Plotkin have provided a categorical account of such transition systems [TP97], whereby the sense in which these and other transition systems are abstractly the same is precisely captured by the idea of *final coalgebra semantics.* Final coalgebra semantics, refers to semantics structured as a final coalgebra, i.e., a view of semantics as an abstract machine. Hutton explains [Hut98]:

> ... the operational approach, in which the meaning of programs is defined using a transition relation that captures single execution steps in an appropriate abstract machine. The transition relation is defined using a set of inference rules, and the meaning of a program is given by repeatedly applying the relation to generate a transition tree that captures all possible execution paths of the program. In fact, the pattern of recursion used to construct transition trees is precisely the pattern of recursion captured by unfold. Hence, an operational semantics can be characterised as a semantics defined by unfolding to transition trees.

This pattern of recursion captured by unfold was formally identifed with the final coalgebra in section 2.3. Thus the relationship between operational semantics and finite coalgebras is established via the pattern of recursion known as unfold. Hutton also points out that this categorical basis for operational semantics is not widely known, and that "most functional programmers are not aware of this connection".

5.1.2 Denotational Semantics are Initial Algebraic

> *Final coalgebras have a dual: initial algebras, that provide the structural basis for denotational semantics, otherwise known as initial algebra semantics.*

Having identified the specific, category-theoretic basis for operational semantics as final coalgebra semantics, we can ask what is dual to this? The answer is initial algebra semantics. It has already been observed in section 2.3 how the notions of initial algebra and final coalgebra are dual to each other. In order to establish that initial F-algebras form the structural basis for denotational semantics, it is necessary to be clear about what, precisely, makes a semantic description 'denotational'. Stoy describes the essence of the denotational approach [Sto77]:

> We give "semantic valuation functions", which map syntactic constructs in the program to the abstract values (numbers, truth values, functions etc.) which they denote. These valuation functions are usually recursively defined: the value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents, and it is this emphasis on the values *denoted* by all these constructs that gives the approach it's name. There may or may not be an obvious way of working out the results of these functions in any particular case: that is, the defining equations may or may not suggest a way of implementing the language. [author's emphasis]

In effect, Stoy is describing a source-oriented system ("which maps syntactic constructs") that is compositional ("the value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents"). As explained below, this compositional structure can be recognised as initial algebraic, i.e., constructor replacement applied to the abstract syntax tree for a program.

The two essential denotational features of source-orientation and compositionality can be seen even in a very simple example of denotational semantics:

$$\mathscr{M}[\![a{+}b]\!] = \mathscr{M}[\![a]\!] + \mathscr{M}[\![b]\!] \tag{5.1}$$

The double brackets due to Scott and Strachey are a notational device whereby denotational semantics is explicitly source-oriented: the contents of the brackets are syntax objects. The meaning function represented by the symbol $\mathscr{M}$ (often pronounced 'means') maps each abstract

source entity to its corresponding meaning or value in the semantic domain. Note the careful use of notation here, + on the left-hand side is different from $+$ on the right-hand side, where the former refers to an abstract syntax element, whereas the latter refers to addition in the denotational domain: $+$ is the single semantic function defined for this tiny semantics.

The identifiers a and b stand for any abstract syntax element. Their use above in defining the meaning of + is explicitly compositional — the meaning of the addition syntax is recursively composed of the meaning of the two arguments a and b (which might in turn consist of other additions).

Note that equational reasoning is explicit in the structure of the semantics here, in contrast to operational semantics, where semantic rules are typically expressed as inference rules in predicate calculus. This is no accident, as both Scott and Strachey were seeking this very feature, precisely to allow easy reasoning about program correctness, without the problems of determining when a transition system is finished processing (e.g. by reaching normal form, in a term-rewriting system), and the halting problem in general. Scott's comment that there "may or may not be an obvious way of working out the results of these functions in any particular case" [Sco70] makes this exact point.

Initial algebras are the natural structure for expressing denotational semantics, as Hutton explains [Hut98]:

> One of the most popular styles of semantics is the *denotational* approach, in which the meaning of programs is defined using a valuation function that maps programs into values in an appropriate semantic domain. The valuation function is defined using a set of recursion equations, and must be compositional in the sense that the meaning of a program is defined purely in terms of the meaning of its syntactic subcomponents. In fact, the pattern of recursion required by compositionality is precisely the pattern of recursion captured by fold. Hence, a denotational semantics can be characterised as a semantics defined by folding over program syntax. [author's emphasis]

This relationship between denotational semantics via folds (over program syntax) to initial algebras (as were defined formally in section 2.2.3) was made explicitly first by Rutten and Turi [RT94], but as Hutton points out this connection is "widely known in certain circles".

Thus, in a deep and mathematically-precise sense, operational and denotational semantics are dual to each other. It should perhaps not be surprising, that computer science developed two different, complementary models of semantics – operational and denotational – that turned out to be, at core, mathematical duals. Therefore, when looking for an alternative to the operational semantic basis for tracing, denotational semantics presents itself via duality as the compelling alternative to consider.

5.2 Tracing Denotational Semantics

The transformation of denotational semantics by the addition of execution tracing is modelled by an algebra homomorphism, more specifically a canonical isomorphism.

Given that the two essential, defining features of denotational semantics are source-orientation and compositionality, and that these are the two key requirements for practical execution traces identified in chapter 3, a semantic basis has been found that appears to be a precise fit. So far it has been shown that:

1. denotational semantics is the dual of operational semantics, the latter providing the implicit, and later explicit basis for the trace monoid, and
2. the essential properties of denotational semantics — source-orientation and compositionality — are an exact fit for the needs of practical execution tracing, i.e. a source-orientated record with complex, nested semantic structure.

It is now possible to ask, in categorical terms: what relationship does tracing have to denotational semantics? More specifically, how can tracing be derived from a denotational-semantic basis by direct analogy with the obvious and automatic derivation of the trace monoid from an operational basis?

We need something to transform existing semantics into semantics which also produces a trace. We want to systematically augment (all) the semantic productions so they produce trace, and we also want to change the type of our semantics. We want to be able to say something like:

$$\mathscr{M}_{\mathscr{T}} = \mathscr{T}(\mathscr{M}) \tag{5.2}$$

Where $\mathscr{M}_{\mathscr{T}}$ is a new semantic specification, augmented by execution tracing, created by applying some tracing transformer $\mathscr{T}$ to an existing denotational semantics $\mathscr{M}$. The essential idea here is that the trace is systematically derived from the existing semantics, rather than in an ad hoc fashion.

5.2.1 Monads for Modular Semantics

Monads allow denotational semantics to be structured in a modular fashion.

In the very early stages, denotational semantics did not have a suitable, mathematical structure with which to model the semantic domain, and Strachey proceeded on the basis that something suitable would be found in due course [Str00]. Indeed, shortly thereafter Scott identified a class of suitable structures, now known as Scott domains [Sco70]. One deficiency of Scott domains

(the first viable denotational model identified) is that the resulting semantics is not modular. Subsequently, various other mathematical structures have been used to model the semantic domain, for this reason as well as others.

The requirement that semantic values are available immediately means that the semantic domain required for denotational semantics may contain some strange entities. In order to support just the lambda calculus ([Bar81]), the semantic domain needs to contain some pretty odd things, including infinite types. Similarly, given that the lambda calculus supports universal computation, it can therefore express programs which do not terminate. This requires the semantic domain to support a 'value' which means 'did not terminate'. Now of course the fact that such a value exists in the semantic domain does not mean that it is effectively calculable for an arbitrary program - in fact it cannot be, as to do so would involve solving the halting problem.

Here we use the monadic approach introduced by Moggi, specifically for the purpose of allowing modular semantics [Mog89]. It has already been seen in section 2.2.1 that monads provide a means to accommodate a wide range of program behaviours in a categorical setting. Modularity in the semantics allows a clear separation between basic semantics for a language of interest and semantics augmented with trace generation. Further, this separation of concerns makes it possible to investigate how tracing can be added to any denotational semantics consistently, as we show in the next section. Moggi's ideas have been successful in the practical exploration of modular (denotational) semantics, such as the work of Liang, Hudak and Jones on modular interpreters [LHJ95]. In particular, they use monad transformers to add features to an existing semantics, a practice that is followed here. One additional, advantageous feature of the denotational approach – which is exploited in the next chapter – is that the semantic equations can be expressed directly in a functional programming language, as a (monadic) interpreter for the language of interest, and a (monad) transformer which can be used to transform this interpreter into one which automatically produces execution trace, in addition to the original semantics.

Because a monadic basis has been chosen for the semantics here (since it is useful subsequently for illustrative purposes), the structure we use for implementation purposes is a *monad transformer*. This allows us to transform our semantic monad in the fashion above, by applying a monad transformer to it. Chapter 6 contains a concrete example of a tracing monad transformer, for a concrete example of semantics[1].

5.2.2 Canonical Tracing

> *The morphism required to transform an initial algebra describing denotational semantics into semantics augmented with tracing output is a homomorphism, more*

[1] Monad transformers have become the standard way of combining the features of two monads in Haskell – the concept is in widespread use in this language [OGS08].

specifically an isomorphism, which should be canonical with respect to the original semantics.

Despite the fact that the current discussion is completely abstract with respect to semantics – no specific example is under discussion – there are some ways in which the monad transformer $\mathscr{T}$, can be characterised in more general terms. The observation that $\mathscr{T}$ can be seen as a monad transformer derives from the specific choice of using monads to structure the semantic domain. While this is a useful approach for practical reasons, it is not the most abstract perspective that can be taken on $\mathscr{T}$.

Firstly, and most fundamentally, it must be remembered that denotational trace as a concept results from an exercise in algebraic semantics: the recognition that that initial F-algebras and final F-coalgebras capture abstractly the consequently dual semantic notions of denotational and operational respectively. This perspective requires simply that denotational semantics are expressed as an initial F-algebra. This means that, most abstractly, $\mathscr{T}$ is an algebra *homomorphism*. However we can qualify $\mathscr{T}$ even more precisely.

The operation $\mathscr{T}$, that adds tracing to denotational semantics, admits an inverse — an operation $\mathscr{T}^{-1}$, to remove tracing from traced denotational semantics. It is axiomatic when tracing that the addition of tracing should not alter the denotational semantics, which means that recovery of the original semantics is assured. $\mathscr{T}$ is therefore an *isomorphism*.

Section 2.1.2 described several notions of isomorphism — the concept of a *canonical* isomorphism captures precisely the important property of "no choice" in the derivation of trace from denotational semantics. There is only one sensible option for a 'canonical' isomorphism to apply tracing, and that is to trace 'everything'. Thus we find the important abstract notion of *canonical tracing*, being tracing applied to 'everything' in the denotational semantics.

Knowing also that the canonical tracing morphism is an algebra homomorphism means that canonical tracing has implementation implications: each semantic function in the algebra should be traced. This in turn provides an exact specification for the operations to be recorded in what we accordingly call a *canonical trace*.

5.2.3 Canonical Traces

We call the maximal set of trace information available from a dual-consequent, denotational perspective, a "canonical trace", when produced by canonical tracing.

In order to answer the question as to what information will be provided by a denotational trace, we would like to have a correspondence between trace objects and the corresponding semantics for each element of syntax. The requirement that the tracing morphism is canonical with respect to the addition of tracing to semantic functions, ensures that all semantic functions will be traced. We might imagine this tracing transformation as consistently wrapping each semantic function in the denotational semantics, i.e., each of the morphisms in the algebra for

the semantics, with trace start and end logging, and the details of the argument(s) and result(s). Each semantic function will typically map to many separate instances of entries in a trace, since each semantic function is generally executed more than once. (So from the programmer's perspective every item of trace can be mapped back to the corresponding semantic function, but many other trace entries may map to that same semantic function.)

By tracing all of the denotational semantic functions – resulting from the use of a canonical isomorphism – a *canonical trace* is produced, that contains all semantically relevant information. A canonical trace therefore represents a maximal bound with respect to the execution tracing information that can be collected, given a denotational semantic specification. A concrete example of a full trace is given in section 6.

It should be emphasised that canonical traces are unlikely to be optimally useful in unmodified form. For example, profiling activities will only focus on timing information and execution counts, which require only a subset of information available in the canonical trace, i.e. a derived trace. In fact, it is an essential feature of many practical uses of tracing that canonical traces will not be used, because appropriate selection of the essential information needing to be included in a derived trace is key to enabling the associated activities effectively. In short, as was pointed out in section 3.2.2, practical tracing systems need "temporal and static selectivity". The practical filtering and query of traces is explored in more detail in chapter 6.

Nevertheless, the purpose here is to highlight that canonical traces represent the maximal set of execution information available, derived via a completely straightforward, canonical transformation applied to the language semantics, and that many (and perhaps all) source-oriented, trace-based activities can be enabled by canonical traces.

5.3 Two Sides to the Semantic Coin

Operational and denotational traces are complementary.

There is no reason per se why the mathematical dual of a familiar, useful structure might itself have concrete realisation as a similarly familiar or useful structure. However, as observed in section 2.3, in many cases mathematical duals do turn our to be familiar, useful objects. To aid the reader in understanding how the the various useful dualities discussed in this chapter are related, their relationships are summarised in table 5.2.

There is also no reason why mathematical duals, even if they are interesting or useful, should necessarily correspond by applying meaningfully to the same concrete object of interest (e.g., share the same carrier type). Nevertheless, once again there are many useful cases where this does happen, such as the example of the types `List` and `Stream` presented in section 2.3, highlighting that cons-lists have a dual nature, being both data and codata. A cons-list can be manipulated through the interface provided by its constructors (i.e., `Cons` and `Nil`, as an initial

object:	initial algebra	final coalgebra
familiar example:	data	codata
recursion strategy:	fold	unfold
morphism:	catamorphism	anamorphism
intuition:	abstract data type	abstract machine
semantics:	denotational	operational
sameness:	equality	bisimilarity
proof technique:	induction	coinduction

Table 5.2: Duality Quick Reference

algebra), or the interface provided by its destructors (i.e., `head` and `tail`, as a final coalgebra).

In addition, there is a much investigated correspondence between operational and denotational semantics when relating to the same programming language [Ong95]. A denotational semantics is said to be *adequate* if, for any equality between denotations, the corresponding operational semantics is observationally equivalent (i.e., bisimilar). When (observationally) distinct operational semantics are also distinct in the corresponding denotational semantics, the denotational semantics is termed *fully abstract*. Thus, not only does duality tells us that initial algebra semantics are dual to final coalgebra semantics, and that this means that both must exist as unique alternatives to each other, but furthermore, each provides a complementary perspective when applied to the same language: they are two sides of the semantic coin. Turi and Plotkin [TP97] explain that both are necessary:

> Both operational and denotational semantics are necessary for a complete description of a programming language: the former for specifying the execution of programs and the latter for reasoning about them in terms of abstract, mathematical entities.

Meseguer and Goguen [MG86] make this point more generally for initial algebras and final coalgebras:

> Much confusion can be avoided by distinguishing sharply between (concrete or abstract) *data types* that are just algebras, and (concrete or abstract) *machines* that in addition may have internal *states* . . . The fact that many common examples can be viewed from either perspective contributes to the potential confusion.
>
> There is for instance, the rather pointless controversy about whether final or initial algebra semantics is 'best'. For abstract machines, it is *behavior* that matters. Machines that represent and manipulate their internal states differently (i.e., are non-isomorphic as data types) can still have the same behavior. A software module can usually be *realized* in many different ways; among these, the *final* one uses as little storage as possible for internal states, while the *initial* one has no sharing at all

> for storage of states. Because the space efficiency of the final realization can greatly reduce its time efficiency, there are many cases where the pragmatically correct choice of data representation is neither initial nor final, but rather something in between ...
>
> In summary, initial realizations are appropriate in case all behavior is visible behavior; final realizations may be appropriate in case there are hidden internal states, but often the most practical realizations, although neither initial nor final, are rather close to being initial. [author's emphasis]

The reality is that the world of programming is inhabited by both species of structure, and they are equally useful for different applications. Neither initial algebra nor final coalgebra semantics are suited to all applications, providing, as they do, the complimentary notions of abstract data and machine. These complimentary perspectives are suited to answering different questions. Each of these dual perspectives come equipped with tools for formal reasoning about them. Notably, coinduction/bisimilarity have been useful in exploring concurrency [RT94], and more recent work has exploited the connections between the trace monoid, co-algebraic structures and coinduction [HJS07]. Jacobs and Rutten provide an overview of the relationship between these dual structures, and the associated tools which come with them (bisimilarity/coinduction vs equality/induction) [JR97]. Jacobs and Rutten provide a tutorial on coinduction, which includes several examples of coinductive reasoning. Section 6.4.3 and section 6.4.2 have examples of reasoning by mathematical induction.

Duality suggests we should find that denotational traces are good for questions of correctness (via equality), will be source-oriented and compositional (as required for practical tracing in section 3.3), and that induction should be a natural proof technique. Knowing that the denotational perspective is unique and dual to the operational one, we get some hints as to what denotational traces might be good for – i.e., precisely those domains where operationally-based trace has traditionally not been used, being the practical tracing activities described in chapter 3.

A key contribution of this thesis is that the semantic basis for tracing is considered explicitly, as is how tracing in both practice and theory relates to both operational and denotational notions of semantics. It is this result which answers the questions:

- Why is the existing trace monoid used so little in practice when it is by contrast a foundational concept in the study of concurrent, distributed and non-deterministic systems with process algebrae? What explains the difference in utility of the trace monoid in these two problem domains?

- Why are practical tracing activities not grounded upon some formal concepts and ideas as is so typically the case of other artefacts in computing? What would a suitable theoretical basis for tracing as it is found in practice look like?

The explanation in both cases is provided by the recognition that the existing theoretical basis provided by the trace monoid is only describing half of the semantic story. The operational and denotational perspectives are complimentary. This duality suggests we will find that practical tracing – the requirements of which cannot be met by the trace monoid – must instead be based on an implicit, denotational semantic assumption. This is indeed what we have found: the common requirements for practical tracing (see section 3.3) of source-orientation and compositionality also are the two key defining features of denotational semantics.

Chapter 6

A Denotational Tracer

To highlight the practical, straightforward nature of both denotational tracing implementation and usage, a canonical, denotational tracer is implemented for a simple language, sufficient to provide some simple examples of nevertheless sophisticated uses for denotational traces, including a specification recovery from execution trace and analyses of space and time complexity using formal reasoning applied to traces (i.e., proof by induction, as implied by categorical reasoning based on the denotational basis, for which induction is the associated proof technique).

In order to demonstrate how easily a denotational trace can be derived from the denotational semantic specification of a language, and thereby show the practical usefulness of denotational traces, a very simple sequential language has been deliberately chosen as the basis for an example to trace. Firstly, this is because simple sequential programming problems are of fundamental importance to most programming tasks, even those involving elements of concurrency. The language we use here is in the tradition of Scott's LCF and Plotkin's PCF, which have a long history of supporting useful research into programming language semantics (and in particular the exploration of the correspondence between denotational and operational semantics [Ong95]). The great usefulness of such tiny, exemplary languages — in this case based ultimately on Church's lambda calculus [Bar81] — is that they are uncluttered by a mass of syntactic and semantic details, making it possible to focus efficiently on a few key details of interest. In general such languages are as simple as possible, although universal. Named declarations are included in the language (i.e. a 'let' or 'while' declaration can be specified). Without this, recursive programs (and therefore loops) cannot be expressed without the use of a recursion combinator, severely impacting readability of source code, for even trivial programs. The execution order is applicative (aka eager evaluation), the familiar order of argument evaluation found in most mainstream languages today. All functions are curried (i.e., have exactly one argument) as this can be used easily to model n-ary functions with any number of arguments. The concrete syntax for the language is a small subset of Haskell, although this is of little consequence here as we are interested in the abstract syntax only. The meta-language used to

implement the denotational semantics here is Haskell [Mar10].

Canonical tracing is added to the denotational semantics, and a simple example program and its traces are explored, as well as the associated reasoning and tools needed for some example software engineering activities, including specification recovery and reasoning about space and time complexity.

6.1 Executable Semantic Specifications

The denotational semantics specified here are implemented in the meta-language Haskell; execution of the denotational specification constitutes an interpreter, a useful experimental tool.

One additional, advantageous feature of the denotational approach to semantics – which we exploit here – is that the semantic equations can be expressed directly in a functional programming language, as a (monadic) interpreter for the language of interest, and a (monad) transformer which can be used to transform this interpreter into one which automatically produces execution trace, in addition to the original semantics.

As early as the 1980's, it was recognised that it is convenient to implement denotational semantics in a form where they can be executed [Wat86]. Advantages include machine-checking of syntax and largely automated testing. Executable denotational semantics effectively provide an interpreter, which can be an extremely useful research platform in itself. In the context of this project, an interpreter provides a natural integration point for automatic tracing.

While denotational semantics is structured such that the semantic domain contains values which in principle are immediately available, these values (such as oddities like 'did-not-terminate') may not be effectively calculable in practice. This presents no contradiction – the existence of denotational semantics does not somehow solve the halting problem. We simply have to accept that if denotational semantics are executable (i.e., computable), then some values (e.g. 'did-not-terminate') are not effectively calculable.

Nothing here is intended to imply this implementation of a denotational tracing system is optimal for any particular application – most practical systems would be quite optimised to the problem at hand – the approach used here is chosen because it is well suited to the exploration of the relationship between semantics and tracing.

6.2 A Simple Denotational Interpreter

A simple language with just enough structure to exercise some interesting examples is constructed: the syntax is a simple variant of ML-style languages, the semantics are simple, with a strict/eager execution order, and the value domain contains just a few basic types.

The concrete syntax for the language is a Haskell-like variant of the lambda calculus. The abstract syntax is very simple. A program consists of an expression, and an optional list of declarations. Expressions may include literals, identifiers, conditional expressions, function applications and lambda expressions:

```
data Prog = Prog Expr [Decl]
data Decl = Decl String Prog
data Expr = Number Integer
          | Boolean Bool
          | Ident String
          | Lambda String Expr
          | IfStmt Expr Expr Expr
          | Apply Expr Expr
```

The semantic domain supports a few basic types of values. Lambda function values are associated with the environment prevailing at the time of their declaration (i.e. the environment is bound statically as is usual in almost all languages).

```
data Value = ValInteger Integer
           | ValBoolean Bool
           | ValLambda String EnvId (() -> Meaning)
```

The meaning of a program or expression is the value it denotes, augmented by both an environment and trace information.

```
type Means a = Env a
```

The monad `Env` provides the mappings of bindings to their values. It can be thought of as a type modifier applied to the value type, that associates appropriate environment bindings with semantic values. We will only be applying `Means` to `Value`, and this will appear many times in subsequent type declarations, so we define a convenient type alias for the semantic value domain:

```
type Meaning = Means Value
```

Being denotational, the semantics here consist of a single function for each of the abstract syntax data structure elements above.

```
declare :: Meaning -> [Meaning] -> Meaning
declare expr decls = local $ do sequence_ decls
                                expr
```

A declaration, or a program, is simply the meaning of an expression, evaluated in the context of a local environment with a set of associated identifier bindings.

```
abstract :: String -> Meaning -> Meaning
abstract arg expr = do
  env <- getEnv
  return $ ValLambda arg env $ \_ -> expr
```

Lambda abstractions are evaluated in the context of the environment that prevailed at the time of definition - the function getEnv simply returns an identifier for the current environment state. Application of lambdas occurs in the context of the environment at the time of abstraction (using withEnv), extended with the additional binding of the argument value to the lambda argument variable identifier. Function application is strict: first the function expression is evaluated to derive a lambda value or error, then the argument is evaluated and bound to the argument in the environment, before finally the body of the lambda is evaluated.

```
apply :: Meaning -> Meaning -> Meaning
apply fn expr = do
  ValLambda arg env str lambda <- fn
  e <- expr
  withEnv env $ do bind arg $ return e
                   lambda ()
```

Choice, provided by the `if` statement, evaluates a (Boolean) condition, then evaluates the `then` or `else` consequent. Binding and finding of identifiers via the bind and find functions in the algebra operate exactly as would be expected, i.e., they add a new binding to the environment, or lookup the environment via an identifier, using the mapping provided by `Env`.

```
choose :: Meaning -> Meaning -> Meaning -> Meaning
choose p e1 e2 = do ValBoolean b <- p
                    if b then e1 else e2
```

Denotational semantics provides an explicit mapping from each abstract syntax element to a meaning in the semantic domain, composed as necessary from the meanings of syntactic sub-components. It is therefore natural to express such semantics as structural induction over the data types describing the abstract syntax, such as those above. The pattern of recursion involved is known as 'fold' to functional programmers [Hut99], and recognised as a functional programming design pattern known as a catamorphism [MFP91]. At this point, the semantics are explicitly structured as initial algebra semantics, and we see how naturally this arises in the context of denotational semantics, in particular due to source-orientation and compositionality.

Using these ideas, we define the type of the fold algebra for the abstract syntax of the essential functional language, in particular we use *updatable fold algebras* as described in [KLV00]:

```
data SynCataAlg uProg uDecl uExpr = SynCataAlg
  { fProg    :: uExpr -> [uDecl] -> uProg
  , fDecl    :: String -> uProg -> uDecl
  , fBoolean :: Bool -> uExpr
  , fNumber  :: Integer -> uExpr
  , fIdent   :: String -> uExpr
  , fLambda  :: String -> uExpr -> uExpr
  , fIfStmt  :: uExpr -> uExpr -> uExpr -> uExpr
  , fApply   :: uExpr -> uExpr -> uExpr }
```

The fold operation (i.e. pattern of recursion) is then defined for the abstract syntax algebra:

```
class Fold alg t a where
  fold :: alg -> t -> a
instance
  Fold (SynCataAlg uProg uDecl uExpr) Prog uProg
  where
    fold alg (Prog e ds) =
      fProg alg (fold alg e) (map (fold alg) ds)
instance
  Fold (SynCataAlg uProg uDecl uExpr) Decl uDecl
  where
    fold alg (Decl i p) = fDecl alg i (fold alg p)
instance
  Fold (SynCataAlg uProg uDecl uExpr) Expr uExpr
  where
    fold alg (Literal s)      = fLiteral alg s
    fold alg (Number n)       = fNumber alg n
    fold alg (LambdaL i e s)  = fLambdaL alg i (fold alg e) s
    fold alg (LambdaS i e s)  = fLambdaS alg i (fold alg e) s
    fold alg (IfStmt p e1 e2) = fIfStmt alg (fold alg p)
                                            (fold alg e1)
                                            (fold alg e2)
    fold alg (Apply f e)      = fApply alg (fold alg f) (fold alg e)
    fold alg (Ident i)        = fIdent alg i
```

We can now express the meaning of a program as an instance of the syntax fold algebra:

```
evalA :: (Meaning -> Meaning -> Meaning)
      -> SynCataAlg Meaning Meaning Meaning
```

```
evalA = SynCataAlg declare
                   bind
                   (return . ValBoolean)
                   (return . ValInteger)
                   find
                   abstract
                   choose
                   apply
```

The primary disadvantage of this technique becomes apparent here: significant boilerplate code needs to be written and maintained for each type. In this case, this is not of concern as the examples we are working with are deliberately chosen in order to be simple and clear for illustrative purposes, rather than entailing all of the details which would be inherent in a full-blown, practically-oriented system, for a complete, general purpose language. As a result of these intentionally small semantics, the boilerplate involved in constructing a syntax fold algebra here is minimal.

Denotational semantics usually uses Scott brackets, to provide an explicit mapping from (abstract) source constructs to semantic domain values. We do not have access to these special symbols in Haskell, however the transliteration between the two is trivial and purely notational.

More importantly, in effect what we have here are denotational semantics structured as initial algebra semantics, i.e. a fold applied to the associated algebra, gives us an interpreter which can be applied to any abstract syntax element:

```
interpret = fold evalA
```

6.3 Trace Generation

> *In a straightforward way, canonical trace structures are defined for the language, and modified semantics with tracing added are generated by application of a suitable monad transformer.*

For the purposes of this thesis, the answer to the question, "what should be traced?" is answered with, "everything," i.e., we want a complete record of the semantics, because we are interested in the limits to the information available on this basis. The trace structure thus directly reflects the compositional structure of the semantic equations in 1:1 correspondence:

```
data Trace
  = Prog [Trace] -- bindings
         Trace   -- execution
  | Bind Identifier Trace  Value
```

```
  | Boolean Bool
  | Number Integer
  | Choose Trace -- condition
           Trace -- consequent
  | Find Identifier Value
  | Abstract Identifier Value
  | Apply Trace -- function
          Trace -- argument
          Trace -- binding
          Trace -- execution
          Value -- result
```

Tracing is added to our semantic domain as follows:

```
type Means' a = (TraceT Means) a
```

TraceT is a monad transformer, i.e., a type modifier applied to our existing value domain that associates appropriate state information with semantic values, being the trace information collected thus far during execution. Once again, we define a convenient type alias:

```
type Meaning' = Means' Value
```

Canonical tracing as specified above is implemented here by defining a transformation (traceT - a higher-order function) that can be applied to the existing semantic fold algebra constructed earlier for the example language:

```
traceT :: SynCataAlg Meaning Meaning Meaning
       -> SynCataAlg Meaning' Meaning' Meaning'
traceT alg = SynCataAlg
  (\ expr decls    -> trace "prog" [] $ fProg alg expr decls)
  (\ ident prog    -> trace "bind" [ident] $ fDecl alg ident prog)
  (\ bool          -> trace "boolean" [show bool] $ fBoolean alg bool)
  (\ num           -> trace "number" [show num] $ fNumber alg num)
  (\ ident         -> trace "find" [ident] $ fIdent alg ident)
  (\ arg thunk str -> trace "abstract" [arg, str]
                    $ fLambda alg arg thunk str)
  (\ p e1 e2       -> trace "choose" [] $ fIfStmt alg p e1 e2)
  (\ fn expr       -> trace "apply" [] $ fApply alg fn expr)
```

This allows us to define a modified interpreter, which augments the original one by collecting trace information as well:

```
interpret' = fold $ traceT evalA
```

The practical motivations for tracing often involve programs that are long-running or do not terminate, or programs that terminate with an exception or error. This suggests that traces should be structured as an ordered, potentially infinite collection of abstract trace events.

At the lowest level we model traces similarly here, using the inbuilt Haskell list type:

```
type Trace = [Event]
```

Further, a list allows us to have ‘begin’ events which do not have a corresponding ‘end’ event, which provides a convenient description for programs which terminate in error, with one or more semantic steps incomplete. While this model of a sequence of events provides a suitable low-level basis, it does not describe richer, compositional structures relating directly to syntax. Without events indicating both the start and end of semantic operations, their exact hierarchical relationship cannot be recorded. Thus, each trace event must encode the details of the associated semantic function beginning (inputs) or ending (outputs) and the time-stamp.

In practice, all that the traceT transformer does is to wrap each function in the existing algebra in a call to trace which has the side-effect of logging begin and end trace events associated with that function; the semantics described by the original algebra are not altered.

```
trace str details action = do bgn
                              result <- action
                              end $ printVal result
                              return result
  where bgn = log Bgn $ args details
        end = log End
```

The `log` function records a trace message, with `args` formatting a list of arguments appropriately.

When working with these low-level traces we will first be parsing them into the trace structures defined above — note that the trace structures could have been generated directly in principle.

6.4 A Simple Example Program

As a demonstration of the practical usefulness of denotational trace to users of tracing systems, a simple example program is executed using the denotational tracer and the resulting execution traces are used to perform several software engineering tasks, including specification recovery and analyses of space and time complexity, with mathematical induction being a natural proof technique.

Having defined canonical tracing for the simple language in previous sections, we can now explore the concept of canonical traces in practice with an example program. Consider the function f (factorial) defined by:

$$f(n) = ff(1, n) \tag{6.1}$$

$$ff(m, 0) = m \tag{6.2}$$

$$ff(m, n + 1) = ff(m \times n, n) \tag{6.3}$$

This function f is in turn defined as a tail-recursive version ff corresponding to the familiar iteration often seen in introductory programming texts. Restating f in the concrete syntax of our simple language[1], we have:

```
f = \n -> ff 1 n;
ff = \m -> \n -> if eq n 0
                 then m
                 else ff (mul m n) (sub n 1);
```

Because canonical traces are inherently and indeed maximally verbose, a small and simple example was chosen deliberately here for reasons of space. Bearing in mind that the semantics of the simple language defined for this example language specify that functions are applied in applicative order (i.e., the function argument is evaluated before the body of the function), executing the expression `f 0` results in the trace below. Some longer trace lines are elided for readability, ellipses at the right-hand side indicate where this has been done. Also, a time-stamp could be included as was done in the example traces in chapter 3, but this was not done here, because the examples explored in this chapter do not require this information, so again, this is elided for readability. Otherwise the trace is exactly as it is produced in full form. The indentation is a simple presentation device to help visualise and highlight the nested structure.

```
> prog
  > bind f
    > prog
      > abstract n (ff 1 n)
      < abstract = \n -> ff 1 n
    < prog = \n -> ff 1 n
  < bind = \n -> ff 1 n
  > bind ff
    > prog
      > abstract m (\n -> ...)
```

[1]The standard library for the simple language includes just three inbuilt, prefix, binary, primitive functions: `eq`, `mul` and `sub`, being equality, multiplication and subtraction respectively.

```
      < abstract = \m -> \n -> ...
    < prog = \m -> \n -> ...
  < bind = \m -> \n -> ...
  > apply
    > find f
    < find = \n -> ff 1 n
    > number 0
    < number = 0
    > bind n
    < bind = 0
    > apply
      > apply
        > find ff
        < find = \m -> \n -> ...
        > number 1
        < number = 1
        > bind m
        < bind = 1
        > abstract n (if ...)
        < abstract = \n -> ...
      < apply = \n -> ...
      > find n
      < find = 0
      > bind n
      < bind = 0
      > choose
        > apply
          > apply
            > find eq
            < find = \v1 -> eq v1
            > find n
            < find = 0
            > bind v1
            < bind = 0
          < apply = \v2 -> eq 0 v2
          > number 0
          < number = 0
          > bind v2
          < bind = 0
        < apply = True
        > find m
        < find = 1
      < choose = 1
    < apply = 1
  < apply = 1
< prog = 1
```

6.4.1 Specification Recovery

Denotational trace is used to recover a specification for the program.

Some examples showing how traces can be useful for reasoning about correctness have already been shown in section 3.1. Beyond these simple examples of debugging already provided, more sophisticated analyses of correctness can be supported, such as specification recovery, where the specification for a program is recovered from its traces. This scenario is directly relevant to a programmer engaged in the practical problem of software maintenance, by attempting to understand a program via reflection on the source code and corresponding traces.

Consider the evaluation of `f 3`:

```
> f 3
  > ff 1 3
    eq 3 0 = False
    mul 1 3 = 3
    sub 3 1 = 2
    > ff 3 2
      eq 2 0 = False
      mul 3 2 = 6
      sub 2 1 = 1
      > ff 6 1
        eq 1 0 = False
        mul 6 1 = 6
        sub 1 1 = 0
        > ff 6 0
          eq 0 0 = True
        < = 6
      < = 6
    < = 6
  < = 6
< = 6
```

The trace above consists of just the `apply` events. A simple program was built to generate this derived trace. Nothing is reordered; some information that is irrelevant to the task at hand has been removed.[2]

Laying out this information a little differently, we can see the execution tracing of `f` with a sample selection of inputs (arguments) `n` = `0`, `1`, `2`, etc. will yield respective information:

[2]Details of the tool used to generate this derived trace can be found in section 6.4.1.1.

```
f 0 = ff 1 0
    = 1
f 1 = ff 1 1
    = ff (mul 1 1) (sub 1 1)
    = 1
f 2 = ff 1 2
    = ff (mul 1 1) (sub 2 1)
    = ff (mul 1 2) (sub 1 1)
    = 2
f 3 = ff 1 3
    = ff (mul 1 3) (sub 3 1)
    = ff (mul 3 2) (sub 2 1)
    = ff (mul 6 1) (sub 1 1)
    = 6
```

...etc.

The great value of these execution traces is that from them may now be inferred an invariant for `ff`. While invariant analysis is usually presented in iterative procedural contexts, it's equally applicable in tail-recursive functions. For initial argument values to `ff` of `m` $= M0$ and `n` $= N0$, then for any subsequent invocation of `ff` we hypothesise:

$$m \times fact(n) = M0 \times fact(N0) \tag{6.4}$$

where:

$$fact(0) = 1 \tag{6.5}$$

$$fact(n+1) = (n+1) \times fact(n) \tag{6.6}$$

Note that this $fact$, while potentially executable, plays the role of a specification rather than an implementation artifact in this scenario.

Subsequently, this invariant hypothesis can be separately validated and used in the discovery of specifications of other components. Deductive validation can be achieved using correctness proof techniques (e.g. fold-unfold program transformations [Dij76] involving factorial). The specification for ff can be derived from conjoining the invariant with the termination condition $ff(m, 0) = m$, thus

$$ff(M0, N0) = M0 \times fact(N0) \tag{6.7}$$

The specification for f is then derived directly

$$f(X) = ff(1, X) = 1 \times fact(X) = fact(X) \tag{6.8}$$

The key to this process, as exemplified by the recovery of a specification for f from its implementation, was the presentation of denotational traces that facilitated reasoning about the source code.

6.4.1.1 'readable' Trace Analysis Tool

> *More specific traces can be derived from canonical traces via structural induction over the trace structures.*

In order to produce the 'readable' summary of the example, canonical trace in this section, we do structural induction over the trace structures:

```
readable (Prog _ p) = readable p
readable (Number _) = ""
readable (Literal _) = ""
readable (Bind _ _ _) = ""
readable (Choose t1 t2) = readable t1 ++ readable t2
readable (AbstractL _ _) = ""
readable (AbstractS _ _) = ""
readable (Find _ _) = ""
readable (Delay _) = ""
readable (Force fs) = concatMap (\(_, ts, _) -> concatMap readable ts) fs
readable a@(Apply _ t1 t2 Nothing r) =
  concat [ maybeReadable t1
         , readable t2
         , name a , " " , args a, " = ", r, "\n" ]
readable a@(Apply _ _ _ (Just t2) r) =
  concatMap readable (argTraces a) ++
    concat [ "> ", name a, " ", args a, "\n"
           , readable t2
           , "< = ", r, "\n" ]

args = unwords . argVals

maybeReadable Nothing  = ""
maybeReadable (Just x) = readable x
```

The algorithm functions by walking the trace structures and producing a human-readable subset of the elements encountered. Functional programmers will recognise this as structural induction, i.e., fold-expressible, or in category-theoretic terms, this is a catamorphic algorithm in structure.

As a simple presentation device, some indentation can be added to the output of `readable` based on the `<` and `>` characters:

```
indent :: String -> String
indent = unlines . ind' 0 . lines
  where
    ind' :: Int -> [String] -> [String]
    ind' i (('>':x):xs) = (spaces i ++ ('>':x))
                        : ind' (i + 2) xs
    ind' i (('<':x):xs) = (spaces (i - 2) ++ ('<':x))
                        : ind' (i - 2) xs
    ind' i (x:xs)       = (spaces i ++ x)
                        : ind' i xs
    ind' _ []           = []
    spaces n = replicate n ' '
```

Indentation helps highlight the nested relationships found in the (denotational) trace.

6.4.2 Time Complexity

The time complexity of the program is analysed using proof by induction.

Because a canonical trace contains a record of all semantic function, the length of such a trace can be used as a direct measure of the time taken to execute a program. A small program was built to compute this metric from a canonical trace (the details of the tool used to extract this information from the trace can be found later in this section). Other measures of time complexity derived from the canonical trace are certainly possible, for example varying time costs could be associated with each syntax element/semantic production.

n	steps (f n)	steps (f n) - steps (f (n-1))
0	28	
1	56	28
2	84	28
3	112	28

Table 6.1: Time Complexity of “f”

Using this simple tool for the factorial example above, we can tabulate the number of semantic steps involved in executing the `f` function in table 6.1. Because the number of steps increases by the same amount with each increment of `n`, we can infer the relationship:

$$steps(f(n+1)) = steps(f(n)) + 28 \tag{6.9}$$

i.e., that for each increment of `n`, 28 more steps will be required. In other words, the programmer at this point may now suspect, based on inspection of the traces for the various inputs above, that the time complexity of this algorithm is linear with respect to the size of the input value.

This relationship clearly holds at least for the first few values shown in the table, and is also suggested by the 'shape' of the readable trace above, which shows an extra nested, recursive call to `ff` for each decrement of its second argument:

```
...
  > ff 1 3
    eq 3 0 = False
    mul 1 3 = 3
    sub 3 1 = 2
    > ff 3 2
      eq 2 0 = False
      mul 3 2 = 6
      sub 2 1 = 1
      > ff 6 1
        eq 1 0 = False
        mul 6 1 = 6
        sub 1 1 = 0
        > ff 6 0
          eq 0 0 = True
        < = 6
      < = 6
    < = 6
  < = 6
...
```

If the programmer now wishes to prove via deductive reasoning that this relationship holds for all (positive) values provided to `f` — say in order to prove to themselves that this function is not implicated in a run-time performance problem they may be debugging — then partial traces like the one above are not sufficient. It is not possible to know whether other paths through the code might be taken, for example when the argument to `f` is `4`, no trace has been seen for this scenario. Because this trace contains only a subset of all denotational-semantic information, it is not in general possible for the programmer to know what else might be happening that is not shown in the trace.

Here the canonical trace comes into its own as a tool, precisely because by definition it provides a complete record, containing all denotationally relevant information. The programmer

can inspect the canonical trace, and in particular the entries relating to the use of the `if` construct in the language, e.g., the relevant extract from the canonical trace provided above for `f 0` is:

```
...
       > choose
         > apply
           > apply
             > find eq
             < find = \v1 -> eq v1
             > find n
             < find = 0
             > bind v1
             < bind = 0
           < apply = \v2 -> eq 0 v2
           > number 0
           < number = 0
           > bind v2
           < bind = 0
         < apply = True
         > find m
         < find = 1
       < choose = 1
...
```

Here the programmer can see that `ff` terminates whenever its second argument is `0`. An inspection of the surrounding trace shows that for `f 0`, this is what will happen, because the argument to `f` is supplied unmodified as the second argument to `ff`. This fact can consequently be taken as the base case for an inductive argument that the time complexity of the algorithm is linear. We know from table 6.1 that the base case holds, i.e., $steps(f(0)) = 28$, so we can assert:

$$steps(f(n + 1)) = 28 + n \times 28 \tag{6.10}$$

The time complexity of `f` for these inputs inspected so far is obviously linear with respect to `n`, or in the familiar "big-oh" notation: $O(n)$. However at this point it is not clear if this relationship holds for values larger than `3`.

The inductive step required for an inductive proof of the time complexity of the algorithm, requires that the programmer convince themselves that for each subsequently larger value supplied to `f`, a fixed number of additional computational steps will be required. Here the programmer already knows from inspection of canonical trace, that there is only one `if` construct in the function `ff`, where the result of the calls to `eq` are used to choose either the `then` or `else` branch. The `then` branch has already been, seen being the base case cited above. The `else` branch occurs when the second argument to `ff` is not equal to `0`, in which case this

value is decremented, and then supplied recursively as the second argument to `ff` once more. There are no other `if` cases encountered in the canonical trace, so there are no other possible paths through the code; the traces show complete code coverage. Since the same additional steps will be required each time the second argument to `ff` is decremented, the programmer can confidently conclude that with each subsequently larger value supplied to `f`, it will require a fixed, additional number of computation steps.

Having established both the base and inductive cases, the inductive proof is complete, and the programmer can be confident that the time complexity of `f` is linear as they inferred from the examples in table 6.2. At this point, the programmer is now confident that the relationship in equation 6.10 holds, as implied by the values in table 6.1; for any given `n`, an additional 28 steps will be required for `n+1`.

Note that at no time did this proof require an inspection of or direct understanding of the source code — canonical traces are sufficient in this case to support a proof by induction, something that would otherwise not have been possible, using traces alone. It is the essential feature of canonical traces that they contain by definition all denotationally semantic information, that makes such reasoning possible, precisely because the programmer can be confident that 'everything' that is happening is visible in such traces. There are obvious limitations to this inductive proof technique as applied to canonical traces, at the very least the function of interest must be idempotent. Nevertheless, it is both somewhat surprising and useful, that at least in some cases such proofs are possible, using trace information alone. This approach, i.e., inferring important relationships from trace information which might subsequently be proven deductively, is complementary to reasoning about the source code itself. In the style of reasoning presented here, the programmer observes the actual behaviour of the program, and from that infers something about the program rather than the reverse. In one case, reasoning starts with what the programmer expects from the code, in the other it begins with the actual behaviour – closing this gap to achieve correspondence between the programmer's intention as expressed in source code and the resulting behaviour is arguably at the core of computer programming, and it is equally useful to approach this problem from either direction.

6.4.2.1 'timecomp' Trace Analysis Tool

> *A measure of algorithmic time complexity can be derived by structural induction over trace structures.*

In order to compute a measure of the time complexity, the algorithm used here counts the number of trace structure elements. Again, this is a simple example of fold-expressible, structural induction. However, the implementation below is optimised to simply divide the number of trace event lines by two and thereby avoids an unnecessary parsing step. This relies on the fact that each trace structure element has a corresponding 'begin' and 'end' trace event, i.e., there are two trace events recorded for each trace structure.

```
main = do xs <- getContents
          putStrLn $ show $ (length $ lines xs) `div` 2
```

6.4.3 Space Complexity

The space complexity of the program is analysed using proof by induction.

Given the simple nature of the example language – which has no heap-allocated space, just a call stack, and where all functions have precisely one argument – a measure of stack space usage can be derived from the canonical trace by counting the depth of nesting of `apply` semantic steps. As for time complexity, other measures of space complexity could also be computed, but are not considered here. NB: the calculation of space complexity here requires access to information about the compositional structure of the events in the trace, i.e., the depth of function call application nesting – without this nothing is known about the nested relationship of function/procedure calls. Therefore a trace monoid cannot support this kind of calculation.

n	space (f n)	space (f n) - space (f (n - 1))
0	4	
1	6	2
2	7	1
3	8	1

Table 6.2: Space Complexity of “f”

Table 6.2 tabulates the maximum stack depth for various values of n. Now we can infer, for $n > 1$, that:

$$space(f(n+1)) = 5 + n \times 1 \tag{6.11}$$

i.e., that the space complexity is linear, or $O(n)$. As before, we can go further – given that the canonical traces provide complete coverage of the `if` branch – and prove via induction in exactly the same was as was done for time complexity. An inspection of the canonical trace shows there is no other `if` branch in the source code which we have not already seen executed, which could result in nesting of function applications differing from the pattern identified above, given any value of `n`. Once again we do not need to refer to the source here or collect any further information – the canonical trace is sufficient in this case to show us that there are no branches that have not already been executed.

6.4.3.1 ‘spacecomp’ Trace Analysis Tool

A measure of algorithmic space complexity can be derived by structural induction over trace structures.

Once again the tool to compute space complexity was in essence very simple and used the structural information in the trace provided by the nesting of function application events:

```
main = do t <- readTrace
          let depths = map fst
                     $ indent 1
                     $ keepOnly "apply" t
          print depths
          print $ maximum depths
```

In this case, it is simplest to filter the trace events so that only the `apply` events are kept – all other details of the trace structure were discarded.

```
keepOnly name = filter $ \ (Event _ _ _ name' _) -> name == name'
```

An 'indent' is computed for each event, i.e., the level of nesting, based on the nesting of `apply` begin and end events.

```
indent :: Int -> [Event] -> [(Int, Event)]
indent i []     = []
indent i (e:es) =
  let Event _ _ et _ _ = e
  in case et of
          Bgn -> (i, e)   : indent (i+1) es
          End -> (i-1, e) : indent (i-1) es
```

The maximum depth encountered is the metric used in this case, a reasonable proxy for memory usage bearing in mind that each `apply` event corresponds to a function application, which in turn will require storage for its argument (e.g., on the stack).

Chapter 7

Conclusions

Thus category theory has delivered another useful result in software engineering as further evidence of its general applicability to this field, thereby reinforcing its usefulness for modeling, design and implementation, and suggesting this toolset should become a standard part of the software engineering curriculum.

7.1 Summary of Results

Category theory has delivered another useful result for the engineering of software by identifying, justifying and describing precisely a unique, abstract alternative to the existing notion of the trace monoid, that turns out to fit well with the requirements for practical tracing systems, and provides an elegant, integrated solution to the software engineering problems inherent in tracing systems of ad hoc design.

Having developed the concept of denotational trace in chapter 5 and illustrated how this works in practice with an example language, program, traces and associated formal reasoning and tools in chapter 6, we can ask how well this denotational tracing fits with the requirements for practical tracing systems identified in chapter 3, summarised in section 3.3 (and the corresponding deficiencies of the existing notion of the trace monoid noted in section 4.5).

The use of category theory in developing the basis for denotational trace has provided particular benefits here. Not only does category theory provide a convenient descriptive framework for discussing tracing independently of specific language semantics and syntax, but more importantly it provides a solid theoretical justification for the existence and usefulness of denotational trace. Denotational trace is not just an arbitrary solution to the practical problems of execution tracing. On the contrary, it is justified by the category-theoretic concept of the duality between operational (final coalgebra) and denotational (initial algebra) semantics. As such it provides a unique, dual alternative to the trace monoid (having an operational semantic basis). It is to be expected, given that these dual models of trace are necessarily complementary, that each

will provide solutions to problems for which the other is poorly suited. In section 4.5 it was shown that in general, the trace monoid lacks the necessary structure to provide source-oriented traces with complex, nested organisation, necessary for the practical activities found in the case studies and examples in chapter 3. We might then expect, that the dual concept of trace, using the dual of the semantic basis, would be suitable for source-oriented tracing. This is indeed what we find – a completely unsurprising result when it is considered that this dual semantic basis is denotational, that by definition is explicitly source-oriented. Similarly, we might expect that many or all of the concepts on the left-hand side of the Duality Quick Reference (i.e., the 'denotational' side of table 5.2) might apply somehow to denotational traces, in particular mathematical induction might be useful for reasoning about traces, and structural induction might be useful for constructing trace analysis tools. Again, we find that this is the case in the examples of tools and reasoning found in chapter 6.

In fact we have found that the denotational (vs. operational) theory of tracing introduced in chapter 5, offers a satisfactory, integrated solution to all of the practical problems identified in section 3.5 arising from the use of ad hoc bases for existing tracing systems, for both the designers and users of tracing systems:

- The concept of canonical denotational traces is introduced in section 5.2.3 to provide a language designer or implementer with the complete set of semantic information available for tracing, including how it is structured and precisely how it relates to a (denotational) semantic specification for the language. The question as to what traces may contain, and how they should be structured, is answered. This in turn provides the users of such traces a clear and precise definition of how they relate to the semantics of a program of interest. A practical example of a simple language extended with denotational trace is provided above, to show the completely straightforward relationship between denotational semantics and canonical traces.

- An example of a program in the simple language and the corresponding traces illustrate how denotational trace can support a complex, high-value software engineering tasks such as the recovery of specification from trace – something of considerable usefulness to an engineer engaged in the maintenance of poorly or incorrectly documented code.

- Given this mathematical basis for practical tracing, the user of a denotational tracing system has access to mathematical tools for reasoning such as proof by induction. In section 6.4.2 and section 6.4.3 examples are given of back-of-the-envelope proofs from traces using mathematical induction – for the space and time complexity of a program – techniques of great usefulness to anyone needing to analyse or debug the performance of real programs.

- Formal, mathematical techniques offer the potential for easy automation, thereby providing better tool support for trace users. Several examples of tools for the analysis of traces

have been provided, being both simple to implement and useful. Structural induction was found to be a useful implementation technique for the implementation of trace analysis tools.

Courtesy of the categorical notion of duality, denotational trace provides a valid, justified alternative to the trace monoid, but unlike the latter provides a good fit with the source-oriented requirement for practical execution tracing as it is found in chapter 3. This in turn addresses the fundamental, practical problems for designers and users outlined in section 3.5, due to the use of ad hoc bases for existing trace systems.

7.2 Significance

Category provides a powerful and appropriate mathematical basis for software engineering.

The specific result derived here has broader implications for the relevance of category theory to practical software engineering. The aim of this thesis was to address a theoretical deficiency — the lack of theoretical basis for execution tracing as it is found in practice. Category theory has been used to solve a long standing, real-world, practical software engineering problem. Taken alongside the growing body of literature showing the relevance of category theory to software engineering (see section 2.4) this thesis further confirms how category theory can enable and illuminate software engineering. In abstract, general terms, category has been used at three levels to provide an integrated solution to a practical, software engineering problem: for modeling, design and implementation.

7.2.1 Modeling of the Problem

The flexible and powerful notions of the category and categorical notions of sameness, provide an effective basis for modeling software abstractions at many levels.

The practice of computer programming consists in large part of managing complexity through a suitable choice of abstractions. The flexible categorical notions of sameness, such as isomorphism and the equivalence of categories, are useful because the give precise description to the intuitive notion of 'abstraction'.

Duality could never have been leveraged in this thesis without first resting on a categorical characterisation of the problem space. It seems likely that the method of analysis used here — i.e., to explore the mathematical foundations of a practical problem space, and then use duality to justify an alternative theoretical basis — could be equally useful when focused on other practical problems unrelated to execution tracing. There is undoubtedly much to explore in the potential for category theory to provide solid mathematical underpinnings for software

engineering. Key to further exploring this useful, engineering relationship between theory and practice, will be introducing key concepts from category theory to software engineers, motivated by real solutions to their practical problems.

7.2.2 Design of the Solution

Tools such as duality can be used for the design of justified, valid solutions.

In this project, category theory has been found to provide a uniquely useful set of conceptual tools, not just for discussing software design patterns in abstract yet precise terms, but also with meta-level design capabilities making to possible to cut the Gordian knot of the ad hoc bases for existing practical tracing systems. Here category theory provided an especially powerful and convenient tool in the notion of duality. The task of designing an alternative theory of tracing was greatly simplified, to the point of being derived 'for free'. Not only does duality provide an uniquely well justified, single alternative, but furthermore, this alternative is guaranteed to be well formed in the categorical sense — validity does not need to be proved as would be the case for some other, arbitrary alternative that might have been proposed. Thus, a complex, open-ended design space, i.e., the development of a new, effective, valid and justified theoretical basis for tracing, suitable for execution tracing as it is used in practice, has been collapsed into a single, uniquely well justified answer. Without the categorical notion of the duality between initial algebras and finite coalgebras, this elegant and appropriate result would not have been possible.

7.2.3 Implementation of an Example

Category theory provides a design pattern catalogue for program implementation.

Having used category theory firstly to model the problem of tracing and then to design the denotational trace, it's usefulness further extended to implementing an example of a denotational trace, and the tools used to inspect and analyse the resultant traces. The practice of engineering stands on the use of clearly defined mathematical abstractions. Category theory provides just such a set of robust, well-understood engineering abstractions for practical programming, i.e., a mathematically-grounded design pattern catalogue. The example denotational tracer in chapter 6 was implemented using categorical constructs.

7.3 Implications and Future Directions

There is interesting work remaining to be done in exploring how operationally and denotationally based traces are complementary, based on the duality and correspondence between operational and denotational semantics, and more broadly in continuing to explore the uses of category theory in software engineering.

7.3.1 Further Exploration of Tracing via Category Theory

> *There is interesting work remaining to be done in exploring how operationally and denotationally based traces are complementary, based on the duality and correspondence between operational and denotational semantics.*

It is clear that tracing has a dual nature. Just as the semantics of languages and programs can be viewed from the complementary, dual perspectives of operational and denotational semantics, so too there are corresponding notions of trace, being the trace monoid and denotational trace respectively. Neither model of tracing is more 'correct' or 'justified' in purely theoretical terms, although there seems to be a clear alignment between the routine practical needs of source-oriented tracing and the source-oriented structure of denotational traces. Nevertheless, these dual perspectives on trace are necessarily complementary. Existing results in computer science which have established full abstraction between operational and denotational semantics under various conditions [Ong95], suggest that it may be possible to both prove the correctness of and/or automatically derive, an (operational) implementation of a language including denotational tracing, given a (denotational) specification of semantics. Further research is required to explore the full potential of these ideas, and their practical implications.

7.3.2 Categories as Domain Specific Languages

> *Categories provide an effective template for domain specific languages.*

It has already been observed in section 2.2 that all of mathematics can be viewed as various categorical domain specific languages. The associative composition provided by the category is also a unbiquitous pattern in computer programming, and thus provides an effective template for a wide range of domain specific languages. A software engineer engaged in developing a domain specific language would be well advised to use a categorical approach to the design.

7.3.3 Categorical Programming Language Design

> *While categorical ideas can be ported into any programming language, the usefulness of category theory to software engineering suggest that programming languages should explicitly support categorical notions, to derive the full benefits of automated tool support.*

The usefulness of category theory as a basis for domain specific languages extends more broadly to the design of general-purpose programming languages. Not only does category theory provide a template for the design of languages themselves, but in addition, programming languages can directly support categorical notions. While categorical concepts can in principle be implemented in a programming language which lacks explicit support for these, the disadvantage is that no

automated tool support is available. In this project the Haskell language was used, which allowed direct expression of some categorical concepts in the type system. To the extent that this was possible, it had the significant advantage that the type system provided automated checks that the implementation was correct. Hagino has developed a research programming language explicitly based on category theory [Hag87], but categorical concepts are yet to be applied to mainsteam languages.

7.3.4 Category Theory in Software Engineering Education

> *Given the useful and appropriate tools that category theory brings to software engineering, and the promise it offers as a mathematical basis for robust software engineering abstractions, category theory should become part of the software engineering curriculum.*

Category theory has had a considerable impact on theoretical computer science in recent decades, to the extent that several textbooks presenting category theory to the computer scientist already exist (see chapter 1). The impact of category theory on practical programming and software engineering is newer, and textbooks are yet to be written with the mainstream, pragmatic programmer in mind. Section 2.4 summarises many applications to which category theory has been applied in computing. Given the diverse, useful applications of category theory in software engineering, Zheng, Shi and Xue have investigated the introduction of category theory into the software engineering curriculum [ZSX07]. As they observe, category theory provides a promising, mathematical basis for software engineering.

Bibliography

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, types and structures - an introduction to category theory for the working computer scientist.* Foundations of computing. MIT Press, 1991.

[APE09] Tristan O.R. Allwood, Simon Peyton Jones, and Susan Eisenbach. Finding the needle: Stack traces for ghc. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 129–140, New York, NY, USA, 2009. ACM.

[Aug98] Lex Augusteijn. Sorting morphisms. In *3rd International Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 1–27. Springer-Verlag, 1998.

[Bar81] Hendrik Pieter Barendregt. *The lambda calculus : Its syntax and semantics.* Elsevier Science B. V., Amsterdam, Holland, 1981.

[BB13] Paul Bailes and Leighton Brough. Making sense of recursion patterns. In *FormSERA 2012: Formal Methods in Software Engineering: Rigorous and Agile Approaches*, pages 16–22, 2013.

[BBK12] Paul Bailes, Leighton Brough, and Colin Kemp. Higher-order catamorphisms as bases for program structuring and design recovery. In *IASTED SE 2013: The 12th IASTED International Conference on Software Engineering*, pages 775–782, 2012.

[BH93] Thomas Ball and Suzan Horowitz. Slicing programs with arbitrary control flow. In Peter A. Fritzson, editor, *AADEBUG '93: Proceedings of the 1st Workshop on Automated and Algorithmic Debugging*, volume 749 of *LNCS*, pages 206–222, 1993.

[BJJM99] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In *3rd International Summer School on Advanced Functional Programming*, pages 28–115. Springer-Verlag, 1999.

[BJW+01] Rhodes Brown, John Jorgensen, Qin Wang, Karel Driesen, Laurie Hendren, and Clark Verbrugge. STOOP: The sable toolkit for object-oriented profiling. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications Conference*, volume 10, pages 1–2. ACM, 2001.

[BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[CDJ01] Robert M. Colomb, C.N.G. Dampney, and Michael Johnson. Category-theoretic fibration as an abstraction mechanism in information systems. *Acta Informatica*, 38(1):1–44, 2001.

[Cop04] B. Jack. Copeland. *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma.* Oxford University Press, 2004.

[CRW00] O. Chitil, C. Runciman, and M. Wallace. Tracing and debugging of lazy functional programs — a comparative evaluation of three systems. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 47–62. RWTH Aachen, 2000.

[CRW03] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for tracing. *Implementation of Functional Languages*, 2670:165–181, 2003.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX Annual Technical Conference*, pages 15–28, 2004.

[Cur94] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer*, pages 267–278, 1994.

[DF02] Andrew Dingwall-Smith and Anthony Finkelstein. From requirements to monitors by way of aspects. In *Early Aspects 2002: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, 2002.

[DGW06] S. Ducasse, T. Girba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, volume 00, pages 37–46. IEEE, 2006.

[DHK+92] P. Dauphin, F. Hartleb, M. Kienow, V. Mertsiotakis, and A. Quick. PEPP: Performance Evaluation of Parallel Programs — User's Guide – version 3.1. Technical Report 5/92, Universitat Erlangen–Nürnberg, April 1992.

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, 1976.

[DK99] Scott A. Deloach and Mieczyslaw Kokar. Category theory approach to fusion of wavelet-based features. In *Proceedings of the Second International Conference on Information Fusion*, pages 117–124, 1999.

[DM12] Zinovy Diskin and T. S. E. Maibaum. Category theory and model-driven engineering: From formal semantics to design patterns and beyond. In Ulrike Golas and Thomas Soboll, editors, *ACCAT*, volume 93 of *EPTCS*, pages 1–21, 2012.

[DR95] V. Diekert and G. Rosenberg. *The book of traces.* World Scientific Publishing, Singapore, 1995.

[Duc92] Mireille Ducassé. A general trace query mechanism based on Prolog. In Martin Wirsing Maurice Bruynooghe, editor, *Proceedings of PLILP'92 (4th International Symposium on Programming Language Implementation and Logic Programming)*, volume 631 of *LNCS*, pages 400–414. Springer-Verlag, August 1992.

[Duc93] Mireille Ducassé. A pragmatic survey of automated debugging. In Peter A. Fritzson, editor, *AADEBUG '93: Proceedings of the 1st Workshop on Automated and Algorithmic Debugging*, volume 749 of *LNCS*, pages 1–15, 1993.

[Duc98] Mireille Ducassé. Coca: A debugger for C based on fine grained control flow and data events. Technical Report 1202, IRISA/ISNA, September 1998.

[Duc99] Mireille Ducassé. OPIUM: An extendable trace analyser for Prolog. *The Journal of Logic Programming, Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41:177–223, 1999.

[EGW98] Hartmut Ehrig, Martin Große-Rhode, and Uwe Wolter. Applications of category theory to the area of algebraic specification in computer science. *Applied Categorical Structures*, 6(1):1–35, 1998.

[EKKL90] S. Eggers, D. Keppel, E. Koldinger, and H. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–47, Boulder, CO, United States, 1990. ACM.

[Ezu95] S. Ezust. TANGO: The Trace ANalysis GeneratOr. Master's thesis, Département d'informatique et de recherche opéationnelle, Université de Montréal, Canada, 1995.

[FHL98] M. Frumkin, R. Hood, and L. Lopez. Trace-driven debugging of message passing programs. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, pages 753–762, Washington, DC, USA, 1998. IEEE Computer Society.

[FOGG05] Michael Fischer, Johann Oberleitner, Harald Gall, and Thomas Gschwind. System evolution tracking through execution trace analysis. In *IWPC '05: Proceedings*

of the 13th International Workshop on Program Comprehension, pages 237–246, Washington, DC, USA, 2005. IEEE Computer Society.

[GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London, UK, UK, 1993. Springer-Verlag.

[GM01] J. Gargiulo and S. Mancoridis. Gadget: A tool for extracting the dynamic structure of java programs. In *Proceedings of SEKE'01 (International Conference on Software Engineering and Knowledge Engineering)*, pages 244–251, 2001.

[GNU14] GNU. GDB: The GNU project debugger. `http://www.gnu.org/software/gdb/`, 2014. Accessed: 2014-03-18.

[GO03] Thomas Gschwind and Johann Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 259–268. IEEE, 2003.

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 385–402, New York, NY, USA, 2005. ACM Press.

[Gog89] Joseph A. Goguen. A categorical manifesto. Technical Report PRG-72, Oxford University, 1989.

[Gon12a] Gabriel Gonzalez. The category design pattern. `http://www.haskellforall.com/2012/08/the-category-design-pattern.html`, 2012. Accessed: 2013-07-02.

[Gon12b] Gabriel Gonzalez. The functor design pattern. `http://www.haskellforall.com/2012/09/the-functor-design-pattern.html`, 2012. Accessed: 2013-07-02.

[GTW78] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[Har02] Bruno Harbulot. An investigation of aspect-oriented programming. Technical report, The University of Manchester, UK, 2002.

[Has14] Haskell Wiki. Monad tutorials timeline. `http://www.haskell.org/haskellwiki/Monad_tutorials_timeline`, 2014. Accessed: 2014-04-09.

[HHJ13] Jennifer Hackett, Graham Hutton, and Mauro Jaskelioff. The Under Performing Unfold: A New Approach to Optimising Corecursive Programs. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, 2013.

[Hil93] Gillian Hill. Category theory for the configuration of complex systems. *Algebraic Methodology and Software Technology*, 1993.

[Hin12] Ralf Hinze. Generic programming with adjunctions. In Jeremy Gibbons, editor, *Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 47–129. Springer Berlin Heidelberg, 2012.

[HJS07] Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4:11):1–36, 2007.

[HL04] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *CASCON '04: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[HMG+97] Jeffrey K. Hollingsworth, Barton P. Miller, M. J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 201–213. IEEE Computer Society, 1997.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - International edition (2. ed)*. Addison-Wesley, 2003.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communcations of the ACM*, 21(8):666–677, August 1978.

[Hua11] Jinzi Huang. *Modeling Multi-Agent Systems with Category Theory*. PhD thesis, Concordia University, 2011.

[Hut98] Graham Hutton. Fold and unfold for program semantics. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 280–288. ACM Press, 1998.

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[HW00] Michael Healy and Keith Williamson. Applying category theory to derive engineering software from encoded knowledge. In Teodor Rus, editor, *Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 484–498. Springer Berlin Heidelberg, 2000.

[JD01] Michael Johnson and C. N. G. Dampney. On category theory as a (meta) ontology for information systems research. In *Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001*, FOIS '01, pages 59–69, New York, NY, USA, 2001. ACM.

[JD02] Erwan Jahier and Mireille Ducassé. Generic program monitoring by trace analysis. *Theory and Practice of Logic Programming*, 2(4):611–643, 2002.

[JDR00] Erwan Jahier, Mireille Ducassé, and Olivier Ridoux. Specifying Prolog trace models with a continuation semantics. In Kung-Kiu Lau, editor, *Logic Based Program Synthesis and Transformation, 10th International Workshop*, volume 2042 of *LNCS*, pages 165–182. Springer, 2000.

[JMP05] Mark W. Johnson, Peter McBurney, and Simon Parsons. A mathematical model of dialog. *Electron. Notes Theor. Comput. Sci.*, 141(5):33–48, December 2005.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[JS94] Dean F. Jerding and John T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Georgia Institute of Technology, Atlanta, GA, USA, July 1994.

[JSB97] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *International Conference on Software Engineering*, pages 360–370, 1997.

[JZTB98] Clinton L. Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A lightweight architecture for program execution monitoring. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 67–74, 1998.

[KHC91] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: a formal framework for specifying, implementing, and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 338–352, 1991.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[Kis92] Amir Shai Kishon. *Theory and art of semantics-directed program execution monitoring.* PhD thesis, Yale University, New Haven, CT, USA, 1992.

[Klo92] J. W. Klop. Term rewriting systems. *Handbook of Logic in Computer Science*, 2:1–116, 1992.

[KLV00] J. Kort, R. Lämmel, and J. Visser. Functional transformation systems. In *WFLP 2000: Proceedings of the 9th International Workshop on Functional and Logic Programming*, pages 154–168, 2000.

[KOK10] Noorulain Khurshid, Olga Ormandjieva, and Stan Klasa. Towards a tool support for specifying complex software systems by categorical modeling language. In Roger Lee, Olga Ormandjieva, Alain Abran, and Constantinos Constantinides, editors, *Software Engineering Research, Management and Applications 2010*, volume 296 of *Studies in Computational Intelligence*, pages 133–149. Springer Berlin Heidelberg, 2010.

[KQS92] Ranier Klar, Andreas Quick, and Franz Sötz. Tools for a model–driven instrumentation for monitoring. In G. Balbo, editor, *Proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 165–180. Elsevier Science Publisher B.V., 1992.

[Kue95] Geoffrey H. Kuenning. Kitrace: Precise interactive measurement of operating system kernels. *Software - Practice and Experience*, 25(1):1–21, 1995.

[LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software - Practice Experience*, 24(2):197–218, 1994.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[LN97] D.B. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, 1997.

[Mac98] Saunders Mac Lane. *Categories for the working mathematician, 2nd ed.* Springer-Verlag, New York, USA, 1998.

[Mar10] Simon Marlow. Haskell 2010 language report. `http://haskell.org/definition/haskell2010.pdf`, 2010.

[Maz08] Barry Mazur. When is one thing equal to some other thing? In *Proof and Other Dilemmas. 1st ed.*, pages 221–242. Mathematical Association of America, Washington DC, 2008.

[MB11] Erik Meijer and Gavin Bierman. A co-relational model of data for large shared data banks. *Queue*, 9(3):30:30–30:48, March 2011.

[MC02] Johan Moe and David A. Carr. Using execution trace data to improve distributed systems. *Software - Practice and Experience*, 32(9):889–906, 2002.

[MCC+95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, London, UK, 1991. Springer-Verlag.

[MG86] J Meseguer and J A Goguen. Initiality, induction, and computability. In Maurice Nivat and John C Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, New York, NY, USA, 1986.

[Mic14] Microsoft Developer Network. Use dump files to debug app crashes and hangs in visual studio. `http://msdn.microsoft.com/en-us/library/d5zhxt22.aspx`, 2014. Accessed: 2014-03-18.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

[Moe11] Geoff Moes. Monoid for the masses. `http://www.elegantcoding.com/2011/05/monoid-for-masses.html`, 2011. Accessed: 2013-07-02.

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh Univ., 1989.

[NB95] Lee Naish and Timothy Barbour. A Declarative Debugger for a Logical-Functional Language. In Graham Forsyth and Moonis Ali, editors, *Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems — Invited and Additional Papers*, pages 91–99, Melbourne, June 1995. DSTO General Document 5.

[NES05] Nan Niu, Steve Easterbrook, and Mehrdad Sabetzadeh. A category-theoretic approach to syntactic software merging. In *ICSM 05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM05)*, pages 197–206. IEEE Computer Society, 2005.

[NF93] Henrik Nilsson and Peter Fritzson. Lazy algorithmic debugging: Ideas for practical implementation. In Peter A. Fritzson, editor, *AADEBUG '93: Proceedings of the 1st Workshop on Automated and Algorithmic Debugging*, volume 749 of *LNCS*, pages 117–134, 1993.

[NS96] Henrik Nilsson and Jan Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Technical Report S-581 83, Linkoping University, Sweden, 1996.

[OD10] Walamitien H. Oyenan and Scott A. DeLoach. Using category theory to compose multiagent organizational design models. Technical Report MACR-TR-2010-01, Kansas State University, 2010.

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[Ong95] C. H. L. Ong. Correspondence between operational and denotational semantics. *Handbook of Logic in Computer Science*, pages 269–356, 1995.

[Pey01] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *Engineering theories of software construction*, pages 47–96, 2001.

[PHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 326–337, 1993.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[Pip06] Dan Piponi. You could have invented monads! (and maybe you already have). `http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html`, 2006. Accessed: 2014-04-09.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, 1981.

[Plo04] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, pages 60–61, 2004.

[PN81] B. Plattner and J. Nievergelt. Monitoring program execution: A survey. *IEEE Computer*, pages 76–93, 1981.

[PN03] Bernard Pope and Lee Naish. Practical aspects of declarative debugging in haskell 98. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 230–240, New York, NY, USA, 2003. ACM Press.

[Pop98] Bernard Pope. Buddha: A declarative debugger for haskell. Technical report, Department of Computer Science, University of Melbourne, Australia, June 1998.

[PY93] David K. Poulsen and Pen-Chung Yew. Execution-driven tools for parallel simulation of parallel architectures and applications. *Supercomputing*, pages 860–869, 1993.

[Pyt] Python 2.7.5 Documentation. The python standard library, debugging and profiling. `http://docs.python.org/2/library/trace.html`. Accessed: 2013-07-02.

[Qui93] Andreas Quick. A new approach to behavior analysis of parallel programs based on monitoring. In G.R. Joubert, D. Trystram, and F.J. Peters, editors, *Proccedings of the International Conference on Parallel Computing*, pages 7–10, September 1993.

[RB88] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory.* Prentice Hall, 1988.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (15th International Conference on Software Maintenance)*, pages 13–22. IEEE CS Press, 1999.

[RD02] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings ICSM'02 (18th International Conference on Software Maintenance)*, pages 34–43. IEEE CS Press, 2002.

[RDW98] Tamar Richner, Stéphane Ducasse, and Roel Wuyts. Understanding object-oriented programs through declarative event analysis. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, volume 1543 of *LNCS*, pages 78–79. Springer-Verlag, 1998.

[Rei93] Steven P. Reiss. Trace-based debugging. In Peter A. Fritzson, editor, *AADEBUG '93: Proceedings of the 1st Workshop on Automated and Algorithmic Debugging*, volume 749 of *LNCS*, pages 305–315, 1993.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[RR99] Manos Renieris and Steven P. Reiss. ALMOST: Exploring program traces. In *Workshop on New Paradigms in Information Visualization and Manipulation*, pages 70–77, 1999.

[RR00] Steven P. Reiss and Manos Renieris. Generating Java trace data. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 71–77. ACM Press, 2000.

[RR01] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230, Toronto, Ontario, Canada, 2001. IEEE.

[RR03] Stephen P. Reiss and Manos Renieris. Languages for dynamic instrumentation. In *Proceedings of ICSE Workshop on Dynamic Analysis (WODA 2003)*, pages 6–9, 2003.

[RT94] Jan J. M. M. Rutten and Daniele Turi. Initial algebra and final coalgebra semantics for concurrency. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 530–582, London, UK, UK, 1994. Springer-Verlag.

[SBP10] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stacktraces help developers fix bugs? In *MSR '10: Proceedings of the 2010 International Working Conference on Mining Software Repositories*, pages 118–122. IEEE, 2010.

[Sco70] Dana Scott. Outline of a mathematical theory of computation. Technical report, Oxford University, 1970.

[SKB03] Maximilian Störzer, Jens Krinke, and Silvia Breu. Trace analysis for aspect application. In *Workshop on Analysis of Aspect-Oriented Software (AAOS)*, 2003.

[Sob08] Thomas Soboll. On the construction of transformation steps in the category of multiagent systems. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics*, volume 5144 of *Lecture Notes in Computer Science*, pages 184–190. Springer Berlin Heidelberg, 2008.

[SR97] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *Proceedings of the Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 291–308. Springer, 1997.

[Sta95] John T. Stasko. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95-03, Georgia Institute of Technology, 1995.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press Series in Computer Science, 1977.

[Str00] C. Strachey. Fundamental concepts in programming languages. In *Higher-Order Symbolic Computation*, volume 13, pages 11–49. Springer-Verlag, 2000.

[Sys98] Tarja Systä. Dynamic modeling in forward and reverse engineering of object-oriented software systems. In *Proceedings of Doctoral Symposium of 13th IEEE International Conference of Automated Software Engineering (ASE'98)*, pages 47–50, 1998.

[TFM+01] Hong-Linh Truong, Thomas Fahringer, Georg Madsen, Allen D. Malony, Hans Moritsch, and Sameer Shende. On using SCALEA for performance analysis of distributed and parallel programs. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 34–34, New York, NY, USA, 2001. ACM Press.

[TH02] M. Tikir and J. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 86–96. ACM Press, July 2002.

[TJ98] Kevin Templer and Clinton L. Jeffery. A configurable automatic instrumentation tool for ANSI C. In *Proceedings of Doctoral Symposium of 13th IEEE International Conference of Automated Software Engineering (ASE'98)*, pages 249–258, 1998.

[TM99] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. *Operating Systems Design and Implementation*, pages 117–130, 1999.

[Tot08] David Toth. Database engineering from the category theory viewpoint. In Václav Snásel, Karel Richta, and Jaroslav Pokorný, editors, *DATESO*, volume 330 of *CEUR Workshop Proceedings*, pages 37–48, 2008.

[TP97] Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Proceedings of the 12th LICS Conference*, pages 280–291. IEEE, Computer Society Press, 1997.

[TSS95] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. Integrating visualization support into distributed computing systems. In *International Conference on Distributed Computing Systems*, pages 19–26, 1995.

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, number 42 in 2. London Mathematical Society, 1936.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[Wat86] David A. Watt. Executable semantic descriptions. *Software - Practice and Experience*, 16(1):13–43, 1986.

[WCBR01] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, 2001.

[WHB01] Keith Williamson, Michael Healy, and Richard Barker. Industrial applications of software synthesis via category theorycase studies using specware. *Automated Software Engineering*, 8(1):7–30, 2001.

[WMSR00] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient mapping of software system traces to architectural views. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, pages 31–40. IBM Press, 2000.

[WS97a] Richard D. Watson and Eric Salzman. A trace browser for a lazy functional language. In *Proceedings of the Twentieth Australian Computer Science Conference*, pages 356–363, 1997.

[WS97b] Richard D. Watson and Eric Salzman. Tracing the evaluation of lazy functional languages: A model and its implementation. In *Asian Computing Science Conference*, pages 336–350, 1997.

[Yor09] Brent Yorgey. Typeclassopedia. `https://wiki.haskell.org/Typeclassopedia`, 2009. Accessed: 2015-03-20.

[ZG04] Xiangyu Zhang and Rajiv Gupta. Whole execution traces. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–116, Washington, DC, USA, 2004. IEEE Computer Society.

[ZS95] Qiang A. Zhao and John T. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95-01, Georgia Institute of Technology, 1995.

[ZSX07] Yujun Zheng, Haihe Shi, and Jinyun Xue. From mathematics to software engineering: Introducing category theory into the computer science curriculum. In Yong Shi, GeertDick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science ICCS 2007*, volume 4489 of *Lecture Notes in Computer Science*, pages 469–476. Springer Berlin Heidelberg, 2007.