Semantic Frameworks for Complexity

Douglas J. Gurr

Doctor of Philosophy

University of Edinburgh

1990

Abstract

This thesis extends denotational semantics to take account of the resource requirements of programs.

We describe the approach we have taken in modelling the resource requirements of programs, and motivate the definition of a monoid $\mathcal{M}$ of resource values. A connection is established with Moggi's categorical semantics of computations, and this connection is exploited to study complexity as a monad constructor.

A formal system, the λ_{com}-calculus, for reasoning the resource requirements of programs is developed. Operational and denotational semantics are defined for this system, and we prove a correspondence theorem.

We show that Moggi's framework is not sufficiently general to capture all the examples of interest to us. Therefore, we define a new class of models based on the idea of an external datum. We investigate the relationship between these two approaches.

The new framework is used to investigate various concepts of importance in complexity theory and the analysis of algorithms. In particular, we show how to capture the notions of input measures, upper bounds on complexity and non-exact complexity

Acknowledgements

I owe a considerable debt to my first supervisor, Gordon Plotkin, for the original suggestion of the subject of this thesis and for the unfailing insight of his comments over the past three years.

Over the past year, I have had the unique experience of having John Power as my second supervisor. It would be impossible to express adequately the debt I owe to John for his guidance and support, and he would doubtless deride any attempt to do so.

In addition to my supervisors, I have benefited considerably from conversations with Stuart Anderson, Rod Burstall, Martin Hyland, Barry Jay, Mark Jerrum, Eugenio Moggi, Valeria de Paiva and Glynn Winskel. All the diagrams in this thesis were produced using Paul Taylor's diagram macros.

My friends and colleagues at Edinburgh have made the whole PhD process much more bearable than it might otherwise have been. In particular, I would like to thank Chris, David, James, Simon and, of course and above all, Carolyn.

Finally, I would like to thank my parents and relatives for their moral and financial support over the past seven years.

This work was supported by an SERC research studentship.

Declaration.

This thesis was composed by myself and the work reported in it is my own.

Douglas J. Gurr

Table of Contents

1. Introduction **10**

1.1 Denotational semantics and complexity theory 10

1.2 Background theory 13

1.2.1 Programming language semantics 13

1.2.2 Complexity theory 15

1.3 Related work 17

1.4 Outline of the thesis 18

2. Motivation and Examples **21**

2.1 Adding complexity 21

2.1.1 Introduction 21

2.1.2 Composing programs 22

2.1.3 Examples 26

2.1.4 Datatypes 34

3. Reasoning in Time **36**

3.1 Introduction 36

3.2 Categorical semantics of computations 38

3.2.1 Introduction 38

3.2.2 Monads 39

3.2.3 Datatypes, strong monads and computational models . . . 40

3.3 The monad for complexity 42

3.3.1 Introduction 42

3.3.2 The monad for complexity 43

3.4 The λ_c-calculus 44

3.4.1 Introduction 44

3.4.2 A language for computation 45

3.4.3 Interpretation of the language 48

3.4.4 The λ_c-calculus 50

3.5 Reasoning about resource 55

3.5.1 Introduction 55

3.5.2 The λ_{com}-calculus 56

3.5.3 Interpretation of the λ_{com}-calculus in complexity models . . 68

3.6 Relating the λ_{com}-calculus to other systems 73

3.6.1 Introduction 73

3.6.2 The relationship between the λ_{com}-calculus and the λ_c-calculus 73

3.6.3 An operational semantics 77

3.6.4 The λ_{com}-calculus and the operational semantics 80

4. Monad Constructors **82**

4.1 Introduction 82

4.2 The monad constructor for complexity 84

4.2.1 Introduction 84

4.2.2 A category of monads 84

4.2.3 The monad constructor $(-)_M$ 86

4.2.4 Properties of $(-)_M$ 88

4.3 Limits in SMon(C) 95

4.3.1 Introduction 95

4.3.2 Limits 96

4.3.3 Equalizers and pullbacks 102

4.3.4 Products as parallel composition 102

4.4 Strong monad constructors 103

4.4.1 Introduction 103

4.4.2 Strong monad constructors 104

4.4.3 The strong monad constructor for complexity 105

5. Internal and External Data **106**

5.1 Introduction 106

5.2 Internal complexity categories 109

5.2.1 Introduction 109

5.2.2 Internal data 109

5.3 External complexity categories 112

5.3.1 Introduction 112

5.3.2 External data 112

5.4 Relationship between **Int** and **Ext** 121

5.4.1 Introduction 121

5.4.2 The main theorem 121

5.4.3 Internal and external complexity categories 124

6. Order and Disorder **127**

6.1 Introduction 127

6.2 Ordered and lax structures 128

6.2.1 Introduction 128

6.2.2 Enriched categories 128

6.2.3 Ordered monoids 133

6.3 Ordered data 136

6.3.1 Introduction 136

6.3.2 Internal ordered data 136

6.3.3 External ordered data 138

6.3.4 Relationship between the internal and external 143

7. Input Measures **147**

7.1 Introduction 147

7.2 Measures 149

7.2.1 Introduction 149

7.2.2 Measures and analyses 150

7.2.3 The model C_S^M 153

7.2.4 Conditions for C_S^M to be a model 154

7.3 A category based on measures 159

7.3.1 Introduction 159

7.3.2 Lax measures 159

7.4 Equivalences 162

7.4.1 Introduction 162

7.4.2 Congruences 162

7.4.3 M-Equivalences 163

7.4.4 Classification results 166

7.4.5 From M-equivalences to congruences 170

8. Non-Exact Complexity **175**

8.1 Introduction 175

8.2 Non-exact data 177

8.2.1 Introduction 177

8.2.2 Non-exact data 177

8.2.3 Relationship between exact and non-exact 180

8.3 Non-exact models 182

8.3.1 Introduction 182

8.3.2 Non-exact measures 185

8.3.3 Constructing a non-exact model 189

8.3.4 Universal property 191

8.3.5 Worst case order of magnitude complexity 197

9. Conclusions and Future Work **199**

9.1 Conclusions 199

9.2 Unsolved Problems 200

9.3 Future Work 201

9.3.1 The complexity of higher order functions 201

9.3.2 Timing concurrent systems 202

9.3.3 A new categorical semantics of computations 204

9.3.4 Logics for complexity 205

A. Proofs from chapter 4 **207**

A.1 Proof of lemma 4.2.9 207

A.2 Proof of lemma 4.2.17 219

A.3 Proof of lemma 4.2.22 221

A.4 Proof of proposition 4.3.11 227

A.5 Proof of lemma 4.4.5 229

A.6 Proof of proposition 4.4.6 230

Bibliogrpahy **234**

Index **239**

Chapter 1

Introduction

1.1 Denotational semantics and complexity theory

A fundamental notion in computer science is that of equivalence between programs. In general, many different programs will solve any given problem and a notion of program equivalence is crucial in making comparisons between them. A large part of theoretical computer science, and in particular the study of language semantics, is motivated by the need to define suitable notions of equivalence between programs.

The paradigm of equivalence between programs is behavioural equivalence. A subclass of the terms of the programming language is defined to be programs and a notion of observable behaviour is defined, often by giving an operational semantics for the language. Two programs are then considered equivalent if they have the same observable behaviour in all program contexts. There are many different notions of observable behaviour, for example, the input-output behaviour of a Pascal program or the non-silent events in Milner's calculus for communicating systems [Mil89]. It is clear that the definition of observable behaviour will determine the notion of program equivalence.

The direct study of equivalence between terms of a programming language is often difficult. The difficulties are essentially those of any formal syntactic system

and are clearly evidenced if one attempts to read directly a piece of program code written in a language such as APL. The theory of denotational semantics was developed in the late 1960's by Scott and Strachey [Sto77] to assist in the analysis of programming languages and in particular in the study of notions of program equivalence.

The approach is to give a model which is intended to reflect the chosen notion of behaviour. Typically, a model will be a mathematical structure such as a category. An interpretation of the terms of the programming language as elements of the model or "denotational semantics" is given. The idea is to consider programs to be equivalent if they are interpreted by the same element of the model. In this way, equivalence between terms is reduced to equality in a suitable mathematical structure. Clearly, for a model to be appropriate, it must capture our notion of behaviour. That is, terms should be identified in the model precisely when we consider them to be behaviourally equivalent. When the notion of behaviour is formalised by an operational semantics, the task of finding such an appropriate model is known as the full abstraction problem [Plo81], [Stou88].

There are a number of applications of these ideas. A semantics for a programming language can be used to assist in checking the correctness of an implementation of the language. In specification theory [BuGo81], [EhMa85] semantic techniques are used to prove the correctness of a particular program against some predetermined specification of a problem. In each of these areas, there is much interest in developing proof systems for the denotational models so that correctness proofs can be automated.

The study of denotational semantics also has applications in the area of programming language design. Studying a model for a programming language can enable the designer to improve the language. For example, a language that is mathematically elegant is easier to reason about. The programming language Standard ML [HMT87] is a fine example of the value of this approach.

An important notion in computer science which has been largely ignored in existing work on semantics is that of resource. Examples of the resource requirements of a program are the time it takes to execute and the amount of memory it

uses. Resource requirements of programs have traditionally been studied in complexity theory and in the analysis of algorithms. Considerations of resource are clearly relevant to the aims of semantics. However, there has been relatively little interaction between semantics and these areas of computer science.

The central aim of this thesis is to study a notion of behaviour in which the resource requirements of a program are considered. Thus, given an existing notion of behavioural equivalence, we wish to consider two programs to be equivalent if they are equivalent in this existing notion and have the same resource requirements.

In this thesis, we apply the techniques of denotational semantics to study this new notion of behaviour. That is, we study models which reflect this notion of behaviour and the interpretation of programming languages in such models and we develop systems for reasoning about these models.

We have three main applications of this work in mind. The first is in the formal specification and verification of programs. At present, much work is required to give a proof of program correctness. For little extra effort, the techniques we develop allow a correctness proof to provide information about the resource requirements of a system as well. An example of a situation where such an analysis would be useful is the verification of part of the control system of a "fly-by-wire" aircraft, such as the European Airbus. It may well be important to know that the part will perform its computation sufficiently rapidly to respond to some particular circumstance, such as the failure of an engine on take-off or landing.

The second application is to provide software tools which can perform an informal analysis of the resource requirements of a program. An example would be a form of compiler which, when given a program, would immediately calculate how long the program would take to evaluate. Alas, it follows from the undecidability of the halting problem that such a program is impossible. However, we indicate how the techniques we develop can be applied to this task. We approach the problems in two ways. Firstly, it is possible to develop a system which, given a program, provides an expression for the exact resource requirements of the program. The catch, of course, is the resource required to evaluate this expression. Secondly, we

could develop a system which produces approximate answers. One can hope to obtain a system which gives a reasonable estimate of the resource requirements.

Finally, we hope to provide some insight into the complexity theory of functional languages. One of the main drawbacks of complexity theory is that it is not directly applicable to languages with higher order functions, such as ML, or even to languages which allow procedures to be passed as parameters, such as Pascal or C. Our results are applicable to both the exact and non-exact complexity analysis of such languages.

The remainder of this chapter contains a summary of the basic ideas of denotational semantics and the basic ideas and definitions of complexity theory and the analysis of algorithms. We assume a knowledge of the basic definitions of category theory such as category, functor, natural transformation and adjunction. Other concepts are introduced as needed in the main text. The standard introductory text on category theory remains Mac Lane's book [Mac71]. A more elementary introduction which emphasises the applications to computer science is Burstall and Rydeheards's book [BuRy88]. Details about denotational semantics may be found in [Sto77], about the analysis of algorithms in [AHU75] and about complexity theory in [GaJo79]. We review the existing work in this area and in section 1.4 we summarise the contents of this thesis.

1.2 Background theory

We give a brief summary of the basic ideas of programming language semantics and of complexity theory and the analysis of algorithms.

1.2.1 Programming language semantics

The notion of a formal language is central to the study of mathematical logic. A special class of formal languages, for specifying mathematical operations, is the

programming languages. The interest in these languages is mainly owing to their implementation on the electronic digital computer.

As with any formal language, semantics are often given to provide an interpretation of the meaning of the syntax. There are three flavours of programming language semantics, operational, axiomatic and denotational.

In the operational approach, a subclass of the terms is designated as values and an evaluation mechanism is given. This is done either by giving an evaluation of the language on some abstract machine, such as the SECD machine [Lan64], [Sto77] or by giving a relational style partial evaluation function from closed terms to values [Plo81], as in the definition of the language Standard ML.

In the axiomatic approach [Hoa69], the language is treated directly as a logic by associating axioms with each statement of the language. This logic is then used to prove properties of programs written in the language.

In the denotational approach, the terms of the programming language are interpreted as elements of a suitable mathematical structure. One can then study and reason about the programming language by studying and reasoning about the model. It is the denotational approach which mostly concerns us here.

The mathematical structures used in denotational semantics are known as domains. A number of properties are required of a mathematical structure in order for it to be a suitable model. For example, a model for the untyped lambda calculus should be isomorphic to the space of endofunctions on it. The first person to solve this problem was Dana Scott [Sco76] who used the idea of partially ordered sets of information to motivate the study of the category **CPO** of complete partial orders and monotone continuous functions. Subsequently, many other categories with structure have been developed and studied as domains. The standard reference on domain theory is Gordon Plotkin's ever expanding but, as yet, still unpublished lecture notes [Plo87].

Denotational semantics has been successful in giving a formal meaning to a wide class of programming languages and in proving properties of languages. A problem of denotational semantics is the fact that it ignores many important

computational features of languages, such as complexity. In this thesis we attempt to address one aspect of this problem.

1.2.2 Complexity theory

The notion of resource is central to the study of dynamic aspects of programs and programming languages. The most important example of resource is time. However, there is considerable interest in other examples of resource such as memory or the number of times a particular operation is used.

In different situations, one is interested in different degrees of precision in the analysis of the resource requirements of programs. The difference corresponds to the difference between a problem, an algorithm and an implementation of an algorithm. Informally, a problem is any task that may be solved using a computer. Examples are sorting a list of integers or finding a minimal spanning tree in a graph. An algorithm is difficult to define precisely. It may best be described as a sequence of steps giving a procedure to solve a particular problem. Some examples are the quicksort or the mergesort algorithms for solving the sorting problem, the greedy algorithm for solving the minimal spanning tree problem or Euclid's algorithm for finding a greatest common divisor. An implementation of an algorithm is a specific computer program which solves a given problem by following the steps of the algorithmic procedure.

A computational model is chosen, often the Turing machine. The time taken by a computation is defined as the number of steps until the computation halts. If a computation halts for all possible inputs x, then the **time complexity** is given by the maximum time taken over all inputs of a given size. Other definitions, such as the average time, are also used.

The resource requirements of problems are the subject of computational complexity theory [GaJo79]. The main question that one asks is whether there exists a feasible solution to a given problem, where "feasible" is usually considered to be synonymous with polynomial time computable.

The resource requirements of algorithms are studied in the analysis of algorithms [AHU75]. In the analysis of algorithms, one is interested in choosing between different algorithms which solve the same problem. The resource requirements are usually given up to order of magnitude. For example, one says the complexity of the quicksort algorithm is $O(n^2)$ whereas the complexity of the mergesort algorithm is $O(nlogn)$.

When analysing the resource requirements of implementations, one is usually interested in the exact complexity, for example, the number of seconds it takes a particular piece of assembler code to perform its operation.

In both complexity theory and the analysis of algorithms, program complexity is expressed by using input measures. An input measure is a function from input data to an object of sizes. For example, for a sorting problem, the size of a problem instance might be the number of items to be sorted. The size of an instance of the travelling salesman problem is the number of edges in the underlying graph. The complexity of the problem is then defined as a function from sizes of input to resources. In order for this to be well defined, some choice has to be made. Typically, this can be worst case complexity where the complexity is defined to be the longest time taken over all inputs of a given size, or average case complexity where the complexity is defined to be the average time, with respect to a suitable distribution, taken over all inputs of a given size.

In complexity theory and the analysis of algorithms, both the size of input and the resource are often expressed as natural numbers. The complexity will then be a function from $\mathbf{N}$ to $\mathbf{N}$. Therefore, one can speak of an algorithm having complexity which is $O(n)$ if the complexity of the most efficient implementation of the algorithm is $O(n)$. One can speak of a problem being of polynomial complexity if there exists some program which solves the problem and whose complexity is polynomial.

1.3 Related work

There has been relatively little work on uniting complexity theory and other areas of computer science.

An early contribution is the thesis of Nielson [Niel84]. She takes an axiomatic approach, extending Hoare Logics to obtain formal systems for reasoning about the exact run time of programs. Subsequently, Bjerner and Holmström [BjHo89] have analysed the exact run time complexity of first order lazy functional programs. They take a denotational approach and use a notion of demand function to obtain lazy expressions for the time complexity. However, their work is based on a specific language and model, it only deals with first order functions and does not consider non-exact complexity.

There has been some work on formalising the notion of polynomial time computability in various logical and mathematical frameworks. This is an easier problem than that of expressing arbitrary non-exact complexity, because of the robustness of the class of polynomial functions. In particular, it is possible to obtain compositional models. An early contribution is that of Asveld and Tucker [AsTu82] who showed how to express notions such as polynomial time and polynomial space within the framework of abstract data types in algebraic specifications. More recently, several authors [GiSS90], [NeRS89] and [See89] have refined the typing systems of various logics so that they contain explicit bounds of the time complexity. The most elegant and interesting of these papers is [GiSS90] which develops a system based on Girard's Linear Logic [Gir87]. The idea is that the type of an expression will yield an upper bound on its time complexity. In each case, the main result is that the typeable terms correspond precisely to the polynomial computable functions. The main drawback with this approach is that it does not yield very good bounds and is only suitable for studying polynomial time computable functions.

One of the drawbacks of complexity theory is that it is not directly applicable to languages with higher order functions, such as ML, or even to languages

which allow procedures to be passed as arguments such as Pascal or C. Cook and Kapron [CoKa90] have attempted to address this issue by studying the complexity of functionals, that is functions which take natural numbers and other functionals as arguments and return natural numbers. They attempt to define a notion of feasible computation for these functionals. Shultis [Shul90] has studied the exact time complexity of a specific higher order language he defines. His approach is similar to the work in section 2.1. However, he does not consider the non-exact complexity of higher order languages. This appears to be a hard problem.

Finally, Flajolet, Salvy and Zimmerman [FSZ89] have developed an implementation of a formal system for the automatic analysis of the average case complexity of certain algorithms. Their approach is rather limited in scope. However, they produce an interesting software tool, the $\Lambda\Upsilon\Omega$-Cookbook, for performing the analysis automatically. This is along the lines of the approach we described in section 1.1.

It seems likely that other authors have worked on this subject. However, none of them has seriously influenced this work.

1.4 Outline of the thesis

This thesis focuses on extending existing work on denotational semantics to incorporate ideas from complexity theory and the analysis of algorithms. The thesis falls naturally into two parts. Chapters 2,3 and 4 are related to Moggi's work on the categorical semantics of computations [Mog88b], [Mog89]. Chapter 5 is the pivotal chapter in which we show why Moggi's framework is not sufficiently general for all of our needs. We define a new class of models and show how Moggi's models are a special case of this. In chapters 6,7 and 8, we use this new framework to study various notions from complexity theory.

In chapter 2, we describe the approach that we have taken in modelling the resource requirements of programs and discuss in some detail the properties required of such models. In particular, we emphasise the importance of having a monoid

$\mathcal{M}$ of resource values in order to model the sequential composition of programs. We show that the models obtained from our approach can be viewed as a special case of Moggi's categorical semantics of computations.

In chapter 3, we study formal systems for reasoning about the models we have developed. In particular, we construct a formal system, the λ_{com}-calculus, based on Moggi's λ_c-calculus for reasoning about the resource requirements of programs. We define operational and denotational semantics for this language and prove a correspondence theorem.

In chapter 4, we exploit the connection with Moggi's work and follow his program for a modular approach to denotational semantics. We show that complexity may be viewed as a monad constructor. We extend his work to define a notion of strong monad constructor and discuss some applications of this extension.

In chapter 5, we show that Moggi's categorical semantics of computations is not sufficiently general to express all of the examples of interest to us. Therefore, we define a new class of models, based on the notion of an external datum. We also define a notion of internal datum which corresponds to Moggi's framework. In order to investigate the relationship between the internal and the external approach, we define categories **Int** and **Ext** of, respectively, internal and external data. We prove that the category **Int** is a full reflective subcategory of the category **Ext**.

In chapters 6, 7 and 8, we show how to extend our work to capture various notions of importance in complexity theory and in the analysis of algorithms. In this work, we are led to consider order enriched structures. In particular, in chapter 6, we replace our monoid $\mathcal{M}$ by an ordered monoid $\mathcal{M}$ of resource values, leading to the definition of an ordered datum. We define internal and external notions of ordered data and study the relationship between them.

In chapter 7, we define input measures and analyses and show how they give rise to models in which complexity is expressed as a map from input size to resource. Unfortunately, these models are unsatisfactory because they destroy the compositionality of exact complexity. A possible solution to this problem is to

consider non-exact complexity. Accordingly, we consider the kind of equivalence relations that we need for a non-exact framework. This leads to the definition of an M-equivalence on an external datum. We prove a characterisation result for M-equivalences and that every M-equivalence gives rise to a congruence on $\mathcal{C}^M$. We show that the models we seek can be expressed as quotients of external complexity categories by congruences generated by M-equivalences.

In chapter 7 we saw that if we use input measures and exact complexity, it is not in general possible to obtain a compositional semantics. Interestingly, it is possible to obtain a compositional semantics for certain types of non-exact complexity, such as polynomial time complexity. In chapter 8, we apply the work of chapters 6 and 7 to develop compositional non-exact models. We define a non-exact datum and study the relationship between non-exact data and external data along similar lines to chapter 5. Finally, we show how to construct models for compositional non-exact complexity.

Chapter 2

Motivation and Examples

2.1 Adding complexity

2.1.1 Introduction

Let $\mathcal{L}$ be a programming language. Let $\mathcal{D}$ be a model for $\mathcal{L}$, for example a domain, and let $[\![-]\!]$ be an interpretation of the language $\mathcal{L}$ in $\mathcal{D}$.

Let p be a program in $\mathcal{L}$ taking input of type A and producing output of type B. Then the denotation of p in $\mathcal{D}$ will be a map from the denotation of A to denotation of B. We shall write this as:

$$fun(p) : [\![A]\!] \longrightarrow [\![B]\!].$$

For each input value $a \in A$ this tells us which output value $b \in B$ is produced by p. Therefore, it represents the input-output behaviour of the program p. We call this the functional behaviour of p.

We want to represent the complexity of the program p. The complexity of p is the amount of resource that p consumes on each input $a \in A$ to produce the output $p(a) \in B$. A first attempt would be to represent the complexity of p as a map:

$$cx(p) : [\![A]\!] \longrightarrow M.$$

That is, a map from values of the input type A to an object M of possible resource values. The idea is that $cx(p)(a)$ is the amount of resource consumed by the program p when applied to the input $a \in A$.

Following this approach, we aim to construct models of programming languages which contain a suitable object M of resource values. A program p of type $A \longrightarrow B$ will then be denoted by a pair of maps:

$$\langle fun(p) : [\![A]\!] \longrightarrow [\![B]\!], cx(p) : [\![A]\!] \longrightarrow M \rangle.$$

2.1.2 Composing programs

It is fundamental to the study of denotational semantics that we can model the sequential composition of programs. Typically, we expect a model $\mathcal{D}$ to be a category. The denotation of a sequential composition of programs $p; q$ is then given by the composition in $\mathcal{D}$ of the denotation of p and the denotation of q. That is:

$$[\![p; q]\!] \;=\; [\![q]\!][\![p]\!].$$

Notation 2.1.1 We note the unfortunate clash between the category theoretic tradition of writing "p followed by q" as qp and the programming language notation $p; q$. We respect both conventions, writing composition of programs on the right and composition in mathematical structures on the left.

The amount of resource consumed by a composite program $p; q$ on an input value $a \in A$ is the amount consumed by p on a together with the amount consumed by q on $p(a)$. Therefore, the complexity of a composite program $p; q$ is determined by the complexity of p, the complexity of q and an operation $\cdot$ which combines them.

Example 2.1.2 To model time complexity, we could take M to be the natural numbers and $\cdot$ to be addition. The time taken by $p; q$ is then the time taken by p plus the time taken by q on the output of p.

Example 2.1.3 To model space complexity, we could take M to be the natural numbers and $\cdot$ to be maximum. The memory used by $p;q$ on a is then the maximum of $t(a)$, the memory used by p on a, and $s(p(a))$, the memory used by q on $p(a)$.

Example 2.1.4 To model time complexity and space complexity, we could take M to be the $\mathbf{N} \times \mathbf{N}$ and $\cdot$ to be the operation $\langle +, max \rangle$.

Let p be a program of type $A \longrightarrow B$ and let q be a program of type $B \longrightarrow C$. The extended denotations of p and q are given by:

$$\begin{array}{ccccc} [\![A]\!] & \xrightarrow{fun(p)} & [\![B]\!] & \xrightarrow{fun(q)} & [\![C]\!] \\ \downarrow{\scriptstyle cx(p)} & & \downarrow{\scriptstyle cx(q)} & & \\ M & & M & & \end{array}$$

The denotation of the program $p;q$ of type $A \longrightarrow C$ is a pair of maps:

$$\begin{array}{ccc} [\![A]\!] & \xrightarrow{fun(p;q)} & [\![C]\!] \\ \downarrow{\scriptstyle cx(p;q)} & & \\ M. & & \end{array}$$

We have mentioned that we take $\mathcal{D}$ to be a category in order to model the composition of programs. In order to express the extended denotation of $p;q$, we require some additional structure on $\mathcal{D}$. In particular, let $\mathcal{D}$ be a category with finite products and let $\cdot$ be a morphism from $M \times M$ to M. Then we can model $p;q$ by:

$$\begin{array}{ccccc} [\![A]\!] & \xrightarrow{fun(p)} & [\![B]\!] & \xrightarrow{fun(q)} & [\![C]\!] \\ \downarrow{\scriptstyle \langle cx(p), cx(q) fun(p) \rangle} & & & & \\ M \times M & & \xrightarrow{\qquad\cdot\qquad} & & M \end{array}$$

so that:

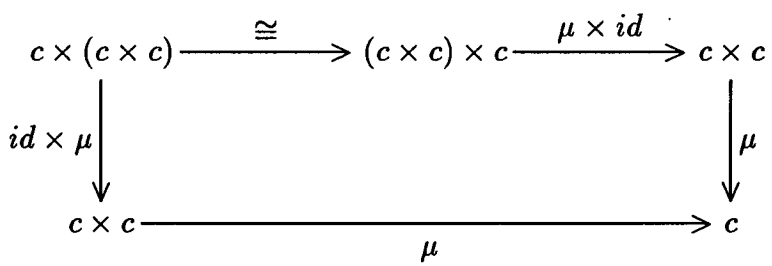
$$fun(p;q) = fun(q)fun(p) \quad \text{and}$$

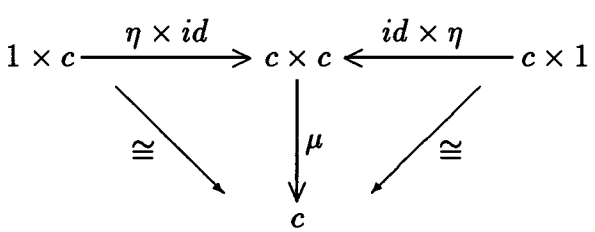
$$cx(p;q) = cx(p) \cdot cx(q)fun(p).$$

There are two basic conditions required of the models in denotational semantics. In order to model trivial computations, that is the process of not performing any computation, we require that $\mathcal{D}$ has all identity maps. Secondly, we require that composition in $\mathcal{D}$ is associative since composition of programs is associative. These two conditions amount to the requirement that $\mathcal{D}$ is a category.

We want our extended models to be categories for the same reasons. The following result shows that this condition corresponds to requiring that M has a monoid structure.

Definition 2.1.5 A **monoid** in a category $\mathcal{C}$ with finite products is a tuple $\langle c, \eta, \mu \rangle$ consisting of an object c of $\mathcal{C}$ and arrows $\mu : c \times c \longrightarrow c$ and $\eta : 1 \longrightarrow c$ such that the following diagrams:





commute.

Notation 2.1.6 Let $\mathcal{C}$ be a category with finite products, let $\mathcal{M} = \langle M, 0, \cdot \rangle$ be a monoid in $\mathcal{C}$ and let $s, t : A \longrightarrow M$ be morphisms in $\mathcal{C}$. We write $s \cdot t$ for the morphism $A \xrightarrow{\langle s,t \rangle} M \times M \longrightarrow M$.

Proposition 2.1.7 Let $\mathcal{C}$ be a category with finite products, let M be an object of $\mathcal{C}$ and let $M \times M \longrightarrow M$ be a morphism in $\mathcal{C}$. Then the following data:

- objects: objects of $\mathcal{C}$,
- morphisms: pairs $\langle f, t\rangle$ where $f : A \longrightarrow B$ and $t : A \longrightarrow M$ in $\mathcal{C}$ and
- composition: given by $\langle g, s\rangle\langle f, t\rangle = \langle gf, t \cdot sf\rangle$

define a category $\mathcal{C}^+$ if there exists a morphism $0 : 1 \longrightarrow M$ such that $\langle M, 0, \cdot\rangle$ is a monoid in $\mathcal{C}$. Conversely, if $\mathcal{C}$ is well pointed and $\mathcal{C}^+$ is a category, then there exists a morphism $0 : 1 \longrightarrow M$ such that $\langle M, 0, \cdot\rangle$ is a monoid in $\mathcal{C}$.

Proof: Suppose that $\langle M, 0, \cdot\rangle$ is a monoid in $\mathcal{C}$.
Then the identity at A is given by $\langle id, 0\rangle$ since:

- $$\begin{aligned}\langle f, t\rangle\langle id, 0\rangle &= \langle fid, 0 \cdot tid\rangle \\ &= \langle f, 0 \cdot t\rangle \\ &= \langle f, t\rangle.\end{aligned}$$
- $$\begin{aligned}\langle id, 0\rangle\langle f, t\rangle &= \langle idf, t \cdot 0f\rangle \\ &= \langle f, t \cdot 0\rangle \\ &= \langle f, t\rangle.\end{aligned}$$

Composition is associative since:

- $$\begin{aligned}\langle f, r\rangle(\langle g, s\rangle\langle h, t\rangle) &= \langle f, r\rangle\langle gh, t \cdot sh\rangle \\ &= \langle f(gh), (t \cdot sh) \cdot rgh\rangle \\ &= \langle (fg)h, t \cdot (sh \cdot rgh)\rangle \\ &= \langle (fg)h, t \cdot (s \cdot rg)h\rangle \\ &= \langle fg, s \cdot rg\rangle\langle h, t\rangle \\ &= (\langle f, r\rangle\langle g, s\rangle)\langle h, t\rangle\end{aligned}$$

and therefore the data defines a category.
Conversely, suppose that $\mathcal{C}$ is well pointed and that $\mathcal{C}^+$ is a category.
Then for each $A \in |\mathcal{C}|$, there exists a map $\langle f_A, t_A\rangle : A \longrightarrow A$ such that for all

$\langle f,t\rangle : B \longrightarrow A$, $\langle f_A,t_A\rangle\langle f,t\rangle = \langle f,t\rangle$.
In particular, for all $t : 1 \longrightarrow M$:

$$\langle f_1,t_1\rangle\langle id,t\rangle = \langle id,t\rangle$$

which implies that $\langle f_1 id, t \cdot t_1 id\rangle = \langle id,t\rangle$ and in particular that $t \cdot t_1 = t$ for all t. Similarly, $t_1 \cdot t = t$ for all t. However, $\mathcal{C}$ is well pointed and so this implies that t_1 is a unit for $\cdot$.
Finally, composition is associative from which we can show that $\cdot$ is associative and therefore that $\langle M,0,\cdot\rangle$ is a monoid in $\mathcal{C}$.
This completes the proof. □

Therefore, a first attempt to model complexity is a category $\mathcal{C}$ with finite products and a monoid $\langle M,0,\cdot\rangle$ in $\mathcal{C}$.

Notation 2.1.8 Let $\mathcal{C}$ be a category with finite products and let $\mathcal{M}$ be a monoid in $\mathcal{C}$. We call $\langle \mathcal{C},\mathcal{M}\rangle$ a complexity datum and we call $\mathcal{C}^+$ the complexity category for $\langle \mathcal{C},\mathcal{M}\rangle$.

Example 2.1.9 Each of the binary operations on M in examples 2.1.2, 2.1.3 and 2.1.4 forms a monoid structure on M in **Set**.

2.1.3 Examples

We give a number of examples to illustrate our approach. In the first example, we show how to extend the semantics for a simple programming language to incorporate time complexity. In the second example, we show how this approach can be applied to an assembler language. In the third example, we use a Pascal program to show how the complexity may depend on the functional behaviour. In the final example, we use the language ML to demonstrate some of the issues involved in modelling functional languages.

Example 2.1.10 We consider a simple language $\mathcal{L}$ given by the following abstract syntax:

Expressions $\mathbf{e} ::= i \mid n \mid e_1 + e_2 \mid e_1 * e_2$
where i is an identifier and n is a numeral.

Commands $\mathbf{c} ::= skip \mid i \leftarrow e_1 \mid do^n c \mid c_1; c_2$.

This language is not very expressive. However, we can write the following command computing the factorial of n:

$$c = i \leftarrow 0; m \leftarrow 1; do^n(i \leftarrow i + 1; m \leftarrow m * i).$$

We give a denotational semantics for $\mathcal{L}$. Let S be a set of states each of which consists of an assignment of positive integer values to each identifier. We write $s[n/i]$ for the state obtained from s by replacing the value of i by the natural number n. Commands denote total functions from S to S as follows:

$[\![skip]\!] = \lambda s.s$
$[\![i \leftarrow e]\!] = \lambda s.s[[\![e]\!] s/i]$

where $[\![e]\!] s$ is the integer given by:
$[\![n]\!] s = n$
$[\![i]\!] s = s(i)$
$[\![e_1 + e_2]\!] s = [\![e_1]\!] s + [\![e_1]\!] s$
$[\![e_1 * e_2]\!] s = [\![e_1]\!] s \times [\![e_1]\!] s.$

$[\![c_1; c_2]\!] = [\![c_2]\!][\![c_1]\!] = \lambda s.[\![c_2]\!]([\![c_1]\!] s)$
$[\![do^n c]\!] = [\![c]\!]^n$ where $[\![c]\!]^0 = \lambda s.s$ and $[\![c]\!]^n = [\![c]\!]^{n-1}[\![c]\!]$.

Evaluating the denotation of c we obtain:

$[\![c]\!] = [\![i \leftarrow 0; m \leftarrow 1; do^n(i \leftarrow i + 1; m \leftarrow m * i)]\!]$
$[\![c]\!] = \lambda s.[\![m \leftarrow 1; do^n(i \leftarrow i + 1; m \leftarrow m * i)]\!]([\![i \leftarrow 0]\!] s)$
$[\![c]\!] = \lambda s.[\![m \leftarrow 1; do^n(i \leftarrow i + 1; m \leftarrow m * i)]\!] s[0/i]$
$[\![c]\!] = \lambda s.(\lambda s.[\![(do^n(i \leftarrow i + 1; m \leftarrow m * i)]\!]([\![m \leftarrow 1]\!] s))s[0/i]$
$[\![c]\!] = \lambda s.[\![i \leftarrow i + 1; m \leftarrow m * i]\!]^n \; s[0/i, 1/m]$
$[\![c]\!] = \lambda s.((\lambda s.\text{if } n = 0 \text{ then } s \text{ else}$

$$[\![i \leftarrow i+1; m \leftarrow m*i]\!]^{n-1}\ (\ [\![i \leftarrow i+1; m \leftarrow m*i]\!]s))s[0/i, 1/m]$$

$$[\![c]\!] = \begin{cases} \lambda s.s[0/i, 1/m] & \text{if } n = 0 \\ \lambda s.[\![i \leftarrow i+1; m \leftarrow m*i]\!]^{n-1}(s[1/i, 1/m]) & \text{if } n \neq 0 \end{cases}$$

and it follows by induction that we can continue this expansion to obtain:

$$[\![c]\!] \ = \ \lambda s.s[n/i, n!/m]$$

which shows that c is a program which computes the factorial function.

We now extend the semantics so a command denotes a pair of functions $S \rightarrow S$ and $S \rightarrow \mathbf{N}$. For simplicity, we shall assume that each basic operation takes constant time although this is not an essential restriction. The new semantics is given by:

$[\![skip\ ]\!] = \lambda s.\langle s, 0\rangle$

$[\![i \leftarrow e]\!] = \lambda s.\langle s[[\![e]\!]_1 s/i], [\![e]\!]_2 s + \alpha\rangle$

where $[\![e]\!]s = \langle [\![e]\!]_1 s, [\![e]\!]_2 s\rangle$ is given by:

$[\![n]\!]s = \langle n, 0\rangle$

$[\![i]\!]s = \langle s(i), 0\rangle$

$[\![e_1 + e_2]\!]s = \langle [\![e_1]\!]_1 s + [\![e_2]\!]_1 s, [\![e_1]\!]_2 s + [\![e_2]\!]_2 s + \beta\rangle$

$[\![e_1 * e_2]\!]s = \langle [\![e_1]\!]_1 s \times [\![e_2]\!]_1 s, [\![e_1]\!]_2 s + [\![e_2]\!]_2 s + \gamma\rangle$

and α, β, γ are natural numbers.

$[\![c_1; c_2]\!] = [\![c_2]\!][\![c_1]\!]$

where $[\![c_2]\!][\![c_1]\!] = \lambda s.\langle [\![c_2]\!]_1([\![c_1]\!]_1 s), [\![c_1]\!]_2 s + [\![c_2]\!]_2([\![c_1]\!]_1 s)\rangle$

$[\![do^n c]\!] = [\![c]\!]^n$ where $[\![c]\!]^0 = \lambda s.\langle s, 0\rangle$ and $[\![c]\!]^n = [\![c]\!]^{n-1}[\![c]\!]$.

Evaluating the new denotation of c we obtain:

$[\![c]\!] = [\![i \leftarrow 0; m \leftarrow 1; do^n(i \leftarrow i+1; m \leftarrow m*i)]\!]$

$[\![c]\!] = [\![m \leftarrow 1; do^n(i \leftarrow i+1; m \leftarrow m*i)]\!][\![i \leftarrow 0]\!]$

$[\![c]\!] = (\lambda s.s[[\![0]\!]_1/i], \lambda s.[\![0]\!]_2 + \alpha)[\![m \leftarrow 1; do^n(i \leftarrow i+1; m \leftarrow m*i)]\!]$

$[\![c]\!] = (\lambda s.s[0/i], \lambda s.\alpha)[\![do^n(i \leftarrow i+1; m \leftarrow m*i)]\!][\![m \leftarrow 1]\!]$

$$[\![c]\!] = (\lambda s.s[0/i], \lambda s.\alpha)(\lambda s.s[1/m], \lambda s.\alpha)[\![do^n(i \leftarrow i+1; m \leftarrow m*i)]\!]$$

$$[\![c]\!] = [\![i \leftarrow i+1; m \leftarrow m*i]\!]^n(\lambda s.s[0/i, 1/m], \lambda s.2\alpha)$$

$$[\![c]\!] = (\lambda s.\text{if } (n=0) \text{ then } (\lambda s.s, \lambda s.0)$$
$$\text{else } [\![i \leftarrow i+1; m \leftarrow m*i]\!]^{n-1})[\![(i \leftarrow i+1; m \leftarrow m*i)]\!](\lambda s.s[0/i, 1/m], \lambda s.2\alpha)$$

$$[\![c]\!] = \begin{cases} (\lambda s.s[0/i, 1/m], \lambda s.2\alpha) & \text{if } n = 0 \\ [\![i \leftarrow i+1; m \leftarrow m*i]\!]^{n-1}[\![m \leftarrow m*i]\!] & \\ [\![i \leftarrow i+1]\!](\lambda s.s[0/i, 1/m], \lambda s.2\alpha) & \text{if } n \neq 0 \end{cases}$$

$$[\![c]\!] = \begin{cases} (\lambda s.s[0/i, 1/m], \lambda s.2\alpha) & \text{if } n = 0 \\ [\![i \leftarrow i+1; m \leftarrow m*i]\!]^{n-1}(\lambda s.s[1/i, 1/m], \lambda s.4\alpha + \beta + \gamma) & \text{if } n \neq 0 \end{cases}$$

and clearly we can continue the expansion to obtain:

$$[\![c]\!] = \lambda s.\langle s[n/i, n!/m], (2n+2)\alpha + n(\beta + \gamma)\rangle$$

giving not only the functional behaviour of the program but also its time complexity.

Example 2.1.11 Although our main interest is in studying high level programming languages such as C or ML, there is no a priori reason why these techniques could not be applied to a much wider class of programming languages. For example, we give an extended semantics to a subset of the instruction set of a Z80/8080 chip.

We assume a set of registers $\mathcal{X}$ with $|\mathcal{X}| \geq 2$ and a set of memory locations $\mathcal{L}$ with $|\mathcal{L}| \geq 1$. The instructions are as follows:

Instruction	Action
LD X,Y	load the value in register Y into register X
LD X,(L)	load the value in location L into register X
LD (L),X	load the value in register X into location L
ADD X,Y	add the value in Y to the value in X
SUB X,Y	subtract the value in Y from the value in X
NOP	no operation

These instructions comprise the basic 16-bit instruction set, without branching, of the Z80 or 8080 chip.

We now give a timed denotational semantics for the language. A value is an integer in the range 0 to 256. A state S is an assignment of values to each of the registers $X_1, ..., X_n$. An environment is an assignment of values to each of the locations $L_1, ..., L_m$. Each instruction c will denote a pair of functions $[\![c]\!]$ from $S \times E$ to $S \times E$ and from $S \times E$ to $\mathbf{N}$.

The semantics is given by the following rules:

$$
\begin{aligned}
[\![\text{LD } X_i, X_j]\!] &= \langle \lambda\langle s, e\rangle.\langle s(i) = s(j), e\rangle, \lambda\langle s, e\rangle.1\rangle \\
[\![\text{LD } X_i, (L_j)]\!] &= \langle \lambda\langle s, e\rangle.\langle s(i) = e(j), e\rangle, \lambda\langle s, e\rangle.10\rangle \\
[\![\text{LD } (L_i), X_j]\!] &= \langle \lambda\langle s, e\rangle.\langle s, e(i) = s(j)\rangle, \lambda\langle s, e\rangle.12\rangle \\
[\![\text{ADD } X_i, X_j]\!] &= \langle \lambda\langle s, e\rangle.\langle s(x) = s(x) + s(y), e\rangle, \lambda\langle s, e\rangle.2\rangle \\
[\![\text{SUB } X_i, X_j]\!] &= \langle \lambda\langle s, e\rangle.\langle s(x) = s(x) - s(y), e\rangle, \lambda\langle s, e\rangle.3\rangle \\
[\![\text{NOP}]\!] &= \langle \lambda\langle s, e\rangle.\langle s, e\rangle, \lambda\langle s, e\rangle.0\rangle \\
[\![I_1; I_2]\!] &= [\![I_2]\!][\![I_1]\!]
\end{aligned}
$$

We use this timed semantics to illustrate the difference between using registers and using memory locations to write a simple program to multiply a value by four.

I_1	I_2
LD A,(L)	LD A,(L)
LD B,A	LD B,(L)
ADD A,B	ADD A,B
LD B,A	LD (L),A
	LD B,(L)
ADD A,B	ADD A,B
LD (L),A	LD (L),A

We evaluate the denotations of I_1 and I_2 using the above semantics to obtain:

$$[\![I_1]\!] = \langle \lambda\langle s, e\rangle.\langle s(A) = 4e(L), s(B) = 2e(L), e(L) = 4e(L)\rangle, \lambda\langle s, e\rangle.28\rangle$$
$$[\![I_2]\!] = \langle \lambda\langle s, e\rangle.\langle s(A) = 4e(L), s(B) = 2e(L), e(L) = 4e(L)\rangle, \lambda\langle s, e\rangle.58\rangle$$

This illustrates the advantage of making extensive use of registers in assembly language programming. The two programs I_1 and I_2 have the same functional behaviour and therefore would not be distinguished by a traditional semantics. However, it is clear from our extended semantics that I_1 is more than twice as fast as I_2.

Example 2.1.12 We give a timed denotational semantics to the following fragment of Pascal:

Command	Action
`read n`	read an integer value into the variable `n`
`write n`	write out the value of the integer variable `n`
`x:=e`	assign to value of the expression `e` to the variable `x`
`while b do c`	repeat the command `c` for as long as the boolean expression `b` holds
`begin ... end`	brackets
$c_1 ; c_2$	execute c_1 then execute c_2

We now give a timed semantics for this fragment. An environment ρ is an assignment of integer values to each variable. The set of environments is denoted by E. We write $\mathcal{I}$ and $\mathcal{O}$ for the input and output types of each command, and each command c will denote a pair of functions $[\![c]\!]$ from $\mathcal{I} \times E$ to $\mathcal{O} \times E$ and from $\mathcal{I} \times E$ to $\mathbf{N}$. In most cases, both $\mathcal{I}$ and $\mathcal{O}$ will be the unit type and are omitted for simplicity. We note that a semantics for this fragment requires partial functions, since a program such as `while true do c` will not terminate.

The semantics is given by the following rules:

$$[\![\texttt{read n}]\!] = \langle \lambda x : \mathbf{N}.\lambda\rho.\rho[x/n], \lambda x : \mathbf{N}.\lambda\rho.1\rangle$$
$$[\![\texttt{write n}]\!] = \langle \lambda\rho.\langle \rho(n), \rho\rangle, \lambda\rho.2\rangle$$
$$[\![\texttt{x := e}]\!] = \langle \lambda\rho.\rho[[\![\texttt{e}]\!]_1\rho/\texttt{x}], \lambda\rho.[\![\texttt{e}]\!]_2\rho + 1\rangle$$

where $[\![\mathtt{e}]\!]\rho = \langle [\![\mathtt{e}]\!]_1\rho, [\![\mathtt{e}]\!]_2\rho \rangle$ is given by:

$[\![\mathtt{n}]\!]\rho = \langle n, 1 \rangle$

$[\![\mathtt{x}]\!]\rho = \langle \rho(\mathtt{x}), 1 \rangle$

$[\![\mathtt{e}_1 + \mathtt{e}_2]\!]\rho = \langle [\![\mathtt{e}_1]\!]_1\rho + [\![\mathtt{e}_2]\!]_1\rho, [\![\mathtt{e}_1]\!]_2\rho + [\![\mathtt{e}_2]\!]_2\rho + [log_2([\![\mathtt{e}_1]\!]_1\rho \cdot [\![\mathtt{e}_2]\!]_1\rho)] \rangle$

$[\![\mathtt{e}_1 - \mathtt{e}_2]\!]\rho = \langle [\![\mathtt{e}_1]\!]_1\rho - [\![\mathtt{e}_2]\!]_1\rho, [\![\mathtt{e}_1]\!]_2\rho + [\![\mathtt{e}_2]\!]_2\rho + [log_2([\![\mathtt{e}_1]\!]_1\rho \cdot [\![\mathtt{e}_2]\!]_1\rho)] \rangle$

$[\![\mathtt{e}_1 * \mathtt{e}_2]\!]\rho = \langle [\![\mathtt{e}_1]\!]_1\rho \times [\![\mathtt{e}_2]\!]_1\rho, [\![\mathtt{e}_1]\!]_2\rho + [\![\mathtt{e}_2]\!]_2\rho + [\![\mathtt{e}_1]\!]_1\rho + [\![\mathtt{e}_2]\!]_1\rho \rangle$

$[\![\mathtt{while\ b\ do\ c}]\!] = Y_\gamma(\lambda\gamma.\lambda\rho.\ \text{if } [\![\mathtt{b}]\!]\rho \text{ then } \gamma[\![c]\!]\rho \text{ else } \langle \rho, 1 \rangle)$

where $[\![\mathtt{e}]\!]\rho$ is true if the boolean expression b holds in the environment ρ and false otherwise, and Y_γ is a suitable fixpoint operator [Sto77].

$[\![\mathtt{begin\ c\ end}]\!] = [\![\mathtt{c}]\!]$

$[\![\mathtt{c}_1; \mathtt{c}_2]\!] = [\![\mathtt{c}_2]\!][\![\mathtt{c}_1]\!]$

The main point of this semantics is that the complexity may depend on the functional behaviour. For example, consider the following program to compute the factorial function:

```
begin
        m:=1;
        read n;
        while n>0 do
                        begin
                                m:=m*n;
                                n:=n-1;
                        end;
        write m;
end.
```

It is easy to show that this program terminates after exactly $3n + 4$ computational steps. If we assume that the complexity does not depend on the functional behaviour, then each operation should take constant time. Therefore, one might estimate that the complexity is $O(n)$. However, if one actually runs this program, it is clear that its complexity is not $O(n)$.

However, we can evaluate the denotation of this program using the above semantics to obtain:

$$\langle \lambda x \lambda \rho.\langle x!, \rho[n/0, m/x!]\rangle, \lambda x \lambda \rho.6(x+1) + 1! + \cdots + x! + [log_2 1] + \cdots + [log_2 x]\rangle$$

which suggests, more accurately, that the complexity is in fact $O(n!)$.

Example 2.1.13 In this example, we seek to illustrate some of the issues involved in giving an extended semantics to functional languages. Consider the following ML function definition:

```
- fun sum 0 = 0 | sum x = x + sum(x-1);
> val sum = fn : int -> int
- sum 7;
> 28 : int
- sum 28;
> 406 : int
```

sum is a program which computes the sum up to n digits. We can write the following ML program to apply twice a function such as sum.

```
- fun twice f x:int = f(f(x));
> val twice = fn : (int -> int) -> (int -> int)

- val sum2 = twice sum;
> val sum2 = fn : int -> int
- sum2 7;
> 406 : int
```

When modelling languages with higher order functions, it is usual to use a cartesian closed category. One then models the input to a program such as `twice` by a suitable element of the exponential object $[\![\texttt{int}]\!]^{[\![\texttt{int}]\!]}$. For example, if the model was **Set** and $[\![\texttt{int}]\!]$ were $\mathbf{N}$, then an input to $[\![\texttt{twice}]\!]$ would be the element $\lambda n.1/2n(n+1)$ of $\mathbf{N}^{\mathbf{N}}$.

In order to give an extended semantics to a functional language such as ML, it is clear that $[\![\texttt{twice}]\!]$ must somehow code up the complexity of its input. Since this complexity will be a function from input type to resource, an input to the denotation of `twice` should be a pair of elements of the exponential objects $[\![\texttt{int}]\!]^{[\![\texttt{int}]\!]}$ and $M^{[\![\texttt{int}]\!]}$. For example, we might have:

$$[\![\texttt{sum}]\!] = \langle \lambda n.1/2n(n+1), \lambda n.2n+4 \rangle.$$

We return to this discussion in the next section.

2.1.4 Datatypes

Many typed programming languages are endowed with datatype constructors which construct new datatypes from existing ones. For example, in Pascal if $\sigma_1, \sigma_2, ..., \sigma_n$ are types then we can form a record type $\sigma_1 \times \sigma_2 \times ... \times \sigma_n$, an element of which is a tuple $\langle a_1, a_2, ..., a_n \rangle$ where each a_i is an element of σ_i. We can form a sum type $\sigma_1 + \sigma_2 + ... + \sigma_n$ an element of which is an element of any one of the σ_i. In standard ML, if $\sigma_1, \sigma_2, ..., \sigma_n$ are types then we can form a tuple type $\sigma_1 \times \sigma_2 \times ... \times \sigma_n$ and a record type which corresponds to the record type in Pascal. We can also form a functional type $\sigma_1 \longrightarrow \sigma_2$, an element of which is a program taking input of type σ_1 and producing output of type σ_2.

In the previous section, we showed why categories form useful denotational models. In denotational semantics, categories with structure are used to model datatype constructors. For example, product types are modelled by categorical product as:

$$[\![\sigma_1 \times \sigma_2]\!] = [\![\sigma_1]\!] \times [\![\sigma_2]\!],$$

sum types are modelled by categorical coproduct as:

$$[\![\sigma_1 + \sigma_2]\!] = [\![\sigma_1]\!] + [\![\sigma_2]\!]$$

and functional types are modelled by internal hom as:

$$[\![\sigma_1 \longrightarrow \sigma_2]\!] = [[\![\sigma_1]\!], [\![\sigma_2]\!]].$$

The usual conditions that one imposes on a domain $\mathcal{D}$ is that $\mathcal{D}$ is a cartesian closed category with finite coproducts.

Remark 2.1.14 *Although most interest has focused on cartesian closed categories since these correspond to simply typed lambda calculii, interest has also centred on categories with other structure. For example, categories which are models for the solutions of recursive domain equations.*

In order to model datatype constructors in our extended semantics, we should like $\mathcal{C}^+$ to have all the structure that $\mathcal{C}$ has. In particular, if $\mathcal{C}$ is cartesian closed then we should like $\mathcal{C}^+$ to be cartesian closed.

Unfortunately, this is not the case. The following example shows that $\mathcal{C}^+$ need not even have finite products.

Example 2.1.15 Let $\mathcal{C}$ be **Set** and let $\mathcal{M}$ be the monoid $\langle \mathbf{N}, 0, + \rangle$ in **Set**. Then the category $\mathbf{Set}^+$ does not have finite products.
Proof: We show that $\mathbf{Set}^+$ does not have a terminal object. In $\mathbf{Set}^+$, $\mathbf{Set}^+(A, B)$ is the set $B^A \times \mathbf{N}^A$ which is a singleton if and only if A is empty. Therefore, there is no terminal object. □

Remark 2.1.16 *It is possible to show that* $\mathbf{Set}^+$ *has lax products (c.f. section 6.2). However, we have not found this observation helpful and do not pursue it.*

The problem is that $\mathcal{C}^+$ does not have the usual structure to model datatype constructors, even when the category $\mathcal{C}$ does. A possible solution is to seek a method of modelling datatype constructors in $\mathcal{C}^+$ which is based on the structure in $\mathcal{C}$. In the next chapter, we show how Eugenio Moggi's categorical semantics of computations [Mog88b] provides one such method, and we apply his framework to our problem.

Chapter 3

Reasoning in Time

3.1 Introduction

In chapter 2, we constructed a class of categories $\mathcal{C}^+$ in order to model programs with complexity. We showed that the categories $\mathcal{C}^+$ do not have the usual structure to model datatype constructors, and suggested that Moggi's categorical semantics of computations provides a solution to this problem.

In [Mog88b], Moggi has proposed the paradigm of a notion of computation, *i.e.* a feature of a real programming language such as partiality or side effects, as a monad over a category with structure.

Viewing, according to a long tradition in the subject, programs as closed λ-terms, this approach seeks to resolve the oversimplification introduced by using the λ-calculus with full $\beta\eta$ equality. This oversimplification has the effect of identifying a program of type $A \to B$ with a total function from the denotation of A to the denotation of B and so destroys important computational information.

In this chapter, we show how complexity can be viewed as a notion of computation. We show that the complexity categories of chapter 2 correspond to Moggi's models for a particular choice of monad.

In the main part of this chapter, we exploit this connection to study formal systems for reasoning about the denotational models that we develop. That is,

we develop formal systems for reasoning about programs with complexity. At first sight, it may seem paradoxical to develop formal systems to reason about models for programming languages which are themselves formal systems. A natural question is why one wants formal systems and not just the denotational models.

The denotational models were introduced in an attempt to reduce some of the arbitrary detail in the programming languages. However, the problems of specification and verification remain considerable for even reasonably simple programs. In order for it to be feasible to verify a program of any significant size, it is essential to use some form of machine assisted proof. However, this requires that we have a formal system for reasoning about the model. Ultimately, the hope would be to use a relatively simple formal system to reason about the models for a programming language and to infer properties of the programming language indirectly by means of a full abstraction result.

Other authors [FSZ89], [Niel84], [Shul90] have developed formal systems for reasoning about the complexity of programs. However, in each case, their systems are based on a single language and only consider natural number time complexity. Our system is based on a class of models and an arbitrary monoid of resource values. Therefore, it is applicable to a broad class of languages and many different types of complexity.

In [Mog88b], Moggi provided a formal system, the λ_c-calculus, for reasoning about programming languages with a notion of computation. The λ_c-calculus is intended to be sound and complete with respect to interpretation in λ_c-models (definition 3.2.8). In this chapter, we develop a calculus, the λ_{com}-calculus, for reasoning about programs with complexity.

In section 3.2, we describe Moggi's categorical semantics for computations and, in section 3.3, we show how our complexity categories may be viewed as a special case of his constructions.

In section 3.4, we define a language for computations and an interpretation of the language in any λ_c-model. We present Moggi's λ_c-calculus over the language for

computations. We define a notion of entailment by λ_c-models and quote Moggi's soundness and completeness results.

In section 3.5, we show how to express resource within the language for computations. We use this to define the λ_{com}-calculus over the language for computations. We define a notion of entailment by complexity models and prove a soundness result for the λ_{com}-calculus.

In section 3.6, we investigate the relationship between the λ_{com}-calculus and other systems. In particular, we show how the λ_{com}-calculus relates to the λ_c-calculus. We define an operational semantics for the language for computations and prove a correspondence between the λ_{com}-calculus and the operational semantics.

3.2 Categorical semantics of computations

3.2.1 Introduction

In [Mog88b], Moggi presents his categorical semantics of computations and the corresponding computational λ-calculus in an attempt to resolve the oversimplification introduced by using full $\beta\eta$ equality on closed lambda terms.

The idea is to distinguish, for each datatype, between the values of type A and the computations of type A. The intention is that a program of type $A \to B$ denotes a morphism from $[\![A]\!]$ to $T[\![B]\!]$ where $T[\![B]\!]$ is the type of computations of type B.

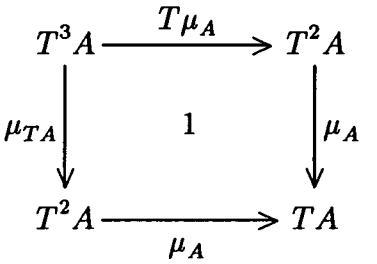
Moggi calls T a notion of computation. He gives several examples such as partiality, side effects and exceptions. In section 3.3, we show how complexity may be viewed as a notion of computation.

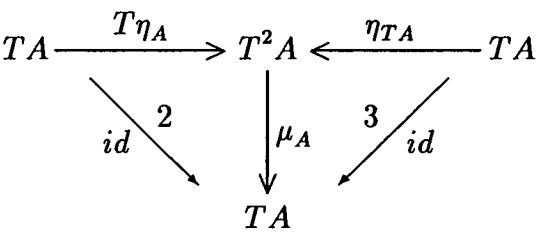
Moggi seeks properties common to a wide class of notions of computations. He imposes the requirement that programs should form a category and shows that this amounts to saying that T forms part of a monad $\langle T, \eta, \mu \rangle$ and that the category of programs is the Kleisli category for this monad.

We briefly review his constructions. Further details appear in [Mog88b].

3.2.2 Monads

Definition 3.2.1 A **monad** on a category $\mathcal{C}$ is a tuple $\langle T, \eta, \mu \rangle$ consisting of an endofunctor T on $\mathcal{C}$, a natural transformation $\eta : Id_{\mathcal{C}} \longrightarrow T$ and a natural transformation $\mu : T^2 \longrightarrow T$ such that the following diagrams:





commute.

Definition 3.2.2 We say a monad $\langle T, \eta, \mu \rangle$ on a category $\mathcal{C}$ satisfies the **monomorphism requirement** if η_A is a monomorphism for all A.

It is well known [Mac71] that any adjunction $\langle F, G, \eta, \epsilon \rangle : \mathcal{C} \rightarrow \mathcal{D}$ gives rise to a monad $\langle GF, \eta, G\epsilon F \rangle$ on $\mathcal{C}$ and furthermore, that any monad arises from an adjunction. This adjunction is not in general unique and there are two standard constructions, the Eilenberg-Moore construction [Mac71] and the Kleisli construction [Mac71] which, given a monad, return an adjunction defining the monad. Although most interest in category theory has centred on the Eilenberg-Moore construction, it is the Kleisli construction that concerns us here.

Definition 3.2.3 Let $\mathcal{C}$ be a category and let $\langle T, \eta, \mu \rangle$ be a monad on $\mathcal{C}$. The **Kleisli category for** T denoted $\mathcal{C}_T$ is given by:

- objects are objects of $\mathcal{C}$,
- a morphism from A to B in $\mathcal{C}_T$ is a morphism from A to TB in $\mathcal{C}$ and
- the composition of g and f in $\mathcal{C}_T$ is the morphism $\mu(Tg)f$ in $\mathcal{C}$.

The basic result relating categories with monads to their Kleisli categories is:

Proposition 3.2.4 Let $\mathcal{C}$ be a category and let $\langle T, \eta, \mu \rangle$ be a monad on $\mathcal{C}$. Then there exists an adjunction $F_T \dashv G_T : \mathcal{C}_T \to \mathcal{C}$ such that the monad T arises from this adjunction.

3.2.3 Datatypes, strong monads and computational models

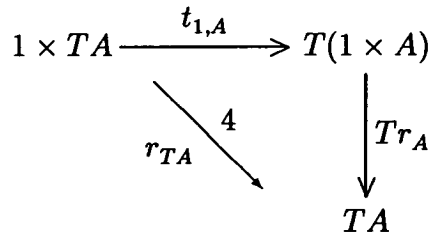
In chapter 2, we described how, in denotational semantics, a program of type $A \to B$ is denoted by a morphism from $[\![A]\!]$ to $[\![B]\!]$. We also mentioned that a product type $B \times C$ is denoted by the object $[\![B]\!] \times [\![C]\!]$. Therefore, a program p of type $A \to B \times C$ is denoted by a morphism from $[\![A]\!]$ to $[\![B]\!] \times [\![C]\!]$.

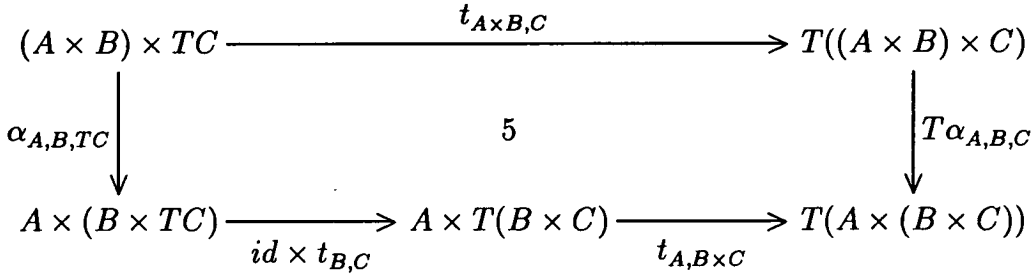
In Moggi's categorical semantics of computations, a program of type $A \to B$ is denoted by a morphism from $[\![A]\!]$ to $T[\![B]\!]$. Therefore, a program p of type $A \to B \times C$ should be denoted by a morphism from $[\![A]\!]$ to $T([\![B]\!] \times [\![C]\!])$. However, the usual pairing operation in a category with finite products takes a morphism from $[\![A]\!]$ to $T[\![B]\!]$ and a morphism from $[\![A]\!]$ to $T[\![C]\!]$ to a morphism from $[\![A]\!]$ to $T[\![B]\!] \times T[\![C]\!]$ and not a morphism from $[\![A]\!]$ to $T([\![B]\!] \times [\![C]\!])$. A similar problem arises with functional types.

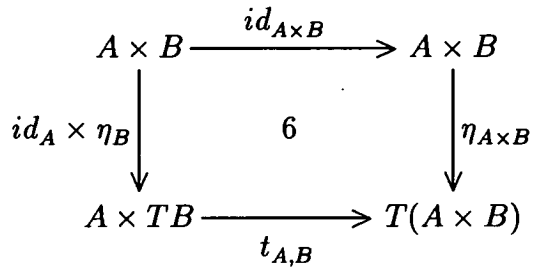
In order to overcome these problems, Moggi introduces the following notion [Koc72].

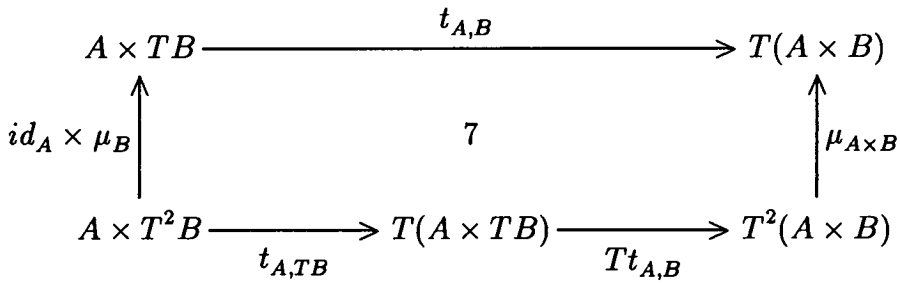
Definition 3.2.5 A **strong monad** on a category $\mathcal{C}$ with finite products, is a tuple $\langle T, \eta, \mu, t \rangle$ where $\langle T, \eta, \mu \rangle$ is a monad on $\mathcal{C}$ and t is a natural transformation

with components $t_{A,B} : A \times TB \to T(A \times B)$ such that the following diagrams:









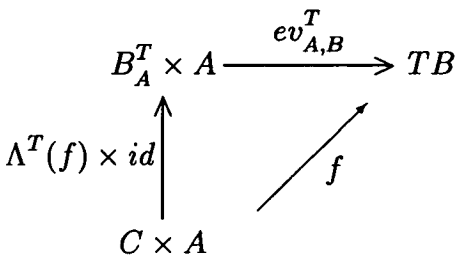
commute, where r and α are the evident natural isomorphisms.

Notation 3.2.6 The natural transformation t is called a tensorial strength for the monad $\langle T, \eta, \mu \rangle$.

The tensorial strength allows us to define a pairing operation with which we can interpret product types in the following way. Let $\langle T, \eta, \mu, t \rangle$ be a strong monad on

a category $\mathcal{C}$ with finite products. Then $\psi_{A,B} = \mu_{A,B}T^2\beta_{B,A}Tt_{B,A}T\beta_{TA,B}t_{TA,B}$ is a natural transformation $\psi : T(-) \times T(-) \to T(- \times -)$ where β is the twist map.

Definition 3.2.7 Let $\mathcal{C}$ be a category with finite products and let $\langle T, \eta, \mu, t\rangle$ be a strong monad on $\mathcal{C}$. A T-**exponential** is a representation of the functor $\mathcal{C}_T(- \times A, B) : \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$, *i.e.* a pair $\langle B_T^A, ev_{A,B}^T : (B_A^T \times A \to TB)\rangle$ with the universal property that for any $f : C \times A \longrightarrow TB$, there exists a unique morphism $\Lambda^T(f) : C \longrightarrow B_T^A$ such that the following diagram:



commutes.

The T-exponential is Moggi's analogue of the exponential object in a cartesian closed category. It is used to interpret functional types.

Definition 3.2.8 A λ_c-**model** is a pair $\langle \mathcal{C}, \mathcal{T}\rangle$ consisting of a category with finite products $\mathcal{C}$ and a strong monad $\mathcal{T} = \langle T, \eta, \mu, t\rangle$ on $\mathcal{C}$ together with a distinguished T-exponential B_T^A for each pair of objects A and B of $\mathcal{C}$.

3.3 The monad for complexity

3.3.1 Introduction

We show how our complexity categories may be viewed as a special case of Moggi's constructions. We discuss the application of this to modelling datatype constructors in complexity categories.

3.3.2 The monad for complexity

Proposition 3.3.1 Let $\mathcal{C}$ be a category with finite products and let $\mathcal{M} = \langle M, 0, \cdot \rangle$ be a monoid in $\mathcal{C}$. Then the following data:

- $T^M(-) = M \times -$,
- $\eta^M_A = A \xrightarrow{\cong} 1 \times A \xrightarrow{\langle 0, id \rangle} M \times A$,
- $\mu^M_A = M \times (M \times A) \xrightarrow{\langle \cdot, id \rangle} M \times A$ and
- $t^M_{A,B} = A \times (M \times B) \xrightarrow{\alpha} M \times (A \times B)$

define a strong monad on $\mathcal{C}$.

Proof: It is well known [LaSc86] that the functor "$M \times -$" forms part of a monad on $\mathcal{C}$ with η and μ given as above. It is routine to verify that t^M is a tensorial strength for this monad. □

Notation 3.3.2 We call this the **strong monad for complexity** and we write $\mathcal{C}_M$ for the Kleisli category corresponding to this strong monad.

The next result is the crucial one which tells us that our construction of $\mathcal{C}^+$ is in fact precisely the Kleisli category for this strong monad.

Proposition 3.3.3 Let $\mathcal{C}$ be a category with finite products and let $\mathcal{M} = \langle M, 0, \cdot \rangle$ be a monoid in $\mathcal{C}$. Then the category $\mathcal{C}^+$ is isomorphic to the category $\mathcal{C}_M$.

Proof: Let A and B be objects of $\mathcal{C}$. A morphism in $\mathcal{C}_M$ between A and B is a morphism $\langle t, f \rangle : A \longrightarrow M \times B$ in $\mathcal{C}$. Therefore, it is clear that the following assignment:

- $A \longmapsto A$ and
- $\langle t, f \rangle \longmapsto \langle f, t \rangle$

defines a bijection ι between $\mathcal{C}_M$ and $\mathcal{C}^+$. It is straightforward to verify that ι is a functor. □

An important application of this result is that Moggi provides categorical structures to model datatype constructors. In chapter 2, we saw that the category $\mathcal{C}^+$ does not have the usual structure to model datatype constructors, even when the category $\mathcal{C}$ does. However, it is not clear that we want the usual categorical structure. For example, Let A and B be objects of $\mathcal{C}^+$. Then to compute an element $\langle a, b\rangle$ of $A \times B$, one has to compute the value a and the value b. If we assume that there is no cost for the paring operation then the amount of resource required to compute $\langle a, b\rangle$ should be the amount required to compute a together with the amount required to compute b. Therefore, the complexity of a pair $\langle p, q\rangle$ should be given by:

$$cx(\langle p, q\rangle) \;=\; cx(p) \cdot cx(q).$$

It is not hard to show that the notion of pairing that arises in Moggi's work gives us precisely this construction. Observe that even if there is a cost for the pairing operation, such as for record types in ML, we can define 'pairing' in terms of this zero-cost pairing operation. Thus, although we do not have the usual categorical structures which model datatype constructors, we do have a framework in which to construct suitable analogies.

3.4 The λ_c-calculus

3.4.1 Introduction

The λ_c-calculus is a version of the typed lambda calculus [Bar84] with product types and a let constructor. The calculus enables us to derive sequents of the form $\Gamma \vdash A$ where Γ is a context, *i.e.* an assignment of types to variables, and A is a formula. There are two basic judgements, existence ($\downarrow$) and equivalence ($\equiv$).

In this section, we define a signature for computations. We then present the language for computations, which is parametric in a signature. We give an interpretation of the language for computations in any λ_c−model and then present the rules of the λ_c-calculus. We state soundness and completeness results for the λ_c-calculus with respect to interpretation in λ_c−models.

3.4.2 A language for computation

Definition 3.4.1 A **signature for computations** Σ consists of:

- A set type(Σ) of base types,
- for each sequence $\tau_1, ..., \tau_n, \tau \in \text{type}(\Sigma)$ with $n \geq 1$, a set $\text{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$ of function symbols,
- for each $\tau \in \text{type}(\Sigma)$, a set $\text{Const}_\Sigma(\tau)$ of constant symbols of type τ and

Definition 3.4.2 Let Σ be a signature for computations. The types of the **language for computation** are given by the following formation rules:

$$\frac{}{\vdash 1 : type}$$

$$\frac{}{\vdash \sigma : type} \; \sigma \in \text{type}(\Sigma)$$

$$\frac{\vdash \sigma : type \quad \vdash \tau : type}{\vdash \sigma \times \tau : type}$$

$$\frac{\vdash \sigma : type \quad \vdash \tau : type}{\vdash \sigma \to \tau : type}$$

$$\frac{\vdash \sigma : type}{\vdash T\sigma : type}$$

Definition 3.4.3 A **context** consists of an assignment of types to a finite set of variables.

Notation 3.4.4 We write contexts either as $x_1 : \tau_1, ..., x_n : \tau_n$ or as Γ. Note that since a context $x_1 : \tau_1, ..., x_n : \tau_n$ is a set, the order is not significant.

Definition 3.4.5 Let Σ be a signature for computations. The terms of the language for computation are given by the following formation rules:

$$\text{var}\ \frac{}{x_1 : \tau_1, ..., x_n : \tau_n \vdash x_i : \tau_i}$$

$$\text{cst}\ \frac{}{\Gamma \vdash c : \tau}\ c \in \text{Const}_\Sigma(\tau)$$

$$\text{fun}\ \frac{\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n}{\Gamma \vdash f(e) : \tau}\ f \in \text{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$$

$$\text{let}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let}\ x = e_1\ \text{in}\ e_2) : \tau_2}$$

$$*\ \frac{}{\Gamma \vdash * : 1}$$

$$\text{pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$\pi\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(e) : \tau_i}$$

$$\mu\ \frac{\Gamma \vdash e : T\tau}{\Gamma \vdash \mu(e) : \tau}$$

$$[] \frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] : T\tau}$$

$$\lambda \frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\lambda x_1 : \tau_1 . e_2) : \tau_1 \to \tau_2}$$

$$\text{app} \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1 \to \tau_2}{\Gamma \vdash e_2(e_1) : \tau_2}$$

Definition 3.4.6 Let Σ be a signature for computations. The sequents **Seq** of the language for computation are constructed from the terms by the following formation rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \downarrow \tau \in \mathbf{Seq}}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \equiv e_2 : \tau \in \mathbf{Seq}}$$

Notation 3.4.7 We write $\mathcal{L}(\Sigma)$ for the language for computations over the signature Σ

Definition 3.4.8 Let e be a term of the language for computations $\mathcal{L}(\Sigma)$. The **free variables** of e, $FV(e)$ are defined inductively by the following rules:

- $FV(x) = \{x\}$,
- $FV(c) = \emptyset$,
- $FV(f(e)) = FV(e)$,
- $FV(\text{let } x{=}e_1 \text{ in } e_2) = FV(e_1) \cup (FV(e_2) - \{x\})$,
- $FV(*) = \emptyset$,

- $FV(\langle e_1, e_2 \rangle) = FV(e_1) \cup FV(e_2)$,
- $FV(\pi(e)) = FV(e)$,
- $FV(\mu(e)) = FV(e)$,
- $FV([e]) = FV(e)$,
- $FV(\lambda x.e) = FV(e) - \{x\}$ and
- $FV(e_1(e_2)) = FV(e_1) \cup FV(e_2)$.

Definition 3.4.9 A term e of the language for computations $\mathcal{L}(\Sigma)$ is **closed** if $FV(e) = \emptyset$.

3.4.3 Interpretation of the language

In section 3.2, we described Moggi's categorical semantics of computations [Mog88b]. We mentioned that the idea is to distinguish, for each datatype A, between the values of type A and the computations of type A. The intention is that a program of type $A \to B$ should be denoted by a morphism from $[\![A]\!]$ to $T[\![B]\!]$ where $T[\![B]\!]$ is the type of computations of type B. The λ_c-calculus is an application of this work.

In this section we describe an interpretation of the language for computations in a λ_c-model (definition 3.2.8). This interpretation is parametric in an interpretation of the base types and function symbols of the calculus as, respectively, objects and morphisms of the Kleisli category of the λ_c-model. The interpretation of a type $T\tau$ is given by $T[\![\tau]\!]$ and the interpretation of a term $x_1 : \tau_1, ..., x_n : \tau_n \vdash e : \tau$ is a morphism from $\times[\![\tau_i]\!]$ to $[\![\tau]\!]$ in the Kleisli category of the λ_c-model. This is consistent with the idea that a program of type $A \to B$, *i.e.* a term $x : A \vdash e : B$ of the language of computations, should denote a morphism from $[\![A]\!]$ to $T[\![B]\!]$.

Notation 3.4.10 Let Γ be the context $x_1 : \tau_1, ..., x_n : \tau_n$. Then we write $[\![\Gamma]\!]$ for $[\![\tau_1]\!] \times ... \times [\![\tau_n]\!]$.

Definition 3.4.11 Let $\langle \mathcal{C}, \mathcal{T} \rangle$ be a λ_c-model and let Σ be a signature for computations. An **interpretation** of the language for computations in $\mathcal{C}_T$ is a pair of functions $[\![-]\!] : types \to |\mathcal{C}_T|$ and $[\![-]\!] : Terms \to \mathcal{C}_T$ such that:

- $[\![\sigma \times \tau]\!] = [\![\sigma]\!] \times [\![\tau]\!]$,
- $[\![\sigma \to \tau]\!] = [\![\tau]\!]_T^{[\![\sigma]\!]}$,
- $[\![T\tau]\!] = T[\![\tau]\!]$ and
- $[\![1]\!] = 1$,

and:

- $[\![f]\!]$ is a morphism from $[\![\tau_1]\!] \times ... \times [\![\tau_n]\!]$ to $[\![\tau]\!]$ in $\mathcal{C}_T$ for each function symbol $f \in \mathrm{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$,
- $[\![c]\!]$ is a morphism from 1 to $[\![\tau]\!]$ in $\mathcal{C}_T$ for each $c \in \mathrm{Const}_\Sigma(\tau)$,
- $$\frac{[\![\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n]\!] = g}{[\![\Gamma \vdash f(e) : \tau]\!] = \mu_{[\![\tau]\!]} T([\![f]\!])g} \quad f \in \mathrm{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$$
- $[\![x_1 : \tau_1, ..., x_n : \tau_n \vdash x_i : \tau_i]\!] = \eta_{[\![\tau_i]\!]} \pi_i$,
- $$\frac{[\![\Gamma \vdash e_1 : \tau_1]\!] = f \quad [\![\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2]\!] = g}{[\![\Gamma \vdash \text{let } x_1 = e_1 \text{ in } e_2 : \tau_2]\!] = \mu_{[\![\tau_2]\!]} T(g) t_{[\![\Gamma]\!],[\![\tau_1]\!]} \langle id_{[\![\Gamma]\!]}, f \rangle},$$
- $[\![\Gamma \vdash * : 1]\!] = \eta_1 !_{[\![\tau]\!]}$
  where $!_{[\![\Gamma]\!]}$ is the unique morphism in $\mathcal{C}$ from $[\![\Gamma]\!]$ to 1,
- $$\frac{[\![\Gamma \vdash e_1 : \tau_1]\!] = f \quad [\![\Gamma \vdash e_2 : \tau_2]\!] = g}{[\![\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2]\!] = \psi_{[\![\tau_1]\!],[\![\tau_2]\!]} \langle f, g \rangle},$$
- $$\frac{[\![\Gamma \vdash e : \tau_1 \times \tau_2]\!] = f}{[\![\Gamma \vdash \pi_i(e) : \tau_i]\!] = T(\pi_i) f},$$

- $$\frac{[\![\Gamma \vdash e : \tau]\!] \;=\; f}{[\![\Gamma \vdash [e] : T\tau]\!] \;=\; \eta_{T[\![\tau]\!]} f},$$

- $$\frac{[\![\Gamma \vdash e : T\tau]\!] \;=\; f}{[\![\Gamma \vdash \mu(e) : \tau]\!] \;=\; \mu_{[\![\tau]\!]} f},$$

- $$\frac{[\![\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2]\!] \;=\; f}{[\![\Gamma \vdash (\lambda x_1 : \tau_1 . e_1) : \tau_1 \to \tau_2]\!] \;=\; \eta_{[\![\tau_1 \to \tau_2]\!]} \Lambda^T_{[\![\tau_1]\!],[\![\tau_2]\!],[\![\Gamma]\!]}(f)}$$ and

- $$\frac{[\![\Gamma \vdash e_1 : \tau_1]\!] \;=\; f \quad [\![\Gamma \vdash e_2 : \tau_1 \to \tau_2]\!] \;=\; g}{[\![\Gamma \vdash e_2(e_1) : \tau_2]\!] \;=\; app_{[\![\tau_1]\!],[\![\tau_2]\!]}\langle f, g\rangle}$$

where $app_{A,B} : T(B_T^A) \times TA \to TB$ is $\mu_B T(eval^T_{A,B}) \psi_{B_T^A, A}$.

Remark 3.4.12 *An interpretation is fixed by giving an interpretation of the base types, and the constant and function symbols.*

The let constructor plays an important role in the interpretation. Computationally, the term $x_1 : \tau_1 \vdash$ (let $x_2 = e_1$ in e_2) corresponds to the sequential composition of the programs $x_1 : \tau_1 \vdash e_1$ and $x_2 : \tau_2 \vdash e_2$. Semantically, it corresponds to composition in the Kleisli category. This is consistent with the idea that composition of morphisms denoting programs should be composition in the Kleisli category.

3.4.4 The λ_c-calculus

In this section, we present the λ_c-calculus over the language for computations. The λ_c-calculus is a formal system for deriving sequents of the form $\Gamma \vdash A$ where Γ is a context and A is a formula. Here we shall be interested in the case where A is an atomic formula ($e_1 \equiv e_2 : \tau$ or $e \downarrow \tau$). For a discussion of the case where A is an arbitrary first order formula see [Mog88b].

The rules of the λ_c-calculus are as follows. We partition them into

- general rules,
- inference rules for let and computational (T) types,
- inference rules for product types and
- inference rules for functional types.

In each case, we assume that the context assigns suitable types to the free variables of each term.

Definition 3.4.13 Let Σ be a signature for computations. The rules of the λ_c-calculus are given by:

- General Rules

$$\text{E.x}\ \frac{}{\Gamma, x : \tau \vdash x \downarrow \tau}$$

$$\text{subst}\ \frac{\Gamma \vdash e \downarrow \tau \quad \Gamma, x : \tau \vdash A}{\Gamma \vdash A[x := e]}$$

where $A[x := e]$ is A with e substituted for each occurrence of x.

Rules to the effect that the equivalence $\equiv$ is a congruence relation.

- Rules for let and computational types

$$\text{id}\ \frac{}{\Gamma \vdash (\text{let } x = e \text{ in } x) \equiv e : \tau}$$

$$\text{comp}\ \frac{}{\Gamma \vdash (\text{let } x_2 = (\text{let } x_1 = e_1 \text{ in } e_2) \text{ in } e) \equiv (\text{let } x_1 = e_1 \text{ in } (\text{let } x_2 = e_2 \text{ in } e)) : \tau}$$

where x_1 does not occur free in e

$$\text{let}.\beta\ \frac{}{\Gamma \vdash (\text{let } x_1 = x_2 \text{ in } e) \equiv e[x_1 := x_2] : \tau}$$

$$\text{let.f}\ \frac{}{\Gamma \vdash f(e) \equiv (\text{let } x = e \text{ in } f(x)) : \tau}$$

$$\text{E.[-]}\ \frac{}{\Gamma \vdash [e] \downarrow T\tau}$$

$$\text{let.}\mu\ \frac{}{\Gamma \vdash \mu(e) \equiv (\text{let } x = e \text{ in } \mu(x)) : \tau}$$

$$\text{T.}\beta\ \frac{}{\Gamma \vdash \mu([e]) \equiv e : \tau}$$

$$\text{T.}\eta\ \frac{}{\Gamma \vdash [\mu(x)] \equiv x : T\tau}$$

- Rules for unit and product types

$$\text{E.}*\ \frac{}{\Gamma \vdash * \downarrow 1}$$

$$1.\eta\ \frac{}{\Gamma \vdash * \equiv x : 1}$$

$$\text{E.}\langle - \rangle\ \frac{}{\Gamma \vdash \langle x_1, x_2 \rangle \downarrow \tau_1 \times \tau_2}$$

$$\text{let.}\langle - \rangle\ \frac{}{\Gamma \vdash \langle e_1, e_2 \rangle \equiv (\text{let } x_1, x_2 = e_1, e_2 \text{ in } \langle x_1, x_2 \rangle) : \tau_1 \times \tau_2}$$

where "let $x_1, x_2 = e_1, e_2$ in e" is an abbreviation for "let $x_1 = e_1$ in (let $x_2 = e_2$ in e)"

$$\text{E.}\pi\ \frac{}{\Gamma \vdash \pi_i(x) \downarrow \tau_i}$$

$$\text{let}.\pi \ \frac{}{\Gamma \vdash \pi_i(e) \equiv (\text{let } x = e \text{ in } \pi_i(x)) : \tau_i}$$

$$\times.\beta \ \frac{}{\Gamma \vdash \pi_i(\langle x_1, x_2 \rangle) \equiv x_i : \tau_i}$$

$$\times.\eta \ \frac{}{\Gamma \vdash \langle \pi_1(x), \pi_2(x) \rangle \equiv x : \tau_1 \times \tau_2}$$

- Rules for functional types

$$\text{E}.\lambda \ \frac{}{\Gamma \vdash (\lambda x : \tau_1.e : \tau_2) \downarrow \tau_1 \to \tau_2}$$

$$\text{let.app} \ \frac{}{\Gamma \vdash e_1(e_2) \equiv (\text{let } x_1, x_2 = e_1, e_2 \text{ in } x_1(x_2)) : \tau_2}$$

$$\beta \ \frac{}{\Gamma \vdash (\lambda x : \tau_1.e : \tau_2)(x) \equiv e : \tau_2}$$

$$\eta \ \frac{}{\Gamma \vdash (\lambda x_1 : \tau_1.x(x_1)) \equiv x : \tau_1 \to \tau_2}$$

Notation 3.4.14 We write $\Gamma \vdash_{\mathcal{F}} A$ if the sequent $\Gamma \vdash A$ is derivable in the formal system $\mathcal{F}$.

It is worth saying a few words about the two judgements of the λ_c-calculus. The equivalence predicate $\equiv$ is the familiar notion of program equivalence, except that equivalence in the λ_c-calculus is intended to reflect our notion of computation. Accordingly, equivalence is to be interpreted by equality of morphisms in the Kleisli category for a strong monad.

The existence predicate $\downarrow$ is derived from the termination predicate in the logic of partial terms [Mog88a]. It is intended to reflect a program that does not require any "computation". Moggi defines values to be those terms e for which $\Gamma \vdash e \downarrow$ is derivable. Thus, for example, a value for partial computations is a terminating program, a value for side-effects is a program with no side effects and in our case, a value should be a program that consumes no resource.

The following definition is intended to capture these intuitions.

Definition 3.4.15 Let $\langle \mathcal{C}, \mathcal{T} \rangle$ be a λ_c–model and let $[\![-]\!]$ be an interpretation. We say $\mathcal{C}_T$ **entails** $\Gamma \vdash e_1 \equiv e_2 : \tau$ if:

$$[\![\Gamma \vdash e_1 : \tau]\!] \;=\; [\![\Gamma \vdash e_2 : \tau]\!].$$

We say $\mathcal{C}_T$ **entails** $\Gamma \vdash e \downarrow \tau$ if $[\![\Gamma \vdash e : \tau]\!]$ factors through the unit map $\eta_{[\![\tau]\!]}$.

Notation 3.4.16 We write "$\mathcal{C}_T$ entails $\Gamma \vdash A$" as $\Gamma \models_{\mathcal{C}_T} A$.

We define notions of soundness and completeness for the λ_c-calculus with respect to interpretation in λ_c–models.

Definition 3.4.17 Let $\langle \mathcal{C}, \mathcal{T} \rangle$ be a λ_c–model and let $[\![-]\!]$ be an interpretation. $\langle \mathcal{C}, \mathcal{T} \rangle$ is **sound** for the λ_c-calculus with respect to $[\![-]\!]$ if for all formulae A, $\Gamma \vdash_{\lambda_c} A$ implies that $\Gamma \models_{\mathcal{C}_T} A$.

Definition 3.4.18 Let $\mathcal{M}$ be a collection of models for a formal system $\mathcal{F}$. We say $\mathcal{M}$ is **complete** for $\mathcal{F}$ if for all formulae A, $\Gamma \models_M A$ for each $M \in \mathcal{M}$ implies that $\Gamma \vdash_{\mathcal{F}} A$.

Moggi [Mog88b] states the following soundness and completeness results.

Theorem 3.4.19 Any λ_c–model is sound for the λ_c-calculus.

Theorem 3.4.20 The class of λ_c–models is complete for the λ_c-calculus.

3.5 Reasoning about resource

3.5.1 Introduction

The rules of the λ_c-calculus are intended to be correct for any notion of computation. That is, they are sound with respect to interpretation in the Kleisli category of any λ_c-model. When one considers a particular notion of computation, for example partial computations [Mog88a] or in our case complexity, we expect that extra rules will hold and that some extension may be required in order to express the particular features of interest to us. In our case, we need some way of expressing resource within the calculus.

In chapter 2, we showed how the basic approach to modelling complexity was to extend our semantics so that a program p of type $A \longrightarrow B$ is denoted by a pair of maps $fun(p) : [\![A]\!] \longrightarrow [\![B]\!]$ and $cx(p) : [\![A]\!] \longrightarrow M$. The map $fun(p)$ models the functional behaviour of p and the map $cx(p)$ models the complexity of p.

A calculus for reasoning about the denotational models we have developed should reflect this approach. In particular, we need to express the functional and complexity parts of each program.

The notion of value in the λ_c-calculus suggests a way of expressing the functional part of each program since, for the monad for complexity, a value is a term which requires no resource. Thus our requirements are a way of expressing resource within the calculus and a predicate which expresses the fact that a term has a certain resource requirement and a certain functional behaviour.

In this section, we show how the terms of unit type can express resource. We use this observation and the above considerations to motivate the definition of the λ_{com}-calculus. This is a calculus for reasoning about complexity.

A natural approach would be to define the λ_{com}-calculus as an extension of the λ_c-calculus. We chose not to do this for the following reason. In chapter 5, we show that the class of models given in this chapter does not capture all the

examples of interest to us, and we define a broader class of models. Unfortunately, it is not possible to extend the interpretation of the λ_c-calculus to this larger class of models. However, it is possible to extend the interpretation of the λ_{com}-calculus to this larger class. We will discuss this point in more detail in chapters 5 and 9.

3.5.2 The λ_{com}-calculus

Our aim is to express resource within the language for computations and use this to split each term into its functional and complexity parts. The following observation suggests that we can use the terms of unit type to model resource.

Lemma 3.5.1 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a complexity model, let Σ be a signature for computations and let $[\![-]\!]$ be an interpretation of $\mathcal{L}(\Sigma)$ in $\mathcal{C}_M$. Let t be a closed term of type 1. Then $[\![\emptyset \vdash t : 1]\!]$ is a global element of M in $\mathcal{C}$.

Proof: $[\![\emptyset \vdash t : 1]\!]$ is a morphism $[\![t]\!]$ from $[\![\emptyset]\!]$ to $[\![1]\!]$ in the Kleisli category $\mathcal{C}_M$. However, $[\![1]\!] = 1$ and $[\![\emptyset]\!] = \times\emptyset = 1$. Thus $[\![t]\!]$ is a morphism from 1 to 1 in $\mathcal{C}_M$. That is, a morphism from 1 to $M \times 1 \cong M$ in $\mathcal{C}$. □

In particular, constants of unit type correspond to (global) elements of M. Of course, this alone does not suffice since we also need to model the composition of programs. In chapter 2, we showed that to model composition, we require a monoid of resource values and not simply a set. Therefore, we require some way of composing resources within our calculus.

In order to do this, we recall the slogan that "let is interpreted as composition of programs". The following result shows the sense in which this captures the idea of putting complexities together.

Lemma 3.5.2 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a complexity model, let Σ be a signature for computations and let $[\![-]\!]$ be an interpretation of $\mathcal{L}(\Sigma)$ in $\mathcal{C}_M$. Let $\emptyset \vdash t_1 : 1$ and $x : 1 \vdash t_2 : 1$ be terms of the language for computations such that $x \notin FV(t_2)$.

Then $[\![\emptyset \vdash \text{let } x{=}t_1 \text{ in } t_2]\!]$ is the morphism:

$$[\![\emptyset \vdash t_1 : 1]\!] \cdot [\![\emptyset \vdash t_2 : 1]\!] : 1 \longrightarrow M.$$

Proof: By definition 3.4.11,

$$\begin{aligned}
[\![\emptyset \vdash \text{let } x{=}t_1 \text{ in } t_2]\!] &= \mu_{[\![1]\!]} T([\![t_2]\!]) t_{[\![\emptyset]\!],[\![1]\!]} \langle id_{[\![\emptyset]\!]}, [\![t_1]\!] \rangle \\
&= \mu_1 T([\![t_2]\!]) t_{1,1} \langle id_1, [\![t_1]\!] \rangle \\
&= \cdot \langle [\![t_2]\!], id_M \rangle \langle id_1, [\![t_1]\!] \rangle \\
&= [\![t_1]\!] \cdot [\![t_2]\!].
\end{aligned}$$

as required. □

This observation motivates using the term "let $x = t_1$ in t_2" to represent the composition of t_1 and t_2.

Notation 3.5.3 Let t_1 and t_2 be terms of unit type. We write $t_1 \cdot t_2$ for the term "let $x{=}t_1$ in t_2" where x is any variable $x : 1$ such that $x \notin FV(t_1) \cup FV(t_2)$.

The following result shows that in the λ_c-calculus, the terms of unit type inherit a monoid structure.

Proposition 3.5.4 Let Σ be a signature for computations. Then the following sequents are derivable in the λ_c-calculus:

- $\Gamma \vdash * \cdot t \equiv t : 1$,
- $\Gamma \vdash t \cdot * \equiv t : 1$ and
- $\Gamma \vdash t_1 \cdot (t_2 \cdot t_3) \equiv (t_1 \cdot t_2) \cdot t_3 : 1$.

Proof: Recall that $t_1 \cdot t_2$ stands for let $x{=}t_1$ in t_2 where x is any variable $x : 1$ such that $x \notin FV(t_1) \cup FV(t_2)$. The sequents are derived as follows:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \text{let } x{=}y \text{ in } t \equiv t[x := y]}}{\Gamma \vdash \text{let } x{=}y \text{ in } t \equiv t} \qquad \dfrac{}{\Gamma \vdash \text{let } x{=}y \text{ in } t \equiv \text{let } x{=}* \text{ in } t}}{\dfrac{\Gamma \vdash \text{let } x{=}* \text{ in } t \equiv t}{\Gamma \vdash * \cdot t \equiv t}}$$

secondly,

$$\frac{\dfrac{}{\Gamma \vdash \text{let } x{=}t \text{ in } * \equiv \text{let } x{=}t \text{ in } x}}{\Gamma \vdash \text{let } x{=}t \text{ in } * \equiv t}$$
$$\Gamma \vdash t \cdot * \equiv t$$

and finally,

$$\frac{}{\Gamma \vdash \text{let } x_1{=}t_1 \text{ in let } x_2{=}t_2 \text{ in } t_3 \equiv \Gamma \vdash \text{let } x_2{=}\text{let } x_1{=}t_1 \text{ in } t_2 \text{ in } t_3}$$
$$\Gamma \vdash t_1 \cdot (t_2 \cdot t_3) \equiv (t_1 \cdot t_2) \cdot t_3 : 1$$

□

Remark 3.5.5 *Note that, for every monad T, the terms of unit type have this monoid structure. The additional condition which classifies the complexity models is that $[\![T\tau]\!]$ is $M \times [\![\tau]\!]$.*

We can now represent both functional behaviour, using values, and resource, using terms of unit type, within the language for computations. However, we require some additional structure in order to give a calculus for reasoning about programs with complexity.

Firstly, for each type τ we require a predicate $\langle -,-,- \rangle \in \mathrm{Pred}_\Sigma(\tau,\tau,1)$ where the intended meaning of $\langle e,v,t \rangle$ is that the term e has functional behaviour v and complexity t. Secondly, for each of the basic constants and basic function symbols, we need to add new constants which are intended to express their functional and complexity parts. Thus for each constant symbol $c \in \mathrm{Const}_\Sigma(\tau)$, we require additional constants $v_c \in \mathrm{Const}_\Sigma(\tau)$ and $t_c \in \mathrm{Const}_\Sigma(1)$ such that $\langle c, v_c, t_c \rangle$ holds. The idea is that t_c is the resource required to evaluate the constant c and v_c is the value which results.

For the function symbols, the situation is a little more complicated. For each basic function symbol $f \in \mathrm{Funct}_\Sigma(\tau_1,...,\tau_n,\tau)$, we require two additional function symbols $v_f \in \mathrm{Funct}_\Sigma(\tau_1,...,\tau_n,\tau)$ and $t_f \in \mathrm{Funct}_\Sigma(\tau_1,...,\tau_n,1)$ such that

$\langle f(x), v_f(x), t_f(x)\rangle$ holds. The idea is that $t_f(v)$ is the cost of evaluating f at the value v and $v_f(v)$ is the value which results. For example, suppose f were the symbol $+$ in standard ML. Then for each x, y of type int, $v_+(x,y)$ would be the value to which $x+y$ evaluates and $t_+(x,y)$ would be the resource required to evaluate it.

These considerations motivate the following definition of a language for complexity and the λ_{com}-calculus as a formal system over the language for complexity.

Definition 3.5.6 Let Σ be a signature for computations. The terms of the language for complexity are generated by the formation rules for $\mathcal{L}(\Sigma)$ together with:

$$v_c \frac{}{\Gamma \vdash v_c : \tau} \; c \in \mathrm{Const}_\Sigma(\tau)$$

$$t_c \frac{}{\Gamma \vdash t_c : 1} \; c \in \mathrm{Const}_\Sigma(\tau)$$

$$v_f \frac{\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n}{\Gamma \vdash v_f(e) : \tau} \; f \in \mathrm{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$$

$$t_f \frac{\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n}{\Gamma \vdash t_f(e) : 1} \; f \in \mathrm{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$$

Definition 3.5.7 Let Σ be a signature for computations. The sequents **Seq** of the language for complexity are constructed from the terms by the following formation rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \downarrow \tau \in \mathbf{Seq}}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \equiv e_2 : \tau \in \mathbf{Seq}}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash t : 1}{\Gamma \vdash \langle e_1, e_2, t \rangle : \tau \in \mathbf{Seq}}$$

We now define the λ_{com}-calculus over the language for computation. We partition the rules as follows:

- rules for the $\downarrow$ predicate,
- rules for the $\equiv$ predicate and
- "triple" rules for the $\langle -, -, - \rangle$ predicate.

Our aim is to provide a calculus in which programs can be split into their functional and complexity parts. Thus, whilst we retain the two judgements, existence ($\downarrow$) and equality ($\equiv$), of the λ_c-calculus, they are no longer central. Instead, the last section of rules is the one of most interest. They derive formulae of the form $\Gamma \vdash \langle e, v, t \rangle : \tau$ whose intended meaning is that the term e has functional behaviour v and complexity t. Thus, for example, the rule *pair* expresses the fact that if e_1 and e_2 have, respectively, functional behaviour v_1 and v_2, and complexity t_1 and t_2, then the term $\langle e_1, e_2 \rangle$ has functional behaviour $\langle v_1, v_2 \rangle$ and complexity $t_1 \cdot t_2$. The rule π expresses the fact that if $e : \tau_1 \times \tau_2$ has functional behaviour v and complexity t, then the term $\pi_i(e) : \tau_i$ has functional behaviour $\pi_i(v)$ and the same complexity t. This approach is consistent with the aim of studying a notion of program equivalence in which two programs are equivalent when they have the same functional behaviour and the same complexity.

Definition 3.5.8 Let Σ be a signature for computations. The rules of the λ_{com}-calculus are given by:

- Rules for the $\downarrow$ predicate

$$\text{E.x} \ \frac{}{\Gamma \vdash x \downarrow \tau}$$

$$\text{E.}[\text{-}]\ \frac{}{\Gamma \vdash [e] \downarrow T\tau}$$

$$\text{E.}*\ \frac{}{\Gamma \vdash * \downarrow 1}$$

$$\text{E.}\langle - \rangle\ \frac{\Gamma \vdash v_1 \downarrow \tau_1 \quad \Gamma \vdash v_2 \downarrow \tau_2}{\Gamma \vdash \langle v_1, v_2 \rangle \downarrow \tau_1 \times \tau_2}$$

$$\text{E.}\pi\ \frac{\Gamma \vdash v \downarrow \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(v) \downarrow \tau_i}$$

$$\text{E.}\lambda\ \frac{}{\Gamma \vdash (\lambda x : \tau_1 . e : \tau_2) \downarrow \tau_1 \to \tau_2}$$

$$\text{E.}c\ \frac{}{\Gamma \vdash v_c \downarrow \tau}\ c \in \text{Const}_\Sigma(\tau)$$

$$\text{E.}f\ \frac{\Gamma \vdash v \downarrow \tau_1 \times \cdots \times \tau_n}{\Gamma \vdash v_f(v) \downarrow \tau}\ f \in \text{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$$

$$\text{E.}\xi\ \frac{\Gamma \vdash v_1 \downarrow \tau \quad \Gamma \vdash v_1 \equiv v_2 : \tau}{\Gamma \vdash v_2 \downarrow \tau}$$

- Rules for the $\equiv$ predicate

$$\text{refl}\ \frac{}{\Gamma \vdash e \equiv e : \tau}$$

$$\text{symm}\ \frac{\Gamma \vdash e_1 \equiv e_2 : \tau}{\Gamma \vdash e_2 \equiv e_1 : \tau}$$

$$\text{trans}\ \frac{\Gamma \vdash e_1 \equiv e_2 : \tau \quad \Gamma \vdash e_2 \equiv e_3 : \tau}{\Gamma \vdash e_1 \equiv e_3 : \tau}$$

$$\text{subst}\ \frac{\Gamma \vdash e \downarrow \tau \quad \Gamma, x : \tau \vdash A}{\Gamma \vdash A[x := e]}$$

where $A[x := e]$ is A with e substituted for each occurrence of x.

$$\text{let}.\xi\ \frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) \equiv (\text{let } x = e_1' \text{ in } e_2') : \tau_2}$$

$$\langle - \rangle.\xi\ \frac{\Gamma \vdash e_1 \equiv e_1' : \tau_1 \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \equiv \langle e_1', e_2' \rangle : \tau_1 \times \tau_2}$$

$$\pi.\xi\ \frac{\Gamma \vdash e_1 \equiv e_2 : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(e_1) \equiv \pi_i(e_2) : \tau_i}$$

$$\lambda.\xi\ \frac{\Gamma, x : \tau_1 \vdash e_1 \equiv e_2 : \tau_2}{\Gamma \vdash \lambda x.e_1 \equiv \lambda x.e_2 : \tau_1 \to \tau_2}$$

$$\mu.\xi\ \frac{\Gamma \vdash e_1 \equiv e_2 : T\tau}{\Gamma \vdash \mu(e_1) \equiv \mu(e_2) : \tau}$$

$$[-].\xi\ \frac{\Gamma \vdash e_1 \equiv e_2 : \tau}{\Gamma \vdash [e_1] \equiv [e_2] : T\tau}$$

$$\text{id}\ \frac{}{\Gamma \vdash (\text{let } x = e \text{ in } x) \equiv e : \tau}$$

$$\text{comp}\ \frac{}{\Gamma \vdash (\text{let } x_2 = (\text{let } x_1 = t_1 \text{ in } t_2) \text{ in } t) \equiv (\text{let } x_1 = t_1 \text{ in } (\text{let } x_2 = t_2 \text{ in } t)) : \tau}$$

where x_1 does not occur free in e

$$\text{let.f}\ \frac{}{\Gamma \vdash f(e) \equiv (\text{let } x = e \text{ in } f(x)) : \tau}$$

$$\text{let.}\beta\ \frac{}{\Gamma \vdash (\text{let } x_1 = x_2 \text{ in } e) \equiv e[x_1 := x_2] : \tau}$$

$$\text{let.}\mu\ \frac{}{\Gamma \vdash \mu(e) \equiv (\text{let } x = e \text{ in } \mu(x)) : \tau}$$

$$\text{T.}\beta\ \frac{}{\Gamma \vdash \mu([e]) \equiv e : \tau}$$

$$\text{T.}\eta\ \frac{}{\Gamma \vdash [\mu(x)] \equiv x : T\tau}$$

$$1.\eta\ \frac{}{\Gamma \vdash * \equiv x : 1}$$

$$\text{let.}\langle - \rangle\ \frac{}{\Gamma \vdash \langle e_1, e_2 \rangle \equiv (\text{let } x_1, x_2 = e_1, e_2 \text{ in } \langle x_1, x_2 \rangle) : \tau_1 \times \tau_2}$$

$$\text{let.}\pi\ \frac{}{\Gamma \vdash \pi_i(e) \equiv (\text{let } x = e \text{ in } \pi_i(x)) : \tau_i}$$

$$\times.\beta\ \frac{}{\Gamma \vdash \pi_i(\langle x_1, x_2 \rangle) \equiv x_i : \tau_i}$$

$$\times.\eta\ \frac{}{\Gamma \vdash \langle \pi_1(x), \pi_2(x) \rangle \equiv x : \tau_1 \times \tau_2}$$

$$\beta\ \frac{}{\Gamma \vdash (\lambda x : \tau_1.e : \tau_2)(x) \equiv e : \tau_2}$$

$$\eta\ \frac{}{\Gamma \vdash (\lambda x_1 : \tau_1.x(x_1)) \equiv x : \tau_1 \to \tau_2}$$

- Rules for $\langle -, -, - \rangle$ predicate

$$\text{value}\ \frac{\Gamma \vdash v \downarrow \tau}{\Gamma \vdash \langle v, v, * \rangle : \tau}$$

$$\text{cst}\ \frac{}{\Gamma \vdash \langle c, v_c, t_c \rangle : \tau}\ c \in \text{Const}_\Sigma(\tau)$$

$$\text{fun}\ \frac{\Gamma \vdash \langle e, v, t \rangle : \tau_1 \times \cdots \times \tau_n}{\Gamma \vdash \langle f(e), v_f(v), t \cdot t_f(v) \rangle : \tau}\ f \in \text{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$$

$$\text{pair}\ \frac{\Gamma \vdash \langle e_1, v_1, t_1 \rangle : \tau_1 \quad \Gamma \vdash \langle e_2, v_2, t_2 \rangle : \tau_2}{\Gamma \vdash \langle \langle e_1, e_2 \rangle, \langle v_1, v_2 \rangle, t_1 \cdot t_2 \rangle : \tau_1 \times \tau_2}$$

$$\pi\ \frac{\Gamma \vdash \langle e, v, t \rangle : \tau_1 \times \tau_2}{\Gamma \vdash \langle \pi_i(e), \pi_i(v), t \rangle : \tau_i}$$

$$\mu\ \frac{\Gamma \vdash \langle e, [e'], t \rangle : T\tau \quad \Gamma \vdash \langle e', v', t' \rangle}{\Gamma \vdash \langle \mu(e), v', t \cdot t' \rangle : \tau}$$

$$\text{let}\ \frac{\Gamma \vdash \langle e_1, v_1, t_1 \rangle : \tau_1 \quad \Gamma \vdash \langle e_2[x := v_1], v_2, t_2 \rangle : \tau_2}{\Gamma \vdash \langle \text{let } x = e_1 \text{ in } e_2, v_2, t_1 \cdot t_2 \rangle : \tau_2}$$

$$\text{app}\ \frac{\Gamma \vdash \langle e_1, v_1, t_1 \rangle : \sigma \to \tau \quad \Gamma \vdash \langle e_2, v_2, t_2 \rangle : \sigma \quad \Gamma \vdash \langle v_1(v_2), v_3, t_3 \rangle : \tau}{\Gamma \vdash \langle e_1(e_2), v_3, t_1 \cdot t_2 \cdot t_3 \rangle : \tau}$$

$$\text{cong}\ \frac{\Gamma \vdash \langle e, v, t\rangle : \tau \quad \Gamma \vdash e' \equiv e : \tau \quad \Gamma \vdash v' \equiv v : \tau \quad \Gamma \vdash t' \equiv t : 1}{\Gamma \vdash \langle e', v', t'\rangle : \tau}$$

Proposition 3.5.9 If $\Gamma \vdash_{\lambda_{com}} \langle e, v, t\rangle : \tau$, then $\Gamma \vdash_{\lambda_{com}} v \downarrow \tau$.

Proof: We proceed by induction on the derivation.

- Base case: suppose that the last rule in the derivation was either *value* or *cst*. Then, in each case, it follows immediately from the axioms of the calculus that $\Gamma \vdash v \downarrow \tau$ is derivable.

- General case: Suppose that the last rule was *pair*. Then we have:

$$\frac{\begin{array}{cc}\dfrac{}{\begin{array}{c}\vdots\\ \Gamma \vdash \langle e_1, v_1, t_1\rangle\end{array}} & \dfrac{}{\begin{array}{c}\vdots\\ \Gamma \vdash \langle e_1, v_1, t_1\rangle\end{array}}\end{array}}{\Gamma \vdash \langle\langle e_1, e_2\rangle, \langle v_1, v_2\rangle, t_1 \cdot t_2\rangle}$$

  Then by the inductive hypothesis, there exist derivations:

$$\frac{}{\begin{array}{c}\vdots\\ \Gamma \vdash v_i \downarrow \tau_i\end{array}}$$

  and the following derivation:

$$\frac{\dfrac{}{\Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash \langle x_1, x_2\rangle \downarrow \tau_1 \times \tau_2} \quad \dfrac{}{\begin{array}{c}\vdots\\ \Gamma \vdash v_1 \downarrow \tau_1\end{array}}}{\Gamma, x_2 : \tau_2 \vdash \langle v_1, x_2\rangle \downarrow \tau_1 \times \tau_2} \quad \dfrac{}{\begin{array}{c}\vdots\\ \Gamma \vdash v_1 \downarrow \tau_1\end{array}}$$
$$\Gamma \vdash \langle v_1, v_2\rangle \downarrow \tau_1 \times \tau_2$$

shows that $\Gamma \vdash \langle v_1, v_2 \rangle \downarrow \tau_1 \times \tau_2$ is derivable in the λ_{com}-calculus.

Similarly if the last rule was *fun*, π, μ, *app* or *cong* the result follows easily.

Suppose the last rule were *let*. Then by the inductive hypothesis, there exist derivations:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash v_1 \downarrow \tau_1\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma, x : \tau_1 \vdash v_2 \downarrow \tau_2\end{array}}{}}{}$$

and the following derivation:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash v_1 \downarrow \tau_1\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma, x : \tau_1 \vdash v_2 \downarrow \tau_2\end{array}}{}}{}}{\Gamma \vdash v_2[x := v_1] \downarrow \tau_2}$$

shows that $\Gamma \vdash v_2[x := v_1] \downarrow \tau_2$ is derivable.

Finally, suppose that the last rule was *subst* so that we have:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, x : \tau_2 \vdash \langle e, v_1, t \rangle : \tau_1\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash v_2 \downarrow \tau_2\end{array}}{}}{}}{\Gamma \vdash \langle e, v_1, t \rangle[x := v_2]}$$

Then $\Gamma, x : \tau_2 \vdash v_1 \downarrow \tau_2$ by the inductive hypothesis and $\Gamma \vdash v_2 \downarrow \tau_2$ by assumption. Therefore, $\Gamma \vdash v_1[x := v_2] \downarrow \tau_2$ by applying the rule *subst* to the predicate $v_1 \downarrow \tau_2$. This completes the proof. □

We give some examples of derivations in the λ_{com}-calculus.

Example 3.5.10 Let Σ be a signature which includes a type *int*, basic constants $3, 4 \in \text{Const}_\Sigma(int)$ and basic function symbols $+, \times \in \text{Funct}_\Sigma(int, int, int)$. We

evaluate the denotation of the term $x : int \vdash +(\times(3,x),4)$. For notational convenience we shall omit the contexts ($x : int$ or $\emptyset$) from the sequents. For reasons of space, we perform the derivation in four parts.

$$\dfrac{\dfrac{\dfrac{}{\langle 3, v_3, t_3\rangle} \quad \dfrac{}{\langle x, x, *\rangle}}{\langle (3,x),(v_3,x),t_3 \cdot *\rangle} \quad \dfrac{}{t_3 \cdot * \equiv *}}{\langle (3,x),(v_3,x),t_3\rangle}$$

We now use the rule fun to evaluate $\times$ applied to $(3,x)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\quad}}{\quad .\quad .\quad .\quad}}{\langle (3,x),(v_3,x),t_3\rangle}}{\langle \times(3,x), v_\times(v_3,x), t_3 \cdot t_\times(v_3,x)\rangle}$$

Similarly, we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\quad}}{\quad .\quad .\quad .\quad}}{\langle \times(3,x), v_\times(v_3,x), t_3 \cdot t_\times(v_3,x)\rangle} \quad \dfrac{}{\langle 4, v_4, t_4\rangle}}{\langle \langle \times(3,x),4\rangle, \langle v_\times(v_3,x), v_4\rangle, t_3 \cdot t_\times(v_3,x) \cdot t_4\rangle}$$

Finally, we can apply the rule fun once more to obtain:

$$\langle +(\langle \times(3,x),4\rangle), v_+(\langle v_\times(v_3,x), v_4\rangle), t_3 \cdot t_\times(v_3,x) \cdot t_4 \cdot t_+(\langle v_\times(v_3,x), v_4\rangle)\rangle$$

We observe a number of points. Firstly, a lot of work is required to evaluate even this simple expression. The advantages of an automated theorem prover are clear. Secondly, the expression for the complexity of this expression contains a free variable x.

Example 3.5.11 Let Σ be a signature which includes a type int, a basic constant $2 \in \mathrm{Const}_\Sigma(int)$ and a basic function symbol $f \in \mathrm{Funct}_\Sigma(int, int)$. We evaluate the denotation of the term $\emptyset \vdash (\text{let } y = \lambda x.f(x) \text{ in } (\lambda x : int.y(x)) : int \to int)(2) : int$. For reasons of space, we perform the derivation in three parts.

$$\dfrac{\dfrac{}{\emptyset \vdash \langle \lambda x.f(x), \lambda x.f(x), * \rangle} \quad \dfrac{\dfrac{}{\emptyset \vdash \lambda x.\lambda x.fxx \downarrow}}{\emptyset \vdash \langle \lambda x.\lambda x.fxx, \lambda x.\lambda x.fxx, * \rangle} \quad \dfrac{}{x \vdash x \cdot * \equiv x}}{\dfrac{\emptyset \vdash \langle \text{let } y = \lambda x.f(x) \text{ in } \lambda x.y(x), \lambda x.\lambda x.f(x)(x), * \cdot * \rangle}{\emptyset \vdash * \cdot * \equiv *}}$$

$$\dfrac{\emptyset \vdash \langle \text{let } y = \lambda x.f(x) \text{ in } \lambda x.y(x), \lambda x.\lambda x.f(x)(x), * \rangle}{\emptyset \vdash \langle \text{let } y = \lambda x.f(x) \text{ in } \lambda x.y(x), \lambda x.f(x), * \rangle}$$

The sequent $\emptyset \vdash \langle 2, v_2, t_2 \rangle$ is an axiom. Finally, in order to apply the *app* rule, we need to derive a triple predicate for the term $f(x)[x := v_2] = f(v_2)$:

$$\dfrac{\dfrac{}{x : int \vdash \langle f(x), v_f(x), t_f(x) \rangle} \quad \dfrac{}{\emptyset \vdash v_2 \downarrow \tau}}{\emptyset \vdash \langle f(v_2), v_f(v_2), t_f(v_2) \rangle}$$

and applying *app* we obtain:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\end{array}}{\langle \text{let } y = \lambda x.f(x) \text{ in } \lambda x.y(x), \lambda x.f(x), * \rangle} \quad \dfrac{}{\langle 2, v_2, t_2 \rangle} \quad \dfrac{\begin{array}{c}\vdots\end{array}}{\langle f(v_2), v_f(v_2), t_f(v_2) \rangle}}{\langle (\text{let } y = \lambda x.f(x) \text{ in } \lambda x.y(x))(2), v_f(v_2), t_2 \cdot t_f(v_2) \rangle}$$

This gives us a much simplified form of the term, since (let $y = f$ in $\lambda x.y(x))(2)$ is reduced to the value $v_f(v_2)$ and the complexity $t_2 \cdot t_f(v_2)$.

3.5.3 Interpretation of the λ_{com}-calculus in complexity models

In section 3.4.3, we gave an interpretation of the language for computations in any λ_c-model. In particular, this gives an interpretation of the language for computations in any complexity model. However, in order to capture the extra features of the λ_{com}-calculus, we require a slight modification of this definition. The main difference is that an interpretation should satisfy the axioms for our extra constants.

In this section, we describe an interpretation of the language for computations in any complexity model. We define a notion of entailment for the λ_{com}-calculus and prove a soundness result.

Definition 3.5.12 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a complexity model and let Σ be a signature for computations. An **interpretation** of the language for complexity in $\mathcal{C}_M$ is a pair of functions $[\![-]\!] : \mathit{types} \to |\mathcal{C}_T|$ and $[\![-]\!] : \mathit{Terms} \to \mathcal{C}_T$ such that:

- $[\![\sigma \times \tau]\!] = [\![\sigma]\!] \times [\![\tau]\!]$,
- $[\![\sigma \to \tau]\!] = [\![\tau]\!]_T^{[\![\sigma]\!]}$,
- $[\![T\tau]\!] = M \times [\![\tau]\!]$ and
- $[\![1]\!] = 1$,

and:

- $$\frac{[\![f]\!] = \langle s, g\rangle : [\![\tau_1 \times \cdots \times \tau_n]\!] \longrightarrow M \times [\![\tau]\!]}{[\![v_f]\!] = \langle 0, g\rangle : [\![\tau_1 \times \cdots \times \tau_n]\!] \longrightarrow M \times [\![\tau]\!]},$$
- $$\frac{[\![f]\!] = \langle s, g\rangle : [\![\tau_1 \times \cdots \times \tau_n]\!] \longrightarrow M \times [\![\tau]\!]}{[\![t_f]\!] = \langle s, !\rangle : [\![\tau_1 \times \cdots \times \tau_n]\!] \longrightarrow M \times 1},$$
- $$\frac{[\![c]\!] = \langle s, g\rangle : 1 \longrightarrow M \times [\![\tau]\!]}{[\![v_c]\!] = \langle 0, g\rangle : 1 \longrightarrow M \times [\![\tau]\!]},$$
- $$\frac{[\![c]\!] = \langle s, g\rangle : 1 \longrightarrow M \times [\![\tau]\!]}{[\![t_c]\!] = \langle s, !\rangle : 1 \longrightarrow M \times 1},$$
- The interpretation of all other terms corresponds to that for the language for computations $\mathcal{L}(\Sigma)$ in the strong monad $M \times -$.

Remark 3.5.13 *An interpretation is fixed by giving an interpretation of the base types, and the constant and function symbols.*

This is consistent with our earlier definition 3.4.11. In particular, the let constructor still corresponds to composition in the Kleisli category.

Our intuition is that the triple predicate $\langle e, v, t\rangle$ should capture the notion of v being the functional part of e and t being the complexity of e. The following definition is intended to capture this intuition.

Definition 3.5.14 Let $\langle \mathcal{C}, \mathcal{M}\rangle$ be a complexity model and let $[\![-]\!]$ be an interpretation. We say $\mathcal{C}_M$ **entails** $\Gamma \vdash v_1 \equiv v_2 : \tau$ if:

$$[\![\Gamma \vdash v_1 : \tau]\!] \;=\; [\![\Gamma \vdash v_2 : \tau]\!].$$

We say $\mathcal{C}_M$ **entails** $\Gamma \vdash e \downarrow \tau$ if $[\![\Gamma \vdash e : \tau]\!]$ factors through the unit map $\eta_{[\![\tau]\!]}$.
We say $\mathcal{C}_M$ **entails** $\Gamma \vdash \langle e, v, t\rangle : \tau$ if $\mathcal{C}_M$ entails $\Gamma \vdash v \downarrow \tau$ and:

$$[\![\Gamma \vdash e : \tau]\!] \;=\; \mu_{[\![\tau]\!]}(id_M \times [\![\Gamma, x : 1 \vdash v : \tau]\!])t_{[\![\Gamma]\!],1}\langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash t : 1]\!]\rangle$$

where this composition is in the complexity category for $\langle \mathcal{C}, \mathcal{M}\rangle$.

Notation 3.5.15 We write "$\mathcal{C}_M$ entails $\Gamma \vdash A$" as $\Gamma \models_{\mathcal{C}_M} A$.

The following lemma gives a more explicit form of $\Gamma \models_{\mathcal{C}_M} \langle e, v, t\rangle : \tau$.

Lemma 3.5.16 Let $\langle \mathcal{C}, \mathcal{M}\rangle$ be a complexity model and let $[\![-]\!]$ be an interpretation. Suppose that $\Gamma \models_{\mathcal{C}_M} v \downarrow \tau$. Then $\Gamma \models_{\mathcal{C}_M} \langle e, v, t\rangle$ if:

$$[\![\Gamma \vdash e : \tau]\!] \;=\; (id_M \times f)t_{[\![\Gamma]\!],1}\langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash t : 1]\!]\rangle$$

where $[\![\Gamma, x : 1 \vdash v : \tau]\!] \;=\; \eta_{[\![\tau]\!]} f$.

Proof: Suppose that $\Gamma \models_{\mathcal{C}_M} \vdash v \downarrow \tau$. Then, by definition, $[\![\Gamma, x : 1 \vdash v : \tau]\!]$ factors though $\eta_{[\![\tau]\!]}$ and so equals $\eta_{[\![\tau]\!]} f$ for some f. Then:

$$\mu_{[\![\tau]\!]}(id_M \times [\![\Gamma, x : 1 \vdash v : \tau]\!])t_{[\![\Gamma]\!],1}\langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash t : 1]\!]\rangle$$

equals:

$$(id_M \times f)t_{[\![\Gamma]\!],1}\langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash t : 1]\!]\rangle$$

by the definition of the monad $M \times -$. □

We define a notion of soundness for the λ_{com}-calculus with respect to interpretation in complexity models.

Definition 3.5.17 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a complexity model and let $[\![-]\!]$ be an interpretation. $\langle \mathcal{C}, \mathcal{M} \rangle$ is **sound** for the λ_{com}-calculus with respect to $[\![-]\!]$ if for all formulae A, $\Gamma \vdash_{\lambda_{com}} A$ implies that $\Gamma \models_{\mathcal{C}_M} A$.

We have the following soundness result for the λ_{com}-calculus.

Theorem 3.5.18 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a complexity model and let $[\![-]\!]$ be an interpretation. Then $\langle \mathcal{C}, \mathcal{M} \rangle$ is sound for the λ_{com}-calculus with respect to $[\![-]\!]$.

Proof: Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a complexity model, and let $[\![-]\!]$ be an interpretation. We need to show that each of the axioms of the λ_{com}-calculus are valid in $\mathcal{C}_M$ and that each of the rules preserve validity.
We consider first, all those formulae of the form $e \downarrow \tau$. Recall that $\Gamma \models_{\mathcal{C}_M} v \downarrow \tau$ if $[\![\Gamma \vdash v : \tau]\!]$ factors through the unit map $\eta_{[\![\tau]\!]}$.

- $[\![\Gamma \vdash x_i : \tau_i]\!] = \eta_{[\![\tau]\!]} \pi_i$ and hence factors through $\eta_{[\![\tau]\!]}$ by definition.

- $[\![\Gamma \vdash * : 1]\!] = \eta_{[\![1]\!]} !_{[\![\tau]\!]}$ and hence factors through $\eta_{[\![1]\!]}$ by definition. Similarly, $\Gamma \models_{\mathcal{C}_M} [-] \downarrow T\tau$ and $\Gamma \models_{\mathcal{C}_M} \lambda x.e \downarrow \sigma \to \tau$.

- $$\begin{aligned} [\![\Gamma \vdash \langle x_1, x_2 \rangle : \tau_1 \times \tau_2]\!] &= \psi_{[\![\tau_1]\!],[\![\tau_2]\!]} \langle [\![\Gamma \vdash x_1 : \tau_1]\!], [\![\Gamma \vdash x_2 : \tau_2]\!] \rangle \\ &= \psi_{[\![\tau_1]\!],[\![\tau_2]\!]} \langle \eta_{[\![\tau_1]\!]} \pi_1, \eta_{[\![\tau_2]\!]} \pi_2 \rangle \\ &= \psi_{[\![\tau_1]\!],[\![\tau_2]\!]} (\eta_{[\![\tau_1]\!]} \times \eta_{[\![\tau_2]\!]}) \langle \pi_1, \pi_2 \rangle \\ &= \eta_{[\![\tau_1 \times \tau_2]\!]} \end{aligned}$$
  and hence factors through $\eta_{[\![\tau_1 \times \tau_2]\!]}$.

- $$\begin{aligned} [\![\Gamma \vdash \pi_i(x) : \tau_i]\!] &= (id_M \times \pi_i)[\![\Gamma \vdash x : \tau_1 \times \tau_2]\!] \\ &= (id_M \times \pi_i)(\eta_{[\![\tau_1 \times \tau_2]\!]} \pi_{1\times 2}) \\ &= \eta_{[\![\tau_1]\!]} \pi_i \pi_{1\times 2} \end{aligned}$$
  and hence factors through $\eta_{[\![\tau_i]\!]}$.

- Finally, it follows immediately from the definition of $[\![-]\!]$ that $\Gamma \models_{\mathcal{C}_M} v_c \downarrow \tau$ and that $\Gamma \models_{\mathcal{C}_M} v_f(x) \downarrow \tau$.

We now consider the formulae of the form $e_1 \equiv e_2 : \tau$. Recall that $\Gamma \models_{\mathcal{C}_M} e_1 \equiv e_2 : \tau$ if $[\![\Gamma \vdash e_1 : \tau]\!]$ equals $[\![\Gamma \vdash e_2 : \tau]\!]$. We do the case for the rule $\times.\beta$ to illustrate the general approach.

- $$\begin{aligned}[\![\Gamma \vdash \pi_i(\langle x_1, x_2\rangle) : \tau_i]\!] &= (id_M \times \pi_i)[\![\Gamma \vdash \langle x_1, x_2\rangle : \tau_1 \times \tau_2]\!] \\ &= (id_M \times \pi_i)\psi_{[\![\tau_1]\!],[\![\tau_2]\!]}\langle [\![\Gamma \vdash x_1 : \tau_1]\!], [\![\Gamma \vdash x_2 : \tau_2]\!]\rangle \\ &= [\![\Gamma \vdash x_i : \tau_i]\!]\end{aligned}$$

  as required.

We now consider the formulae of the form $\langle e, v, t\rangle : \tau$. Recall that $\Gamma \models_{\mathcal{C}_M} \langle e, v, t\rangle$ if:

$$[\![\Gamma \vdash e : \tau]\!] = (id_M \times f)t_{[\![\Gamma]\!],1}\langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash t : 1]\!]\rangle.$$

Suppose that $\Gamma \models_{\mathcal{C}_M} \langle e, v, t\rangle$ and that $[\![\Gamma \vdash v : \tau]\!] = \eta_{\tau_1 \times \tau_2} f$. Then we have:

- $$\begin{aligned}[\![\Gamma \vdash \pi_i(e) : \tau_i]\!] &= (id_M \times \pi_i)[\![\Gamma \vdash e : \tau_1 \times \tau_2]\!] \\ &= (id_M \times \pi_i)(id_M \times f)t_{[\![\Gamma]\!],1}\langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash t : 1]\!]\rangle \\ &= (id_M \times \pi_i f)t_{[\![\Gamma]\!],1}\langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash t : 1]\!]\rangle\end{aligned}$$

  However, $$\begin{aligned}[\![\Gamma \vdash \pi_i(v) : \tau_i]\!] &= (id_M \times \pi_i)[\![\Gamma \vdash v : \tau_1 \times \tau_2]\!] \\ &= (id_M \times \pi_i)\eta_{\tau_1 \times \tau_2} f \\ &= \eta_{\tau_1 \times \tau_2}\pi_i f\end{aligned}$$

  and therefore $\Gamma \models_{\mathcal{C}_M} \langle \pi_i(e), \pi_i(v), t\rangle$ as required.

The other cases are similar and are omitted. □

We discuss completeness issues in section 3.6.

3.6 Relating the λ_{com}-calculus to other systems

3.6.1 Introduction

We characterise the relationship between the λ_{com}-calculus and the λ_c-calculus and the relationship between the λ_{com}-calculus and an operational semantics for the language for computations.

3.6.2 The relationship between the λ_{com}-calculus and the λ_c-calculus

Definition 3.6.1 Let Σ be a signature for computations and let $\mathcal{L}(\Sigma)$ be the language for computations over Σ. The rules $Ax(\Sigma)$ consist of the following axioms:

- $$\frac{}{\emptyset \vdash v_c \downarrow \tau} \quad c \in \mathrm{Const}_\Sigma(\tau)$$

- $$\frac{}{x : \tau_1 \times \cdots \times \tau_n \vdash v_f(x) \downarrow \tau} \quad f \in \mathrm{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$$

- $$\frac{}{\emptyset \vdash c \equiv \mathrm{let}\ x = t_c\ \mathrm{in}\ v_c}$$

- $$\frac{}{x : \tau_1 \times \cdots \times \tau_n \vdash f(x) \equiv \mathrm{let}\ z = t_f(x)\ \mathrm{in}\ v_f(x)} \quad z \neq x$$

Notation 3.6.2 We write $\lambda_c + Ax$ for the formal system whose rules are those of the λ_c-calculus plus the set $Ax(\Sigma)$.

The next theorem summarises the relationship between the λ_c-calculus and the λ_{com}-calculus. First, we have a technical lemma.

Lemma 3.6.3 Let Σ be a signature for computations and let A be a formula of the form $e_1 \equiv e_2 : \tau$ or $v \downarrow \tau$. Suppose that the sequent $\Gamma \vdash A$ is derivable in the λ_{com}-calculus. Then the sequent $\Gamma \vdash A$ is derivable in the calculus $\lambda_c + Ax$.

Proof: A simple induction on the length of the derivation of the sequent. □

Theorem 3.6.4 Let Σ be a signature for computations and suppose that the sequent $\Gamma \vdash \langle e, v, t\rangle : \tau$ is derivable in the λ_{com}-calculus. Then both of the sequents $\Gamma \vdash v \downarrow \tau$ and $\Gamma \vdash e \equiv \text{let } x{=}t \text{ in } v : \tau$ are derivable in the calculus $\lambda_c + Ax$.

Notation 3.6.5 In the following proof, for convenience, we write:

$$\dfrac{\dfrac{\dfrac{}{\vdots}}{\Gamma \vdash e_1 \equiv e_2} \quad \dfrac{\dfrac{}{\vdots}}{\Gamma \vdash e_2 \equiv e_3}}{\Gamma \vdash e_1 \equiv e_3}$$

as:

$$\dfrac{\dfrac{\dfrac{}{\vdots}}{\Gamma \vdash e_1 \equiv e_2}}{\Gamma \vdash e_1 \equiv e_3}$$

Proof: Suppose that $\Gamma \vdash \langle e, v, t\rangle : \tau$ is derivable in the λ_{com}-calculus. Then by lemma 3.5.9, $\Gamma \vdash v \downarrow \tau$ is derivable in the λ_{com}-calculus and by lemma 3.6.3, it is derivable in the $\lambda_c + Ax$.
We verify that "$e \equiv \text{let } x{=}t \text{ in } v$", by induction on the number of triple rules in the derivation.

- Base case: the cases *cst* and *fun* follow by definition since they are all contained in the set $Ax(\Sigma)$. The case *value* is also immediate since the sequent $\Gamma \vdash e \equiv \text{let } x{=}* \text{ in } e : \tau$ is derivable for all terms $\Gamma \vdash e : \tau$.

- General case: suppose that the last rule was *pair*. Then by the inductive hypothesis, we have derivations of the sequents $\Gamma \vdash e_i \equiv \text{let } x_i = t_i \text{ in } v_i : \tau_i$ and $\Gamma \vdash v_i \downarrow \tau_i$ in the calculus $\lambda_c + Ax$ and we have:

$$\frac{}{\langle e_1, e_2\rangle \equiv \text{let } y_1 = e_1 \text{ in } (\text{let } y_2 = e_2 \text{ in } \langle y_1, y_2\rangle)}$$

$$\frac{\langle e_1, e_2\rangle \equiv \text{let } y_1 = (\text{let } x_1 = t_1 \text{ in } v_1) \text{ in } (\text{let } y_2 = (\text{let } x_2 = t_2 \text{ in } v_2) \text{ in } \langle y_1, y_2\rangle)}{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } y_1 = v_1 \text{ in } (\text{let } y_2 = (\text{let } x_2 = t_2 \text{ in } v_2) \text{ in } \langle y_1, y_2\rangle))}$$

$$\frac{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } y_1 = v_1 \text{ in } (\text{let } y_2 = (\text{let } x_2 = t_2 \text{ in } v_2) \text{ in } \langle y_1, y_2\rangle))}{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } y_2 = (\text{let } x_2 = t_2 \text{ in } v_2) \text{ in } \langle y_1, y_2\rangle)[y_1 := v_1]}$$

$$\frac{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } y_2 = (\text{let } x_2 = t_2 \text{ in } v_2) \text{ in } \langle y_1, y_2\rangle)[y_1 := v_1]}{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } y_2 = (\text{let } x_2 = t_2 \text{ in } v_2) \text{ in } \langle v_1, y_2\rangle)}$$

$$\frac{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } y_2 = (\text{let } x_2 = t_2 \text{ in } v_2) \text{ in } \langle v_1, y_2\rangle)}{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } x_2 = t_2 \text{ in } (\text{let } y_2 = v_2 \text{ in } \langle v_1, y_2\rangle))}$$

$$\frac{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } x_2 = t_2 \text{ in } (\text{let } y_2 = v_2 \text{ in } \langle v_1, y_2\rangle))}{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } x_2 = t_2 \text{ in } \langle v_1, y_2\rangle[y_2 := v_2])}$$

$$\frac{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } x_2 = t_2 \text{ in } \langle v_1, y_2\rangle[y_2 := v_2])}{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } x_2 = t_2 \text{ in } \langle v_1, v_2\rangle)}$$

$$\frac{\langle e_1, e_2\rangle \equiv \text{let } x_1 = t_1 \text{ in } (\text{let } x_2 = t_2 \text{ in } \langle v_1, v_2\rangle)}{\langle e_1, e_2\rangle \equiv \text{let } x_2 = (\text{let } x_1 = t_1 \text{ in } t_2) \text{ in } \langle v_1, v_2\rangle}$$

$$\frac{\langle e_1, e_2\rangle \equiv \text{let } x_2 = (\text{let } x_1 = t_1 \text{ in } t_2) \text{ in } \langle v_1, v_2\rangle}{\langle e_1, e_2\rangle \equiv \text{let } x_2 = t_1 \cdot t_2 \text{ in } \langle v_1, v_2\rangle}$$

Suppose that the last rule was π. Then by the inductive hypothesis, we have derivations of the sequents $\Gamma \vdash e \equiv \text{let } x = t \text{ in } v$ and $\Gamma \vdash v \downarrow \tau$ and we have:

$$\frac{\begin{array}{c}\\ \cdot \\ \cdot \\ \cdot \\ \\ \Gamma \vdash e \equiv \text{let } x = t \text{ in } v\end{array}}{\Gamma \vdash \pi(e) \equiv \pi(\text{let } x = t \text{ in } v)}$$

$$\frac{\Gamma \vdash \pi(e) \equiv \pi(\text{let } x = t \text{ in } v)}{\Gamma \vdash \pi(e) \equiv \text{let } y = \text{let } x = t \text{ in } v \text{ in } \pi(y)}$$

$$\frac{\Gamma \vdash \pi(e) \equiv \text{let } y = \text{let } x = t \text{ in } v \text{ in } \pi(y)}{\Gamma \vdash \pi(e) \equiv \text{let } x = t \text{ in } (\text{let } y = v \text{ in } \pi(y))}$$

$$\frac{\Gamma \vdash \pi(e) \equiv \text{let } x = t \text{ in } (\text{let } y = v \text{ in } \pi(y))}{\Gamma \vdash \pi(e) \equiv \text{let } x = t \text{ in } \pi(y)[y := v]}$$

$$\frac{\Gamma \vdash \pi(e) \equiv \text{let } x = t \text{ in } \pi(y)[y := v]}{\Gamma \vdash \pi(e) \equiv \text{let } x = t \text{ in } \pi(v)}$$

Suppose that the last rule was *let*. Then by the inductive hypothesis, we have derivations of the sequents $\Gamma \vdash e_1 \equiv \text{let } x = t_1 \text{ in } v_1 : \tau_1$ and $\Gamma \vdash e_2[x := v_1] \equiv \text{let } x = t_2 \text{ in } v_2$ and we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{let } x=e_1 \text{ in } e_2 \equiv \text{let } x=(\text{let } x_1=t_1 \text{ in } v_1) \text{ in } e_2}}{\text{let } x=e_1 \text{ in } e_2 \equiv \text{let } x_1=t_1 \text{ in } (\text{let } x=v_1 \text{ in } e_2)}}{\text{let } x=e_1 \text{ in } e_2 \equiv \text{let } x_1=t_1 \text{ in } e_2[x:=v_1]}}{\text{let } x=e_1 \text{ in } e_2 \equiv \text{let } x_1=t_1 \text{ in } (\text{let } x_2=t_2 \text{ in } v_2)}}{\text{let } x=e_1 \text{ in } e_2 \equiv (\text{let } x_2=(\text{let } x_1=t_1 \text{ in } t_2) \text{ in } v_2)}}{\text{let } x=e_1 \text{ in } e_2 \equiv (\text{let } x_2=t_1 \cdot t_2 \text{ in } v_2)}$$

Suppose that the last rule was *app*. Then by the inductive hypothesis, we have derivations of the sequents $\Gamma \vdash e_1 \equiv \text{let } x_1=t_1 \text{ in } v_1 : \sigma \to \tau$, $\Gamma \vdash e_2 \equiv \text{let } x_2=t_2 \text{ in } v_2 : \sigma$ and $\Gamma \vdash v_1(v_2) \equiv \text{let } x_3=t_3 \text{ in } v_3 : \tau$ and we have:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{e_1(e_2) \equiv \text{let } y_1=e_1 \text{ in } (\text{let } y_2=e_2 \text{ in } y_1(y_2))}}{e_1(e_2) \equiv \text{let } y_1=\text{let } x_1=t_1 \text{ in } v_1 \text{ in } (\text{let } y_2=\text{let } x_2=t_2 \text{ in } v_2 \text{ in } y_1(y_2))}}{\vdots}}{e_1(e_2) \equiv \text{let } x_1=t_1 \text{ in } (\text{let } x_2=t_2 \text{ in } v_1(v_2))}}{e_1(e_2) \equiv \text{let } x_1=t_1 \text{ in } (\text{let } x_2=t_2 \text{ in let } x_3=t_3 \text{ in } v_3)}}{\vdots}}{e_1(e_2) \equiv \text{let } x_3=(\text{let } x_1=t_1 \text{ in } (\text{let } x_2=t_2 \text{ in } t_3)) \text{ in } v_3}}{e_1(e_2) \equiv \text{let } x_3=t_1 \cdot t_2 \cdot t_3 \text{ in } v_3}$$

Finally, suppose the last the last rule was *cong*. Then by the inductive hypothesis, we have derivation of the sequent $\Gamma \vdash e \equiv \text{let } x=t \text{ in } v$ and by lemma 3.6.3 we have derivations of the sequents $\Gamma \vdash v \equiv v'$ and $\Gamma \vdash t \equiv t'$. However $\equiv$ is a congruence relation and so we have a derivation of the sequent $\Gamma \vdash e \equiv \text{let } x=t' \text{ in } v'$.

This completes the proof. □

Unfortunately, we have not succeeded in proving the converse. In particular, it is not clear to us what extra rules we would need to add to $\lambda_c + Ax$ in order to obtain completeness with respect to complexity models.

Remark 3.6.6 *In Moggi's metalanguage [Mog88b], we can add for each type τ a function symbol:*

$$val \ : \ T\tau \longrightarrow \tau$$

which returns the value part of each term, and define an operation:

$$com(e) \ = \ \mathrm{let}_T \ x{=}e \ \mathrm{in} \ [*]_T$$

which returns the complexity of each term. Then the following additional rules:

- $val([e]_T) \ \equiv \ e$,
- $val(\mathrm{let}_T \ x{=}e_1 \ \mathrm{in} \ e_2) \ \equiv \ val(e_2[x := val(e_1)])$,
- $com(\mathrm{let}_T \ x{=}e_1 \ \mathrm{in} \ e_2) \ \equiv \ com(e_1) \cdot com(e_2[x := val(e_1)])$ *and*
- $\mathrm{let}_T \ x{=}com(e) \ \mathrm{in} \ val(e) \ \equiv \ e$

suffice to prove that $T\tau \cong T1 \times \tau$ in any model. It is also not hard to show that $T1$ has a monoid structure.

3.6.3 An operational semantics

We give an operational semantics for the λ_{com}-calculus which is an evaluation function from closed terms to values. We modify the usual form of evaluation, $e \Longrightarrow v$, and obtain evaluations of the form:

$$e \stackrel{t}{\Longrightarrow} v$$

The idea is that this represents the term e evaluating to the value v and consuming resource t in the process.

There is an unfortunate conflict between the operational semantics notion of value and the notion of value in Moggi's λ_c-calculus. In the operational semantics, the values are a syntactically classified subclass of the terms and are intended to represent terms which are fully evaluated. In the λ_c-calculus, values are terms v for which $\Gamma \vdash v \downarrow$ is derivable, and are intended to represent terms which have a

trivial "computational" component. The following example shows that these two notions do not coincide.

Example 3.6.7 Let v be a closed λ_c-calculus value of type τ. Then there exits a derivation:

$$\dfrac{\begin{array}{c}\\ \vdots \\ \\ \hline \end{array}}{\emptyset \vdash v \downarrow \tau}$$

of the sequent $\emptyset \vdash v \downarrow \tau$. Therefore we can derive the sequent $\emptyset \vdash \pi_1\langle v, v\rangle \downarrow \tau$ as follows:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\\ \vdots \\ \\ \hline \end{array}}{\emptyset \vdash \langle x, x\rangle \downarrow \tau \times \tau} \quad \dfrac{\begin{array}{c}\\ \vdots \\ \\ \hline \end{array}}{\emptyset \vdash v \downarrow \tau}}{\emptyset \vdash \langle v, v\rangle \downarrow \tau \times \tau}}{\emptyset \vdash \pi_1\langle v, v\rangle \downarrow \tau}$$

Therefore the term $\pi_1\langle v, v\rangle$ is a value in the λ_c sense. However, $\pi_1\langle v, v\rangle$ is not fully evaluated since it is equivalent to the simpler form v.

Accordingly, we define a subclass of the λ_c values to be syntactic values. This definition is intended to correspond to the operational semantics notion of fully evaluated terms. The operational semantics we give will be an evaluation function from closed terms to syntactic values.

Definition 3.6.8 The class SynVal of **syntactic values** is given by the following BNF:

$sv ::= * \mid v_c \mid v_f(sv) \mid \lambda x.e \mid [e] \mid \langle sv_1, sv_2\rangle.$

provided that $\lambda x.e$ and $[e]$ are closed.

The following result shows that the syntactic values are values in the λ_c sense.

Lemma 3.6.9 Let sv be a syntactic value of type τ. Then the sequent $\emptyset \vdash sv \downarrow \tau$ is derivable in the λ_{com}-calculus.

Proof: All cases except $\langle sv_1, sv_2 \rangle$ and $v_f(sv)$ follow immediately from the axioms of the λ_{com}-calculus. For $\langle sv_1, sv_2 \rangle$, we proceed by induction on the number of brackets. The base case we have just established. Suppose $\langle sv_1, sv_2 \rangle$ is a syntactic value of type $\sigma \times \tau$. Then the sequents $\emptyset \vdash v_1 \downarrow \sigma$ and $\emptyset \vdash v_2 \downarrow \tau$ are derivable by the inductive hypothesis. The following derivation:

$$\frac{\emptyset \vdash \langle x_1, x_2 \rangle \downarrow \sigma \times \tau \quad \emptyset \vdash v_1 \downarrow \sigma \quad \emptyset \vdash v_2 \downarrow \tau}{\emptyset \vdash \langle v_1, v_2 \rangle \downarrow \sigma \times \tau}$$

establishes that $\emptyset \vdash \langle v_1, v_2 \rangle \downarrow \sigma \times \tau$ is derivable. The case for $v_f(sv)$ follows similarly and this completes the proof. □

We note that this inclusion is strict since, for example, the term $\pi_1 \langle v_c, v_c \rangle$ is a value but not a syntactic value.

Unfortunately, the syntactic values are not necessarily unique. For example, if $x : \tau \vdash e \equiv e'$ then $\emptyset \vdash \lambda x.e \equiv \lambda x.e'$ and both of these are syntactic values. For this reason, we do not use the term canonical values.

We now present the rules of the operational semantics.

Definition 3.6.10 Let Σ be a signature for computations. The **operational semantics** for $\mathcal{L}(\Sigma)$ is given by:

$$\text{value} \ \frac{}{v \stackrel{*}{\Longrightarrow} v} \ v \in \text{SynVal}$$

$$\text{cst} \ \frac{}{c \stackrel{t_c}{\Longrightarrow} v_c}$$

$$\text{fun}\ \frac{e \overset{t}{\Longrightarrow} v}{f(e) \overset{t \cdot t_f(v)}{\Longrightarrow} v_f(v)}\quad f \in \text{Funct}_\Sigma(\tau_1, ..., \tau_n, \tau)$$

$$\text{pair}\ \frac{e_1 \overset{t_1}{\Longrightarrow} v_1 \quad e_2 \overset{t_2}{\Longrightarrow} v_2}{\langle e_1, e_2 \rangle \overset{t_1 \cdot t_2}{\Longrightarrow} \langle v_1, v_2 \rangle}$$

$$\pi\ \frac{e \overset{t}{\Longrightarrow} \langle v_1, v_2 \rangle}{\pi_i(e) \overset{t}{\Longrightarrow} v_i}$$

$$\mu\ \frac{e_1 \overset{t_1}{\Longrightarrow} [e_2] \quad e_2 \overset{t_2}{\Longrightarrow} v_2}{\mu(e_1) \overset{t_1 \cdot t_2}{\Longrightarrow} v_2}$$

$$\text{let}\ \frac{e_1 \overset{t_1}{\Longrightarrow} v_1 \quad e_2[x := v_1] \overset{t_2}{\Longrightarrow} v_2}{\text{let } x = e_1 \text{ in } e_2 \overset{t_1 \cdot t_2}{\Longrightarrow} v_2}$$

$$\lambda.\text{app}\ \frac{e_1 \overset{t_1}{\Longrightarrow} \lambda x.e \quad e_2 \overset{t_2}{\Longrightarrow} v_2 \quad e[x := v_2] \overset{t_3}{\Longrightarrow} v_3}{e_1(e_2) \overset{t_1 \cdot t_2 \cdot t_3}{\Longrightarrow} v_3}$$

Remark 3.6.11 *It is clear that if* $e \overset{t}{\Longrightarrow} v$ *then* v *is a syntactic value. In particular, by lemma 3.6.9, if* $e \overset{t}{\Longrightarrow} v$ *then the sequent* $\emptyset \vdash v \downarrow \tau$ *is derivable in the* λ_{com}*-calculus.*

3.6.4 The λ_{com}-calculus and the operational semantics

Theorem 3.6.12 Let Σ be a signature for computations, let e be term of $\mathcal{L}(\Sigma)$ and suppose that $e \overset{t}{\Longrightarrow} v$. Then the sequent $\Gamma \vdash \langle e, v, t \rangle$ is derivable in the λ_{com}-calculus.

Proof: We proceed by induction on the length of the derivation of $e \overset{t}{\Longrightarrow} v$.

- Base case: if $e = c$ then the result follows immediately since the required sequent is an axiom of the λ_{com}-calculus.

- General case: suppose that the last rule was

$$\text{pair}\ \frac{e_1 \stackrel{t_1}{\Longrightarrow} v_1 \quad e_2 \stackrel{t_2}{\Longrightarrow} v_2}{\langle e_1, e_2\rangle \stackrel{t_1 \cdot t_2}{\Longrightarrow} \langle v_1, v_2\rangle}\ .$$

  Then the $e_i \stackrel{t_i}{\Longrightarrow} v_i$ have shorter derivations, and so by the inductive hypothesis, the sequents $\Gamma \vdash \langle e_i, v_i, t_i\rangle$ are derivable in the λ_{com}-calculus. Then, applying the *pair* triple rule, we obtain $\Gamma \vdash \langle\langle e_1, e_2\rangle, \langle v_1, v_2\rangle, t_1 \cdot t_2\rangle$ as required.

  The other cases are similar. For example, if the last rule was *let* then, by the inductive hypothesis, we have $\Gamma \vdash \langle e_1, v_1, t_1\rangle$ and $\Gamma \vdash \langle e_2[x := v_1], v_2, t_2\rangle$ and applying the *let* triple rule, we obtain $\Gamma \vdash \langle \text{let } x = e_1 \text{ in } e_2, v_2, t_1 \cdot t_2\rangle$ as required.

This completes the proof. □

We had hoped to include the following result. Unfortunately, at the last moment, a flaw was discovered in the proof.

Conjecture 3.6.13 Let Σ be a signature for computations and let e be a closed term of $\mathcal{L}(\Sigma)$. Suppose that the sequent $\emptyset \vdash \langle e, v, t\rangle : \tau$ is derivable in the λ_{com}-calculus. Then there exists terms $v' : \tau$, $t' : 1$ such that $\emptyset \vdash v \equiv v' : \tau$ and $\emptyset \vdash t \equiv t' : 1$ are derivable in the λ_{com}-calculus and $e \stackrel{t'}{\Longrightarrow} v'$.

Chapter 4

Monad Constructors

4.1 Introduction

In chapter 3, we described how Moggi [Mog88b] has proposed the paradigm of a notion of computation as a monad over a category with structure. In [Mog89], Moggi goes further and studies modularity in denotational semantics by viewing notions of computation not as monads but as monad constructors, *i.e.* endofunctions on the object set of the category SMon($\mathcal{C}$) of strong monads on C. The idea is that we can add a new feature, for example side effects, which has the effect of taking a monad representing T-computations, where T is an arbitrary monad representing some other feature, to one representing T-computations with side-effects.

Ideally we should like these monad constructors to come with the following properties and structure.

- **They preserve the monomorphism requirement.**
  The importance of the monomorphism requirement, that η_A is a monomorphism for each A, is discussed in section 4.2.4.

- **They are endofunctors on SMon($\mathcal{C}$).**
  This allows monad morphisms from S to T to lift to monads morphisms from

S^+ to T^+ where $(-)^+$ is our constructor. Thus if we have an interpretation of S in T we can obtain one of S^+ in T^+.

- **They are monads on SMon($\mathcal{C}$).**
  In order to add features in a stepwise fashion, we need an embedding η of T in T^+. In order to expresses the idea that adding a feature twice gives no more expressive power than adding it once, we should like to have $T^{++} = T^+$. This does not hold in general but it suffices to have a map, μ, from T^{++} to T^+ such that $\langle (-)^+, \eta, \mu \rangle$ is a monad on SMon($\mathcal{C}$).

- **They are strong monads on SMon($\mathcal{C}$).**
  Moggi does not consider the question of strong monads on SMon($\mathcal{C}$). However, they may have applications to concurrent computation. We discuss this in section 4.3.

In this chapter, we first apply Moggi's ideas to the monad for complexity. We then extend his work to consider limits in SMon($\mathcal{C}$) and strong monads on SMon($\mathcal{C}$).

In section 4.2, we define the category SMon($\mathcal{C}$) of strong monads and strong monad morphisms over $\mathcal{C}$. We then define the monad constructor for complexity, $(-)_M$ and prove the following results:

- $(-)_M$ preserves the monomorphism requirement.

- $(-)_M$ extends naturally to an endofunctor on SMon($\mathcal{C}$).

- $(-)_M$ extends naturally to a monad on SMon($\mathcal{C}$).

In section 4.3 we extend Moggi's work in the following ways:

- We prove that the forgetful functor U : SMon($\mathcal{C}$)$\longrightarrow$ $[\mathcal{C}, \mathcal{C}]$ creates limits.

- We show how limits in SMon($\mathcal{C}$) can be useful in studying the relationships between monads.

- We present reasons to regard the notion of a strong monad on $\mathrm{SMon}(\mathcal{C})$ as useful in developing a theory of parallel computations with features.

Finally in section 4.4 we prove:

- $(-)_M$ extends naturally to a strong monad on $\mathrm{SMon}(\mathcal{C})$.

4.2 The monad constructor for complexity

4.2.1 Introduction

This section is based on [Mog89]. We apply his framework to the monad for complexity.

4.2.2 A category of monads

We extend the definitions of monad and strong monad to obtain categories $\mathrm{Mon}(\mathcal{C})$ and $\mathrm{SMon}(\mathcal{C})$ of monads and strong monads.

Definition 4.2.1 Let $\langle S, \eta^S, \mu^S \rangle$ and $\langle T, \eta^T, \mu^T \rangle$ be monads on a category $\mathcal{C}$. A **monad morphism** σ is a natural transformation $\sigma : S \longrightarrow T$ such that the following diagrams:

$$\begin{array}{ccc} A & \xrightarrow{\eta^S_A} & SA \\ {\scriptstyle id_A}\downarrow & 1 & \downarrow{\scriptstyle \sigma_A} \\ A & \xrightarrow[\eta^T_A]{} & TA \end{array}$$

$$\begin{array}{ccc} S^2A & \xrightarrow{\mu^S_A} & SA \\ {\scriptstyle (\sigma\sigma)_A}\downarrow & 2 & \downarrow{\scriptstyle \sigma_A} \\ T^2A & \xrightarrow[\mu^T_A]{} & TA \end{array}$$

commute, where $\sigma\sigma$ is defined to be $\sigma_T S\sigma$ which by naturality is also equal to $T\sigma\sigma_S$.

Definition 4.2.2 Let $\langle S, \eta^S, \mu^S, t^S \rangle$ and $\langle T, \eta^T, \mu^T, t^T \rangle$ be strong monads on a category $\mathcal{C}$ with finite products. A **strong monad morphism**, σ is a monad morphism $\sigma : S \longrightarrow T$ such that the following diagram:

$$\begin{array}{ccc} A \times SB & \xrightarrow{t^S_{A,B}} & S(A \times B) \\ \downarrow{\scriptstyle id_A \times \sigma_A} & 3 & \downarrow{\scriptstyle \sigma_{A\times B}} \\ A \times TB & \xrightarrow[t^T_{A,B}]{} & T(A \times B) \end{array}$$

commutes.

Remark 4.2.3 *Let S, T and U be monads on a category $\mathcal{C}$. Given monad morphisms $\sigma : S \longrightarrow T$ and $\tau : T \longrightarrow U$, the composite natural transformation $\tau\sigma$ is a monad morphism $\tau\sigma : S \longrightarrow U$ and for all monads S on $\mathcal{C}$, the identity natural transformation on S is a monad morphism.*

If $\mathcal{C}$ has finite products and S, T and U are strong monads on $\mathcal{C}$, then if $\sigma : S \longrightarrow T$ and $\tau : T \longrightarrow U$ are strong monad morphisms, the composite monad morphism $\tau\sigma$ is a strong monad morphism. Identity monad morphisms on strong monads are always strong.

Definition 4.2.4 Let $\mathcal{C}$ be a category. We define $\mathrm{Mon}(\mathcal{C})$ to be the category of monads and monad morphisms on $\mathcal{C}$ together with the evident composition.

Definition 4.2.5 Let $\mathcal{C}$ be a category with finite products. We define $\mathrm{SMon}(\mathcal{C})$ to be the category of strong monads and strong monad morphisms on $\mathcal{C}$ together with the evident composition.

Remark 4.2.6 *The category of principal interest to us is SMon($\mathcal{C}$).*

4.2.3 The monad constructor $(-)_M$

Definition 4.2.7 Let $\mathcal{C}$ be a category with finite products. A **monad constructor for** $\mathcal{C}$ is an endofunction on $|\mathrm{SMon}(\mathcal{C})|$.

Henceforth, we shall assume that $\mathcal{C}$ is a category with finite products and a monoid object $\mathcal{M} = \langle M, 0, \cdot \rangle$.

Remark 4.2.8 *Warning! We have defined the strong monad for complexity $\mathcal{T}^M$ (c.f. proposition 3.3.1) as a strong monad on $\mathcal{C}$, i.e. an object of SMon($\mathcal{C}$). We are going to use $\mathcal{T}^M$ to define a monad constructor for complexity, i.e. a function $|SMon(\mathcal{C})| \to |SMon(\mathcal{C})|$ which takes an arbitrary strong monad T to a strong monad T_M of T-computations with complexity.*

It is important to avoid the natural confusion.

Lemma 4.2.9 Let $\langle T, \eta^T, \mu^T, t^T \rangle$ be a strong monad on $\mathcal{C}$. Then the following data:

- $T_M(-) = T(M \times -)$,
- $\eta_A^{T_M} = \eta_{M\times A}^T \eta_A^M$,
- $\mu_A^{T_M} = \mu_{M\times A}^T T^2 \mu_A^M T t_{M,M\times A}^T$ and
- $t_{A,B}^{T_M} = T t_{A,B}^M t_{A,M\times B}^T$

define a strong monad $\langle T_M, \eta^{T_M}, \mu^{T_M}, t^{T_M} \rangle$ on $\mathcal{C}$.

Proof: It is routine to verify that T_M is an endofunctor on $\mathcal{C}$ and that η^{T_M}, μ^{T_M} and t^{T_M} are all natural. The verification that $\langle T_M, \eta^{T_M}, \mu^{T_M}, t^{T_M} \rangle$ satisfies the axioms of definitions 3.2.1 and 3.2.5 is routine but extremely tedious and is relegated to appendix A.1. □

Remark 4.2.10 *The definition of T_M is not arbitrary, as the tensorial strength $t^T_{M,M\times A}$ is a distributive law for the monads T and $M \times -$ [BaWe85].*

Remark 4.2.11 *There is another natural candidate for the monad constructor $(-)_M$ where $T_M(-) = M \times T(-)$. However, the first definition appears to be the more natural since, for example, if T is the lifting monad $(-)_\perp$ then the second definition gives $T_M A = M \times A_\perp$. This leads to the difficulty of a program returning a value for the time taken even when it does not terminate.*

Definition 4.2.12 Let $\langle T, \eta^T, \mu^T, t^T\rangle$ be a strong monad on $\mathcal{C}$. The strong monad of **T-computations with M-complexity** is defined by $\langle T_M, \eta^{T_M}, \mu^{T_M}, t^{T_M}\rangle$.

Remark 4.2.13 *The lemma shows that assignment $T \mapsto T_M$ takes strong monads to strong monads. We call this function the* **monad constructor for complexity.**

Moggi [Mog89] has used a metalanguage to define monad constructors. For completeness, we give such a definition below.

Proposition 4.2.14 Let T be a type constructor for computational types in the metalanguage. Given an interpretation of T as a monad on $\mathcal{C}$, the interpretation of the following metalanguage terms:

- $T_M(-) = T(M \times -)$,
- $\eta^{T_M}_A : a \longmapsto [\langle 0, a\rangle]_T$,
- $\mu^{T_M}_A : c \longmapsto \mathrm{let}_T\ \langle m_1, c_1\rangle = c$ in $(\mathrm{let}_T\ \langle m_2, c_2\rangle = c_1$ in $[\langle m_1 \cdot m_2, c_2\rangle]_T)$ and
- $t^{com}_{A,B} : \langle a, c\rangle \mapsto \mathrm{let}_T\ \langle a_1, \langle m_1, b_1\rangle\rangle = t^T_{A,B}(\langle a, c\rangle)$ in $[\langle m_1, \langle a_1, b_1\rangle\rangle]_T$

in $\mathcal{C}$ gives the strong monad T_M.

Proof: We compute the interpretation of the required terms.

$$\begin{aligned}[\![\eta^{T_M}]\!] &= [\![a \longmapsto [\langle 0, a\rangle]_T]\!] \\ &= \eta^T_{M\times A}[\![a \longmapsto \langle 0, a\rangle]\!] \\ &= \eta^T_{M\times A}\eta^M_A.\end{aligned}$$

as required.

The other cases follow similarly. □

Remark 4.2.15 *We have defined $(-)_M$ as a function from $|SMon(\mathcal{C})|$ to $|SMon(\mathcal{C})|$. In general, it is not possible to extend this naturally to a function from $|Mon(\mathcal{C})|$ to $|Mon(\mathcal{C})|$ because the definition of μ^{T_M} requires T to have a tensorial strength.*

With the metalanguage formulation of proposition 4.2.14, the problem is that the term $[\langle m_1 \cdot m_2, c_2\rangle]_T$ has two different free variables. It is shown in [Mog88b] that we need a tensorial strength in order to interpret terms with multiple distinct free variables.

4.2.4 Properties of $(-)_M$

Proposition 4.2.16 $(-)_M$ preserves the monomorphism requirement.

Proof: Suppose T satisfies the monomorphism requirement, that is that η^T_A is a monomorphism for all A. Then, by definition, $\eta^{T_M}_A = \eta^T_{M\times A}\eta^M_A$. Now η^M_A is a monomorphism for every A, as $\eta^M_A : a \longmapsto \langle 0, a\rangle$ and $\eta^T_{M\times A}$ is a monomorphism, as η^T_A is a monomorphism for every A. Therefore, $\eta^{T_M}_A$ is a monomorphism, as monomorphisms are closed under composition.

Thus $(-)_M$ preserves the monomorphism requirement. □

Computationally, the idea is that η represents the inclusion of values into computations. The importance of the monomorphism requirement is twofold. Firstly, it allows us to deduce equality of values from equality of computations. Secondly,

according to the paradigm of categorical logic, formulae should be interpreted as subobjects. The predicate $- \downarrow \tau$ in the λ_c-calculus [Mog88b], existence of computations of type τ, is interpreted by $\eta_{[\![\tau]\!]}$. This is a monomorphism if the monomorphism requirement holds.

Hence, it is desirable that a monad constructor preserves the monomorphism requirement and proposition 4.2.16 shows that $(-)_M$ does.

We now show that $(-)_M$ extends naturally to an endofunctor on SMon($\mathcal{C}$).

Definition 4.2.17 Let $\sigma : S \longrightarrow T$ be a strong monad morphism in SMon($\mathcal{C}$). We define the natural transformation $\sigma_M : S_M \longrightarrow T_M$ by:

$$(\sigma_M)_A \;=\; \sigma_{M\times A}.$$

Lemma 4.2.18 Let $\sigma : S \longrightarrow T$ be a strong monad morphism in SMon($\mathcal{C}$). With the above notation, the natural transformation $\sigma_M : S_M \longrightarrow T_M$ is a strong monad morphism.

Proof: We must verify that the following diagrams:

$$\begin{array}{ccccc}
A & \xrightarrow{\eta_A^{S_M}} & S_M A & \xleftarrow{\mu_A^{S_M}} & S_M^2 A \\
{\scriptstyle id_A}\downarrow & I & \downarrow{\scriptstyle \sigma_A^M} & II & \downarrow{\scriptstyle (\sigma^M\sigma^M)_A} \\
A & \xrightarrow[\eta_A^{T_M}]{} & T_M A & \xleftarrow[\mu_A^{T_M}]{} & T_M^2 A
\end{array}$$

$$\begin{array}{ccc}
A \times S_M B & \xrightarrow{t^{S_M}_{A,B}} & S_M(A\times B) \\
{\scriptstyle id\times\sigma_A^M}\downarrow & III & \downarrow{\scriptstyle \sigma^M_{A\times B}} \\
A \times T_M B & \xrightarrow[t^{T_M}_{A,B}]{} & T_M(A\times B)
\end{array}$$

commute. This is routine but tedious and is relegated to appendix A.2. □

Proposition 4.2.19 The function $(-)_M : |\mathrm{SMon}(\mathcal{C})| \longrightarrow |\mathrm{SMon}(\mathcal{C})|$ in lemma 4.2.14 together with the function on arrows in definition 4.2.17 define a functor

$$(-)_M : \mathrm{SMon}(\mathcal{C}) \longrightarrow \mathrm{SMon}(\mathcal{C}).$$

Proof: We observe that:

$(id_S)_M$ is given by $((id_S)_M)_A = (id_S)_{M\times A} = id_{S(M\times A)}$ and thus $(id_S)_M = id_{S_M}$.

$(\tau\sigma)_M$ is given by $((\tau\sigma)_M)_A = (\tau\sigma)_{M\times A} = \tau_{M\times A}\sigma_{M\times A}$ and thus $(\tau\sigma)_M = \tau_M\sigma_M$.

Therefore, $(-)^M$ extends naturally to an endofunctor on $\mathrm{SMon}(\mathcal{C})$ as claimed. □

The importance of considering the category $\mathrm{SMon}(\mathcal{C})$ of strong monads over $\mathcal{C}$, rather the set $|\mathrm{SMon}(\mathcal{C})|$, is that it allows us to give an interpretation of one monad in another. In other words, identifying monads with models of particular notions of computation, we can relate denotational models corresponding to features of a programming language.

The importance of monad constructors lifting naturally to endofunctors on $\mathrm{SMon}(\mathcal{C})$ is that it allows us to lift naturally an interpretation of S-computations in T-computations to an interpretation of S^+-computations in T^+-computations. This captures the idea that our monad constructor is in some sense additive.

We use this property of monad constructors to build a modular view of denotational semantics in which a model can be extended by adding a new feature to the existing ones. This operation will be modular precisely when the monad constructor is an endofunctor.

There are other ideas that we would like to capture. We view a monad constructor $(-)^+$ as adding a new feature and therefore would like to be able to interpret T-computations in T^+-computations. This requires that for each T, we have a monad morphism:

$$\eta^+_T : T \to T^+$$

We would like to capture the idea that adding a feature twice gives no more expressive power than adding it once. This could be captured by requiring that $T^{++} = T^{+}$. However none of the examples of principle interest satisfy this condition. Instead we can require a monad morphism:

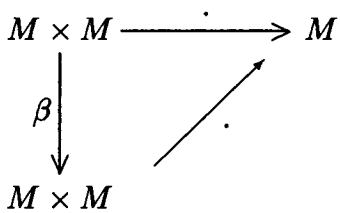
$$\mu_T^+ : T^{++} \to T^+.$$

such that $\mu_T^+ \eta_{T+}^+ = id$ and $\mu_T^+ (\eta_T^+)^+ = id$.

These conditions correspond to the tuple $\langle (-)^+, \eta^+, \mu^+ \rangle$ satisfying the last two diagrams defining it as a monad on $\mathrm{SMon}(\mathcal{C})$. In fact, we can gain some elegance, without losing any of the examples of primary interest to us, by requiring that $\langle (-)^+, \eta^+, \mu^+ \rangle$ is a monad on $\mathrm{SMon}(\mathcal{C})$.

Alas, $(-)_M$ does not, in general, extend to a monad on $\mathrm{SMon}(\mathcal{C})$. However, as we shall make precise, if $\mathcal{M}$ is a commutative monoid then we can extend $(-)_M$ to a monad on $\mathrm{SMon}(\mathcal{C})$.

Definition 4.2.20 A monoid $\langle M, 0, \cdot \rangle$ is **commutative** if the following diagram:



commutes, where $\beta = \langle \pi_2, \pi_1 \rangle$ is the twist map.

Henceforth in this section we assume that $\mathcal{M}$ is commutative.

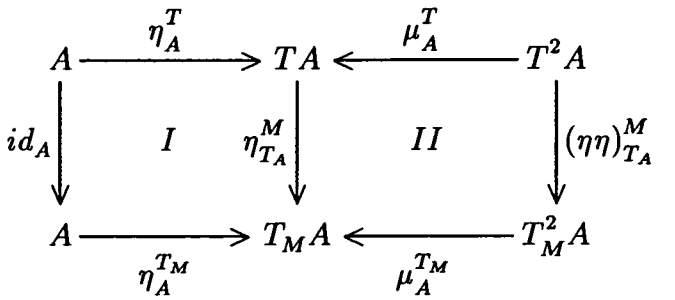
Remark 4.2.21 *In fact, most of the examples of primary interest to us are commutative monoids. However, the commutativity of the monoid $\mathcal{M}$ is only needed to show that diagram V in lemma 4.2.23 commutes.*

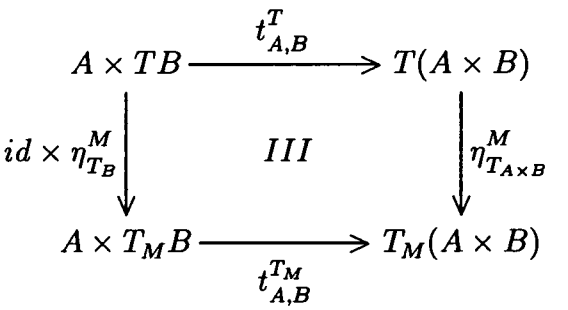
Definition 4.2.22 Let $\langle T, \eta^T, \mu^T, t^T \rangle$ be a strong monad on $\mathcal{C}$. We define the natural transformations $\eta_T^M : T \longrightarrow T_M$ and $\mu_T^M : T_{MM} \longrightarrow T_M$ by:

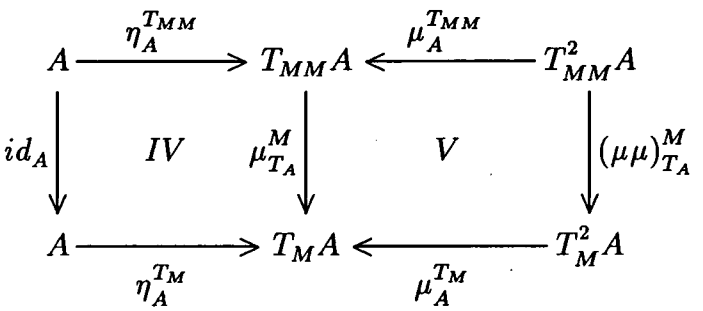
$$\eta_T^M = T\eta^M \quad \text{and} \quad \mu_T^M = T\mu^M.$$

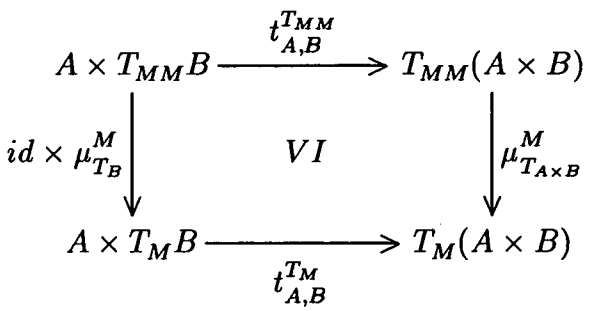
Lemma 4.2.23 η_T^M and μ_T^M are monad morphisms.

Proof: The coherence conditions η_T^M and μ_T^M to be monad morphisms are:









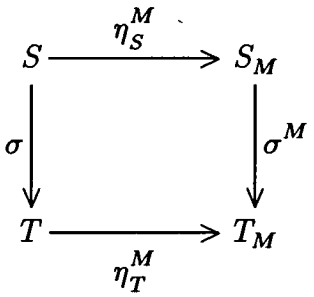
It is routine but tedious to verify that these commute and we relegate the proof to appendix A.3. □

For every monad T, we have shown that η^M_T and μ^M_T are monad morphisms. In fact if we write $\eta^M = \{\eta^M_T \; s.t. \; T \in |\mathrm{SMon}(\mathcal{C})|\}$ and $\mu^M = \{\mu^M_T \; s.t. \; T \in |\mathrm{SMon}(\mathcal{C})|\}$ then we have:

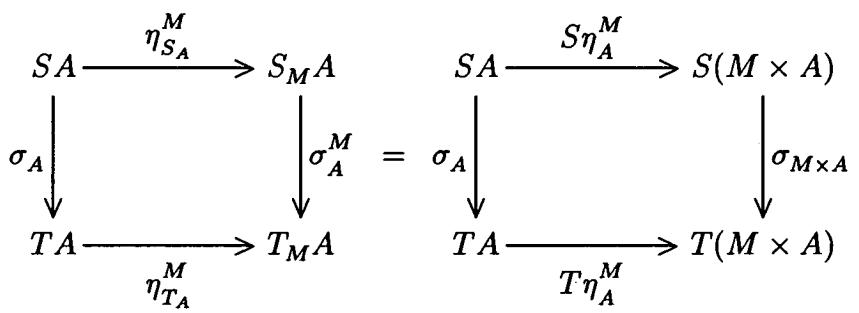
Lemma 4.2.24 η^M and μ^M are natural transformations:

$$\eta^M : Id_{SMon(\mathcal{C})} \longrightarrow (-)_M \quad \text{and} \quad \mu^M : ((-)_M)_M \longrightarrow (-)_M.$$

Proof: For η^M we require the following diagram:

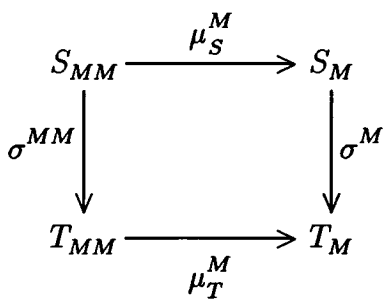


which holds if it holds pointwise, *i.e.* if for each A, we have:



which follows from the naturality of σ.

For μ^M we require the following diagram:



which holds if it holds pointwise, *i.e.* if we have:

$$\begin{array}{ccc} S(M \times (M \times A)) & \xrightarrow{S\mu^M_A} & S(M \times A) \\ \downarrow{\scriptstyle \sigma_{M\times(M\times A)}} & & \downarrow{\scriptstyle \sigma_{M\times A}} \\ T(M \times (M \times A)) & \xrightarrow[T\mu^M_A]{} & T(M \times A) \end{array}$$

which follows from the naturality of σ.

Thus η^M and μ^M are natural as was to be shown. □

Remark 4.2.25 *Warning! We have used η^M for two different natural transformations:*

$$\eta^M : Id_{\mathcal{C}} \longrightarrow M \times - \quad \text{and} \quad \eta^M : Id_{SMon(\mathcal{C})} \longrightarrow (-)_M$$

and similarly for μ^M. We do this for simplicity of notation and it will always be clear from the context which of the natural transformations is intended.

Finally we establish:

Proposition 4.2.26 $\langle (-)_M, \eta^M, \mu^M \rangle$ is a monad on SMon($\mathcal{C}$).

Proof: From lemmas 4.2.23 and 4.2.24 and proposition 4.2.19 it follows that $(-)_M$ is an endofunctor on SMon($\mathcal{C}$), that η^M is a natural transformation from $Id_{SMon(\mathcal{C})}$ to $(-)_M$ and that μ^M is a natural transformation from $((-)_M)_M$ to $(-)_M$. It remains to show that $\langle (-)_M, \eta^M, \mu^M \rangle$ satisfies diagrams 1 to 3 in definition 3.2.1.

Diagram 1 We require:

$$\begin{array}{ccc} T_{MMM} & \xrightarrow{\mu^{MM}_T} & T_{MM} \\ \downarrow{\scriptstyle \mu^M_{T_M}} & & \downarrow{\scriptstyle \mu^M_T} \\ T_{MM} & \xrightarrow[\mu^M_T]{} & T_M \end{array}$$

which commutes if it does so pointwise, *i.e.* if we have for each $A \in |\mathcal{C}|$:

$$\begin{array}{ccc} T(M \times (M \times (M \times A))) & \xrightarrow{T\mu^M_{M\times A}} & T(M \times (M \times A)) \\ \Big\downarrow{\scriptstyle T(id_M \times \mu^M_A)} & & \Big\downarrow{\scriptstyle T\mu^M_A} \\ T(M \times (M \times A)) & \xrightarrow[T\mu^M_A]{} & T(M \times A) \end{array}$$

which commutes because M is associative (recall that $\mu^M_A = \langle \cdot, id \rangle$).

Diagrams 2 and 3 at $A \in |\mathcal{C}|$ are:

$$\begin{array}{ccccc} T(M \times A) & \xrightarrow{T(id \times \eta^M_A)} & T(M \times (M \times A)) & \xleftarrow{T\eta^M_{M\times A}} & T(M \times A) \\ & \searrow^{T(id_{M\times A})} & \Big\downarrow{\scriptstyle T\mu^M_A} & \swarrow_{T(id_{M\times A})} & \\ & & T(M \times A) & & \end{array}$$

which commute because they are T applied to diagrams 2 and 3 in definition 3.2.1.

Thus $\langle (-)_M, \eta^M, \mu^M \rangle$ is a monad over SMon($\mathcal{C}$) as stated. □

4.3 Limits in SMon($\mathcal{C}$)

4.3.1 Introduction

We have shown that the monad constructor for complexity extends naturally to a monad on SMon($\mathcal{C}$), the category of strong monads and strong monad morphisms.

It is a mathematically natural question to ask under what conditions a monad constructor, which is a monad on SMon($\mathcal{C}$), might extend naturally to a strong monad on SMon($\mathcal{C}$). It is a computationally natural question to ask what this might mean.

To study strong monads on $\mathrm{SMon}(\mathcal{C})$ requires that $\mathrm{SMon}(\mathcal{C})$ has finite products. More generally, this leads naturally to the question of what limits might exist in $\mathrm{SMon}(\mathcal{C})$.

We prove that the forgetful functor $U : \mathrm{SMon}(\mathcal{C}) \longrightarrow [\mathcal{C},\mathcal{C}]$ creates limits that exist in $[\mathcal{C},\mathcal{C}]$ so that $\mathrm{SMon}(\mathcal{C})$ has whatever limits $\mathcal{C}$ has. This result has a number of applications:

- A major aim of Moggi's work [Mog89] was to study modularity in denotational semantics. We show how limits in $\mathrm{SMon}(\mathcal{C})$ can be valuable in studying the relationships between denotational models.

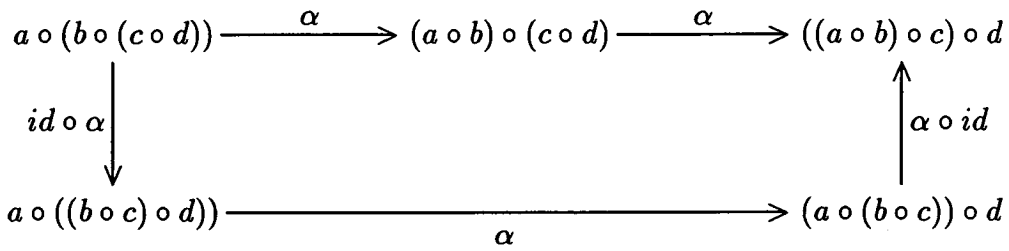
- A number of authors, for example [BrGu90], [MOMe89] and [Win88], have used categorical limits in various categories of models of computation to study notions of parallel or concurrent computation. Our result allows us to do the same in $\mathrm{SMon}(\mathcal{C})$ but with the advantage that we can study parallel and concurrent computation with features.

4.3.2 Limits

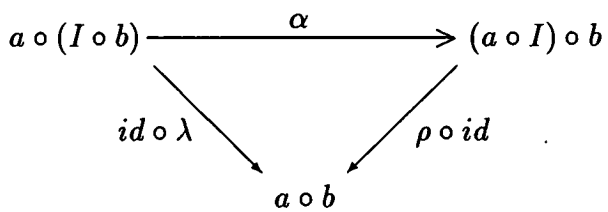
At this point, we introduce some additional notions from category theory. In particular, the definition of a monoidal category. These definitions are required in order to prove the results of this section, and also play an important role in chapters 5 and 6.

Definition 4.3.1 A **monoidal category** $\langle K, \circ, I, \alpha, \lambda, \rho\rangle$ is a category $\mathcal{K}$, together with a functor $\circ : \mathcal{K} \times \mathcal{K} \longrightarrow \mathcal{K}$, an object I of $\mathcal{K}$ and three natural isomorphisms, α, λ, ρ such that:

$$\alpha = \alpha_{a,b,c} : a \circ (b \circ c) \cong (a \circ b) \circ c. \quad \text{and}$$



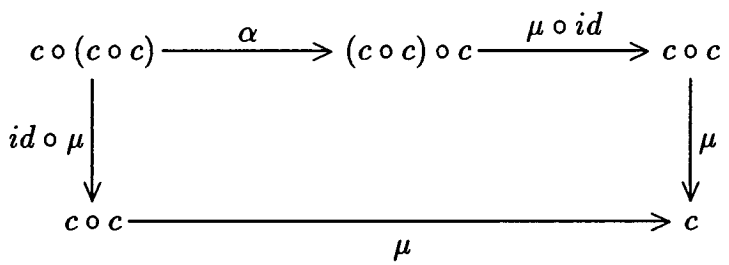
$\lambda = \lambda_a : I \circ a \cong a \quad \rho = \rho_a : a \circ I \cong a$ and

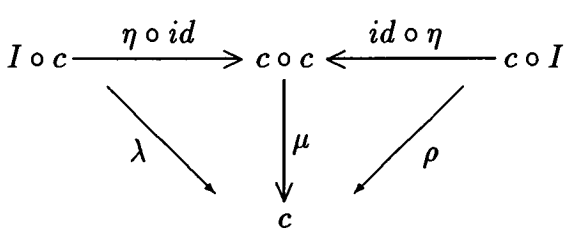


Definition 4.3.2 A **strict** monoidal category, $\langle \mathcal{K}, \circ, I \rangle$ is a monoidal category, $\langle \mathcal{K}, \circ, I, \alpha, \lambda, \rho \rangle$ in which the natural isomorphisms α, λ, ρ are all identities.

Example 4.3.3 Let $\mathcal{C}$ be a category with finite products. Then $\mathcal{C}$ is a monoidal category with the monoidal structure given by $\langle \mathcal{C}, \times, 1 \rangle$.

Definition 4.3.4 A **monoid** in a monoidal category, $\langle \mathcal{K}, \circ, I, \alpha, \lambda, \rho \rangle$ is an object, c of $\mathcal{K}$ and arrows $\mu : c \circ c \longrightarrow c$ and $\eta : I \longrightarrow c$ such that the following diagrams:





commute.

Example 4.3.5 Some examples of monoidal categories and monoids therein are:

$\langle \mathcal{K}, \circ, I \rangle$	Monoids in $\mathcal{K}$
$\langle \mathbf{Set}, \times, 1 \rangle$	Monoids
$\langle \mathbf{Ab}, \otimes, \mathbb{Z} \rangle$	Rings
$\langle \mathbf{SupLat}, \otimes, \mathcal{P}1 \rangle$	Quantales
$\langle \mathbf{Cat}, \times, \mathbf{1} \rangle$	Strict monoidal categories
$\langle \mathbf{o-Graph}, \times_o, o \to o \rangle$	Categories

Details appear in [Mac71] and [JoTi84].

Definition 4.3.6 A **morphism** $f : \langle c, \mu, \eta \rangle \longrightarrow \langle c', \mu', \eta' \rangle$ of monoids is an arrow $f : c \longrightarrow c'$ such that:

$$f\mu = \mu'(f \circ f) \quad \text{and} \quad f\eta = \eta'.$$

Proposition 4.3.7 Monoids and monoid morphisms in $\mathcal{K}$ together with the evident composition form a category, Monoids($\mathcal{K}$). Furthermore, the assignment $\langle c, \mu, \eta \rangle \mapsto c$ defines a forgetful functor, U : Monoids($\mathcal{K}$) $\longrightarrow \mathcal{K}$.

Notation 4.3.8 Let $\mathcal{C}$ be a category, we write $[\mathcal{C}, \mathcal{C}]$ for the category whose objects are functors from $\mathcal{C}$ to $\mathcal{C}$, and whose morphisms are natural transformations.

Remark 4.3.9 $[\mathcal{C}, \mathcal{C}]$ *is a strict monoidal category with the monoidal structure given by functor composition.*

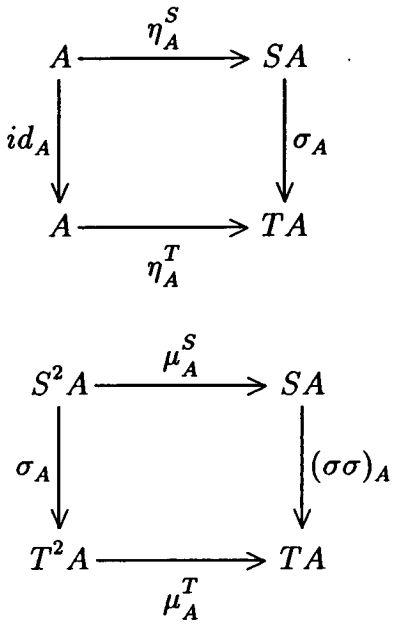
Proposition 4.3.10 Mon($\mathcal{C}$) $\cong$ Monoids($[\mathcal{C}, \mathcal{C}]$).

Proof: Let $\mathcal{C}$ be a category. It is routine to verify that the functor category $[\mathcal{C}, \mathcal{C}]$ is a strict monoidal category where $\circ$ is functor composition and I is $Id_{\mathcal{K}}$.

Trivially a monoid in $[\mathcal{C}, \mathcal{C}]$ is precisely a monad on $\mathcal{C}$.

It remains to show that a monad morphism is precisely a morphism of monoids.

Recall from definition 4.2.2 that a monad morphism $\sigma : S \longrightarrow T$ is a natural transformation such that the following diagrams:



commute, *i.e.* a monad morphism is an arrow $\sigma : S \longrightarrow T$ in $[\mathcal{C},\mathcal{C}]$ such that:

$$\mu^S\sigma \;=\; \mu^T(\sigma\sigma) \quad \text{and} \quad \sigma\eta_S \;=\; \eta^T$$

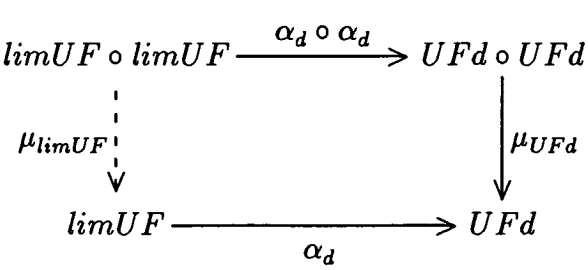
and thus monad morphisms on $\mathcal{C}$ are precisely morphisms of monoids in $[\mathcal{C},\mathcal{C}]$.

□

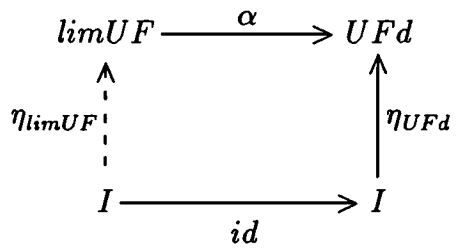
Proposition 4.3.11 Let $\langle \mathcal{K}, \circ, I, \alpha, \lambda, \rho \rangle$ be a monoidal category. If $\mathcal{K}$ has small limits then so does Monoids($\mathcal{K}$) and the forgetful functor U : Monoids($\mathcal{K}$) $\longrightarrow \mathcal{K}$ preserves them.

Proof: Let $\mathcal{D}$ be a small category and $F : \mathcal{D} \longrightarrow$ Monoids($\mathcal{K}$) a functor.

Then $limUF$ exits since $\mathcal{K}$ has all limits and we can define μ_{limUF} by:



Similarly we define η_{limUF} by:



It is routine but tedious to verify that $\langle limUF, \eta_{limUF}, \mu_{limUF}\rangle$ satisfies the conditions of definition 4.3.4 and thus is a monoid in $\mathcal{K}$.

Moreover, the comparison maps lift to maps in Monoids($\mathcal{K}$) and form a limiting cone for F.

The proof is relegated to appendix A.4. □

Proposition 4.3.12 If $\mathcal{C}$ has small limits, then so does $[\mathcal{C},\mathcal{C}]$ and they are computed pointwise.

Proof: Well known. □

Finally we can establish:

Theorem 4.3.13 If $\mathcal{C}$ has small limits then so does Mon($\mathcal{C}$) and these are computed pointwise.

Proof: By proposition 4.3.10, Mon($\mathcal{C}$) $\cong$ Monoids($[\mathcal{C},\mathcal{C}]$).
Hence by proposition 4.3.11 and proposition 4.3.12, Mon($\mathcal{C}$) has small limits and they are computed pointwise. □

Proposition 4.3.14 Let $\mathcal{C}$ be a category with small limits. Then SMon($\mathcal{C}$) has, and the forgetful functor U: SMon($\mathcal{C}$)$\longrightarrow$Mon($\mathcal{C}$) preserves, small limits.

Proof: Let $F : \mathcal{D} \longrightarrow$ SMon($\mathcal{C}$) be a functor with $\mathcal{D}$ small. Let $T = limUF$ in Mon($\mathcal{C}$) with projections $(\alpha_d | d \in \mathcal{D})$, this limit exists by theorem 4.3.13. Since the

forgetful functor $U : \mathrm{Mon}(\mathcal{C}) \longrightarrow [\mathcal{C},\mathcal{C}]$ preserves all limits, the collection $(t_d | d \in \mathcal{D})$ of tensorial strengths of each Fd yields a unique map $t : - \times T- \longrightarrow T(- \times -)$ commuting with the t_d's and the evident projections. Since limits in $[\mathcal{C},\mathcal{C}]$ are given pointwise, it follows that t satisfies the axioms to make it a tensorial strength for T, and the definition of t forces each α_d to be a strong monad morphism.
Now, given any strong monad S and a cone $S \longrightarrow F$, there exists a unique monad morphism from US to T which commutes with the projection maps. This monad morphism is necessarily strong since limits in $\mathrm{SMon}(\mathcal{C})$ are given pointwise. □

Corollary 4.3.15 Let $\mathcal{C}$ be a category with finite products. If $\mathcal{C}$ has small limits then so does $\mathrm{SMon}(\mathcal{C})$ and these are computed pointwise.

Proof: Immediate from theorem 4.3.13 and proposition 4.3.14. □

Example 4.3.16 Let $\mathcal{C}$ be a category with binary products, and let T and S be strong monads on $\mathcal{C}$. Now by corollary 4.3.15, limits are given pointwise in $\mathrm{SMon}(\mathcal{C})$ and therefore, the strong monad $T \times S$ is given by:

- $T \times S(-) = T(-) \times S(-)$,
- $\eta_A^{T\times S} = \langle \eta_A^T, \eta_A^S \rangle$,
- $\mu_A^{T\times S} = (\mu_A^T T\pi_1) \times (\mu_A^S S\pi_2)$ and
- $t_{A,B}^{T\times S} = \langle id \times \pi_1 t_{A,B}^T, id \times \pi_2 t_{A,B}^S \rangle$.

Example 4.3.17 Let $\mathcal{C}$ be a category with a terminal object. Then the terminal object, $1 \in \mathrm{SMon}(\mathcal{C})$ is given by:

- $1(-) = 1$,
- $\eta_A^1 = A \xrightarrow{*} 1$,

- $\mu^1_A = id_1$ and
- $t^1_{A,B} = A \times 1 \xrightarrow{*} 1$.

4.3.3 Equalizers and pullbacks

In a modular approach to denotational semantics, it is important to develop techniques for comparing denotational models such as, for example, the η map interpreting T in T^+. An example of the sort of question which arises is:

- Given two interpretations of S in T what part of S do they agree on?

Let $\mathcal{C}$ be a category with finite limits. Then this question can be answered as follows.

Suppose we have two monad morphisms, *i.e.* interpretations of S in T:

$$S \underset{\tau}{\overset{\sigma}{\underset{\times}{\rightrightarrows}}} T$$

Then consider the equalizer in $\mathrm{SMon}(\mathcal{C})$:

$$E \rightarrowtail S \underset{\tau}{\overset{\sigma}{\underset{\times}{\rightrightarrows}}} T$$

Since E is an object of $\mathrm{SMon}(\mathcal{C})$ it gives us the largest strong monad contained in S on which σ and τ agree, *i.e.* E represents the largest notion of computation which is common to the two interpretations.

4.3.4 Products as parallel composition

Given a monad T on a category with finite products, an arrow from A to B in the Kleisli category for T is regarded as a program taking a value of type A to a computation of type B. So, given monads T and S over a category with finite products, an arrow in the Kleisli category for the monad $T \times S$ is regarded as

a program taking a value of type A to a pair consisting of a T-computation of type B and an S-computation of type B. The composition in the Kleisli category keeps these two computation paths separate, as a consequence of the π terms in the definition of $\mu^{T\times S}$. Thus we can see that the monad $T \times S$ should represent non-communicating parallel computation, as indicated in the introduction to this section. In the category SMon($\mathcal{C}$), cartesian products should give us not just parallel composition but parallel composition of computations with features.

Unfortunately, it does not suffice to take the product of two monads to obtain parallel composition with features, since the product of a S^+-computation and a T^+-computation is an $S^+ \times T^+$-computation and not an $(S \times T)^+$-computation as we should expect. One way of obtaining an $(S \times T)^+$-computation from an $S^+ \times T^+$-computation is to require that the monad constructor $(-)^+$ be a strong monad on SMon($\mathcal{C}$).

In section 4.4 we show that $(-)_M$ extends naturally to a strong monad on SMon($\mathcal{C}$).

4.4 Strong monad constructors

4.4.1 Introduction

In section 4.3.4, we mentioned that it does not suffice to take the product of two monads to obtain parallel composition with features, since the product of an S^+-computation and a T^+-computation is an $S^+ \times T^+$-computation and not an $(S \times T)^+$-computation as is required. We said that one way of obtaining an $(S\times T)^+$-computation from an $S^+ \times T^+$-computation is to demand that the monad constructor $(-)^+$ be a strong monad on SMon($\mathcal{C}$).

This is analogous to the need for a strong monad in Moggi's original work [Mog88b]. There, a strong monad is needed in order to construct computations of pairs of values. Here, a strong monad is needed in order to construct pairs of parallel computations. However, there is an important difference.

In [Mog88b], an operation of pairing of morphisms is defined which is essentially sequential composition, *i.e.* to compute a pair we compute the first argument and then compute the second argument. An example is the monad for time complexity in which the time taken to compute a pair is the sum of the times taken to compute each individual component of the pair. In general, there are two different pairing operations depending on which argument is to be computed first.

We wish to give a monad representing parallel composition in which, for example, the time taken to compute a pair of values is the maximum of the times taken to compute each individual component of the pair.

In this section, we define strong monad constructors and prove that $(-)_M$ extends naturally to a strong monad constructor.

4.4.2 Strong monad constructors

Definition 4.4.1 A **strong monad constructor** over a category $\mathcal{C}$ with finite products is a strong monad on $\mathrm{SMon}(\mathcal{C})$.

Remark 4.4.2 *It is immediate that a strong monad constructor is also a monad constructor.*

Example 4.4.3 The monad constructor for exceptions [Mog89] extends naturally to a strong monad constructor via:

- $T_E = T(- + E)$,
- $\eta^E_T = T(inl)$,
- $\mu^E = T([id, inr])$ and
- $t^E_{S,T} = \langle S(inl), id \rangle$.

We omit the proof.

4.4.3 The strong monad constructor for complexity

We prove that $(-)_M$ extends naturally to a strong monad constructor.

Definition 4.4.4 Let $\langle S, \eta^S, \mu^S, t^S \rangle$ and $\langle T, \eta^T, \mu^T, t^T \rangle$ be a strong monads on $\mathcal{C}$. We define the natural transformation $t^M_{S,T} : S \times T_M \longrightarrow (S \times T)_M$ by:

$$t^M_{S,T_A} = S\eta^M_A \times id_{T(M\times A)} = \eta^M_{S_A} \times id_{T(M\times A)}.$$

Lemma 4.4.5 $t^M_{S,T}$ is a monad morphism.

Proof: This is routine but tedious and is relegated to appendix A.5. □

Proposition 4.4.6 $\langle (-)_M, \eta^M, \mu^M, t^M \rangle$ is a strong monad constructor on $\mathcal{C}$.

Proof: By proposition 4.2.26, $\langle (-)_M, \eta^M, \mu^M \rangle$ is a monad on $\mathrm{SMon}(\mathcal{C})$.

It remains to verify that t^M is natural is S and T and that it satisfies diagrams 4-7 in definition 3.2.5.

This is routine but tedious and is relegated to appendix A.6. □

Chapter 5

Internal and External Data

5.1 Introduction

In chapter 3, in order to reason about complexity, we extended existing semantic frameworks by considering a category $\mathcal{C}$ with finite products and a monoid $\mathcal{M}$ in $\mathcal{C}$. We then modelled programs in the Kleisli category of the monad $M \times -$. In chapter 4, we extended this to a monad constructor, motivated by the search for modularity in denotational semantics. Unfortunately, this approach is not completely satisfactory for a number of reasons.

Firstly, when one is modelling the complexity of a program, one expects that the complexity should eventually yield an actual natural number. This restricts us to the case where $\mathcal{C}$ is a suitable subcategory of **Set**. However, it is well known [Ren84], [Pit89], [RoRo90] that there are languages which cannot be satisfactorily modelled purely set theoretically.

Secondly, we are often interested in comparing the complexity of programs written in different programming languages. For example, suppose that p was a quicksort program written in C and that q was a quicksort program written in ML. Then, in practice, we can compare the complexities of p and q by simply executing them. However, in order to compare their complexities in our current framework, we would need to model both languages in the same category. One might argue

that there should be a single category which models all programming languages, and this would enable us to use our existing framework. However, one sometimes wants mutually inconsistent properties of models for different languages, such as requiring a category to be cartesian closed and have an initial object and least fixed points for all maps. In practice, many different categories are used to model programming languages. We would like to compare the complexity of program modelled in a category $\mathcal{C}$ with one modelled in a category $\mathcal{C}'$.

Thirdly, there are various problems with modelling languages with higher-order functions, such as ML. We discuss some of the issues involved in chapter 9.

Finally, in the analysis of algorithms and in complexity theory, program complexity is often given non-exactly. For example, in the analysis of algorithms, one is often interested in the order of magnitude of the complexity maps. In complexity theory, one often asks whether a complexity map is polynomial or super-polynomial. In order to model non-exact complexity, we seek models in which the functional maps are morphisms and the complexity maps are equivalence classes of morphisms. Unfortunately, this cannot be done within a single category such as $\mathcal{C}_M$. These issues are studied in detail in chapters 6, 7 and 8.

The underlying problem in each of these cases is that we are modelling functional maps and complexity maps in the same category.

In this chapter, we extend our framework to incorporate these examples and compare our new framework with the work in chapter 4.

In section 5.2 we give an internal version of complexity categories which correspond to the definitions in chapter 4. We define an internal datum and a category **Int** of internal data and an internal complexity category and prove that:

- every internal datum generates an internal complexity category, and
- this construction extends to a functor $\mathcal{I} : \mathbf{Int} \longrightarrow \mathbf{Cat}$.

In section 5.3 we explore the idea of functional and complexity maps living in different categories, leading to the definition of an external datum and a category **Ext** of external data. We define an external complexity category and we prove:

- Every external datum generates an external complexity category.

- This construction extends to a functor $\mathcal{E}$: **Ext**$\longrightarrow$**Cat**.

We show that this captures all the standard examples of complexity theory.

Finally in section 5.4 we explore the relationship between the internal and the external approach. We define functors:

$$out : \mathbf{Int} \longrightarrow \mathbf{Ext} \quad \text{and} \quad in : \mathbf{Ext} \longrightarrow \mathbf{Int}$$

and prove that there is an adjunction $in \dashv out$: **Int** $\rightarrow$ **Ext** and that *out* is fully faithful. Therefore:

- **Int** is a full reflective subcategory of **Ext**.

Finally we show that:

- $\mathcal{I} \cong \mathcal{E}\ out$ in the functor category [**Int**, **Cat**].

These results show that **Ext** is more general than **Int** and we give an example in **Ext** which is not naturally an example in **Int**.

In chapters 6, 7 and 8, we show how the external approach can be used to capture several concepts of importance in complexity theory and the analysis of algorithms. In particular, we show how to capture the notion of upper bounds on complexity, input measures and non-exact complexity.

Nielson [Niel84] has studied the notion of upper bounds on time complexity for her specific languages, and the system of Flajolet, Salvy and Zimmerman [FSZ89] produces an estimate of the average complexity of programs. However, in this thesis we attempt to give a coherent general account of these issues. As far as we are aware, there have been no previous studies of this kind.

5.2 Internal complexity categories

5.2.1 Introduction

We define the notion of an internal datum and show how to construct an internal complexity category from an internal datum. We construct a category **Int** of internal data and show that the construction of internal complexity categories yields a functor from **Int** to **Cat**.

5.2.2 Internal data

Definition 5.2.1 An **internal datum** is a pair $\langle \mathcal{C}, \mathcal{M} \rangle$ consisting of a category $\mathcal{C}$ with finite products and a monoid $\mathcal{M} = \langle M, 0, \cdot \rangle$ in $\mathcal{C}$.

The idea is that the monoid of resource values $\mathcal{M}$ is an object of $\mathcal{C}$, and the complexity maps are maps in $\mathcal{C}$.

We proceed to define a category of internal data.

Definition 5.2.2 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ and $\langle \mathcal{C}', \mathcal{M}' \rangle$ be internal data. A **morphism of internal data** is a functor $F : \mathcal{C} \longrightarrow \mathcal{C}'$ such that:

- F preserves finite products, and
- $F(M) \cong M'$ and this isomorphism is coherent with respect to the monoid operations.

Proposition 5.2.3 Internal data and morphisms of internal data together with functor composition define a category **Int**.

Proof: As functor composition is associative, it suffices to verify that if $F : \mathcal{C} \longrightarrow \mathcal{C}'$ and $G : \mathcal{C}' \longrightarrow \mathcal{C}''$ are morphisms of internal data, then so is GF, and that

identity functors are morphisms of internal data. This is straightforward. □

Internal data allow us to define complexity categories as follows. We recall the following result from chapter 4:

Proposition 5.2.4 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be an internal datum. The following data:

- $T(-) = M \times -$,
- $\eta_A = A \xrightarrow{\cong} 1 \times A \xrightarrow{\langle 0, id \rangle} M \times A$,
- $\mu_A = M \times (M \times A) \xrightarrow{\langle \cdot, id \rangle} M \times A$ and
- $t_{A,B} = A \times (M \times B) \xrightarrow{\alpha} M \times (A \times B)$

define a strong monad on $\mathcal{C}$.

Notation 5.2.5 We shall abbreviate this to "the monad $M \times -$".

Definition 5.2.6 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be an internal datum. The internal complexity category $\mathcal{C}_M$ is the Kleisli category for the monad $M \times -$ on $\mathcal{C}$.

A morphism from A to B in $\mathcal{C}_M$ is a map $\langle f, t \rangle : A \longrightarrow B \times M$ in $\mathcal{C}$ and composition of maps is given by $\langle g, s \rangle \langle f, t \rangle = \langle gf, t \cdot sf \rangle$. The idea is that the first component of the pair models the functional behaviour of the program and the second component models its complexity. We can extend definition 5.2.6 to a functor from **Int** to **Cat** as follows:

Lemma 5.2.7 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ and $\langle \mathcal{C}', \mathcal{M}' \rangle$ be internal data and let F be a morphism of internal data from $\langle \mathcal{C}, \mathcal{M} \rangle$ to $\langle \mathcal{C}', \mathcal{M}' \rangle$. Then the assignment:

$$\langle f, t \rangle : A \longrightarrow B \longmapsto \langle Ff, Ft \rangle : FA \longrightarrow FB$$

defines a functor $F^* : \mathcal{C}_M \longrightarrow \mathcal{C}'_{M'}$.

Proof: F^* is well-defined since by assumption it preserves finite products and the monoid structure.

F^* preserves composition since:

- $$\begin{aligned}\langle Fg, Fs\rangle\langle Ff, Ft\rangle &= \langle FgFf, Ft \cdot FsFf\rangle \\ &= \langle F(gf), F(t \cdot sf)\rangle \\ &= F^*\langle gf, t \cdot sf\rangle \\ &= F^*(\langle g, s\rangle\langle f, t\rangle).\end{aligned}$$

This completes the proof. □

Proposition 5.2.8 The assignment:

- $\mathcal{C} \longmapsto \mathcal{C}_M$,
- $F \longmapsto F^*$,

defines a functor $\mathcal{I} : \mathbf{Int} \longrightarrow \mathbf{Cat}$.

Proof: $\mathcal{I}$ is well-defined by lemma 5.2.7 and it is clear that $\mathcal{I}$ preserves identities. It remains to verify that $\mathcal{I}$ preserves composition:

- $$\begin{aligned}\mathcal{I}(FG) &= (FG)^* \\ &= F^*G^* \\ &= \mathcal{I}(F)\mathcal{I}(G).\end{aligned}$$

This completes the proof. □

5.3 External complexity categories

5.3.1 Introduction

We define the notion of an external datum and show how to construct an external complexity category from an external datum. We construct a category **Ext** of external data and show that the construction of external complexity categories yields a functor from **Ext** to **Cat**.

5.3.2 External data

Recall definitions 4.3.1, 4.3.2, 4.3.4 , 4.3.6 and 4.3.7 of monoidal category, strict monoidal category, monoid, monoid morphism and Monoids($\mathcal{K}$).

Remark 5.3.1 *Let* Monoids *be the category* Monoids(**Set**). *Then the category* Monoids([$\mathcal{C}$,**Set**]) *is isomorphic to the functor category* [$\mathcal{C}$,Monoids].

Definition 5.3.2 An **external datum** is a tuple $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ consisting of a category $\mathcal{C}$, a category with finite products $\mathcal{D}$, a functor $U : \mathcal{C} \to \mathcal{D}$, a monoid $\mathcal{M} = \langle M, 0, \cdot \rangle$ in $\mathcal{D}$ and a subobject $\phi^{\mathcal{C}}$ of the functor $\mathcal{D}(U(-), M)$ in the category Monoids([$\mathcal{C}^{\mathrm{op}}$,**Set**]).

The idea is as follows. Let $\mathcal{C}$ be a category which is a model for a programming language, for example a domain. We map $\mathcal{C}$ via the functor U into a category $\mathcal{D}$ which has finite products and contains a monoid $\mathcal{M}$. The idea is that our complexity maps live naturally in $\mathcal{D}$. The functor U allows us to have both functional maps and complexity maps in the same category, so that we can compose them, without requiring that the complexity maps have to live in our model $\mathcal{C}$. Typically, $\mathcal{D}$ might be **Set**.

Now, for each object A of $\mathcal{C}$, we need to specify a set of maps $\phi^{\mathcal{C}}(A)$ from UA to M in $\mathcal{D}$ which are the possible complexity maps for A. For example, if $\mathcal{D}$ was

Set and both UA and M were $\mathbf{N}$, we might choose the polynomial functions. We could simply take $\phi^{\mathcal{C}}(A)$ to be all the maps from UA to M in $\mathcal{D}$, but this is not sufficiently general, (*c.f.* example 5.3.14).

Example 5.3.3 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be an internal datum. Then the tuple $\langle \mathcal{C}, \mathcal{C}, id, \mathcal{M}, \mathcal{C}(-, M) \rangle$ is an external datum.

Example 5.3.4 Let $\mathcal{C}$ be **CPO**, the category of complete partial orders and monotone continuous maps and let $\mathcal{M}$ be the flat lattice of the natural numbers. Note that this is the natural numbers object in **CPO**. Then any complexity map $t : A \longrightarrow M$ in the internal complexity category $\mathbf{CPO}_M$ must be monotone and continuous. However, in M, we have $\perp < n$ for all n and no other relations. Therefore, any complexity map must be constant on any values which are related. This is clearly unsatisfactory, as very few programs have constant complexity.

However, let $\mathcal{D}$ be **Set** and let U be the forgetful functor from **CPO** to **Set** mapping a complete partial order to its underlying set. Then the tuple $\langle \mathbf{CPO}, \mathbf{Set}, U, U(\mathcal{M}), \mathbf{Set}(-, M) \rangle$ is an external datum and allows arbitrary functions from A to $\mathbf{N} \cup \{\perp\}$ to be complexity maps.

Further examples appear at the end of this section.

In order to model trivial computations and the composition of programs, we require certain properties of the sets $\phi^{\mathcal{C}}(A)$. These are, that each $\phi^{\mathcal{C}}(A)$ contains the zero map, that if s and t are elements of $\phi^{\mathcal{C}}(A)$ then so is $s \cdot t$, and that if s is an element of $\phi^{\mathcal{C}}(B)$ and f is a morphism from B to A in $\mathcal{C}$, then sUf is an element of $\phi^{\mathcal{C}}(A)$. The following result shows that definition 5.3.2 captures exactly these conditions.

Proposition 5.3.5 To give a subobject $\phi^{\mathcal{C}} \leq \mathcal{D}(U(-), M)$ in Monoids($[\mathcal{C}^{\mathrm{op}}$,**Set**]) is to give for each object A of $\mathcal{C}$, a subset $\phi^{\mathcal{C}}(A)$ of $\mathcal{D}(UA, M)$ such that:

- $t \in \phi^{\mathcal{C}}(B),\ f : A \longrightarrow B \in \mathcal{C} \ \Rightarrow\ tUf \in \phi^{\mathcal{C}}(A),$

- $s, t \in \phi^{\mathcal{C}}(A) \Rightarrow s \cdot t \in \phi^{\mathcal{C}}(A)$ and
- $UA \xrightarrow{*} 1 \xrightarrow{0} M \in \phi^{\mathcal{C}}(A)$.

Proof: Let $\phi^{\mathcal{C}}$ be a subobject of $\mathcal{D}(U-, M)$ in Monoids($[\mathcal{C}^{\mathrm{op}}$,**Set**]). Then for each $f : A \longrightarrow B$ in $\mathcal{C}$ we have:

$$\begin{array}{ccc} \phi^{\mathcal{C}}(A) & \xrightarrow{\iota_A} & \mathcal{D}(UA, M) \\ {\scriptstyle \phi^{\mathcal{C}}(f)}\uparrow & & \uparrow{\scriptstyle \mathcal{D}(Uf, M)} \\ \phi^{\mathcal{C}}(B) & \xrightarrow[\iota_B]{} & \mathcal{D}(UB, M) \end{array}$$

but $\mathcal{D}(Uf, M)(t) = tUf$ and thus $t \in \phi^{\mathcal{C}}(B)$ implies $tUf \in \phi^{\mathcal{C}}(A)$.
Since $\phi^{\mathcal{C}} \leq \mathcal{D}(U-, M)$ in Monoids($[\mathcal{C}^{\mathrm{op}}$,**Set**]), $\phi^{\mathcal{C}}(A)$ inherits the monoid structure from $\mathcal{D}(UA, M)$ *i.e.*

- $s, t \in \phi^{\mathcal{C}}(A) \Rightarrow s \cdot t \in \phi^{\mathcal{C}}(A)$ and
- $UA \xrightarrow{*} 1 \xrightarrow{0} M \in \phi^{\mathcal{C}}(A)$.

Conversely, it is trivial to see that a collection of subsets $\phi^{\mathcal{C}}(A) \leq \mathcal{D}(UA, M)$ satisfying the above conditions define a subobject $\phi^{\mathcal{C}}$ of $\mathcal{D}(U(-), M)$.
This completes the proof. □

We now proceed to define a category **Ext** of external data.

Definition 5.3.6 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$ be external data. A **morphism of external data** is a pair of functors, $\langle F : \mathcal{C} \longrightarrow \mathcal{C}', G : \mathcal{D} \longrightarrow \mathcal{D}' \rangle$ such that:

- $GU = U'F$,
- G preserves finite products,
- As monoids, $G(\mathcal{M}) = \mathcal{M}'$ up to coherent isomorphism and

- $G\phi^{\mathcal{C}} \leq \phi^{\mathcal{C}'}F$ in $[\mathcal{C}^{\mathbf{op}},\mathbf{Set}]$.

This last condition is characterized by the following result:

Proposition 5.3.7 Given external data $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$ and functors $F : \mathcal{C} \to \mathcal{C}'$ and $G : \mathcal{D} \to \mathcal{D}'$ such that $GU = U'F$, the condition $G\phi^{\mathcal{C}} \leq \phi^{\mathcal{C}'}F$ is equivalent to:

$$\forall\, A \in |\mathcal{C}|,\ t \in \phi^{\mathcal{C}}(A) \ \Rightarrow\ Gt \in \phi^{\mathcal{C}'}(FA).$$

Proof: $G\phi^{\mathcal{C}} \leq \phi^{\mathcal{C}'}F$ implies that we have a monomorphic natural transformation $\iota : G\phi^{\mathcal{C}} \longrightarrow \phi^{\mathcal{C}'}F$ *i.e.* $\forall A \in |\mathcal{C}|\ \exists \iota_A$ monomorphic such that:

$$G\phi^{\mathcal{C}}(A) \overset{\iota_A}{\longmapsto} \phi^{\mathcal{C}'}(FA)$$

which implies $\forall t \in \phi^{\mathcal{C}}(A).\ Gt \in \phi^{\mathcal{C}'}(FA)$.

Conversely, for each $A \in |\mathcal{C}|$, let ι_A be the inclusion of $\phi^{\mathcal{C}}(A)$ in $\phi^{\mathcal{C}'}(FA)$. These will form the components of an inclusion $G\phi^{\mathcal{C}} \leq \phi^{\mathcal{C}'}F$. It remains to prove naturality.

Let $f : A \longrightarrow B \in \mathcal{C}$, we require:

$$\begin{array}{ccc} G\phi^{\mathcal{C}}(A) & \overset{\iota_A}{\rightarrowtail} & \phi^{\mathcal{C}'}(FA) \\ {\scriptstyle G\phi^{\mathcal{C}}(f)}\downarrow & & \downarrow{\scriptstyle \phi^{\mathcal{C}'}(Ff)} \\ G\phi^{\mathcal{C}}(B) & \underset{\iota_B}{\rightarrowtail} & \phi^{\mathcal{C}'}(FB) \end{array}$$

Suppose that $Gt \in G\phi^{\mathcal{C}}(B)$, then:

- $$\begin{aligned} G\phi^{\mathcal{C}}(f)(Gt) &= G(tUf) \\ &= (Gt)(GUf) \\ &= (Gt)(U'Ff) \\ &= \phi^{\mathcal{C}'}(Ff)(Gt) \end{aligned}$$

as required. This completes the proof. □

Proposition 5.3.8 External data, morphisms of external data and composition defined by:

$$\langle F_2, G_2\rangle\langle F_1, G_1\rangle \;=\; \langle F_2F_1, G_2G_1\rangle$$

determine a category **Ext**.

Proof: Composition is clearly associative. It is clear that $\langle Id, Id\rangle$ is a morphism of external data and defines the identity. It remains only to verify that $\langle F_2F_1, G_2G_1\rangle$ is a morphism of external data.

- $G_1U_1 \;=\; U_2F_1$ and $G_2U_2 \;=\; U_3F_2$.
  Therefore $G_2G_1U_1 \;=\; G_2U_2F_1 \;=\; U_3F_2F_1$ as required.
- G_1 and G_2 preserve finite products so G_2G_1 does.
- $G_1\mathcal{M}_1 \;=\; \mathcal{M}_2$ and $G_2\mathcal{M}_2 \;=\; \mathcal{M}_3$ therefore $G_2G_1\mathcal{M}_1 \;=\; \mathcal{M}_3$.
- $G_1\phi^{\mathcal{C}_1} \;\leq\; \phi^{\mathcal{C}_2}F_1$ and $G_2\phi^{\mathcal{C}_2} \;\leq\; \phi^{\mathcal{C}_3}F_2$.
  Therefore $G_2G_1\phi^{\mathcal{C}_1} \;\leq\; G_2\phi^{\mathcal{C}_2}F_1 \;\leq\; \phi^{\mathcal{C}_3}F_2F_1$ as required.

This completes the proof. □

External data allow us to define external complexity categories as follows:

Proposition 5.3.9 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$ be an external datum. The following data:

- objects: objects of $\mathcal{C}$,
- morphisms: pairs $\langle f, t\rangle$ where $f : A \longrightarrow B \in \mathcal{C}$ and $t : UA \longrightarrow M \in \phi^{\mathcal{C}}(A)$ and
- composition: $\langle g, s\rangle\langle f, t\rangle \;=\; \langle gf, t \cdot sUf\rangle$

define a category $\mathcal{C}^{\mathcal{M}}$.

Proof: The identity at A is $\langle id, 0\rangle$ and:

- $\langle f,t\rangle\langle id,0\rangle = \langle fid, 0\cdot tUid\rangle$
  $= \langle f, 0\cdot t\rangle$
  $= \langle f,t\rangle.$

- $\langle id,0\rangle\langle f,t\rangle = \langle idf, t\cdot 0f\rangle$
  $= \langle f, t\cdot 0\rangle$
  $= \langle f,t\rangle.$

Composition is well-defined since,

- $\langle f,t\rangle, \in \mathcal{C}^M(A,B),\ \langle g,s\rangle \in \mathcal{C}^M(B,C) \Rightarrow\ f\in\mathcal{C}(A,B),\ g\in\mathcal{C}(B,C)$
  $\Rightarrow\ sf\in\mathcal{C}(B,C),$

- $\langle f,t\rangle, \in \mathcal{C}^M(A,B),\ \langle g,s\rangle \in \mathcal{C}^M(B,C) \Rightarrow\ t\in\phi^{\mathcal{C}}(A),\ s\in\phi^{\mathcal{C}}(B)$
  $\Rightarrow\ t\in\phi^{\mathcal{C}}(A),\ sUf\in\phi^{\mathcal{C}}(A)$
  $\Rightarrow\ t\cdot sUf\in\phi^{\mathcal{C}}(A)$ and

- $\langle f,t\rangle, \in \mathcal{C}^M(A,B),\ \langle g,s\rangle \in \mathcal{C}^M(B,C) \Rightarrow\ \langle gf, t\cdot sUf\rangle\in\mathcal{C}^M(A,C).$

Composition is associative since,

- $\langle f,r\rangle(\langle g,s\rangle\langle h,t\rangle) = \langle f,r\rangle\langle gh, t\cdot sUh\rangle$
  $= \langle f(gh), (t\cdot sUh)\cdot rUgh\rangle$
  $= \langle (fg)h, t\cdot(sUh\cdot rUgh)\rangle$
  $= \langle (fg)h, t\cdot(s\cdot rUg)Uh\rangle$
  $= \langle fg, s\cdot rUg\rangle\langle h,t\rangle$
  $= (\langle f,r\rangle\langle g,s\rangle)\langle h,t\rangle.$

This completes the proof. □

Notation 5.3.10 We call $\mathcal{C}^M$ an external complexity category.

We give examples of external complexity categories at the end of this section. We can extend the construction of $\mathcal{C}^M$ to a functor from **Ext** to **Cat** as follows:

Lemma 5.3.11 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$ be external data and let $\langle F : \mathcal{C} \longrightarrow \mathcal{C}', G : \mathcal{D} \longrightarrow \mathcal{D}' \rangle$ be a morphism of external data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ to $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$. Then the assignment:

$$\langle f, t \rangle : A \longrightarrow B \longmapsto \langle Ff, Gt \rangle : FA \longrightarrow FB$$

defines a functor $F \cdot G : \mathcal{C}^{\mathcal{M}} \longrightarrow \mathcal{C}'^{\mathcal{M}'}$.

Proof: $F \cdot G$ is well-defined since $G\phi^{\mathcal{C}} \leq \phi^{\mathcal{C}'} F$ and therefore $Gt \in \phi^{\mathcal{C}'}(FA)$.
$F \cdot G$ preserves composition since:

- $$\begin{aligned} \langle Fg, Gs \rangle \langle Ff, Gt \rangle &= \langle FgFf, Gt \cdot GsU'Ff \rangle \ \text{(defn)} \\ &= \langle F(gf), Gt \cdot GsGUf \rangle \ (U'F = GU) \\ &= \langle F(gf), G(t \cdot sUf) \rangle \ (G \text{ preserves } \mathcal{M}) \\ &= F \cdot G(\langle gf, t \cdot sUf \rangle) \ \text{(defn)} \\ &= F \cdot G(\langle g, s \rangle \langle f, t \rangle) \ \text{(defn)}. \end{aligned}$$

This completes the proof. □

Proposition 5.3.12 The assignment:

- $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle \longmapsto \mathcal{C}^{\mathcal{M}}$,
- $\langle F, G \rangle \longmapsto F \cdot G$,

defines a functor $\mathcal{E} : \mathbf{Ext} \longrightarrow \mathbf{Cat}$.

Proof: $\mathcal{E}$ is well-defined by lemma 5.3.11. It is clear that $\mathcal{E}$ preserves identities. It remains to verify that $\mathcal{E}$ respects composition.

- $$\begin{aligned} \mathcal{E}(\langle F_1F_0, G_1G_0 \rangle) &= (F_1F_0) \cdot (G_1G_0) \\ &= (F_1 \cdot G_1)(F_0 \cdot G_0) \\ &= \mathcal{E}(\langle F_1, G_1 \rangle)\mathcal{E}(\langle F_0, G_0 \rangle). \end{aligned}$$

This completes the proof. □

Remark 5.3.13 *Note that the construction of the functor* $\mathcal{E} : \mathbf{Ext} \longrightarrow \mathbf{Cat}$ *relies on all the conditions in the definition of a morphism of external data.*

We show how our framework captures the examples mentioned in the introduction to this chapter.

We often model possibly non-terminating programs. The following example shows how we can model the complexity of such partial computations within our framework.

Example 5.3.14 Let $\mathbf{Set_p}$ be the category of sets and partial functions, let $(-)_\perp : \mathbf{Set_p} \to \mathbf{Set}$ be the lifting functor, let $\mathcal{M}$ be the monoid $\langle \omega + 1, 0, + \rangle$ and for each $A \in |\mathbf{Set_p}|$, let $\phi^{\mathbf{Set_P}}(A)$ be the set of functions from $A_\perp$ to $\mathcal{M}$ mapping $\perp$ to ω. The idea is that the top element ω of $\omega + 1$ represents infinite time. The restriction to functions mapping $\perp$ to ω is the requirement that non terminating programs are given infinite time. The tuple $\langle \mathbf{Set_p}, \mathbf{Set}, (-)_\perp, \mathcal{M}, \phi^{\mathbf{Set_P}} \rangle$ is then an external datum whose complexity category can model the complexity of partial computations. Note that $\phi^{\mathbf{Set_P}}$ is a proper subobject of $\mathbf{Set}(-, \omega + 1)$.

The examples from chapter 2 are all examples here.

Example 5.3.15 Recall that in example 2.1.10, we gave a semantics for a simple language $\mathcal{L}$ in which commands denoted functions from states to states. Let $\mathcal{S}$ be the full subcategory of **Set** with the single object S of states, as defined in example 2.1.10. Let ι be the inclusion of $\mathcal{S}$ in **Set** and let $\mathcal{M}$ be the monoid $\langle \mathbf{N}, 0, + \rangle$ in **Set**. Then the tuple $\langle \mathcal{S}, \mathbf{Set}, \iota, \mathcal{M}, \mathbf{Set}(-, \mathbf{N}) \rangle$ is an external datum and the external complexity category for $\langle \mathcal{S}, \mathbf{Set}, \iota, \mathcal{M}, \mathbf{Set}(-, \mathbf{N}) \rangle$ is a model for our extended semantics of $\mathcal{L}$.

Example 5.3.16 Recall that in example 2.1.11, we gave a semantics for a simple assembler language in which commands denoted functions from $S \times E$ to $S \times E$. Let $\mathcal{S}$ be the full subcategory of **Set** with the two objects S, of states, and E, of environments, as defined in example 2.1.11. Let ι be the inclusion of $\mathcal{S}$ in **Set** and let $\mathcal{M}$ be the monoid $\langle \mathbf{N}, 0, + \rangle$ in **Set**. Then the tuple $\langle \mathcal{S}, \mathbf{Set}, \iota, \mathcal{M}, \mathbf{Set}(-, \mathbf{N}) \rangle$ is an

external datum and the external complexity category for $\langle \mathcal{S}, \mathbf{Set}, \iota, \mathcal{M}, \mathbf{Set}(-, \mathbf{N}) \rangle$ is a model for our extended semantics of the assembler language.

Example 5.3.17 Recall that in example 2.1.12, we gave an extended semantics for a fragment of the programming language Pascal in which commands denoted pairs of partial functions from $\mathcal{I} \times E$ to $\mathcal{O} \times E$ and from $\mathcal{I} \times E$ to $\mathbf{N}$. Let $\mathcal{S}$ be the full subcategory of $\mathbf{Set}_\mathbf{p}$ with the two objects $\mathbf{N}$ and E of environments, as defined in example 2.1.12. Let $(-)_\perp$, $\mathcal{M}$ and $\phi^{\mathcal{S}}$ be given as in example 5.3.14. Then the tuple $\langle \mathcal{S}, \mathbf{Set}, (-)_\perp, \mathcal{M}, \phi^{\mathcal{S}} \rangle$ is an external datum and the external complexity category for $\langle \mathcal{S}, \mathbf{Set}, (-)_\perp, \mathcal{M}, \phi^{\mathcal{S}} \rangle$ is a model for our extended semantics of Pascal.

Suppose that we wished to compare the complexities of the two programs, in examples 2.1.10 and 2.1.12, for computing the factorial function. In this case, we could do so internally, provided we could model both $\mathcal{L}$ and Pascal in the same category with a suitable monoid. However, we cannot always do this.

Example 5.3.18 Recall the following program to compute the factorial function from example 2.1.12:

```
begin
        m:=1;
        read n;
        while n>0 do
                        begin
                                m:=m*n;
                                n:=n-1;
                        end;
        write m;
end.
```

The following recursive program in the language ML also computes the factorial function.

```
- fun fact n:int = if n=1 then 1 else n*fact(n-1);
```

In order to compare the complexity of these two programs internally, we would need a single category that could model the extended semantics of both ML and Pascal.

In order to compare the complexity of these two programs externally, we simply require a model $\mathcal{C}$ for the functional behaviour of ML, and a functor U from $\mathcal{C}$ to **Set** such that $U([\![\text{int}]\!])$ equals $\mathbf{N}$.

Remark 5.3.19 *An important question arises at this point. This is whether the interpretation of the language for computations in internal complexity categories, which we described in chapter 3, can be extended to the external complexity categories. The short answer to this question is no, as there is a problem with the computational types. An explanation of the difficulties, together with a discussion as to how to overcome them, is contained in section 9.3.*

5.4 Relationship between Int and Ext

5.4.1 Introduction

In this section, we study the relationship between the internal and external approaches. In particular we show that **Int** may be regarded as an internalisation of **Ext**.

5.4.2 The main theorem

Proposition 5.4.1 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ and $\langle \mathcal{C}', \mathcal{M}' \rangle$ be internal data and let $F : \mathcal{C} \longrightarrow \mathcal{C}'$ be a morphism of internal data. Then the map *out* given by:

- $out(\langle \mathcal{C}, \mathcal{M} \rangle) = \langle \mathcal{C}, \mathcal{C}, id, \mathcal{M}, \mathcal{C}(-, M) \rangle$ and
- $out(F) = \langle F, F \rangle$

defines a functor out : **Int**$\longrightarrow$**Ext**.

Proof: First observe that $out(F) = \langle F, F\rangle$ is a morphism of external data since:

- Clearly, $Fid = idF$.
- F preserves finite products by assumption.
- F preserves the monoid $\mathcal{M}$ by assumption.
- Suppose $t \in \mathcal{C}(-, M)$, then $Ft \in \mathcal{C}'(F-, FM) = \mathcal{C}'(F-, M')$ thus $F(\mathcal{C}(-, M)) \leq \mathcal{C}'(F-, M')$ as required.

Trivially, *out* preserves identities.
Finally:

- $$\begin{aligned} out(F_1F_0) &= \langle F_1F_0, F_1F_0\rangle \\ &= \langle F_1, F_1\rangle\langle F_0, F_0\rangle \\ &= out(F_1)out(F_0) \end{aligned}$$

whence *out* preserves composition and this completes the proof. □

Proposition 5.4.2 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'}\rangle$ be external data and let $\langle F, G\rangle$ be a morphism of external data. Then the map *in* given by:

- $in(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle) = \langle \mathcal{D}, \mathcal{M}\rangle$ and
- $in(\langle F, G\rangle) = G$

defines a functor in : **Ext**$\longrightarrow$**Int**.

Proof: *in* is well-defined since G preserves finite products and the monoid $\mathcal{M}$ by assumption. It is clear that *in* preserves identities and composition. □

Lemma 5.4.3 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$ be an external datum and let $\langle \mathcal{E}, \mathcal{N}\rangle$ be an internal datum. The function from $\mathbf{Ext}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle, out(\langle \mathcal{E}, \mathcal{N}\rangle))$ to $\{G : \mathcal{D} \to \mathcal{E} |\ G$ preserves finite products and the monoid $\mathcal{M}\}$ mapping $\langle F, G\rangle$ to G is a bijection of sets, natural in $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$ and $\langle \mathcal{E}, \mathcal{N}\rangle$.

Proof: $out(\langle \mathcal{E}, \mathcal{N} \rangle) = \langle \mathcal{E}, \mathcal{E}, id, \mathcal{N}, \mathcal{E}(-, N) \rangle$ so

$\mathbf{Ext}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, out(\langle \mathcal{E}, \mathcal{N} \rangle)) = \mathbf{Ext}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, \langle \mathcal{E}, \mathcal{E}, id, \mathcal{N}, \mathcal{E}(-, N) \rangle)$

an element of which is a pair of functors $\langle F : \mathcal{C} \to \mathcal{E}, G : \mathcal{D} \to \mathcal{E} \rangle$ such that:

- $GU = idF$ *i.e.* $F = GU$,
- G preserves finite products and the monoid $\mathcal{M}$ and
- $G\phi^{\mathcal{C}} \leq \mathcal{E}(F-, N)$.

However, $\phi^{\mathcal{C}} \leq \mathcal{D}(U-, M)$ by definition and for any G, which preserves finite products and maps $\mathcal{M}$ to $\mathcal{N}$, we have:

$$G\phi^{\mathcal{C}} \leq \mathcal{E}(GU-, N)$$

and therefore, putting $F = GU$, we have $G\phi^{\mathcal{C}} \leq \mathcal{E}(F-, N)$.

Hence to give $\langle F, G \rangle$ is to give G.

Naturality is evident and this completes the proof. □

Theorem 5.4.4 We have an adjunction $in \dashv out : \mathbf{Ext} \to \mathbf{Int}$

Proof: Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle \in |\mathbf{Ext}|$ and $\langle \mathcal{E}, \mathcal{N} \rangle \in |\mathbf{Int}|$. We need to show that there is a natural bijection:

$$\mathbf{Int}(in(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle), \langle \mathcal{E}, \mathcal{N} \rangle) \cong \mathbf{Ext}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, out(\langle \mathcal{E}, \mathcal{N} \rangle)).$$

Now $in(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle) = \langle \mathcal{D}, \mathcal{M} \rangle$ so

$$\mathbf{Int}(in(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle), \langle \mathcal{E}, \mathcal{N} \rangle) = \mathbf{Int}(\langle \mathcal{D}, \mathcal{M} \rangle, \langle \mathcal{E}, \mathcal{N} \rangle)$$

an element of which is a functor $G : \mathcal{D} \to \mathcal{E}$ such that G preserves finite products and the monoid $\mathcal{M}$.

Hence by lemma 5.4.3 we are done. □

Corollary 5.4.5 **Int** is a full reflective subcategory of **Ext**.

Proof: Trivially *out* is injective on objects.
It follows immediately from lemma 5.4.3 that *out* is fully faithful.
Finally *out* has a left adjoint by theorem 5.4.4 and the result follows. □

Remark 5.4.6 *Let* $\mathbf{Ext}^\dagger$ *be the full subcategory of* $\mathbf{Ext}$ *consisting of those objects for which* U *is an isomorphism. Let* ι *be the inclusion of* $\mathbf{Ext}^\dagger$ *into* $\mathbf{Ext}$. *Then there are functors* $out^\dagger : \mathbf{Int} \to \mathbf{Ext}^\dagger$, $in^\dagger : \mathbf{Ext}^\dagger \to \mathbf{Int}$ *and* $squash : \mathbf{Ext} \to \mathbf{Ext}^\dagger$ *such that:*

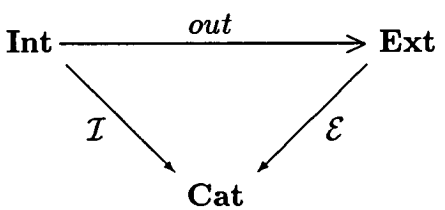
- $out = \iota\, out^\dagger$ *and* $in = in^\dagger\, squash$,
- $in^\dagger \dashv out^\dagger : \mathbf{Int} \to \mathbf{Ext}^\dagger$ *and*
- $squash \dashv \iota : \mathbf{Ext}^\dagger \to \mathbf{Ext}$.

The construction of the functor squash is rather delicate.

5.4.3 Internal and external complexity categories

We have related the categories **Int** and **Ext**. The following result shows that the adjunction $in \dashv out : \mathbf{Int} \to \mathbf{Ext}$ is "compatible" with the functors $\mathcal{I} : \mathbf{Int} \longrightarrow \mathbf{Cat}$ and $\mathcal{E} : \mathbf{Ext} \longrightarrow \mathbf{Cat}$.

Theorem 5.4.7 The diagram:



commutes up to natural isomorphism.

Proof: We require a natural isomorphism from $\mathcal{I}$ to $\mathcal{E}\ out$.
Let $\langle \mathcal{C}, \mathcal{M} \rangle \in |\mathbf{Int}|$. Then $\mathcal{I}(\langle \mathcal{C}, \mathcal{M} \rangle) = \mathcal{C}_\mathcal{M}$ is given by:

- $|\mathcal{C}_M| = |\mathcal{C}|$,
- $\mathcal{C}_M(A,B) = \mathcal{C}(A, B \times M)$ and
- $\langle g,s\rangle\langle f,t\rangle = \langle gf, t \cdot fs\rangle$.

$out(\langle \mathcal{C}, \mathcal{M}\rangle)$ is given by $\langle \mathcal{C}, \mathcal{C}, id, \mathcal{M}, \mathcal{C}(-,M)\rangle$ and therefore $\mathcal{E}\ out(\langle \mathcal{C}, \mathcal{M}\rangle) = \mathcal{C}^M$ is given by:

- $|\mathcal{C}^M| = |\mathcal{C}|$,
- $\mathcal{C}^M(A,B) = \{\langle f,t\rangle : f \in \mathcal{C}(A,B),\ t \in \mathcal{C}(A,M)\}$ and
- $\langle g,s\rangle\langle f,t\rangle = \langle gf, t \cdot sf\rangle$.

We define the functor $\iota : \mathcal{C}_M \longrightarrow \mathcal{C}^M$ in the evident way by:

$$\iota(\langle f,t\rangle : A \longrightarrow B) = \langle f,t\rangle : A \longrightarrow B.$$

It is clear that ι is an isomorphism from $\mathcal{C}_M$ to $\mathcal{C}^M$.
It remains simply to check naturality, *i.e.* given $F : \langle \mathcal{C}, \mathcal{M}\rangle \longrightarrow \langle \mathcal{C}', \mathcal{M}'\rangle \in \mathbf{Int}$ we require:

$$\begin{array}{ccc} \mathcal{C}_M & \xrightarrow{\iota} & \mathcal{C}^M \\ {\scriptstyle F^*}\downarrow & & \downarrow{\scriptstyle F \cdot F} \\ \mathcal{C}'_{M'} & \xrightarrow[\iota]{} & \mathcal{C}'^{M'} \end{array}$$

and this is clear. □

Theorem 5.4.8 Let $\alpha : \mathcal{E} \longrightarrow \mathcal{I}\ in$ be the natural transformation defined by:

$$\alpha = \mathcal{E}\eta : \mathcal{E} \longrightarrow \mathcal{E}\ out\ in \cong \mathcal{I}\ in$$

where η is the unit of the adjunction $in \dashv out$. Then each component of α is full (faithful) if and only if U is full (faithful).

Proof: The components of the natural transformation α are given explicitly by the following:
Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle \in |\mathbf{Ext}|$. Then $\mathcal{E}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle) = \mathcal{C}^{M}$ is given by:

- $|\mathcal{C}^{M}| = |\mathcal{C}|$,
- $\mathcal{C}^{M}(A, B) = \{\langle f, t \rangle : f \in \mathcal{C}(A, B),\ t \in \phi^{\mathcal{C}}\}$ and
- $\langle g, s \rangle \langle f, t \rangle = \langle gf, t \cdot sUf \rangle$.

$in(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle)$ is given by $\langle \mathcal{D}, \mathcal{M} \rangle$ and therefore $\mathcal{I}\, in(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle) = \mathcal{C}_{M}$ is given by:

- $|\mathcal{C}_{M}| = |\mathcal{D}|$,
- $\mathcal{C}_{M}(A, B) = \mathcal{D}(A, B \times M)$ and
- $\langle g, s \rangle \langle f, t \rangle = \langle gf, t \cdot sf \rangle$.

We define the functor $\alpha : \mathcal{C}^{M} \longrightarrow \mathcal{C}_{M}$ in the evident way by:

$$\alpha(\langle f, t \rangle : A \longrightarrow B) = \langle Uf, t \rangle : UA \longrightarrow UB.$$

It is clear that this is full (faithful) precisely when U is full (faithful). □

Chapter 6

Order and Disorder

6.1 Introduction

In chapter 5, we extended existing semantic frameworks in order to reason about the exact complexity of programs. However, in the analysis of algorithms, we are also often interested in non-exact complexity. For example we say that a program has complexity $p(n)$ if its run-time is bounded above by $p(n)$. In the non-exact analysis of program complexity, input measures are often used to express program complexity. An input measure on a type A is a map from the denotation of A to an object $s(A)$ of possible sizes of elements of A. In chapter 7 we study input measures and in chapter 8 we consider the non-exact analysis of program complexity using input measures.

In the study of non-exact complexity, we are led to consider order enriched structures. In particular, we must replace our monoid $\mathcal{M}$ by an ordered monoid $\mathcal{M}$ of resource values. Accordingly, in section 6.2, we review the basic definitions of enriched category theory, which covers order enriched structures and gives some technical advantages. In particular, we introduce order enriched categories and lax functors and we define the notion of an ordered monoid.

In section 6.3, we consider an ordered monoid of resource values, leading to the definition of an ordered datum. We define internal and external notions of ordered data and study the relationship between them along similar lines to chapter 5.

6.2 Ordered and lax structures

6.2.1 Introduction

In this work, we use 2-categories and locally ordered categories. We could define these concepts individually. However, it is convenient to introduce the basic definitions of enriched category theory [Kel82]. Both locally ordered categories and 2-categories can then be defined as particular instances of enriched categories.

We also discuss relations in an arbitrary category, in order to define the notion of an ordered monoid.

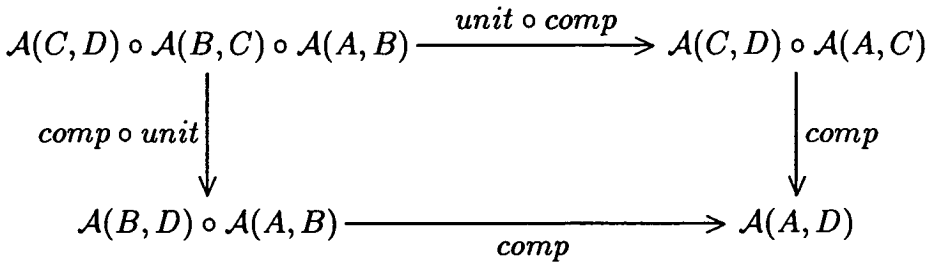
6.2.2 Enriched categories

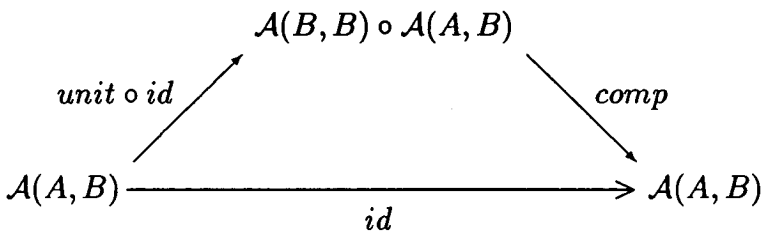
Definition 6.2.1 Let $\mathcal{V} = \langle \mathcal{V}_o, \circ, I \rangle$ be a monoidal category. A **$\mathcal{V}$-category** $\mathcal{A}$ consists of:

- a set $|\mathcal{A}|$,
- for each pair A, B of elements of $|\mathcal{A}|$, an object $\mathcal{A}(A,B)$ of $\mathcal{V}$ and
- for each triple A, B, C of elements of $|\mathcal{A}|$ and each element A of $|\mathcal{A}|$, morphisms in $\mathcal{V}$:

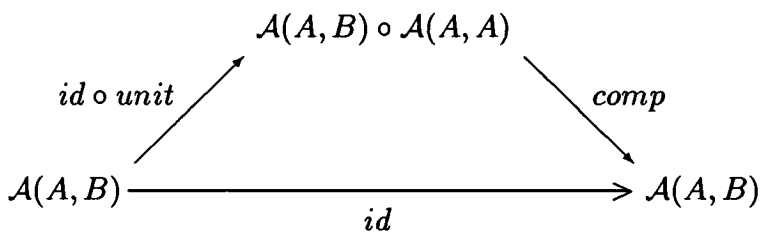
$$comp : \mathcal{A}(B,C) \circ \mathcal{A}(A,B) \longrightarrow \mathcal{A}(A,C) \quad \text{and} \quad unit : I \longrightarrow \mathcal{A}(A,A)$$

such that the following diagrams:





and



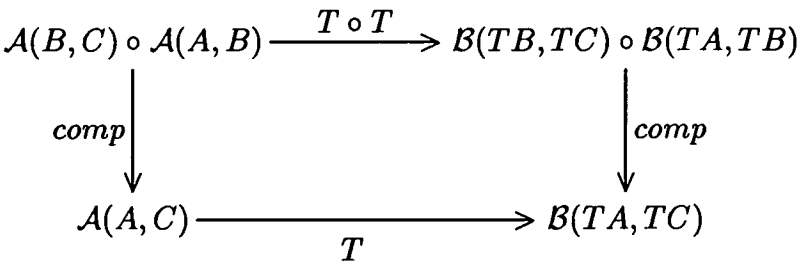
commute, suppressing the structural isomorphisms of $\mathcal{V}$.

Notation 6.2.2 We call $|\mathcal{A}|$ the **object set** of $\mathcal{A}$ and $\mathcal{A}(A,B)$ the **hom-object** of A and B.

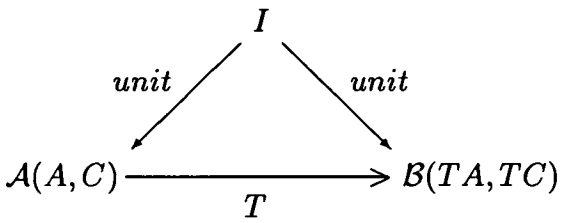
Definition 6.2.3 Let $\mathcal{V}$ be a monoidal category and let $\mathcal{A}$ and $\mathcal{B}$ be $\mathcal{V}$-categories. A **$\mathcal{V}$-functor** $T : \mathcal{A} \to \mathcal{B}$ consists of:

- a function $T : |\mathcal{A}| \to |\mathcal{B}|$ and
- for each A, B an arrow $T : \mathcal{A}(A,B) \longrightarrow \mathcal{B}(TA,TB)$ in $\mathcal{V}$

such that the following diagrams:

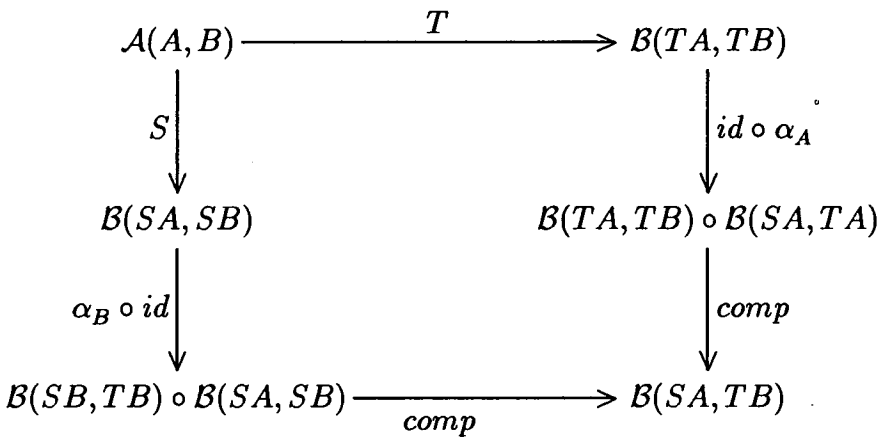


and



commute.

Definition 6.2.4 Let $\mathcal{V}$ be a monoidal category. Let $\mathcal{A}$ and $\mathcal{B}$ be $\mathcal{V}$-categories. Let S and T be $\mathcal{V}$-functors from $\mathcal{A}$ to $\mathcal{B}$. A **$\mathcal{V}$-natural transformation**, α from S to T consists of, for each A in $|\mathcal{A}|$, an arrow $\alpha_A : I \longrightarrow \mathcal{B}(SA,TA)$ such that:



commutes.

Notation 6.2.5 We write $\alpha : S \to T : A \to B$.

Example 6.2.6 Some examples of enriched categories are:

$\mathcal{V}$	$\mathcal{V}$-categories	$\mathcal{V}$-functors	$\mathcal{V}$-nts
$\langle \mathbf{Set}, \times, 1 \rangle$	Categories	Functors	nts
$\langle \mathbf{Cat}, \times, 1 \rangle$	2-Categories	2-Functors	2-nts
$\langle$(n-1)-$\mathbf{Cat}, \times, 1\rangle$	n-Categories	n-Functors	n-nts
$\langle \mathbf{Poset}, \times, 1 \rangle$	Loc. Ord. Categories	ord. Functors	ord. nts
$\langle \bullet \to \bullet, inf, \bullet \rangle$	Posets	ord. pres. maps.	"$f \leq g$"
$\langle \mathbf{Ab}, \otimes, \mathbb{Z} \rangle$	Additive Categories	Add. Functors	Add. nts

The above examples are all rather mathematical in flavour. However, the following example is of direct interest to computer science.

Example 6.2.7 Let $\mathcal{V}$ be the category of omega-complete partial orders. Then a $\mathcal{V}$-category is exactly an O-category in the sense of Smyth and Plotkin [PlSm82].

Proposition 6.2.8 Let $\mathcal{V}$ be a monoidal category. Then $\mathcal{V}$-categories, $\mathcal{V}$-functors and $\mathcal{V}$-natural transformations with the evident composition define a 2-category, $\mathcal{V}$-**Cat**.

Remark 6.2.9 *There is an evident question of size here. We evade the issue by, where necessary, assuming the existence of at least two strongly inaccessible cardinals.*

Corollary 6.2.10 Locally ordered categories and ordered functors with the evident composition form a 2-category, **OrdCat**.

Proposition 6.2.11 Let $\mathcal{V} = \langle \mathcal{V}_o, \circ, I \rangle$ be a monoidal category and let $\mathcal{A}$ be a $\mathcal{V}$-category. Then the elements of $|\mathcal{A}|$ together with, for each $A, B \in |\mathcal{A}|$ the elements of $\mathcal{V}_o(I, \mathcal{A}(A, B))$ define a category $\mathcal{A}_o$.

Notation 6.2.12 $\mathcal{A}_o$ is called the underlying category of $\mathcal{A}$.

In the following we shall be most concerned with locally ordered, that is poset enriched categories. The following result characterises locally ordered categories.

Lemma 6.2.13 A locally ordered category, $\mathcal{C}$ consists of:

- A set $|\mathcal{C}|$ of objects,
- $\forall a, b \in |\mathcal{C}|$ a poset $\mathcal{C}(a, b)$,
- $\forall a \in |\mathcal{C}|$ an element id_a of $\mathcal{C}(a, a)$ and
- $\forall a, b, c \in |\mathcal{C}|$ a monotone function $comp : \mathcal{C}(b, c) \times \mathcal{C}(a, b) \to \mathcal{C}(a, c)$

such that:

- $comp(comp(f, g), h) = comp(f, comp(g, h))$ and
- $comp(id, f) = comp(f, id) = f$.

Notation 6.2.14 We write fg for $comp(f, g)$.

Definition 6.2.15 Let $\mathcal{C}$ and $\mathcal{D}$ be locally ordered categories. A **lax functor**, F from $\mathcal{C}$ to $\mathcal{D}$ consists of a function $F : |\mathcal{C}| \to |\mathcal{D}|$ and a collection of order preserving functions $F : \mathcal{C}(A, B) \to \mathcal{D}(FA, FB)$ such that:

- $F(f)F(g) \leq F(fg)$ and
- $id \leq F(id)$.

Definition 6.2.16 Let $\mathcal{C}$ and $\mathcal{D}$ be locally ordered categories. A lax functor $F : \mathcal{C} \to \mathcal{D}$ is **strict** if:

- $F(fg) = F(f)F(g)$ and
- $F(id) = id$.

Remark 6.2.17 *Let $\mathcal{C}$ be a category. Then $\mathcal{C}$ may be viewed as a locally ordered category by taking the trivial order on $\mathcal{C}(A, B)$ for each pair A, B of objects of $\mathcal{C}$.*

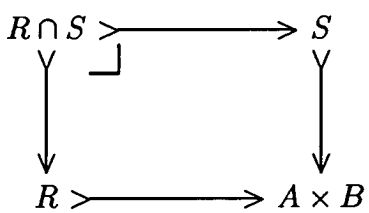
6.2.3 Ordered monoids

Definition 6.2.18 Let $\mathcal{C}$ be a category with finite products and let A and B be objects of $\mathcal{C}$. A **relation** in $\mathcal{C}$ from A to B is a subobject R of $A \times B$.

Notation 6.2.19 We write $R \in Rel_{\mathcal{C}}(A, B)$ or $Rel(A, B)$ where $\mathcal{C}$ is clear.

Lemma 6.2.20 Let $\mathcal{C}$ be a category and let $R \in Rel_{\mathcal{C}}(A, B)$. Then the map $R \rightarrowtail A \times B \xrightarrow{\cong} B \times A$ defines a relation $R^o \in Rel_{\mathcal{C}}(B, A)$.

Lemma 6.2.21 Let $\mathcal{C}$ be a category with finite limits. Let $R \in Rel(A, B)$ and $S \in Rel(A, B)$ be relations in $\mathcal{C}$. Then the following pullback:

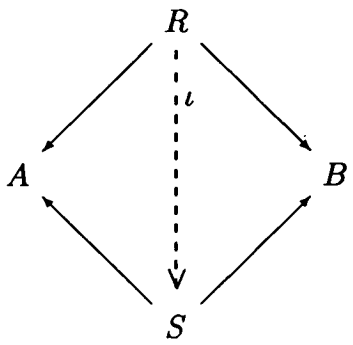


defines a relation $R \cap S \in Rel(A, B)$.

Remark 6.2.22 *Let $R \in Rel(A, B)$ and $S \in Rel(A, B)$ be relations in $\mathcal{C}$. Then if $\mathcal{C}$ has enough structure, we can construct a composite relation $RS \in Rel(A, C)$. In order for this to give a well-behaved category of relations, $\mathcal{C}$ should be a regular category. However, for our purposes it suffices that $\mathcal{C}$ has finite limits. For more details, refer to [FrSc90].*

Definition 6.2.23 Let $\mathcal{C}$ be a category, let S be a relation from A to B in $\mathcal{C}$ and let $A \leftarrow R \rightarrow B$ be a pair of morphisms in $\mathcal{C}$. We say $R \subseteq S$ if there exists a

morphism $\iota : R \to S$ in $\mathcal{C}$ such that the following diagram:



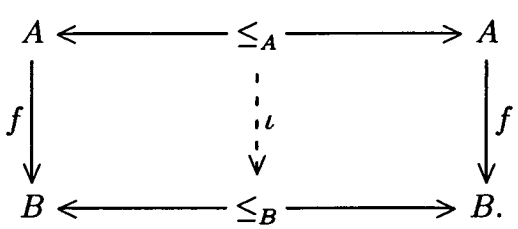
commutes.

Remark 6.2.24 *If R is a relation then any such map is necessarily monomorphic.*

Definition 6.2.25 Let $\mathcal{C}$ be a category with finite limits, let $R \in Rel(A, A)$ and let A be the diagonal relation $\Delta : A \to A \times A$. We say R is:

- a **reflexive** relation if $\Delta \subseteq R$,
- a **symmetric** relation if $R^o \subseteq R$,
- a **transitive** relation if $RR \subseteq R$,
- an **antisymmetric** relation if $R \cap R^o \subseteq \Delta$,
- a **partial order** if R is a reflexive, transitive and antisymmetric relation and
- an **equivalence relation** if R is a reflexive, symmetric and transitive relation.

Definition 6.2.26 Let $\mathcal{C}$ be a category with finite limits, let $\leq_A \in Rel(A, A)$ and $\leq_B \in Rel(B, B)$ be partial orders. A map $f : A \to B$ is **monotone** if there exists a map $\iota : \leq_A \to \leq_B$ such that the following diagram:



commutes.

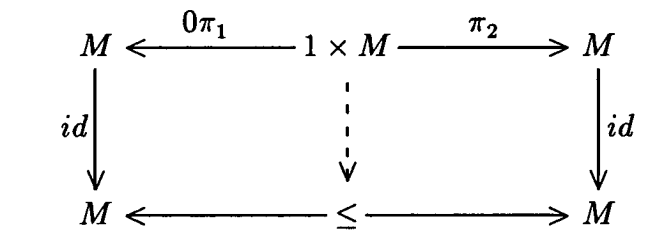
Example 6.2.27 In **Set** all these definitions correspond to the usual ones.

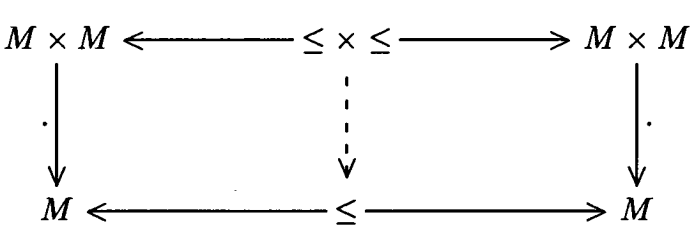
Definition 6.2.28 An ordered monoid in **Set** is a tuple $\langle M, 0, \cdot, \leq \rangle$ consisting of a monoid $\langle M, 0, \cdot \rangle$ and a partial order $\leq$ on M such that:

- $0 \leq m$ for all m in M,
- $m_0 \leq m_1$ implies $m \cdot m_0 \leq m \cdot m_1$ for all m in M and
- $m_0 \leq m_1$ implies $m_0 \cdot m \leq m_1 \cdot m$ for all m in M.

We generalise this definition to an arbitrary category with finite limits.

Definition 6.2.29 Let $\mathcal{C}$ be a category with finite limits. An **ordered monoid**, $\mathcal{M}$ is a tuple $\langle M, 0, \cdot, \leq \rangle$ consisting of a monoid $\langle M, 0, \cdot \rangle$ and a partial order $\leq$ on M such that the following diagrams:





commute.

Notation 6.2.30 Let $\mathcal{C}$ be a category with finite limits and let $\mathcal{M} = \langle M, 0, \cdot, \leq \rangle$ be an ordered monoid in $\mathcal{C}$. We write $\mathcal{M}_o$ for the underlying monoid $\langle M, 0, \cdot \rangle$.

Example 6.2.31 The monoids $\langle \mathbf{N}, 0, + \rangle$, $\langle \mathbf{N}, 0, sup \rangle$ and $\langle \mathbf{N}, 1, \times \rangle$ with the usual ordering are ordered monoids.

6.3 Ordered data

6.3.1 Introduction

We define internal and external notions of ordered datum and show how to construct ordered complexity categories from ordered data. We construct categories **OInt** and **OExt** of internal and external ordered data and investigate the relationship between them. This investigation follows the lines of chapter 5.

6.3.2 Internal ordered data

As in chapter 5, there is an internal version of ordered datum.

Definition 6.3.1 An **internal ordered datum** is a pair $\langle \mathcal{C}, \mathcal{M} \rangle$ consisting of a category $\mathcal{C}$ with finite limits and an ordered monoid $\mathcal{M} = \langle M, 0, \cdot, \leq \rangle$ in $\mathcal{C}$.

Definition 6.3.2 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ and $\langle \mathcal{C}', \mathcal{M}' \rangle$ be internal ordered data. A **morphism of internal ordered data** is a functor $F : \mathcal{C} \longrightarrow \mathcal{C}'$ such that:

- F is a morphism of internal data from $\langle \mathcal{C}, \mathcal{M}_o \rangle$ to $\langle \mathcal{C}', \mathcal{M}'_o \rangle$ and
- $F(\leq) \subseteq \leq'$

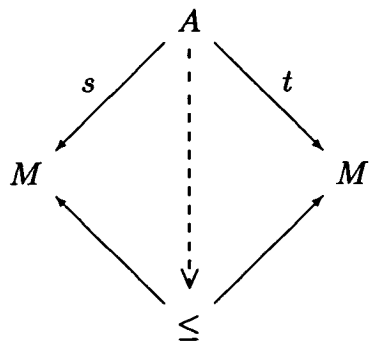
where $\mathcal{M}_o$ is the underlying monoid of $\mathcal{M}$ (notation 6.2.30).

Remark 6.3.3 *We could require that* $F(\leq) = \leq'$. *However, this gives a very restrictive notion of morphism.*

Proposition 6.3.4 Internal ordered data and morphisms of internal ordered data together with functor composition define a category **OInt**.

Definition 6.3.5 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be an internal ordered datum. The internal ordered complexity category $\mathcal{C}_M$ is the Kleisli category for the monad $M \times -$ on $\mathcal{C}$.

Proposition 6.3.6 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be an internal ordered datum, and let $\leq$ be the relation on $\mathcal{C}(A, M)$ given by $s \leq t$ if there exists a map from A to $\leq$ such that the following diagram:



commutes. Then $\mathcal{C}_M$ has an ordered structure given by:

$$\langle f, t \rangle \leq \langle g, s \rangle \quad \text{if} \quad f = g \text{ and } t \leq s.$$

Proof: It suffices to verify the functoriality of composition. This follows as a special case of the argument in the proof of proposition 6.3.15. □

We can extend definition 6.3.5 to a functor from **OInt** to **OrdCat** as follows:

Lemma 6.3.7 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ and $\langle \mathcal{C}', \mathcal{M}' \rangle$ be internal ordered data and let F be a morphism of internal ordered data from $\langle \mathcal{C}, \mathcal{M} \rangle$ to $\langle \mathcal{C}', \mathcal{M}' \rangle$. Then the assignment:

$$\langle f, t \rangle : A \longrightarrow B \longmapsto \langle Ff, Ft \rangle : FA \longrightarrow FB$$

defines a 2-functor $F^* : \mathcal{C}_M \longrightarrow \mathcal{C}'_{M'}$.

Proof: F^* is well-defined since by assumption F preserves finite products and the monoid structure. It is clear from the **Int** case that F^* preserves composition. Finally, suppose that $\langle f, t \rangle \leq \langle g, s \rangle$. Then $f = g$ and $t \leq s$ whence $Ff = Fg$ and $Ft \leq' Fs$ since $F(\leq) \subseteq \leq'$. Thus:

$$F^*(\langle f, t \rangle) \leq F^*(\langle g, s \rangle).$$

This establishes that F^* is a 2-functor and completes the proof. □

Proposition 6.3.8 The assignment:

- $\mathcal{C} \longmapsto \mathcal{C}_M$ and
- $F \longmapsto F^*$

defines a functor $\mathcal{I}$: **OInt**$\longrightarrow$**OrdCat.**

Proof: $\mathcal{I}$ is well-defined by lemma 6.3.7 and it is clear that $\mathcal{I}$ preserves identities and composition. □

6.3.3 External ordered data

Definition 6.3.9 An **external ordered datum** is a tuple $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ consisting of a category $\mathcal{C}$, a category with finite limits $\mathcal{D}$, a functor $U : \mathcal{C} \to \mathcal{D}$, an ordered monoid $\mathcal{M} = \langle M, 0, \cdot, \leq \rangle$ in $\mathcal{D}$ and a subobject $\phi^{\mathcal{C}}$ of the functor $\mathcal{D}(U(-), M)$ in Monoids($[\mathcal{C}^{\mathrm{op}}$,**Set**]).

Remark 6.3.10 *Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum. Then the tuple $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}_o, \phi^{\mathcal{C}} \rangle$ is an external datum.*

Lemma 6.3.11 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum. Then $\phi^{\mathcal{C}}(A)$ is an ordered monoid for each object A of $\mathcal{C}$.

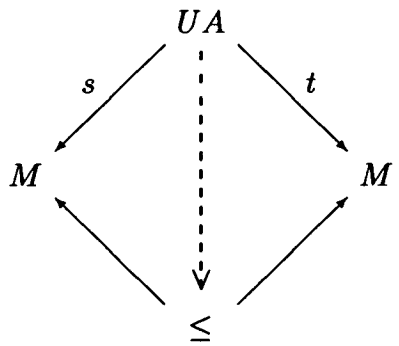
Proof: By proposition 5.3.5, $\phi^{\mathcal{C}}(A)$ is a monoid in **Set** with zero element the map:

$$UA \xrightarrow{*} 1 \xrightarrow{0} M$$

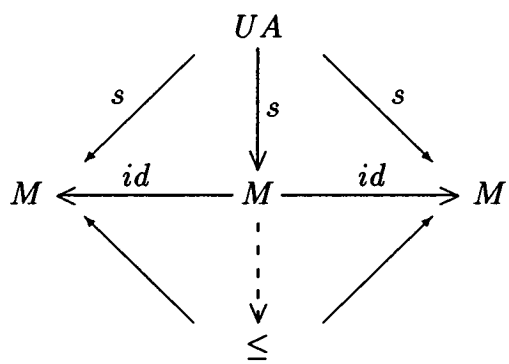
and monoid operation given by:

$$s \cdot t \;=\; UA \xrightarrow{\langle s,t \rangle} M \times M \xrightarrow{\cdot} M.$$

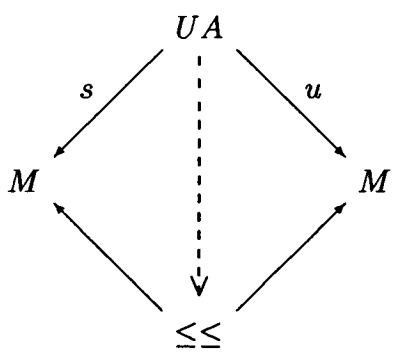
Let $\leq$ be the relation on $\phi^{\mathcal{C}}(A)$ given by: $s \leq t$ if there exists a map from UA to $\leq$ in $\mathcal{D}$ such that the following diagram:



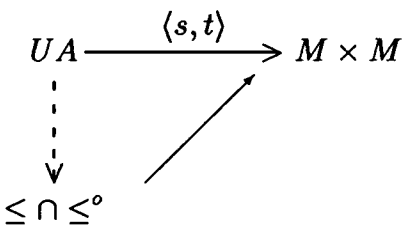
commutes. We claim that $\langle \phi^{\mathcal{C}}(A), 0, \cdot, \leq \rangle$ is an ordered monoid.
The following diagram:



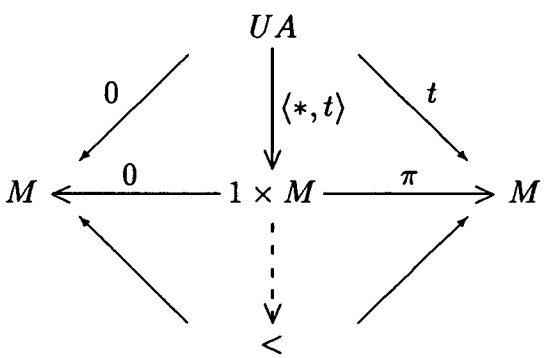
commutes because $\leq$ is a reflexive relation and this shows that $\leq$ is reflexive.
Suppose that $s \leq t$ and $t \leq u$. Then a simple diagram chase implies:



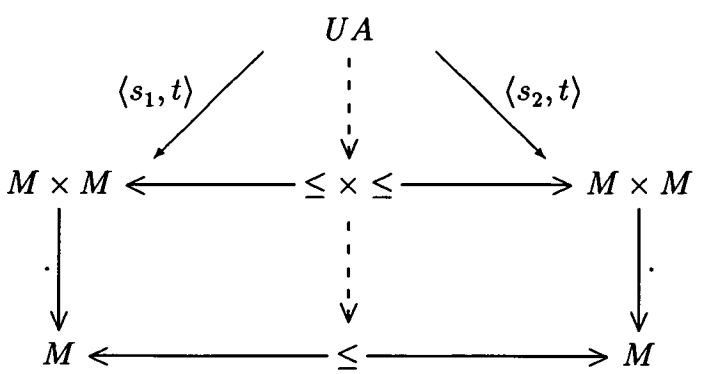
but $\leq$ is transitive so it follows that $s \leq u$ and hence that $\leq$ is transitive.
Suppose that $s \leq t$ and $t \leq s$. Then a simple diagram chase shows:



but $\leq$ is antisymmetric which implies that $\langle s,t\rangle$ factors through $M \xrightarrow{\Delta} M \times M$ and hence that $s = t$. Thus $\leq$ is antisymmetric and this establishes that $\leq$ is a partial order on $\phi^{\mathcal{C}}(A)$.
The first condition in definition 6.2.29 is verified by the following diagram:



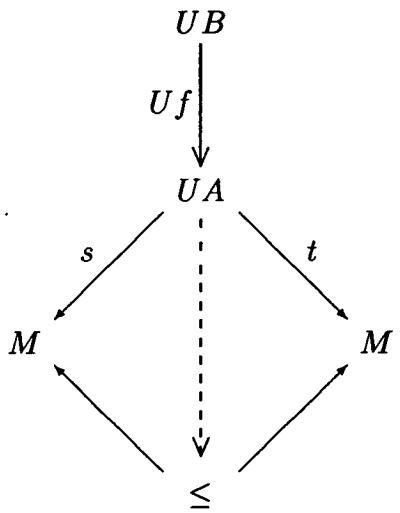
Suppose that $s_1 \leq s_2$. Then a moment's consideration will convince the reader that the following diagram:



verifies that $s_1 \cdot t \leq s_2 \cdot t$. Similarly, $s_1 \leq s_2$ implies that $t \cdot s_1 \leq t \cdot s_2$ and this completes the proof □

Lemma 6.3.12 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum and suppose that $s \leq t$ in $\phi^{\mathcal{C}}(A)$ and $f : B \longrightarrow A$ in $\mathcal{C}$. Then $sUf \leq tUf$.

Proof: Follows from the diagram:



□

Definition 6.3.13 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$ be external ordered data. A **morphism of external ordered data** is a pair of functors $\langle F, G \rangle$ such that:

- $\langle F, G \rangle$ is a morphism of external data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}_o, \phi^{\mathcal{C}} \rangle$ to $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}'_o, \phi^{\mathcal{C}'} \rangle$
- $G(\leq) \subseteq \leq'$.

Proposition 6.3.14 External ordered data, morphisms of external ordered data and composition defined by:

$$\langle F_2, G_2 \rangle \langle F_1, G_1 \rangle \;=\; \langle F_2 F_1, G_2 G_1 \rangle$$

determine a category **OExt**.

Proof: By proposition 5.3.8 it only remains to verify that G_2G_1 preserves finite products and that $G_2G_1(\leq) \subseteq \leq''$. Both of these are clear. □

Proposition 6.3.15 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum. Then the following data:

- objects: objects of $\mathcal{C}$,
- morphisms: pairs $\langle f, t \rangle$ where $f : A \longrightarrow B \in \mathcal{C}$ and $t : UA \longrightarrow M \in \phi^{\mathcal{C}}(A)$,
- composition: $\langle g, s \rangle \langle f, t \rangle = \langle gf, t \cdot sUf \rangle$ and
- an ordering $\langle f, t \rangle \leq \langle g, s \rangle$ if $f = g$ and $t \leq s$ in $\phi^{\mathcal{C}}(A)$

define a locally ordered category $\mathcal{C}^{\mathcal{M}}$.

Proof: For each $A \in |\mathcal{C}|$, $\leq$ defines a partial order on $\phi^{\mathcal{C}}(A)$ by lemma 6.3.11. It follows immediately that $\leq$ defines a partial order on $\mathcal{C}^{\mathcal{M}}(A, B)$. Suppose now that $\langle f, t \rangle \leq \langle f, t' \rangle : A \to B$ and $\langle g, s \rangle \leq \langle g, s' \rangle : B \to C$. Then $t \leq t'$ and $s \leq s'$ so $sUf \leq s'Uf$ by lemma 6.3.12 and $t \cdot sUf \leq t' \cdot s'Uf$ since $\phi^{\mathcal{C}}(A)$ is an ordered monoid by lemma 6.3.11. Thus:

$$\langle g, s \rangle \langle f, t \rangle = \langle gf, t \cdot sUf \rangle \leq \langle gf, t' \cdot s'Uf \rangle = \langle g, s' \rangle \langle f, t' \rangle$$

so composition is functorial. The associativity and identity axioms follow by lemma 5.3.9 and this completes the proof. □

Notation 6.3.16 We call $\mathcal{C}^{\mathcal{M}}$ an external ordered complexity category.

Lemma 6.3.17 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$ be external ordered data and let $\langle F : \mathcal{C} \longrightarrow \mathcal{C}', G : \mathcal{D} \longrightarrow \mathcal{D}' \rangle$ be a morphism of external ordered data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ to $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$. Then the assignment:

$$\langle f, t \rangle : A \longrightarrow B \longmapsto \langle Ff, Gt \rangle : FA \longrightarrow FB$$

defines a 2-functor $F \cdot G : \mathcal{C}^{\mathcal{M}} \longrightarrow \mathcal{C}^{\mathcal{M}'}$.

Proof: $F \cdot G$ is well-defined by lemma 5.3.11.

$F \cdot G$ preserves composition since:

- $$\begin{aligned} \langle Fg, Gs\rangle\langle Ff, Gt\rangle &= \langle FgFf, Gt \cdot GsU'Ff\rangle \text{ (defn)} \\ &= \langle F(gf), Gt \cdot GsGUf\rangle \ (U'F = GU) \\ &= \langle F(gf), G(t \cdot sUf)\rangle \ (G \text{ preserves } \mathcal{M}) \\ &= F \cdot G(\langle gf, t \cdot sUf\rangle) \text{ (defn)} \\ &= F \cdot G(\langle g, s\rangle\langle f, t\rangle) \text{ (defn).} \end{aligned}$$

Finally, suppose that $\langle f, t\rangle \leq \langle g, s\rangle$. Then $f = g$ and $t \leq s$ whence $Ff = Fg$ and $Gt \leq' Gs$ since $G(\leq) \subseteq \leq'$. Thus $F \cdot G(\langle f, t\rangle) \leq F \cdot G(\langle g, s\rangle)$.

This establishes that $F \cdot G$ is a 2-functor and completes the proof. □

Proposition 6.3.18 The assignment:

- $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle \longmapsto \mathcal{C}^{\mathcal{M}}$ and
- $\langle F, G\rangle \longmapsto F \cdot G$

defines a functor $\mathcal{E}$: **OExt**$\longrightarrow$**OrdCat**.

Proof: $F \cdot G$ is well defined by lemma 6.3.17. It is clear that $\mathcal{E}$ preserves identities. It remains to verify that $\mathcal{E}$ respects composition.

- $$\begin{aligned} \mathcal{E}(\langle F_1F_0, G_1G_0\rangle) &= (F_1F_0) \cdot (G_1G_0) \\ &= (F_1 \cdot G_1)(F_0 \cdot G_0) \\ &= \mathcal{E}(\langle F_1, G_1\rangle)\mathcal{E}(\langle F_0, G_0\rangle). \end{aligned}$$

This completes the proof. □

6.3.4 Relationship between the internal and external

As with **Int** and **Ext** in chapter 5, we show that **OInt** is a full reflective subcategory of **OExt**.

Proposition 6.3.19 Let $\langle \mathcal{C}, \mathcal{M} \rangle$ and $\langle \mathcal{C}', \mathcal{M}' \rangle$ be internal ordered data and let $F : \mathcal{C} \longrightarrow \mathcal{C}'$ be a morphism of internal ordered data. Then the assignment:

- $\langle \mathcal{C}, \mathcal{M} \rangle \longmapsto \langle \mathcal{C}, \mathcal{C}, id, \mathcal{M}, \mathcal{C}(-, M) \rangle$ and
- $F \longmapsto \langle F, F \rangle$

defines a functor oout: **OInt**→**OExt**.

Proof: $oout(F)$ is a morphism of external data by proposition 5.4.1. It is a morphism of external ordered data since $F(\leq) \subseteq \leq'$ by assumption.
The verification that *oout* preserves composition and identities follows the proof of proposition 5.4.1. □

Proposition 6.3.20 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$ be external ordered data and let $\langle F, G \rangle$ be a morphism of external ordered data. Then the assignment:

- $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle \longmapsto \langle \mathcal{D}, \mathcal{M} \rangle$ and
- $\langle F, G \rangle \longmapsto G$

defines a functor *oin* : **OExt**→**OInt**.

Proof: *oin* is well defined since, by assumption, G preserves finite limits and $G(\leq) \subseteq \leq'$. □

Theorem 6.3.21 We have an adjunction $oin \dashv oout : \mathbf{OInt} \rightarrow \mathbf{OExt}$.

Proof: Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle \in |\mathbf{OExt}|$ and $\langle \mathcal{C}', \mathcal{M}' \rangle \in |\mathbf{OInt}|$. We need to show that there is a natural bijection:
$\mathbf{OInt}(oin(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle), \langle \mathcal{C}', \mathcal{M}' \rangle) \cong \mathbf{OExt}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, oout(\langle \mathcal{C}', \mathcal{M}' \rangle))$.
Now $\mathbf{OExt}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, oout(\langle \mathcal{C}', \mathcal{M}' \rangle)) =$
$\mathbf{OExt}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, \langle \mathcal{C}', \mathcal{C}', id, \mathcal{M}', \mathcal{C}'(-, M) \rangle)$
an element of which is a pair of functors $\langle F, G \rangle$ such that:

- $\langle F, G\rangle$ is a morphism of external data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}_o, \phi^{\mathcal{C}}\rangle$ to $\langle \mathcal{C}', \mathcal{C}', id, \mathcal{M}'_o, \mathcal{C}'(-, M)\rangle$ and

- $G(\leq) \subseteq \leq'$.

However, by lemma 5.4.3, $\langle F, G\rangle$ is fully determined by G as a morphism of internal data from $\langle \mathcal{D}, \mathcal{M}_o\rangle$ to $\langle \mathcal{C}', \mathcal{M}'_o\rangle$.
Now $\mathbf{OInt}(oin(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle), \langle \mathcal{C}', \mathcal{M}'\rangle) = \mathbf{OInt}(\langle \mathcal{D}, \mathcal{M}\rangle, \langle \mathcal{C}', \mathcal{M}'\rangle)$
an element of which is a functor G such that:

- G is a morphism of internal data from $\langle \mathcal{D}, \mathcal{M}_o\rangle$ to $\langle \mathcal{C}', \mathcal{M}'_o\rangle$ and

- $G(\leq) \subseteq \leq'$.

It now follows immediately that we have the required bijection.
Naturality is evident and this completes the proof. □

Corollary 6.3.22 **OInt** is a full reflective subcategory of **OExt**.

Proof: Trivially *oout* is injective on objects. It follows mutatis mutandis from lemma 5.4.3 that *oout* is fully faithful. Finally *oout* has a left adjoint by theorem 6.3.21 and the result follows. □

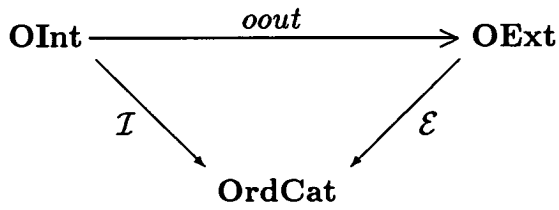
Corollary 6.3.23 Let U_e : **OExt** → **Ext** and U_i : **OInt** → **Int** be the evident forgetful functors. Then both squares of:

$$\begin{array}{ccc} \mathbf{OExt} & \underset{oout}{\overset{oin}{\rightleftarrows}}\ (\perp) & \mathbf{OInt} \\ \Big\downarrow U_e & & \Big\downarrow U_i \\ \mathbf{Ext} & \underset{out}{\overset{in}{\rightleftarrows}}\ (\perp) & \mathbf{Int} \end{array}$$

commute.

We have related the categories **OInt** and **OExt**. The following results show that this relationship is compatible with the functors $\mathcal{I}$ and $\mathcal{E}$.

Theorem 6.3.24 The diagram:



commutes up to natural isomorphism.

Proof: By theorem 5.4.7, if suffices to show that the map $\iota : \mathcal{I}(\langle \mathcal{C}, \mathcal{M} \rangle) \to \mathcal{E}(oout(\langle \mathcal{C}, \mathcal{M} \rangle))$ given by:

$$\langle f, t \rangle : A \longrightarrow B \times M \longmapsto \langle f, t \rangle : A \longrightarrow B$$

is an ordered functor. This is clear. □

These results show that **OExt** is more general than **OInt**. In the study of non-exact complexity we shall be most interested in external ordered data. In particular, as in chapter 5, there are examples of principal interest which cannot be expressed using only internal ordered data.

Example 6.3.25 Let $\mathbf{Set_p}$ be the category of sets and partial functions, let $(-)_\perp : \mathbf{Set_p} \to \mathbf{Set}$ be the lifting functor, let $\mathcal{M}$ be the ordered monoid $\langle \omega + 1, 0, +, \leq \rangle$ and for each $A \in |\mathbf{Set_p}|$, let $\phi^{\mathbf{Set_P}}(A)$ be the set of functions from $A_\perp$ to $\mathcal{M}$ mapping $\perp$ to ω. The idea is that the top element ω of $\omega + 1$ represents infinite time. The restriction to functions mapping $\perp$ to ω is the requirement that non terminating programs are given infinite time. The tuple $\langle \mathbf{Set_p}, \mathbf{Set}, \perp, \mathcal{M}, \phi^{\mathbf{Set_P}} \rangle$ is then an external ordered datum whose complexity category can model the complexity of partial computations.

Chapter 7

Input Measures

7.1 Introduction

In chapter 5, we expressed the exact complexity of a program p of type $A \to B$ as a map from the denotation of A to a monoid $\mathcal{M}$ of resource values. In this chapter, we express the complexity of p as a map $cx(p)$ from the object of sizes $s(A)$ to the monoid $\mathcal{M}$ of resource values. The idea is that $cx(p)(s)$ is the amount of resource consumed by the program p when applied to an input of size s.

Example 7.1.1 In standard ML, lists are an important datatype constructor. In many applications, such as the following program to duplicate each element in a list:

```
- fun double [] = [] | double (hd::tl) = hd::hd::double(tl);
> val double = fn : ('a list) -> ('a list)
- double [1,2,3];
> [1,1,2,2,3,3] : int list
- double ["what"];
> ["what","what"] : string list
```

the measure of the size of a list is its length. The complexity is then expressed as a function from integers to integers, and we say that the complexity is $t(n)$ if this is the resource required to apply `double` to a list of length n.

For this to be well-defined, it is necessary to make some choice of the type of analysis, for example, the greatest or the average resource consumed by p on all possible input values of size s.

Example 7.1.2 Worst case analysis: Let A be $\mathbb{Z}$, let the object $s(A)$ of sizes of A be $\mathbb{N}$ and let the monoid $\mathcal{M}$ be $\langle \mathbb{N}, 0, + \rangle$. Let $p : \mathbb{Z} \to \mathbb{Z}$ be a program which computes x^2 and let $m : \mathbb{Z} \to \mathbb{N}$ be the modulus function. The worst case complexity for p is the function $T : \mathbb{N} \to \mathbb{N}$ given by:

$$T(n) \;=\; max\{t(a) | m(a) = n\} \;=\; max\{t(n), t(-n)\}$$

where $t(a)$ is the time taken by p on an input a in A.
We say that the worst case complexity of p is $O(n^\alpha)$ if $\exists n_0$ such that $\forall n \geq n_0$, $T(n) \leq n^\alpha$.

Example 7.1.3 Average case analysis: Let A be $\mathbb{N} \times \mathbb{N}$, let the object $s(A)$ of sizes of A be $\mathbb{N}$ and let the monoid $\mathcal{M}$ be $\langle \mathbb{N}, 0, + \rangle$. Let $p : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a program which computes $x \times y$ and let $m : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be addition. The average case complexity for p is the function $T : \mathbb{N} \to \mathbb{N}$ given by:

$$T(n) \;=\; mean\{t(a) | m(a) = n\} \;=\; mean\{t(\langle p, q \rangle) | p + q = n\}$$

where $t(a)$ is the time taken by p on an input a in A.
We say the average case complexity of p is $O(n^\alpha)$ if $\exists n_0$ such that $\forall n \geq n_0$, $T(n) \leq n^\alpha$.

In this chapter, we aim to capture the notion of input measure within our framework. In chapter 8 we relate this to a non-exact framework. In fact, if we use input measures and we insist that our semantics be compositional, that is the complexity of a sequential composition of programs $p; q$ is obtainable from the complexity of p and of q, then we have to use a non-exact framework.

In section 7.2, we define input measures and analyses. We show how they give rise to models in which complexity is expressed as a map from input size to resource. These models are unsatisfactory because they destroy the compositionality of exact complexity. However, we can recover compositionality by moving to a non-exact framework.

In section 7.3, we use the enriched structures introduced in chapter 7 to formulate a definition of lax measure. This allows us to express the idea of an upper bound on the complexity of a program.

In section 7.4, we consider the kind of equivalence relations that we need for a non-exact framework. This leads to the definition of an M-equivalence on an external datum. We prove a characterisation result for M-equivalences and that every M-equivalence gives rise to a congruence on $\mathcal{C}^M$. We show that the models we seek can be expressed as quotients of external complexity categories by congruences generated by M-equivalences. In chapter 8, we apply this work to develop compositional non-exact models.

7.2 Measures

7.2.1 Introduction

We define the concept of measure and analysis and give examples. The idea of a measure for an object A is that it gives the size of each element of A. The idea of an analysis is that it gives the type of complexity, such as worst case or average case. We classify the conditions under which a measure gives rise to a suitable model of complexity.

In the following section, we shall restrict attention to the case where the category $\mathcal{D}$ is **Set**. This is not an essential restriction but it simplifies the theory and captures the examples of principal interest to us.

7.2.2 Measures and analyses

Definition 7.2.1 A **measure** for an external ordered datum $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ is a pair $\langle s, \mu \rangle$ consisting of:

- a function $s : |\mathcal{C}| \to |\mathbf{Set}|$ and
- for each A in $|\mathcal{C}|$, a function $\mu_A : UA \to s(A)$

such that the set $\mu_A^{-1}(s)$ is finite for each s in $s(A)$.

The idea is that $s(A)$ is the object of sizes for A and that the map μ_A is the input measure for A. The condition on μ_A corresponds to the reasonable condition that there are at most finitely many values of any given size.

Example 7.2.2 In example 2.1.12, we gave an example of a Pascal program which takes an input value n of type Nat, and produces the output $n!$. We showed that the complexity of the program was $O(n!)$ where n is the input value. However, in complexity theory, the size of an input is often taken to be the length of its binary representation.

In example 5.3.17, we gave an external datum $\langle \mathcal{S}, \mathbf{Set}, (-)_\perp, \mathcal{M}, \phi^{\mathcal{S}} \rangle$ whose external complexity category was a model for our extended semantics for Pascal. Using this external datum, we can express the above measure by taking $s(\mathbf{N})$ to be $\mathbf{N}$ and $\mu_{\mathbf{N}}$ to be the function:

$$\lambda n.\ \lceil log_2 n \rceil$$

Example 7.2.3 Let $\mathcal{C}$ be the single object category consisting of the object $\Sigma = (0,1)^*$ and all computable functions from Σ to Σ, so that $\mathcal{C}$ is a model for Turing Machine computations. Let $\mathcal{M}$ be the ordered monoid $\langle \mathbf{N}, 0, +, \leq \rangle$ in **Set**. Then the following data:

- $s(\Sigma) = \mathbf{N}$ and
- $\mu : \Sigma \to \mathbf{N}$ given by $\mu(\sigma) = length(\sigma)$

defines a measure for $\langle \mathcal{C}, \mathbf{Set}, \iota, \mathcal{M}, \mathbf{Set}(-, \mathbf{N}) \rangle$.

Example 7.2.4 In each of the above examples, the object of size has been the natural numbers. However, this is not always the case. For example, consider the following Pascal program:

```
begin
        s:=1.0;
        read n,r;
        while n>0 do
                        begin
                                s:=s/r;
                                n:=n-1;
                        end;
        write s;
end.
```

The input type of the program is `int`$\times$`real`, and we might take the size of an input $\langle n, r \rangle$ to be the pair $\langle n, \lceil ln\ r \rceil \rangle$. In order to model this, the size object $s([\![\texttt{int}]\!] \times [\![\texttt{real}]\!])$ would need to be $\mathbf{N} \times \mathbf{N}$.

Example 7.2.5 Let $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum. Then the following data:

- $U : |\mathcal{C}| \to |\mathbf{Set}|$ and
- $\forall A,\ id : UA \to UA$

defines a measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$. This is the trivial case where the "size" of an input is simply the input.

In the introduction, we stated that it is necessary to make some choice of the type of analysis, such as, the greatest or the average resource consumed by a program p on all possible inputs of a given size. These are the two principal

examples. In fact, it is not too hard to unify these examples, and this leads to the following definition of an analysis as a function Ω from the finite powerset of the monoid $\mathcal{M}$ to M. For technical reasons, we require that Ω satisfies two simple conditions.

Notation 7.2.6 Let $\mathcal{M} = \langle M, 0, \cdot, \leq \rangle$ be an ordered monoid in **Set**. Let P_f be the covariant finite powerset functor. We define a monoid $\langle P_f M, \{0\}, \cdot \rangle$ by:

$$M_o \cdot M_1 = \{m_o \cdot m_1 | m_0 \in M_o \text{ and } m_1 \in M_1\}$$

Definition 7.2.7 An **analysis** for an ordered monoid $\mathcal{M} = \langle M, 0, \cdot, \leq \rangle$ in **Set** is a function $\Omega : P_f M \to M$ such that:

- $\Omega\{m\} = m$.
- $\Omega(M' \cdot M'') \leq \Omega(M') \cdot \Omega(M'')$

Remark 7.2.8 *The condition* $\Omega(M' \cdot M'') = \Omega(M') \cdot \Omega(M'')$ *would define* Ω *as a morphism of monoids in* **Set**. *However, this condition is not satisfied by all of our examples (c.f. example 7.2.17).*

Example 7.2.9 Let $\mathcal{N}$ be the ordered monoid $\langle \mathbf{N}, 0, +, \leq \rangle$ in **Set** and let $\Omega : P_f \mathbf{N} \to \mathbf{N}$ be given by:

$$\Omega(X) = \begin{cases} max\{x \in X\} & \text{if } X \neq \emptyset \\ 0 & \text{if } X = \emptyset. \end{cases}$$

It is clear that $\Omega(\{m\}) = m$ and that $\Omega(M' \cdot M'') \leq \Omega(M') \cdot \Omega(M'')$ and so Ω is an analysis for $\mathcal{N}$.

This example corresponds to worst case analysis.

Example 7.2.10 Let $\mathcal{N}$ be the ordered monoid $\langle \mathbf{N}, 0, +, \leq \rangle$ in **Set** and let $\Omega : P_f \mathbf{N} \to \mathbf{N}$ be given by:

$$\Omega(X) = mean\{n \in X\}.$$

It is clear that $\Omega(\{m\}) = m$ and that $\Omega(M' \cdot M'') \leq \Omega(M') \cdot \Omega(M'')$ and so Ω is an analysis for $\mathcal{N}$.

This example corresponds to average case analysis.

7.2.3 The model $\mathcal{C}_S^M$

Notation 7.2.11 Let $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum, let $\langle s, \mu \rangle$ be a measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and let Ω be an analysis for $\mathcal{M}$. For each A, let $n_A : \phi^{\mathcal{C}}(A) \to \mathbf{Set}(s(A), M)$ be the function given by the following commuting diagram:

$$\begin{array}{ccc} s(A) & \xrightarrow{\quad n_A(t) \quad} & M \\ {\scriptstyle \mu_A^{-1}}\downarrow & & \uparrow{\scriptstyle \Omega} \\ P_f(UA) & \xrightarrow[P_f(t)]{\qquad} & P_f(M) \end{array}$$

Remark 7.2.12 *Note that n_A is well defined because μ_A is image finite*

The idea is as follows. Given a size s in $s(A)$ and a morphism $\langle f, t \rangle$ in $\mathcal{C}^M$, $\mu_A^{-1}(s)$ is the set of elements of size s. Now $t(a)$ is the resource required to evaluate f at the element a in A, and so $P_f(t)(\mu_A^{-1}(s))$ is the set of resource requirements of the elements a in A of size s. Finally, $\Omega(P_f(t)(\mu_A^{-1}(s)))$ selects an element of the set such as the maximum or the average. Thus, n_A sends a complexity map t in $\phi^{\mathcal{C}}(A)$ to the corresponding map from sizes to resource, with respect to the input measure μ and the choice of the type of complexity Ω.

Using measures, the denotation of a program p of type $A \to B$ is a pair of maps $\langle fun(p) : [\![A]\!] \longrightarrow [\![B]\!], cx(p) : s([\![A]\!]) \longrightarrow M \rangle$ where the map $cx(p)$ represents the amount of resource consumed by the program p on an input of a given size.

The composite of pairs $\langle f, t \rangle$ and $\langle g, s \rangle$ is $\langle gf, t \cdot n_A(s\mu_B Uf) \rangle$ where $s\mu_B Uf$ is the map:

$$UA \xrightarrow{Uf} UB \xrightarrow{\mu_B} s(B) \xrightarrow{s} M.$$

The idea is that the amount of resource consumed by the program $p; q$ on an input of size $s \in s(A)$ is that consumed by p on s, together with that consumed by q on the size of output produced by an input of size s.

Consider the graph $\mathcal{C}_S^M$ whose objects are objects of $\mathcal{C}$ and whose arrows are pairs $f : A \longrightarrow B$ in $\mathcal{C}$ and $t : s(A) \longrightarrow M$ in $\mathcal{D}$. Suppose that we have two arrows $\langle f,t\rangle : A \longrightarrow B$ and $\langle g,s\rangle : B \longrightarrow C$ in $\mathcal{C}_S^M$. Then $s\mu_B Uf$ is a map from UA to M in $\phi^{\mathcal{C}}(A)$ and thus $n_A(s\mu_B Uf)$ is a map from $s(A)$ to M in $\mathcal{D}$. Therefore, we have an evident composition given by:

$$\langle g,s\rangle\langle f,t\rangle \;=\; \langle gf, t\cdot n_A(s\mu_B Uf)\rangle$$

Alas, this does not define a category since this composition need not be associative. The following example shows that, even in the "worst case" example, $\mathcal{C}_S^M$ may not even have identities.

Example 7.2.13 Let $\mathbf{N}$ be the one object category with object $\mathbf{N}$, morphisms all computable functions from $\mathbf{N}$ to $\mathbf{N}$ and functional composition, let $\mathcal{M}$ be the monoid $\langle \mathbf{N}, 1, \times, \leq\rangle$ in **Set**, let $s = id$, let $\mu = \lceil log_2 \rceil$ and let $\Omega : P_f\mathbf{N} \to \mathbf{N}$ be max. Then $\langle s,\mu\rangle$ is a measure for $\langle \mathbf{N}, \mathbf{Set}, \iota, \mathcal{M}, (-,\mathbf{N})\rangle$ and Ω is an analysis for $\mathcal{M}$, but $\mathcal{C}_S^M$ does not have identities.

Proof: It is clear that $\mu^{-1}(n)$ is a finite set for each n in $\mathbf{N}$, and, by example 7.2.9, Ω is an analysis for $\mathcal{M}$.
Now the left identity in $\mathcal{C}_S^M$ is given by $\langle id, 0\rangle$ since:

$$\langle id,0\rangle\langle f,t\rangle \;=\; \langle idf, t\cdot n_A(0s_AUf)\rangle \;=\; \langle f, t\cdot n_A(0)\rangle \;=\; \langle f,t\rangle.$$

However this is not necessarily a right identity since:

$$\begin{aligned}\langle f,t\rangle\langle id,0\rangle \;&=\; \langle fid, 0\cdot n_A(ts_AU(id))\rangle \\ &\quad\; \langle f, n_A(ts_A)\rangle\end{aligned}$$

and $n_A(ts_A)$ is the function $m \longmapsto 2^{\lceil log_2 m\rceil}$ which is not the identity. Therefore, $\langle f,t\rangle\langle id,0\rangle \;=\; \langle f, n_A(ts_A)\rangle \neq \langle f,t\rangle$ and $\mathcal{C}_S^M$ does not have identities. □

7.2.4 Conditions for $\mathcal{C}_S^M$ to be a model

The following result classifies the conditions under which $\mathcal{C}_S^M$ is a category.

Proposition 7.2.14 Let $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum, let $\langle s, \mu \rangle$ be a measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and let Ω be an analysis for $\mathcal{M}$. Then $\mathcal{C}_S^M$ is a category and the assignment $\langle f, t \rangle \longmapsto \langle f, n_A(t) \rangle$ defines a functor $n : \mathcal{C}^M \to \mathcal{C}_S^M$ if and only if:

- $n(s \cdot t) = n(s) \cdot n(t)$,
- $n(t\mu_A) = t$ and
- $n_A(tUf) = n_A(n_B(t)\mu_B U f)$.

Proof: Suppose the conditions hold. We claim that the identity at A is $\langle id, 0 \rangle$ since:

- $\begin{aligned} \langle f, t \rangle \langle id, 0 \rangle &= \langle fid, 0 \cdot n(t\mu_A U(id)) \rangle \\ &= \langle f, n(t\mu_A) id \rangle \\ &= \langle f, n(t\mu_A) \rangle \\ &= \langle f, t \rangle. \end{aligned}$
- $\begin{aligned} \langle id, 0 \rangle \langle f, t \rangle &= \langle idf, t \cdot n(0\mu_A U(f)) \rangle \\ &= \langle f, t \cdot n(0) \rangle \\ &= \langle f, t \rangle. \end{aligned}$

It is clear that composition is well-defined and composition is associative since:

- $\begin{aligned} (\langle h, t \rangle \langle g, s \rangle) \langle f, r \rangle &= \langle hg, s \cdot n_B(t\mu_C U(g)) \rangle \langle f, r \rangle \\ &= \langle (hg)f, r \cdot n_A((s \cdot n_B(t\mu_C U(g)))\mu_B U(f)) \rangle \\ &= \langle h(gf), r \cdot n_A(s\mu_B U(f) \cdot n_B(t\mu_C U(g))\mu_B U(f)) \rangle \\ &= \langle h(gf), r \cdot n_A(s\mu_B U(f)) \cdot n_A(n_B(t\mu_C U(g))\mu_B U(f)) \rangle \\ &= \langle h(gf), r \cdot n_A(s\mu_B U(f)) \cdot n_A(t\mu_C U(g) U(f)) \rangle \\ &= \langle h(gf), r \cdot n_A(s\mu_B U(f)) \cdot n_A(t\mu_C U(gf)) \rangle \\ &= \langle h(gf), (r \cdot n_A(s\mu_B U(f))) \cdot n_A(t\mu_C U(gf)) \rangle \\ &= \langle h, t \rangle (\langle gf, r \cdot n_A(s\mu_B U(f)) \rangle) \\ &= \langle h, t \rangle (\langle g, s \rangle \langle f, r \rangle) \end{aligned}$

Thus $\mathcal{C}_S^M$ is a category.

Suppose that the conditions hold. Then $n : \mathcal{C}^M \to \mathcal{C}_S^M$ preserves identities since:

- $n(\langle id, 0\rangle) = \langle id, n(0)\rangle$
  $= \langle id, 0\rangle).$

and n preserves compositions since:

- $n(\langle gf, t \cdot sUf\rangle) = \langle gf, n(t \cdot sUf)\rangle$
  $= \langle gf, n(t) \cdot n(sUf)\rangle$
  $= \langle gf, n(t) \cdot n(n(s)\mu_B Uf)\rangle$
  $= \langle g, n(s)\rangle\langle f, n(t)\rangle$
  $= n(\langle g, s\rangle)n(\langle f, t\rangle).$

and so n is a functor.

Conversely, suppose that C_S^M is a category and that n is a functor. Then:

- $n(\langle id, 0\rangle) = \langle id, n(0)\rangle = \langle id, 0\rangle$

whence $n(0) = 0$ and:

- $n(\langle gf, t \cdot sUf\rangle) = n(\langle g, s\rangle)n(\langle f, t\rangle)$ thus
  $\langle gf, n(t \cdot sUf)\rangle = \langle g, n(s)\rangle\langle f, n(t)\rangle$ and
  $\langle gf, n(t \cdot sUf)\rangle = \langle gf, n(t) \cdot n(n(s)\mu_B Uf)\rangle.$

In particular taking $t = 0$ we obtain:

- $n(sUf) = n(0 \cdot sUf)$
  $= n(0) \cdot n(n(s)\mu_B Uf)$
  $= 0 \cdot n(n(s)\mu_B Uf)$
  $= n(n(s)\mu_B Uf).$

and taking $f = id$ we obtain:

- $n(t \cdot s) = n(t \cdot sU(id))$
  $= n(t) \cdot n(n(s)\mu_B U(id))$
  $= n(t) \cdot n(sU(id))$
  $= n(t) \cdot n(sid)$
  $= n(t) \cdot n(s).$

This completes the proof. □

The next result simplifies these conditions.

Corollary 7.2.15 Let $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum, let $\langle s, \mu \rangle$ be a measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$, let Ω be an analysis for $\mathcal{M}$ and suppose that the image of U in **Set** includes the constant functions. Then $\mathcal{C}_S^M$ is a category and the assignment $\langle f, t \rangle \longmapsto \langle f, n_A(t) \rangle$ defines a functor $n : \mathcal{C}^M \to \mathcal{C}_S^M$ if and only if:

$$\Omega(M' \cdot M'') \;=\; \Omega(M') \cdot \Omega(M'') \;\text{ for all } M', M'' \in\; P_f(M),$$

and the following diagram:

$$\begin{array}{ccccc} UA & & \xrightarrow{\qquad\qquad t \qquad\qquad} & & M \\ {\scriptstyle \mu_A}\downarrow & & & & \uparrow{\scriptstyle \Omega} \\ s(A) & \xrightarrow[\mu_A^{-1}]{} & P_f(UA) & \xrightarrow[P_f(t)]{} & P_f(M) \end{array}$$

commutes for all $A \in |\mathcal{C}|$ and for all $t : UA \longrightarrow M$.

Proof: Suppose that these conditions hold. Then:

- $$\begin{aligned} n_A(s \cdot t) &= \Omega P(s \cdot t)\mu_A^{-1} \\ &= \Omega(P(s)\mu_A^{-1} \cdot P(t)\mu_A^{-1}) \\ &= \Omega P(s)\mu_A^{-1} \cdot \Omega P(t)\mu_A^{-1} \\ &= n_A(s) \cdot n_A(t). \end{aligned}$$

Now $n_A(tUf) = \Omega P(tUf)\mu_A^{-1}$ and $n_A(n_B(t)\mu_B Uf) = \Omega P(\Omega P(t)\mu_B^{-1}\mu_B U(f))\mu_A^{-1}$. However, $t = \Omega P(t)\mu_A^{-1}\mu_A$ and thus $tUf = \Omega P(t)\mu_A^{-1}\mu_A Uf$ and so $\Omega P(tUf)\mu_A^{-1} = \Omega P(\Omega P(t)\mu_B^{-1}\mu_B U(f))\mu_A^{-1}$ and $n_A(tUf) = n_A(n_B(t)\mu_B Uf)$ as required.
The other cases follow similarly and this completes the proof. □

Remark 7.2.16 *The condition that the image of U includes the constant functions is mild.*

Unfortunately, most of the examples of principal interest do not satisfy these conditions.

Example 7.2.17 Consider the analysis Ω of example 7.2.10. Then Ω does not satisfy the condition that $\Omega(M' \cdot M'') = \Omega(M') \cdot \Omega(M'')$.

Proof: We have:

- $$\begin{aligned}\Omega(X + X') &= mean\{n + n' | n \in X,\ n' \in X'\} \\ &\neq mean\{n | n \in X\} + mean\{n | n \in X'\} \\ &\neq \Omega(X) + \Omega(X').\end{aligned}$$

This completes the proof. □

In the case where Ω is *max* (example 7.2.9), then Ω does satisfy the condition that $\Omega(M' \cdot M'') = \Omega(M') \cdot \Omega(M'')$. However, as the following example shows, this is still not sufficient.

Example 7.2.18 Let Σ be the one object category with object $(0,1)^*$ and morphisms all computable functions, so that Σ is a model for Turing machine computations. Let $\mathcal{M}$ be the ordered monoid $\langle \mathbf{N}, 0, +, \leq \rangle$ in **Set**, let $s(\Sigma) = \mathbf{N}$, let $\mu : \Sigma \to \mathbf{N}$ be given by $\mu(\sigma) = length(\sigma)$ and let $\Omega(N) = max\{n \in N\}$. Then $\Omega(M' \cdot M'') = \Omega(M') \cdot \Omega(M'')$ but n does not satisfy the condition of corollary 7.2.15 and hence C_S^M is not a category.

Proof: We have $X + X' = \emptyset$ if $X = \emptyset$ or $X' = \emptyset$ whence $\Omega(X + X') = 0 = \Omega(X) + \Omega(X')$. Otherwise:

- $$\begin{aligned}\Omega(X + X') &= max\{n + n' | n \in X,\ n' \in X'\} \\ &= max\{n | n \in X\} + max\{n | n \in X'\} \\ &= \Omega(X) + \Omega(X').\end{aligned}$$

and therefore $\Omega(M' \cdot M'') = \Omega(M') \cdot \Omega(M'')$.

Now, let $t : \Sigma \to \mathbf{N}$ be given by:

$$t(\sigma) = \#1\text{'s in } \sigma.$$

Then since $\mu(\sigma) = length(\sigma)$,

$$\mu^{-1}\mu(\sigma) = \{\sigma' | length(\sigma') = length(\sigma)\}$$

so $P(t)\mu^{-1}\mu(\sigma) = \{\#1\text{'s in } \sigma' | length(\sigma') = length(\sigma)\}$ and

$$\Omega P(t)\mu^{-1}\mu(\sigma) = max\{\#1\text{'s in } \sigma' | length(\sigma') = length(\sigma)\} = length(\sigma).$$

Thus $t(\sigma) \neq \Omega P(t)\mu^{-1}\mu(\sigma)$ for all σ. This completes the proof. □

We conclude that these conditions are too strong and therefore that $\mathcal{C}_S^M$ does not give a suitable model.

7.3 A category based on measures

7.3.1 Introduction

In the previous section, we saw that most of the examples of principal interest to us do not satisfy the conditions for $\mathcal{C}_S^M$ to be a category. However, many of the examples do satisfy these conditions if each of the equalities in proposition 7.2.14 is replaced by $\leq$.

In this section, we use the enriched structures introduced in chapter 7 to formulate a definition of lax measure. This does not lead to a category, but to an ordered category $\mathcal{S}^M$ and a lax functor $\langle s, n\rangle : \mathcal{C}^M \longrightarrow \mathcal{S}^M$. This allows us to express the idea of an upper bound on the complexity of a program.

7.3.2 Lax measures

Definition 7.3.1 A **lax measure** for an external ordered datum $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$ is a tuple $\langle S, V, s, n\rangle$ consisting of a locally ordered category S, an ordinary functor $V : S_o \to \mathcal{D}$, a lax functor $s : \mathcal{C} \to S$, and for each A, a function $n_A : \phi^{\mathcal{C}}(A) \to \mathcal{D}(V(s(A)), M)$ such that:

- $n(0) = 0$,
- $n(t \cdot s) \leq n(t) \cdot n(s)$ and
- $n_A(tUf) \leq n_A(t)V(s(f))$.

Example 7.3.2 Let S be the single object ordered category with object $\mathbf{N}$, morphisms all functions from $\mathbf{N}$ to $\mathbf{N}$, functional composition and the pointwise ordering. Let V be the inclusion of S_o into **Set**, let Σ, $\mathcal{M}$ and n be as in example 7.2.18 and finally, let s be the lax functor from Σ to S which sends a function $p : \Sigma \longrightarrow \Sigma$ to the function:

$$\hat{p}(n) = max\{length(p(\sigma))|length(\sigma) = n\}.$$

Then the tuple $\langle S, V, s, n\rangle$ is a lax measure for the external ordered datum $\langle \Sigma, \mathbf{Set}, \iota, \mathcal{M}, \mathbf{Set}(-, M)\rangle$.

Remark 7.3.3 *The collection of maps n_A seem suggestive of the components of a natural transformation from $\phi^{\mathcal{C}}$ to $\mathcal{D}(UsV(-), M)$. Unfortunately the laxness of s introduces some complications which means that the most that we can say is that the n_A are the components of a lax natural transformation from $\phi^{\mathcal{C}}$ to $\mathcal{D}(UsV(-), M)$ considered as graph morphisms.*

Lemma 7.3.4 Let $\langle S, V, s, n\rangle$ be a lax measure for an external ordered datum $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$. Then the tuple $\langle S_o, \mathcal{D}, V, \mathcal{M}, \mathcal{D}(V(-), M)\rangle$ is an external ordered datum.

Proof: Clear. □

Now by proposition 6.3.15 a lax measure gives rise to an ordered category $\mathcal{S}^M$. A morphism in $\mathcal{S}^M$ is a pair $\langle f, t\rangle$. The idea is that f maps sizes of input to sizes of output and t maps sizes of input to resources. We have the following result.

Proposition 7.3.5 Let $\langle S, V, s, n\rangle$ be a lax measure for an external ordered datum $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$. The assignment:

- $A \longmapsto s(A)$,

- $\langle f,t\rangle \longmapsto \langle s(f), n(t)\rangle$

defines a lax functor $\langle s,n\rangle : \mathcal{C}^M \to \mathcal{S}^M$.

Proof: We require $\langle s,n\rangle(id_A) \leq id_{\langle s,n\rangle(A)}$ and $\langle s,n\rangle(\langle f,t\rangle\langle g,s\rangle) \leq \langle s,n\rangle(\langle f,t\rangle)\langle s,n\rangle(\langle g,s\rangle)$.

- $\begin{aligned}\langle s,n\rangle(id_A) &= \langle s,n\rangle(\langle id,0\rangle)\\ &= \langle s(id), n(0)\rangle\\ &= \langle s(id),0\rangle\\ &\leq \langle id,0\rangle\\ &\leq id_{\langle s,n\rangle(A)}.\end{aligned}$

- $\begin{aligned}\langle s,n\rangle(\langle f,t\rangle\langle g,s\rangle) &= \langle s,n\rangle(\langle fg, t\cdot fs\rangle)\\ &= \langle s(fg), n(t\cdot fs)\rangle\\ &= \langle s(fg), n(t)\cdot n(fs)\rangle\\ &\leq \langle s(f)s(g), n(t)\cdot n(fs)\rangle\\ &\leq \langle s(f)s(g), n(t)\cdot s(f)n(s)\rangle\\ &\leq \langle s(f), n(t)\rangle\langle s(g), n(s)\rangle\\ &\leq \langle s,n\rangle(\langle f,t\rangle)\langle s,n\rangle(\langle g,s\rangle).\end{aligned}$

This completes the proof. □

What $\mathcal{S}^M$ gives us is an ordered category of complexities based on measures. This is an improvement on $\mathcal{C}_S^M$ but it is still not satisfactory because the functor $\langle s,n\rangle : \mathcal{C}^M \to \mathcal{S}^M$ is not strict. This means that we lose compositionality in going from exact complexity to complexity based on measures.

In chapter 8, we show how compositionality can be recovered by considering non-exact complexity. In the next section, we consider the kind of equivalence relations that we require for a non-exact framework.

7.4 Equivalences

7.4.1 Introduction

In this section, we introduce the notion of an M-equivalence on an external ordered datum. The idea is that an M-equivalence is a collection of equivalence relations on complexity maps. In chapter 8, we use M-equivalences to express non-exact complexity.

We give examples and classify M-equivalences. We prove that an M-equivalence gives a congruence on the complexity category.

7.4.2 Congruences

We introduce some notions from category theory [Mac71].

Definition 7.4.1 A **congruence** $\sim$ on a category $\mathcal{C}$ is a collection of equivalence relations $\sim_{A,B}$ such that:

- $\sim_{A,B}$ is an equivalence relation on $\mathcal{C}(A,B)$ and
- $f_1 \sim f_2$ implies that $gf_1 \sim gf_2$ and $f_1 h \sim f_2 h$.

Lemma 7.4.2 Let $\mathcal{C}$ be a category and let $\sim$ be a congruence on $\mathcal{C}$. Then the following data:

- objects of $\mathcal{C}$,
- equivalence classes of morphisms in $\mathcal{C}$ under $\sim$ and
- composition given by $[g][f] = [gf]$

defines a category $\mathcal{C}/\sim$. We call this the quotient category of $\mathcal{C}$ by $\sim$.

Proposition 7.4.3 Let $\mathcal{C}$ be a category and let $\sim$ be a congruence on $\mathcal{C}$. Then the assignment:

- $f : A \longrightarrow B \longmapsto [f] : A \longrightarrow B$

defines a quotient functor $q : \mathcal{C} \to \mathcal{C}/\sim$.

7.4.3 M-Equivalences

Definition 7.4.4 An **M-equivalence** $\sim$ on an external datum $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ is a collection of equivalence relations $\sim_A$ such that:

- $\sim_A$ is an equivalence relation on $\phi^{\mathcal{C}}(A)$,
- $s_1 \sim s_2$ implies that $t \cdot s_1 \sim t \cdot s_2$ and $s_1 \cdot t \sim s_2 \cdot t$ and
- $s_1 \sim_A s_2$ implies that $s_1 U f \sim_B s_2 U f$

where s_1, s_2 and t are elements of $\phi^{\mathcal{C}}(A)$ and f is any map from B to A in $\mathcal{C}$.

Notation 7.4.5 The subscripts on $\sim$ should be clear from the context and will usually be omitted.

Remark 7.4.6 *Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum and let $\mathcal{M}_o$ be the underlying monoid. Then $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}_o, \phi^{\mathcal{C}} \rangle$ is an external datum and thus we can form an M-equivalence on it.*

Example 7.4.7 The identity equivalence $\sim$ given by:

$$s \sim_A t \text{ if } s = t$$

is an M-equivalence.

Example 7.4.8 Let $\mathcal{C}$ be the full subcategory of **Set** with the two objects $\mathbf{N}$ and $1 = \{*\}$. Let $\mathcal{M}$ be the monoid $\langle \mathbf{N}, 0, + \rangle$ in **Set** and consider the external datum $\langle \mathcal{C}, \mathbf{Set}, \iota, \mathcal{M}, \mathbf{Set}(\iota(-), \mathbf{N}) \rangle$ where $\phi^{\mathcal{C}}(\mathbf{N})$ is the set all functions $\mathbf{N} \to \mathbf{N}$ and $\phi^{\mathcal{C}}(1)$ is all integers considered as maps $n : 1 \longrightarrow \mathbf{N}$.
Then given two maps $s, t : \mathbf{N} \longrightarrow \mathbf{N}$ define $s \approx t$ if:

$\exists c, C, n_0 \in \mathbf{N} - \{0\}$ such that $\forall\ n \geq n_0,\ s(n) \leq c(t(n)+1)$ and $t(n) \leq C(s(n)+1)$

The idea is that two complexity maps are to be considered equivalent if they are of the same order of magnitude.

Given two maps $n, m : 1 \longrightarrow \mathbf{N}$, define:

$$n \approx m \text{ if } n = m.$$

We claim that $\approx$ is an M-equivalence.
Proof: Since $n \approx m$ if and only if $n = m$ for maps $n, m : 1 \longrightarrow \mathbf{N}$, it is immediate that $\approx$ satisfies the conditions to be an M-equivalence for maps $n, m : 1 \longrightarrow \mathbf{N}$.

It remains to verify that $\approx$ satisfies the conditions to be an M-equivalence for maps $\mathbf{N} \to \mathbf{N}$.

We first show that $\approx$ is an equivalence relation on $\phi^{\mathcal{C}}(\mathbf{N})$

- $t(n) \leq 1.t(n) + 1\ \forall\ n$ and so $t \approx t$.

- $s \approx t \Rightarrow\ \exists c, C, n_0 \neq 0.$ s.t. $\forall n \geq n_0,\ s(n) \leq c(t(n)+1)$ and $t(n) \leq C(s(n)+1)$
  $\Rightarrow\ \exists c, C, n_0 \neq 0.$ s.t. $\forall n \geq n_0,\ t(n) \leq C(s(n)+1)$ and $s(n) \leq c(t(n)+1)$
  $\Rightarrow\ \exists c', C', n_0 \neq 0.$ s.t. $\forall n \geq n_0,\ t(n) \leq c'(s(n)+1)$ and $s(n) \leq C'(t(n)+1)$
  $\Rightarrow\ t \approx s.$

- $r \approx s \approx t \Rightarrow (\exists c, C, n_0 \neq 0.$ s.t. $\forall n \geq n_0,\ r(n) \leq c(s(n)+1)$ and
  $s(n) \leq C(r(n)+1))$ and
  $(\exists c', C', n_0' \neq 0.$ s.t. $\forall n \geq n_0',\ s(n) \leq c'(t(n)+1)$ and $t(n) \leq C'(s(n)+1))$
  $\Rightarrow \forall n \geq max\{n_0, n_0'\},\ r(n) \leq c(c'+1)(t(n)+1)$ and $t(n) \leq C(C'+1)(r(n)+1)$
  $\Rightarrow \exists c'', C'', n_0'' \neq 0$ s.t. $\forall n \geq n_0'',\ r(n) \leq c''(t(n)+1)$ and
  $t(n) \leq C''(r(n)+1)$
  $\Rightarrow r \approx t.$

We now show that $s_1 \approx s_2$ implies that $\forall t.\ t \cdot s_1 \approx t \cdot s_2$ and $s_1 \cdot t \approx s_2 \cdot t$.

- $s_1 \approx s_2 \Rightarrow \exists c, C, n_0 \neq 0.$ s.t. $\forall n \geq n_0,\ s_1(n) \leq c(s_2(n)+1)$ and
  $s_2(n) \leq C(s_1(n)+1)$
  $\Rightarrow \forall n \geq n_0,\ s_1(n)+t(n) \leq max(c,1)(s_2(n)+t(n)+1)$ and
  $s_2(n)+t(n) \leq max(C,1)(s_1(n)+t(n)+1)$
  $\Rightarrow \forall n \geq n_0,\ (s_1+t)(n) \leq max(c,1)((s_2+t)(n)+1)$ and
  $(s_2+t)(n) \leq max(C,1)((s_1+t)(n)+1)$
  $\Rightarrow \exists c', C', n_0 \neq 0.$ s.t. $\forall n \geq n_0,$
  $(s_1+t)(n) \leq c'((s_2+t)(n)+1)$ and $(s_2+t)(n) \leq C'((s_1+t)(n)+1)$
  $\Rightarrow s_1 \cdot t \approx s_2 \cdot t.$

- $s_1 \approx s_2 \Rightarrow \exists c, C, n_0 \neq 0.$ s.t. $\forall n \geq n_0,\ s_1(n) \leq c(s_2(n)+1)$ and $s_2(n) \leq C(s_1(n)+1)$
  $\Rightarrow \forall n \geq n_0,\ t(n)+s_1(n) \leq max(c,1)(t(n)+s_2(n)+1)$ and
  $t(n)+s_2(n) \leq max(C,1)(t(n)+s_1(n)+1)$
  $\Rightarrow \forall n \geq n_0,\ (t+s_1)(n) \leq max(c,1)((t+s_2)(n)+1)$ and
  $(t+s_2)(n) \leq max(C,1)((t+s_1)(n)+1)$
  $\Rightarrow \exists c', C', n_0 \neq 0.$ s.t. $\forall n \geq n_0,\ (t+s_1)(n) \leq c'((t+s_2)(n)+1)$ and $(t+s_2)(n) \leq C'((t+s_1)(n)+1)$
  $\Rightarrow t \cdot s_1 \approx t \cdot s_2.$

Finally, we show $s \approx t$ implies that $sUf \approx tUf$.

- $s \approx t \Rightarrow \exists c, C, n_0 \neq 0$. s.t. $\forall n \geq n_0$, $s(n) \leq c(t(n)+1)$ and $t(n) \leq C(s(n)+1)$
  $\Rightarrow \forall n \geq n_0$, $s(Uf(n)) \leq c'(t(Uf(n))+1)$ and $t(Uf(n)) \leq C'(s(Uf(n))+1)$
  where $c' = \max\{s(n)+c \mid n < n_0\}$ and $C' = \max\{t(n)+C \mid n < n_0\}$
  $\Rightarrow \exists c, C, n_0 \neq 0$. s.t. $\forall n \geq n_0$, $s(Uf(n)) \leq c(t(Uf(n))+1)$ and $t(Uf(n)) \leq C(s(Uf(n))+1)$
  $\Rightarrow sUf \approx tUf$.

This establishes that $\approx$ is an M-equivalence. □

7.4.4 Classification results

One would imagine than any equivalence relation on M should extend pointwise to an M-equivalence. However, the definition would not be satisfactory if all M-equivalences were of this form. The following results show that we have a consistent but non-trivial class of equivalence relations.

Let $\mathcal{C}$ be a category with finite products. We recall from chapter 6 that an equivalence relation on an object A of $\mathcal{C}$ is a monomorphism $\alpha : A' \longrightarrow A \times A$ which satisfies certain reflexivity, symmetry and transitivity conditions.

The next result shows that every equivalence relation on M, satisfying a certain condition, induces an M-equivalence.

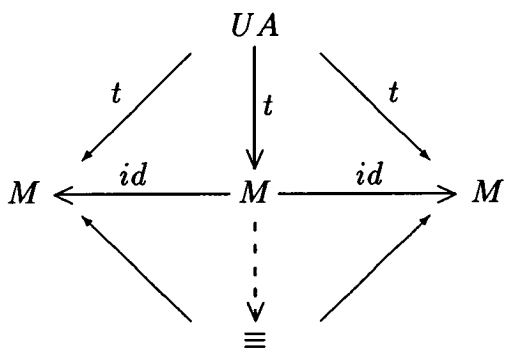
Lemma 7.4.9 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external datum. Let $\alpha : \equiv \longrightarrow M \times M$ be an equivalence relation on M such that there exists a map from $\equiv \times \equiv$ to $\equiv$ making the following diagram:

$$\begin{array}{ccc} \equiv \times \equiv & \xrightarrow{\alpha \times \alpha} & (M \times M) \times (M \times M) \\ \vdots & & \downarrow \cdot \times \cdot \\ \equiv & \xrightarrow[\alpha]{} & M \times M \end{array}$$

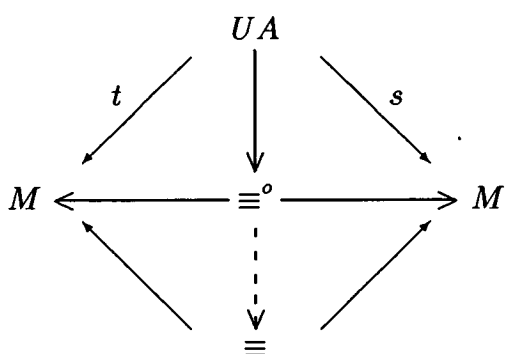
commute. Then the collection of relations on $\phi^{\mathcal{C}}$ given by $s \sim t$ if the map $\langle s,t\rangle : UA \longrightarrow M \times M$ factors through $\alpha : \equiv \longrightarrow M \times M$, defines an M-equivalence on $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$.

Remark 7.4.10 *Informally, the condition on $\equiv$ in lemma 7.4.9 states that, if $m_0 \equiv m_0'$ and $m_1 \equiv m_1'$, then $m_0 \cdot m_1 \equiv m_0' \cdot m_1'$. The M-equivalence $\sim$ then corresponds to the pointwise equivalence.*

Proof: We first show that $\sim$ gives an equivalence relation on each $\phi^{\mathcal{C}}(A)$. The following diagram:

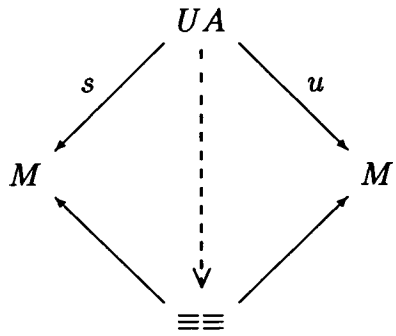


commutes because $\equiv$ is a reflexive relation and this shows that $\sim$ is reflexive. Suppose that $s \sim t$. Then the following diagram:



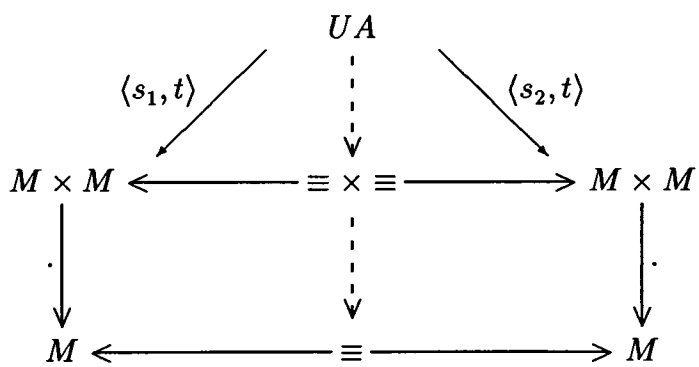
commutes because $\equiv$ is a symmetric relation and this shows that $\sim$ is symmetric. Suppose that $s \sim t$ and $t \sim u$. Then a simple diagram chase implies that the

following diagram:



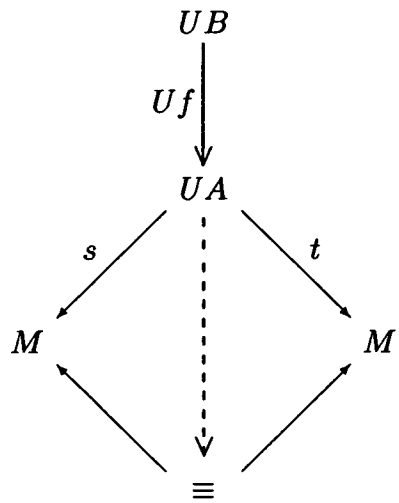
commutes. However, $\equiv$ is transitive so it follows that $s \sim u$. Thus $\sim$ is transitive and this establishes that $\sim$ is an equivalence relation on each $\phi^{\mathcal{C}}$.

Suppose now that $s_1 \sim s_2$. Then a moment's consideration will convince the reader that the following diagram:



verifies that $s_1 \cdot t \sim s_2 \cdot t$. Similarly, $s_1 \sim s_2$ implies that $t \cdot s_1 \sim t \cdot s_2$.

Finally, the following diagram:



verifies that $s \sim t$ implies that $sUf \sim tUf$ and this completes the proof. □

Notation 7.4.11 We call such an M-equivalence **trivial.**

One might imagine that all M-equivalences are trivial. If this were the case then the definition of M-equivalence would not be satisfactory since we want to consider a more general class of equivalences. However, this is not the case.

Proposition 7.4.12 Not every M-equivalence is trivial.

Proof: Let $\sim$ be the M-equivalence of example 7.4.8.
Suppose that $\sim$ were trivial. Then assume, without loss of generality, that $\sim$ is generated by an equivalence relation $\approx$ on $\mathbf{N}$.
Let id be the identity on $\mathbf{N}$ and let s be the successor function on $\mathbf{N}$.
Then $id \sim s$ since:

$$id(n) = n \leq n+1 = s(n) \quad \text{for all } n > 0 \quad \text{and}$$

$$s(n) = n+1 \leq 2n = 2id(n) \quad \text{for all } n > 0.$$

Now by assumption,

$$f \sim g \text{ if and only if } \forall n \in \mathbf{N}.\ f(n) \approx g(n)$$

and so for all n, $id(n) \approx s(n)$, that is $n \approx n+1$. By a trivial induction, this implies that

$$n \approx m \; \forall \; n, m \in \mathbf{N}.$$

Therefore, $f \sim g$ for all $f, g : \mathbf{N} \to \mathbf{N}$ since clearly $f(n) \approx g(n)$ for all n. This is a contradiction since, for example,

$$f(n) = n \not\approx g(n) = n^2.$$

This completes the proof. □

7.4.5 From M-equivalences to congruences

We show how M-equivalences give rise to congruences on complexity categories.

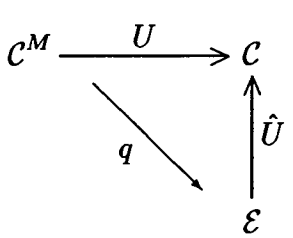
Lemma 7.4.13 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external datum and let $\mathcal{C}^M$ be the complexity category. Then the assignment:

- $A \longmapsto A$ and
- $\langle f, t \rangle \longmapsto f$

defines a functor $U : \mathcal{C}^M \to \mathcal{C}$.

Proof: Clear. □

Definition 7.4.14 Let $\mathcal{E}$ be a category, let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external datum and let $q : \mathcal{C}^M \to \mathcal{E}$ be a functor. $\mathcal{E}$ has the **functional behaviour of** $\mathcal{C}^M$ if there exists a functor $\hat{U} : \mathcal{E} \to \mathcal{C}$ such that the following diagram:



commutes.

Proposition 7.4.15 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external datum and let $\sim$ be an M-equivalence on $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$. Then the relation:

$$\langle f, t \rangle \sim \langle g, s \rangle \text{ if } f = g \text{ and } t \sim s$$

defines a congruence $\sim$ on $\mathcal{C}^{\mathcal{M}}$. Furthermore, the quotient category $\mathcal{C}^{\mathcal{M}}/\sim$ has the functional behaviour of $\mathcal{C}^{\mathcal{M}}$.

Proof: It is easy to see that $\sim$ is an equivalence relation on each $\mathcal{C}^{\mathcal{M}}(A, B)$ as:

- $f = f$ and $t \sim t$ implies that $\langle f, t \rangle \sim \langle f, t \rangle$.
- $\langle f, t \rangle \sim \langle g, s \rangle \Rightarrow f = g$ and $t \sim s$
  $\Rightarrow g = f$ and $s \sim t$
  $\Rightarrow \langle g, s \rangle \sim \langle f, t \rangle$.
- $\langle f, t \rangle \sim \langle g, s \rangle \sim \langle h, r \rangle \Rightarrow (f = g$ and $t \sim s)$ and $(g = h$ and $s \sim r)$
  $\Rightarrow f = g = h$ and $t \sim s \sim r$
  $\Rightarrow f = h$ and $t \sim r$
  $\Rightarrow \langle f, t \rangle \sim \langle h, r \rangle$.

We now show that $\sim$ is a congruence on $\mathcal{C}^{\mathcal{M}}$. Suppose that we have:

$$\langle f_1, t_1 \rangle \sim \langle f_2, t_2 \rangle.$$

Then given $\langle g, s \rangle$ and $\langle h, r \rangle$ we have:

- $\langle g, s \rangle \langle f_i, t_i \rangle = \langle g f_i, s \cdot t_i U g \rangle$

and so:

- $\langle f_1, t_1 \rangle \sim \langle f_2, t_2 \rangle \Rightarrow f_1 = f_2$ and $t_1 \sim t_2$
  $\Rightarrow g f_1 = g f_2$ and $t_1 U g \sim t_2 U g$
  $\Rightarrow g f_1 = g f_2$ and $s \cdot t_1 U g \sim s \cdot t_2 U g$
  $\Rightarrow \langle g f_1, s \cdot t_1 U g \rangle \sim \langle g f_2, s \cdot t_2 U g \rangle$
  $\Rightarrow \langle g, s \rangle \langle f_1, t_1 \rangle \sim \langle g, s \rangle \langle f_2, t_2 \rangle$.
- $\langle f_i, t_i \rangle \langle h, r \rangle = \langle f_i h, t_i \cdot r U f_i \rangle$

and so:

- $\langle f_1, t_1\rangle \sim \langle f_2, t_2\rangle \Rightarrow f_1 = f_2$ and $t_1 \sim t_2$
  $\Rightarrow f_1 h = f_2 h$ and $t_1 \sim t_2$ and $rUf_1 = rUf_2$
  $\Rightarrow f_1 h = f_2 h$ and $t_1 \cdot rUf_1 \sim t_2 \cdot rUf_2$
  $\Rightarrow \langle f_1 h, t_1 \cdot rUf_1\rangle \sim \langle f_1 h, t_2 \cdot rUf_2\rangle$
  $\Rightarrow \langle f_1, t_1\rangle\langle h, r\rangle \sim \langle f_2, t_2\rangle\langle h, r\rangle$.

Finally, we define the map $\hat{U} : \mathcal{C}^M/\sim \to \mathcal{C}$ by:

$$[\langle f, t\rangle] \longmapsto f.$$

We have to show that $\hat{U}$ is well-defined, that $\hat{U}$ is a functor and that $U = q\hat{U}$. We have $\hat{U}([\langle f, t\rangle]) = f$. Suppose that $\langle g, s\rangle \in \mathcal{C}^M$ were another choice for a representative of $[\langle f, t\rangle]$. Then:

- $\langle f, t\rangle \sim \langle g, s\rangle$ implies that $f = g$

and so $\hat{U}([\langle f, t\rangle]) = f = g = \hat{U}([\langle g, s\rangle])$, and thus $\hat{U}$ is well-defined.
To see that $\hat{U}$ is a functor, we have:

- $\hat{U}([\langle id, 0\rangle]) = id$ and

- $\hat{U}([\langle g, s\rangle][\langle f, t\rangle]) = \hat{U}([\langle g, s\rangle\langle f, t\rangle])$
  $= \hat{U}([\langle gf, t \cdot sUf\rangle])$
  $= gf$
  $= \hat{U}([\langle g, s\rangle])\hat{U}([\langle f, t\rangle])$

and so $\hat{U}$ preserves identities and composition. Lastly,

- $\hat{U}(q(\langle f, t\rangle)) = \hat{U}([\langle f, t\rangle])$
  $= f$
  $= U(\langle f, t\rangle)$.

and this completes the proof. □

Example 7.4.16 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external datum and let id be the identity M-equivalence of example 7.4.7. Then:

$$\mathcal{C}^M \cong \mathcal{C}^M/id.$$

Proof: Let $q : \mathcal{C}^M \to \mathcal{C}^M/id$ be the quotient functor. Since **Cat** is regular and q is necessarily epimorphic, it suffices to show that q is monomorphic, *i.e.* that q is one-one. Now:

- $$\begin{aligned} q(\langle f,t\rangle) = q(\langle g,s\rangle) &\Rightarrow [\langle f,t\rangle] = [\langle g,s\rangle] \\ &\Rightarrow f = g \text{ and } t \; id \; s \\ &\Rightarrow f = g \text{ and } t = s \\ &\Rightarrow \langle f,t\rangle = \langle g,s\rangle \end{aligned}$$

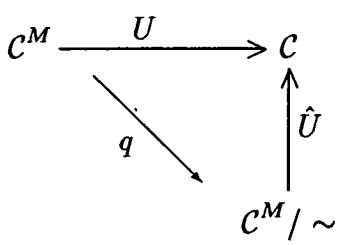
and so q is one-one and hence an isomorphism. This completes the proof. □

This is not the most general class of equivalences we could take. However, the next result shows that any suitable congruence on $\mathcal{C}^M$ can be bounded by a pair of M-equivalences. Furthermore, any congruence which satisfies:

$$\exists f : A \to B.\ s.t.\ \langle f,t\rangle \sim \langle f,s\rangle \quad \text{implies that} \quad \forall g.\ \langle g,t\rangle \sim \langle g,s\rangle$$

can be obtained from an M-equivalence. This is a reasonable condition since we consider complexity maps to be equivalent regardless of which program they are measuring the complexity of.

Proposition 7.4.17 Let $\sim$ be a congruence on $\mathcal{C}^M$ such that the following diagram:



commutes. Then the relations:

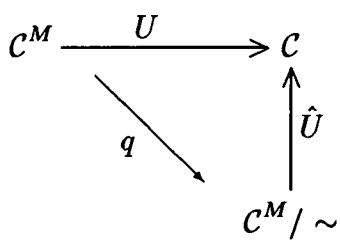
- $t \sim_m s$ if $\exists f \in C$ such that $\langle f, t\rangle \sim \langle f, s\rangle$ and
- $t \sim_M s$ if $\forall f \in C$ such that $\langle f, t\rangle \sim \langle f, s\rangle$

define, respectively minimal and maximal M-equivalences such that

$$\sim_M \leq \sim \leq \sim_m .$$

Proof: Since $U = q\hat{U}$, $\langle f, t\rangle \sim \langle g, s\rangle$ implies $f = g$.
It is not hard to verify that $\sim_M$ and $\sim_m$ are M-equivalences and satisfy the required property. □

Remark 7.4.18 *Let C be a groupoid. Let $\sim$ be a congruence on C^M such that:*



Then the M-equivalence $\sim_M$ of proposition 7.4.17 satisfies:

$$\sim = \sim_M .$$

Proof: *Suppose that $t \sim_m s$. Then there exists $f \in C$ such that $\langle f, t\rangle \sim \langle f, s\rangle$.*
However C is a groupoid and so there exists f^{-1} such that $ff^{-1} = id$.
Then $\langle f, t\rangle \sim \langle f, s\rangle$ implies $\langle f, t\rangle\langle f^{-1}g, 0\rangle \sim \langle f, s\rangle\langle f^{-1}g, 0\rangle$ for all g. Hence $\langle g, t\rangle \sim \langle g, s\rangle$ for all g and therefore $t \sim_m s$.
This completes the proof. □

Chapter 8

Non-Exact Complexity

8.1 Introduction

This chapter contains a study of non-exact complexity. This study is motivated by two main considerations.

In the analysis of algorithms and in complexity theory, program complexity is often given non-exactly. For example, in the analysis of algorithms, one is often interested in the order of magnitude of the complexity maps. In complexity theory, one often asks whether a complexity map is polynomial or super-polynomial in the size of its input.

In chapter 7, we showed how to capture the notion of input measures within our framework. We also showed that if we use input measures and exact complexity, then our semantics will not be compositional. That is the complexity of a sequential composition of programs $p; q$ will not be obtainable from the complexity of p and of q.

The aim of this chapter is the following. In chapter 5 we saw how an external datum gives rise to a model for the semantics of a programming language with exact complexity. We seek a method for taking an external datum, specifying the degree of non-exactness we wish to have in the complexity and then generating

a model for the semantics of a programming language with this degree of non-exactness in the complexity.

This method should satisfy certain conditions that arise from complexity theory or denotational semantics. It should be possible to specify non-exact complexity by means of input measures as in examples 7.1.1, 7.1.2 and 7.1.3. This is important to us because it corresponds to practice in complexity theory.

Secondly, we seek models with non-exact complexity but the same functional behaviour as the original model. In section 7.3, we constructed a category $\mathcal{S}^M$, based on input measures, by expressing the functional part of the denotation of a program as a map from size of input to size of output. This is the minimal functional information that is required in order to model program complexity. However, our original semantics gave us exact functional behaviour and we do not want our extended semantics to destroy this information. For example, suppose we only know how size of input maps to size of output. Then we could not distinguish between the program "times" of example 7.1.2 and the program "$-$times" which computes $-x^2$. This is clearly unsatisfactory for purposes of program verification. Therefore, our models should have exact functional behaviour.

Finally, we want our semantics to be compositional, that is the behaviour of a sequential composition of programs $p; q$ should be obtainable from the behaviour of p and of q. This is an important condition in the structured development of programs from specifications ([BrGu90] , [EhMa85] , [Win88] et al). In chapter 7, we saw that if we use input measures and exact complexity, it is not in general possible to obtain a compositional semantics. The laxness of the functor $\langle s, n\rangle$ in section 7.3 arises precisely because our constructions are not compositional. In this chapter, we investigate the extent to which we can obtain compositional non-exact complexity.

The approach we take is as follows. We consider models in which a program p of type $A \to B$ denotes a pair $\langle fun(p), [cx(p)]\rangle$ where $fun(p)$ is a morphism from $[\![A]\!]$ to $[\![B]\!]$ and $[cx(p)]$ is an equivalence class of maps from $[\![A]\!]$ to M. The idea is that $fun(p)$ models the exact functional behaviour of p and $[cx(p)]$ models the non-exact complexity of p.

Example 8.1.1 Let $\sim$ be the equivalence relation on functions from $\mathbf{N}$ to $\mathbf{N}$ given by:

$f \sim g$ if $\exists c, C, n_0 \in \mathbf{N} - \{0\}$ such that $\forall\ n \geq n_0,\ f(n) \leq cg(n)$ and $g(n) \leq Cf(n)$

Consider the program "square" of example 7.1.2 with worst case complexity $T(n)$. The order of magnitude non-exact complexity of "square" is the equivalence class $[T]$ under the relation $\sim$.

In section 8.2 we study the relationship between exact and non-exact models. We define a non-exact datum and study the relationship between non-exact data and external data along similar lines to chapter 5.

In section 8.3 we consider the question of compositionality. We show how to construct models for non-exact complexity which satisfy all the above conditions, and describe some examples and applications of this work.

8.2 Non-exact data

8.2.1 Introduction

We define the notion of a non-exact datum and show how to construct a complexity category from a non-exact datum. We construct a category **Nex** of non-exact data and show that the construction of complexity categories yields a functor from **Nex** to **Cat**. We then investigate the relationship between **Ext** and **Nex**.

8.2.2 Non-exact data

Definition 8.2.1 A **non-exact datum** is a tuple $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim \rangle$ such that $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ is an external datum and $\sim$ is an M-equivalence on $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$.

We now define a category **Nex** of non-exact data.

Definition 8.2.2 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim \rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'}, \sim' \rangle$ be non-exact data. A **morphism of non-exact data** is a pair of functors, $\langle F : \mathcal{C} \longrightarrow \mathcal{C}', G : \mathcal{D} \longrightarrow \mathcal{D}' \rangle$ such that:

- $\langle F, G \rangle$ is a morphism of external data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ to $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$
- $G(\sim) \leq \sim'$, that is $\forall A, \forall s, t \in \phi^{\mathcal{C}}(A),\ s \sim_A t \ \Rightarrow\ G(s) \sim_{FA} G(t)$.

Proposition 8.2.3 Non-exact data, morphisms of non-exact data and composition defined by:

$$\langle F_2, G_2 \rangle \langle F_1, G_1 \rangle \ = \ \langle F_2 F_1, G_2 G_1 \rangle$$

determine a category **Nex**.

Proof: Follows from proposition 5.3.8 and the observation that,

$$G(\sim) \leq \sim' \quad \text{and} \quad G'(\sim') \leq \sim'' \quad \text{imply that} \quad G'G(\sim) \leq \sim'' .$$

This completes the proof. □

Non-exact data allow us to define non-exact complexity categories as follows.

Definition 8.2.4 The **non-exact complexity category** of a non-exact datum $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim \rangle$ is the quotient category $\mathcal{C}^{\mathcal{M}} / \sim$.

We can extend the construction of $\mathcal{C}^{\mathcal{M}} / \sim$ to a functor from **Nex** to **Cat** as follows.

Lemma 8.2.5 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim \rangle$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'}, \sim' \rangle$ be non-exact data and let $\langle F, G \rangle$ be a morphism of non-exact data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim \rangle$ to $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'}, \sim' \rangle$. Then the assignment:

$$\langle f, [t] \rangle : A \longrightarrow B \ \longmapsto\ \langle Ff, [Gt] \rangle : FA \longrightarrow FB$$

defines a functor $F \cdot G : \mathcal{C}^{\mathcal{M}} / \sim \ \rightarrow\ \mathcal{C}^{\mathcal{M}'} / \sim'$.

Proof: $F \cdot G$ is well-defined by lemma 5.3.11 and the condition $G(\sim) \leq \sim'$ which ensures that $[Gt]$ is independent of the choice of t. To see that $F \cdot G$ is a functor, we observe that:

- $G([0]) = [G0]$ by definition and 0 is the map:

$$UA \xrightarrow{*} 1 \xrightarrow{0} M,$$

so $G0$ is the map:

$$GUA \xrightarrow{G(*)} G1 \xrightarrow{G(0)} G(M)$$

which is:

$$U'(FA) \xrightarrow{*} 1 \xrightarrow{0} M'$$

since G preserves finite limits and the monoid $\mathcal{M}$.

Thus $F \cdot G$ preserves identities. Finally,

- $$\begin{aligned} \langle Fg, [Gs]\rangle\langle Ff, [Gt]\rangle &= \langle FgFf, [Gt \cdot GsU'Ff]\rangle \\ &= \langle F(gf), [Gt \cdot GsGUf]\rangle \\ &= \langle F(gf), [G(t \cdot sUf)]\rangle \\ &= F \cdot G(\langle gf, [t \cdot sUf\rangle]) \\ &= F \cdot G(\langle g, [s]\rangle\langle f, [t]\rangle). \end{aligned}$$

so $F \cdot G$ preserves composition.
This completes the proof. □

Proposition 8.2.6 The assignment:

- $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim\rangle \longmapsto \mathcal{C}^{\mathcal{M}}/\sim$ and
- $\langle F, G\rangle \longmapsto F \cdot G$

defines a functor $\mathcal{N}$: **Nex**→**Cat**.

Proof: $\mathcal{N}$ is well-defined by lemma 8.2.5. It is clear that $\mathcal{N}$ preserves identities. It remains to verify that $\mathcal{N}$ respects composition.

- $\mathcal{N}(\langle F_1F_0, G_1G_0\rangle) = (F_1F_0)\cdot(G_1G_0)$
  $= (F_1 \cdot G_1)(F_0 \cdot G_0)$
  $= \mathcal{N}(\langle F_1, G_1\rangle)\mathcal{N}(\langle F_0, G_0\rangle).$

This completes the proof. □

8.2.3 Relationship between exact and non-exact

In this section, we study the relationship between exact and non-exact data. We show that **Ext** is a full coreflective subcategory of **Nex**.

Proposition 8.2.7 The assignment:

- $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle \longmapsto \langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, id\rangle$ and
- $\langle F, G\rangle \longmapsto \langle F, G\rangle$

defines a functor nex : **Ext**→**Nex**.

Proof: nex is well-defined since the identity relation is an M-equivalence (*c.f.* example 7.4.7) and it is clear that $G(id) \leq id$.
It is also clear that nex preserves identities and composition. □

Proposition 8.2.8 The assignment:

- $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim\rangle \longmapsto \langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$ and
- $\langle F, G\rangle \longmapsto \langle F, G\rangle$

defines a functor U: **Nex**→**Ext**.

Proof: Clear. □

Theorem 8.2.9 We have an adjunction $nex \dashv U$: **Nex**→**Ext**.

Proof: Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle \in |\mathbf{Ext}|$ and $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'}, \sim' \rangle \in |\mathbf{Nex}|$. We need to show that there is a natural bijection:
$\mathbf{Nex}(nex(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle), \langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim' \rangle) \cong$
$\mathbf{Ext}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, U(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim' \rangle))$.
Now $\mathbf{Nex}(nex(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle), \langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim' \rangle) =$
$\mathbf{Nex}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, id \rangle, \langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'}, \sim' \rangle)$
an element of which is a pair of functors $\langle F, G \rangle$ such that:

- $\langle F, G \rangle$ is a morphism of external data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ to $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$ and

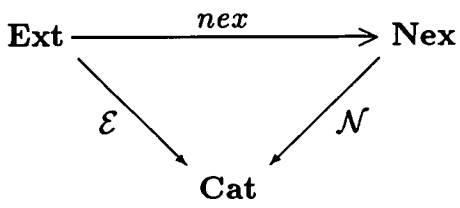
- $G(id) \leq \sim'$.

However, $G(id) \leq \sim'$ if for all t, $Gt \sim' Gt$ which is clearly satisfied since $\sim'$ is an equivalence relation.
Thus an element of $\mathbf{Nex}(nex(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle), \langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim' \rangle)$ is simply a morphism of external data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ to $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$.
Now $\mathbf{Ext}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, U(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim' \rangle)) =$
$\mathbf{Ext}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle, \langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle)$
an element of which is a morphism of external data from $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ to $\langle \mathcal{C}', \mathcal{D}', U', \mathcal{M}', \phi^{\mathcal{C}'} \rangle$ and so we have the required bijection.
Naturality is evident and this completes the proof. □

Corollary 8.2.10 **Ext** is a full coreflective subcategory of **Nex**.

Proof: Trivially, nex is injective on objects. It is clear from the proof of theorem 8.2.9 that nex is fully faithful since $G(id) \leq id$. Finally, nex has a right adjoint by theorem 8.2.9 and the result follows. □

Theorem 8.2.11 The diagram:



commutes up to natural isomorphism.

Proof: We require a natural isomorphism between $\mathcal{E}$ and $\mathcal{N}nex$ in the functor category **[Ext, Cat]**.
Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle \in |\mathbf{Ext}|$. Then $\mathcal{E}(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle) = \mathcal{C}^{\mathcal{M}}$.
$nex(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle)$ is $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, id \rangle$ and therefore,
$\mathcal{N}nex(\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle) = \mathcal{C}^{\mathcal{M}}/id$.
However, by example 7.4.16, the quotient functor $q : \mathcal{C}^{\mathcal{M}} \to \mathcal{C}^{\mathcal{M}}/id$ is an isomorphism of categories as required.
The naturality of q is evident and the result follows. □

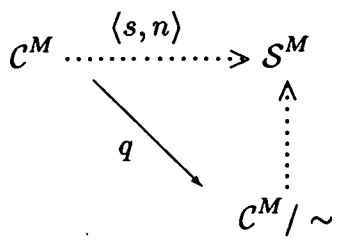
8.3 Non-exact models

8.3.1 Introduction

In chapter 7, we defined a lax measure. We showed that a lax measure allows us to construct an ordered category $\mathcal{S}^{\mathcal{M}}$ and a lax functor $\langle s, n \rangle : \mathcal{C}^{\mathcal{M}} \to \mathcal{S}^{\mathcal{M}}$.

We explained that $\mathcal{S}^{\mathcal{M}}$ is an unsatisfactory model for two reasons. It does not have the same functional behaviour as $\mathcal{C}^{\mathcal{M}}$ (definition 7.4.14) and the laxness of the functor $\langle s, n \rangle$ means that it does not give a compositional semantics. In this section, we overcome these difficulties.

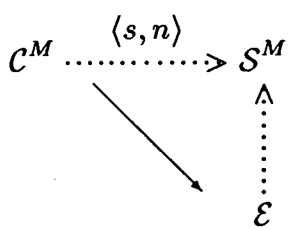
The approach we take is as follows. We seek to construct an M-equivalence $\sim$ on $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ such that the lax functor $\langle s, n \rangle : \mathcal{C}^M \to \mathcal{S}^M$ can be factored as:



where the dotted lines are lax functors.

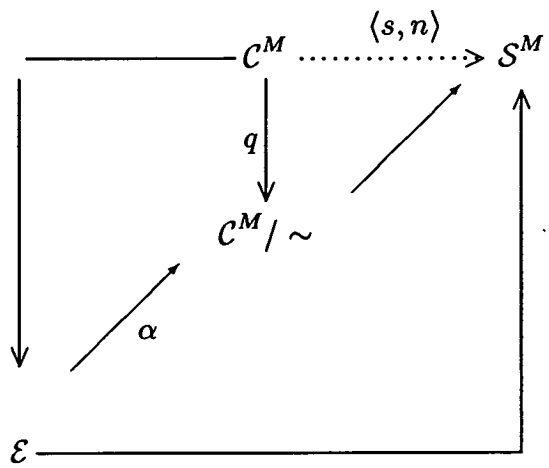
The idea is that complexity maps $s : UA \longrightarrow M$ and $t : UA \longrightarrow M$ should be identified in $\mathcal{C}^M / \sim$ whenever $\langle s, n \rangle$ identifies them in $\mathcal{S}^M$. The results of section 7.4 show that $\mathcal{C}^M / \sim$ will have the same functional behaviour as $\mathcal{C}^M$ and that it will give a compositional semantics.

We wish to identify as many maps as is consistent with having a compositional semantics, when they are identified in $\mathcal{S}^M$. Accordingly, we would like to require that $\mathcal{C}^M / \sim$ be universal amongst all such categories. That is, given a category $\mathcal{E}$ having the same functional behaviour as $\mathcal{C}^M$ and such that the following diagram:

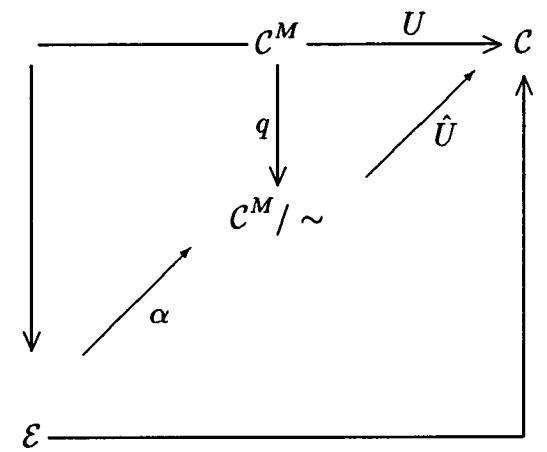


commutes, then there exists a unique functor α from $\mathcal{E}$ to $\mathcal{C}^M / \sim$ such that the

following diagrams:



and



commute. The idea is that $\mathcal{C}^M/\sim$ corresponds to the "compositional part" of $\mathcal{S}^M$.

In fact, this approach does not quite capture all the examples of interest to us. More generally, we want to consider a collection of equivalence relations $\approx$ on the morphisms of $\mathcal{S}^M$ and, as far as is consistent with maintaining compositionality, identify two maps $s : UA \longrightarrow M$ and $t : UA \longrightarrow M$ whenever they are equivalent in $\mathcal{S}^M$.

In this section we follow the above approach, leading to the definition of a non-exact measure. Given a non-exact measure for an external datum $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$, we construct an M-equivalence $\sim$ such that $\mathcal{C}^M/\sim$ satisfies all of our conditions.

8.3.2 Non-exact measures

Definition 8.3.1 A **non-exact measure** for an external ordered datum $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ is a tuple $\langle S, V, s, n, \approx \rangle$ where $\langle S, V, s, n \rangle$ is a lax measure for $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and $\approx$ is a collection of equivalence relations $\approx_A$ such that:

- $\approx_A$ is an equivalence relation on $\mathcal{D}(V(s(A)), M)$,
- $s_1 \approx s_2 \Rightarrow t \cdot s_1 \approx t \cdot s_2$ and $s_1 \cdot t \approx s_2 \cdot t$ and
- $n(s \cdot t) \approx n(s) \cdot n(t)$.

Example 8.3.2 Let $\langle S, V, s, n \rangle$ be a lax measure for an external ordered datum $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$. Then $\langle S, V, s, n, id \rangle$ is a non-exact measure for $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$.

Example 8.3.3 Let $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum and let $\langle \mathbf{N}, \iota, s, n \rangle$ be a lax measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$. Let $\approx$ be the equivalence relation on $\mathbf{Set}(\mathbf{N}, \mathbf{N})$ given by:

$$f \approx g \quad \text{if} \quad f = O(g) \ \text{ and } \ g = O(f).$$

Then $\langle \mathbf{N}, \iota, s, n, \approx \rangle$ is a non-exact measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$.

Of course, $\approx$ is not in general an M-equivalence on $\langle \mathcal{S}_o, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{S}} \rangle$ and therefore we cannot define a category $\mathcal{S}^M / \approx$. However, it is convenient to define a graph $\mathcal{S}^M / \approx$.

Notation 8.3.4 In the remainder of this section, we shall assume that $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ is an external ordered datum and $\langle S, V, s, n, \approx \rangle$ is a non-exact measure for $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$.

Lemma 8.3.5 The following data:

- objects of $\mathcal{S}$ and
- equivalence classes of morphisms in $\mathcal{S}^M$ under $\approx$

defines a graph $\mathcal{S}^M/\approx$.

We now construct a non-exact datum from a non-exact measure. We use the idea that complexity maps should be identified when their images in $\mathcal{S}^M$ are equivalent.

The natural approach is to define $s \sim t$ if $n(s) \approx n(t)$ in $\mathcal{C}_S^M$. Unfortunately, this does not necessarily define an M-equivalence.

Proposition 8.3.6 The collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \cong t \text{ if } n(s) \approx n(t)$$

defines an M-equivalence on $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ if and only if:

$$n(s) \approx n(t) \text{ implies that } n(sUf) \approx n(tUf) \text{ for all } f.$$

Proof: It is clear that $\cong_A$ is an equivalence relation for each A.
Suppose that $s_1 \cong s_2$. Then $n(s_1) \approx n(s_2)$ so:

$$n(s_1 \cdot t) \approx n(s_1) \cdot n(t) \approx n(s_2) \cdot n(t) \approx n(s_2 \cdot t)$$

and so $s_1 \cdot t \cong s_2 \cdot t$. Similarly, $t \cdot s_1 \cong t \cdot s_2$.
Finally, suppose that $s_1 \cong s_2$. Then $n(s_1) \approx n(s_2)$ and $n(s_1 Uf) \approx n(s_2 Uf)$, by assumption, and thus $s_1 Uf \cong s_2 Uf$. Therefore, $\cong$ is an M-equivalence.
Conversely, suppose that $\cong$ is an M-equivalence. Then:

$$s_1 \cong s_2 \Rightarrow s_1 Uf \cong s_2 Uf$$

which is precisely the condition:

$$n(s) \approx n(t) \Rightarrow n(sUf) \approx n(tUf) \text{ for all } f.$$

This completes the proof. □
This candidate is therefore unsatisfactory and we are led to the following definition.

Proposition 8.3.7 The collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \sim t \quad \text{if} \quad \forall f : B \longrightarrow A,\ n(sUf) \approx n(tUf)$$

defines an M-equivalence on $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$.

Proof: We first show that $\sim$ gives an equivalence relation on each $\phi^{\mathcal{C}}(A)$.

- $n(tUf) \approx n(tUf)\ \forall f$ since $tUf = tUf$ and so $t \sim t$.

- $s \sim t \Rightarrow \forall f.\ n(sUf) \approx n(tUf)$
  $\Rightarrow \forall f.\ n(tUf) \approx n(sUf)$
  $\Rightarrow t \sim s.$

- $r \sim s \sim t \Rightarrow \forall f.\ n(rUf) \approx n(sUf)$ and $\forall g\ n(sUg) \approx n(tUg)$
  $\Rightarrow \forall f.\ n(rUf) \approx n(sUf)$ and $n(sUf) \approx n(tUf)$
  $\Rightarrow \forall f.\ n(rUf) \approx n(tUf)$
  $\Rightarrow r \sim t.$

We now show that $s_1 \sim s_2$ implies that $\forall t.\ t \cdot s_1 \sim t \cdot s_2$ and $s_1 \cdot t \sim s_2 \cdot t$.

- $s_1 \sim s_2 \Rightarrow \forall f.\ n(s_1Uf) \approx n(s_2Uf)$
  $\Rightarrow \forall f.\ n(tUf) \cdot n(s_1Uf) \approx n(tUf) \cdot n(s_2Uf)$
  $\Rightarrow \forall f.\ n(tUf \cdot s_1Uf) \approx n(tUf \cdot s_2Uf)$
  $\Rightarrow \forall f.\ n((t \cdot s_1)Uf) \approx n((t \cdot s_2)Uf)$
  $\Rightarrow t \cdot s_1 \sim t \cdot s_2.$

- $s_1 \sim s_2 \Rightarrow \forall f.\ n(s_1Uf) \approx n(s_2Uf)$
  $\Rightarrow \forall f.\ n(s_1Uf) \cdot n(tUf) \approx n(s_2Uf) \cdot n(tUf)$
  $\Rightarrow \forall f.\ n(s_1Uf \cdot tUf) \approx n(s_2Uf \cdot tUf)$
  $\Rightarrow \forall f.\ n((s_1 \cdot t)Uf) \approx n((s_2 \cdot t)Uf)$
  $\Rightarrow s_1 \cdot t \sim s_2 \cdot t.$

Finally, we show $s \sim t$ implies that $sUg \sim tUg$.

- $s \sim t \Rightarrow \forall f.\ n(sUf) \approx n(tUf)$
  $\Rightarrow$ in particular, $\forall f.\ n(sU(fg)) \approx n(tU(fg))$
  $\Rightarrow \forall f.\ n((sUg)Uf) \approx n((tUg)Uf)$
  $\Rightarrow sUg \sim tUg.$

This completes the proof. □

Corollary 8.3.8 Let $\sim$ be the collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \sim t \quad \text{if} \quad \forall f : B \longrightarrow A,\ n(sUf) \approx n(tUf).$$

Then $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}}, \sim \rangle$ is a non-exact datum.

Corollary 8.3.9 Let $\sim$ be the collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \sim t \quad \text{if} \quad \forall f : B \longrightarrow A,\ n(sUf) \approx n(tUf).$$

Then $\sim$ is a congruence on $\mathcal{C}^M$ and $\mathcal{C}^M/\sim$ has the functional behaviour of $\mathcal{C}^M$.

Proof: Follows by proposition 7.4.15 and corollary 8.3.8. □

The next result shows that $\sim$ can be characterised as the maximum M-equivalence contained in $\cong$.

Notation 8.3.10 Let $\mathcal{C}$ be a category with finite products, let I be a set and let R and S be I-indexed collections of relations in $\mathcal{C}$. We say $R \leq S$ if $R_i \subseteq S_i$ for each i in I.

Proposition 8.3.11 Let $\sim$ be the collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \sim t \quad \text{if} \quad \forall f : B \longrightarrow A,\ n(sUf) \approx n(tUf)$$

and let $\cong$ be the collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \cong t \text{ if } n(s) \approx n(t).$$

Then $\sim \leq \cong$ and for all M-equivalences $\equiv$, if $\equiv \leq \cong$ then $\equiv \leq \sim$.

Proof: Note first that $\sim \leq \approx$ since:

- $s \sim t \Rightarrow \forall f.\ n(sUf) \approx n(tUf)$
  $\Rightarrow n(sid) \approx n(tid)$
  $\Rightarrow n(s) \approx n(t)$
  $\Rightarrow s \cong t.$

Now suppose that $\equiv$ is another M-equivalence and that $\equiv \leq \cong$ then:

- $s \equiv t \Rightarrow \forall f.\ sUf \equiv tUf$
  $\Rightarrow \forall f.\ sUf \cong tUf$
  $\Rightarrow \forall f.\ n(sUf) \approx n(tUf)$
  $\Rightarrow s \sim t.$

and so $\equiv \leq \sim$ and this completes the proof. □

8.3.3 Constructing a non-exact model

Proposition 8.3.12 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum, let $\langle S, V, s, n, \approx \rangle$ be a non-exact measure for $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and let $\sim$ be the collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \sim t \quad \text{if} \quad \forall f : B \longrightarrow A.\ n(sUf) \approx n(tUf).$$

Then the assignment:

- $A \longmapsto s(A)$ and
- $[\langle f, t \rangle] \longmapsto [\langle s(f), n(t) \rangle]$

defines a graph morphism $T : \mathcal{C}^M / \sim \ \rightarrow \mathcal{S}^M / \approx$.

Proof: We show that T is well-defined.
Suppose that $\langle g, s \rangle$ were another representative of $[\langle f, t \rangle]$. Then:

- $\langle f, t \rangle \sim \langle g, s \rangle \Rightarrow f = g$ and $t \sim s$
  $\Rightarrow f = g$ and $\forall f.\ n(tUf) \approx n(sUf)$
  $\Rightarrow f = g$ and $n(tid) \approx n(sid)$
  $\Rightarrow f = g$ and $n(t) \approx n(s)$
  $\Rightarrow T([\langle f, t \rangle]) = T([\langle g, s \rangle])$

and so T is well-defined.

This completes the proof. □

In the particular case where $\approx$ is the identity relation, this graph morphism is a lax functor:

Corollary 8.3.13 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum, let $\langle S, V, s, n \rangle$ be a lax measure for $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and let $\sim$ be the collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \sim t \quad \text{if} \quad \forall f : B \longrightarrow A.\ n(sUf) = n(tUf).$$

Then the assignment:

- $A \longmapsto s(A)$ and
- $[\langle f, t \rangle] \longmapsto \langle s(f), n(t) \rangle$

defines a lax functor $T : \mathcal{C}^M / \sim \longrightarrow \mathcal{S}^M$.

Proof: T is well-defined by proposition 8.3.12.

To show that T is a lax functor we have:

- $$\begin{aligned} T([\langle fg, t \cdot sUf \rangle]) &= \langle s(fg), n(t \cdot sUf) \rangle \\ &= \langle s(fg), n(t) \cdot n(sUf) \rangle \\ &\leq \langle s(f)s(g), n(t) \cdot n(sUf) \rangle \\ &\leq \langle s(f)s(g), n(t) \cdot s(f)n(s) \rangle \\ &= \langle s(f), n(t) \rangle \langle s(g), n(s) \rangle \\ &= T([\langle f, t \rangle])T([\langle g, s \rangle]). \end{aligned}$$
- $$\begin{aligned} T([\langle id, 0 \rangle]) &= \langle s(id), n(0) \rangle \\ &= \langle s(id), 0 \rangle \\ &\leq \langle id, 0 \rangle. \end{aligned}$$

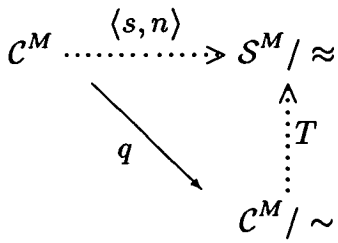
This completes the proof. □

Remark 8.3.14 *There is another natural candidate for $\sim$. This is the smallest M-equivalence containing $\approx$. This does exist, but it would not work here since $s \sim t$ would not imply $n(s) \approx n(t)$ and thus T would not be well-defined.*

Proposition 8.3.15 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum, let $\langle S, V, s, n, \approx \rangle$ be a non-exact measure for $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and let $\sim$ be the collection of relations on $\phi^{\mathcal{C}}$ given by:

$$s \sim t \quad \text{if} \quad \forall f : B \longrightarrow A,\ n(sUf) \approx n(tUf).$$

Then the following diagram:



commutes, where the dotted arrows are graph morphisms.

Proof: Let $\langle f, t \rangle \in \mathcal{C}^M$ then:

- $T(q(\langle f, t \rangle)) = T([\langle f, t \rangle])$
  $= [\langle s(f), n(t) \rangle]$
  $= [\langle s, n \rangle (\langle f, t \rangle)]$.

Thus $qT = \langle s, n \rangle$ as required. □

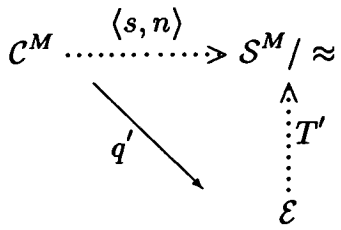
8.3.4 Universal property

We now show that $\mathcal{C}^M / \sim$ has the required universal property.

Theorem 8.3.16 Let $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ be an external ordered datum, let $\langle S, V, s, n, \approx \rangle$ be a non-exact measure for $\langle \mathcal{C}, \mathcal{D}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$ and let $\sim$ be the collection of relations on $\phi^{\mathcal{C}}$ given by:

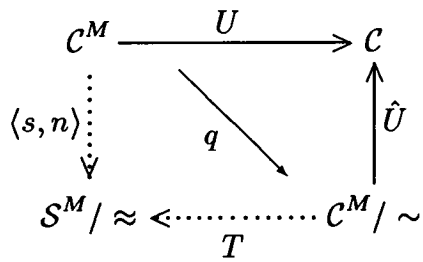
$$s \sim t \quad \text{if} \quad \forall f : B \longrightarrow A,\ n(sUf) \approx n(tUf).$$

Then $\mathcal{C}^M/\sim$ is universal amongst all categories $\mathcal{E}$ with the functional behaviour of $\mathcal{C}^M$ and a functor $q' : \mathcal{C}^M \to \mathcal{E}$ which is surjective on morphisms and such that the following diagram:



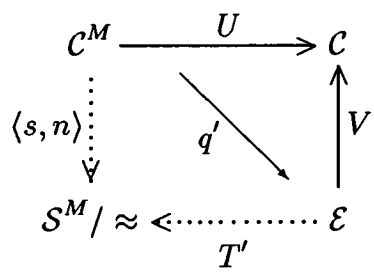
commutes.

Proof: By corollary 8.3.8, proposition 8.3.12 and proposition 8.3.15, we have established that $\mathcal{C}^M/\sim$ is a cone. That is, that the following diagram:

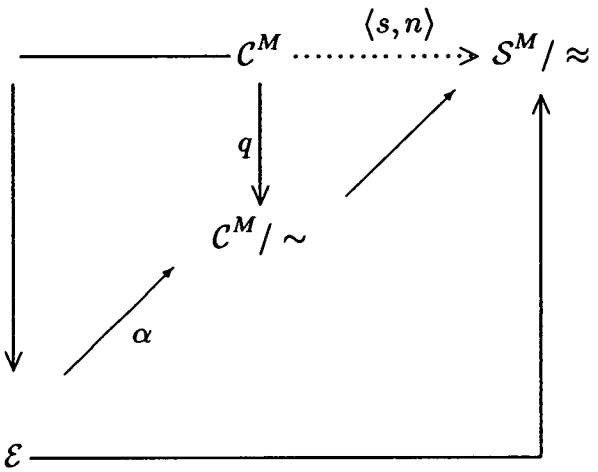


commutes.

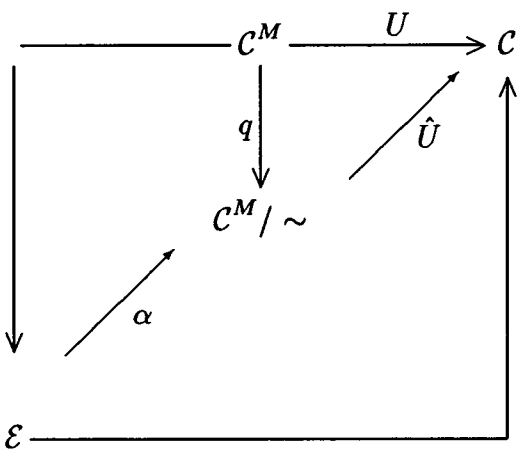
Now, given functors $q' : \mathcal{C}^M \longrightarrow \mathcal{E}$, $T' : \mathcal{E} \longrightarrow \mathcal{S}^M$ and $V : \mathcal{E} \longrightarrow \mathcal{C}$ such that q' is surjective on morphisms and the following diagram:



commutes. Then $\mathcal{C}^M/\sim$ is universal if there exists a unique functor $\alpha : \mathcal{E} \to \mathcal{C}^M/\sim$ such that the following diagrams:



and



commute.

Let $g : d \longrightarrow d'$ be a morphism in $\mathcal{E}$. By assumption, the functor $q' : \mathcal{C}^M \longrightarrow \mathcal{E}$ is surjective on morphisms.

Thus:

$$\exists \langle f, t \rangle \in \mathcal{C}^M \text{ such that } g = q'(\langle f, t \rangle)$$

and:

- $T'(g) = T'(q'(\langle f,t\rangle))$
  $= \langle s,n\rangle(\langle f,t\rangle)$
  $= \langle s(f),n(t)\rangle.$

We now define the functor $\alpha : \mathcal{E} \longrightarrow \mathcal{C}^M/\sim$. Let $g : d \longrightarrow d'$ be a morphism in $\mathcal{E}$, so that $g = q'(\langle f,t\rangle)$ for some $\langle f,t\rangle$ in $\mathcal{C}^M$. Then define:

$$\alpha(g : d \longrightarrow d') = [\langle V(g),t\rangle].$$

We need to show that α is well-defined, that it is a functor, that it makes all the required diagrams commute and that it is unique up to unique isomorphism.
We show that α is well-defined.
Suppose that $\langle f',t'\rangle$ were another choice, *i.e.* $q'(\langle f',t'\rangle) = g$. We need to show that $[\langle V(g),t\rangle] = [\langle V(g),t'\rangle]$.
We have:

- $T'(q'(\langle f',t'\rangle)) = T'(g)$
  $= T'(q'(\langle f,t\rangle))$
  $\Rightarrow \langle s(f'),n(t')\rangle \approx \langle s(f),n(t)\rangle$
  $\Rightarrow n(t') \approx n(t).$

This does not yet suffice since:

$$[\langle V(g),t\rangle] = [\langle V(g),t'\rangle] \text{ if } \forall h.\ n(ht') \approx n(ht).$$

However, consider that $\langle h,0\rangle\langle f,t\rangle = \langle hf,0\cdot ht\rangle$ and that $\langle h,0\rangle\langle f',t'\rangle = \langle hf',0\cdot ht'\rangle$.
Now q' is functorial so:

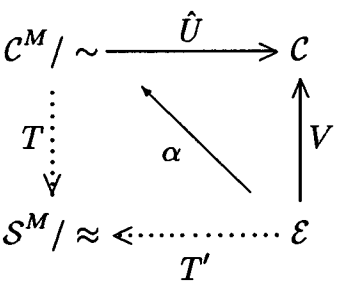
- $q'(\langle hf,ht\rangle) = q'(\langle hf,0\cdot ht\rangle)$
  $= q'(\langle h,0\rangle\langle f,t\rangle)$
  $= q'(\langle h,0\rangle)q'(\langle f,t\rangle)$
  $= q'(\langle h,0\rangle)g$ and
- $q'(\langle hf',ht'\rangle) = q'(\langle hf',0\cdot h't\rangle)$
  $= q'(\langle h,0\rangle\langle f',t'\rangle)$
  $= q'(\langle h,0\rangle)q'(\langle f',t'\rangle)$
  $= q'(\langle h,0\rangle)g.$

Therefore:

- $q'(\langle hf, ht\rangle) = q'(\langle hf', ht'\rangle) \Rightarrow T'(q'(\langle hf, ht\rangle)) = T'(q'(\langle hf', ht'\rangle))$
  $\Rightarrow \langle s, n\rangle(\langle hf, ht\rangle) \approx \langle s, n\rangle(\langle hf', ht'\rangle)$
  $\Rightarrow \langle s(hf), n(ht)\rangle \approx \langle s(hf'), n(ht')\rangle$
  $\Rightarrow n(ht) \approx n(ht')$
  $\Rightarrow \forall h.\, n(ht) \approx n(ht')$
  $\Rightarrow [\langle V(g), t\rangle] = [\langle V(g), t'\rangle]$

and thus α is well-defined.

We now show that the following diagram:



commutes. We have:

- $\hat{U}(\alpha(g)) = \hat{U}([\langle V(g), t\rangle])$
  $= V(g)$
  $\Rightarrow \hat{U}\alpha = V.$

and:

- $T'(g) = T'(q'(\langle f, t\rangle))$ some $\langle f, t\rangle$
  $= \langle s, n\rangle(\langle f, t\rangle)$
  $= \langle s(f), n(t)\rangle$
  $= T([\langle f, t\rangle])$
  $= T([\langle V(q'(\langle f, t\rangle)), t\rangle])$
  $= T([\langle V(g), t\rangle])$
  $= T(\alpha(g))$

and thus it commutes.

We now show that α is a functor.

Suppose that $g = q'(\langle f, t\rangle)$ and $g' = q'(\langle f', t'\rangle)$ are morphisms in $\mathcal{E}$. Then:

- $\alpha(gg') = \alpha(q'(\langle f,t\rangle)q'(\langle f',t'\rangle))$
  $= \alpha(q'(\langle f,t\rangle\langle f',t'\rangle))$
  $= \alpha(q'(\langle ff',t\cdot tUf'\rangle))$
  $= \alpha(q'(\langle ff',t\cdot tUf'\rangle))$
  $= [\langle ff',t\cdot tUf'\rangle]$
  $= [\langle f,t\rangle][\langle f',t'\rangle]$
  $= \alpha(q'(\langle f,t\rangle))\alpha(q'(\langle f',t'\rangle))$
  $= \alpha(g)\alpha(g')$

and thus α is functorial.

It remains to show that α is unique up to unique isomorphism.

Suppose that $\beta : \mathcal{E} \longrightarrow \mathcal{C}^M/\sim$ is a functor making all the above diagrams commute.

Then we have the following commuting diagram:

$$\mathcal{C}^M \xrightarrow{q'} \mathcal{E} \underset{\beta}{\overset{\alpha}{\underset{\times}{\rightrightarrows}}} \mathcal{C}^M/\sim$$

Now, by assumption, the following diagram:

$$\begin{array}{ccc} \mathcal{C}^M & \xrightarrow{q'} & \mathcal{E} \\ & \searrow^{q} & \downarrow \alpha \times \beta \\ & & \mathcal{C}^M/\sim \end{array}$$

commutes. However, $q' : \mathcal{C}^M \longrightarrow \mathcal{E}$ is surjective on morphisms and so:

$$\mathcal{C}^M \xrightarrow{q'} \mathcal{E} \underset{\beta}{\overset{\alpha}{\rightrightarrows}} \mathcal{C}^M/\sim .$$

Thus α is unique as required.

This completes the proof □

Remark 8.3.17 *In the case where $\approx$ is the identity relation, the map $\langle s,n\rangle$: $\mathcal{C}^M \longrightarrow \mathcal{S}^M/id$ is a lax functor of ordered categories. Theorem 8.3.16 specialises*

to the case of maximally factorising a lax functor $\langle s, n\rangle : \mathcal{C}^{\mathcal{M}} \longrightarrow \mathcal{S}^{M}/id$ *as a strict functor* $q : \mathcal{C}^{\mathcal{M}} \longrightarrow \mathcal{C}^{\mathcal{M}}/\sim$ *followed by a lax functor* $\mathcal{C}^{\mathcal{M}}/\sim \longrightarrow \mathcal{S}^{M}/id$. *This is interesting since, in general, it is not possible to give a maximal factorisation of a lax functor as a strict functor followed by a lax functor.*

8.3.5 Worst case order of magnitude complexity

We show how to express one of the principal examples of complexity theory.

Lemma 8.3.18 Let $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$ be an external ordered datum, let $\langle \mathbf{N}, \iota, s, n\rangle$ be a lax measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$ and let $\approx$ be the collection of relations on $\mathbf{Set}(\mathbf{N}, \mathbf{N})$ given by:

$f \approx g$ if $\exists c, C, n_0 \in \mathbf{N} - \{0\}$ such that $\forall\ n \geq n_0,\ f(n) \leq cg(n)$ and $g(n) \leq Cf(n)$

Then $\langle \mathbf{N}, \iota, s, n, \approx\rangle$ is a non-exact measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}}\rangle$.

We recall example 7.2.18. We show how this non-example from chapter 7 becomes an example here.

Example 8.3.19 Consider the following model for Turing machine computations. Let Σ be the one object category with object $(0,1)^*$ and morphisms all computable functions. The idea is that a Turing machine program which takes an input σ and terminates with output $p(\sigma)$, will denote a function $p : \Sigma \longrightarrow \Sigma$.
Let $\mathcal{M}$ be the ordered monoid $\langle \mathbf{N}, 0, +, \leq\rangle$ in **Set** , let $s(\Sigma) = \mathbf{N}$, let $\mu : \Sigma \to \mathbf{N}$ be given by $s(\sigma) = length(\sigma)$, let $\Omega(N) = max\{n \in N\}$ and let n be the collection of functions given by the following commuting diagram:

$$\begin{array}{ccc} s(A) & \xrightarrow{n_A(t)} & M \\ {\scriptstyle \mu_A^{-1}}\downarrow & & \uparrow{\scriptstyle \Omega} \\ P_f(UA) & \xrightarrow[P_f(t)]{} & P_f(M) \end{array}$$

Then $\langle s, \mu \rangle$ does not satisfy the conditions of corollary 7.2.15 and hence $\mathcal{C}_S^M$ is not a category. However, let $\approx$ be the collection of relations on $\mathbf{Set}(\mathbf{N}, \mathbf{N})$ given by:

$f \approx g$ if $\exists c, C, n_0 \in \mathbf{N} - \{0\}$ such that $\forall\ n \geq n_0,\ f(n) \leq cg(n)$ and $g(n) \leq Cf(n)$

Then by lemma 8.3.18, $\langle \mathbf{N}, \iota, s, n, \approx \rangle$ is a non-exact measure for $\langle \mathcal{C}, \mathbf{Set}, U, \mathcal{M}, \phi^{\mathcal{C}} \rangle$. Therefore, by theorem 8.3.16, we can construct a category $\Sigma^N / \sim$ in which $\langle f, t \rangle \sim \langle g, s \rangle$ if $f = g$ and $n(t) \approx n(s)$.
Maps in $\Sigma^N / \sim$ are pairs:

$$f : \Sigma \to \Sigma \quad \text{and} \quad [t : \Sigma \to \mathbf{N}]$$

where $[t] = \{s | n(t) \approx n(s)\}$. In particular, if $n(t) = cn^{\alpha}$ and $n(s) = c'n^{\alpha'}$, then $s \in [t]$ if and only if $\alpha = \alpha'$.
Thus we have a model in which we can represent Turing machine input-output behaviour together with worst case order of magnitude complexity.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, we have extended denotational semantics in order to model the resource requirements of programs, as well as their functional behaviour.

In chapter 3, we have established a connection between this work and Moggi's categorical semantics of computations. We have used this connection to develop a formal system for reasoning about the resource requirements of programs.

In chapter 4, we have shown how to regard complexity as one of Moggi's monad constructors. We have succeeded in carrying out his program for a modular approach to denotational semantics for the monad constructor for complexity.

In chapter 5, we have argued that Moggi's framework is not sufficiently general to capture all the examples of interest to us. We have provided an alternative framework, based on the notion of an external datum, and have provided a precise characterisation of the relationship between the two approaches.

In chapters 6, 7 and 8, we have shown that many concepts of importance in complexity theory and the analysis of algorithms can be captured within our framework. In particular, we have shown how to capture the notions of upper bounds on complexity, input measures and non-exact complexity. Finally, we have

made substantial progress towards providing a compositional theory of non-exact complexity.

9.2 Unsolved Problems

There are a number of questions that we have failed to find satisfactory answers for.

- In chapter 3, we have not provided a proof of conjecture 3.6.13.

- In chapter 3, we showed that if $\Gamma \vdash_{\lambda_{com}} \langle e, v, t \rangle : \tau$ then $\Gamma \vdash_{\lambda_c + Ax} e \equiv t \cdot v : \tau$ and $\Gamma \vdash_{\lambda_c + Ax} v \downarrow \tau$. However, we have failed to prove the converse. In particular, it is not clear to us what rules are needed to be added to $\lambda_c + Ax$ in order for it to be complete with respect to interpretation in complexity models.

- In chapter 4, we stated that most of the examples of primary interest to us are commutative, but that the commutativity is usually not required. It would be nice to find a natural example of a non-commutative monoid. Otherwise, it may be more sensible to require that our monoid of resource values is always commutative.

- In chapter 5, we have not studied lazy languages [BjHo89].

9.3 Future Work

We describe some promising areas of future work.

9.3.1 The complexity of higher order functions

In this thesis, we have not really dwelt on the difficulties in modelling the complexity of higher order functions. However, this is clearly central to the problem of modelling the complexity of functional languages such as ML.

There are two issues involved, the exact complexity of higher order functions and the non-exact complexity of higher order functions. Even for exact complexity, there appear to be substantial difficulties. For example, consider the following ML program:

```
- fun apply f = f(2) : int;
> val apply = fn : (int -> int) -> int
```

We observe that the time taken to evaluate the application of `apply` to a program p of type `int -> int` will depend on the complexity of p and not just its functional behaviour. Therefore, the interpretation of the type `int -> int` needs to code up the complexity of programs of type `int -> int`. In the internal complexity categories, the solution is to interpret the type $A \longrightarrow B$ by the object $(M \times [\![B]\!])^{[\![A]\!]}$ of $\mathcal{C}$. Thus, the interpretation of a program such as `apply` will be a morphism from $(M \times [\![\texttt{int}]\!])^{[\![\texttt{int}]\!]}$ to $M \times [\![\texttt{int}]\!]$ in $\mathcal{C}$, rather than a morphism from $[\![\texttt{int}]\!]^{[\![\texttt{int}]\!]}$ to $M \times [\![\texttt{int}]\!]$. In the external complexity categories, we need to construct a suitable analogy of the object $(M \times B)^A$. However, we are hopeful that this will not prove impossible.

An even more interesting and difficult problem is modelling the non-exact complexity of higher order functions. In particular, the problem of defining input measures for functional types. As an example, consider the following ML program:

```
- fun nasty f = 3*f(2);
> val nasty = fn : (int -> int) -> int
```

As we mentioned in chapter 7, the basic idea of input measures is that it takes equivalent resource to apply a program to two different input values of the same size. However, the resource required to compute **nasty** on an input p is the resource required to compute p on the value 2, together with the resource required to compute $*$ on $\langle 3, p(2)\rangle$. Thus, it is clear that the resource required to compute **nasty** depends on both the complexity and the functional behaviour of the program p. Therefore, the size of an input p should be a measure of both the complexity of p, which is a morphism from $[\![\,\texttt{int}\,]\!]$ to M, and its functional behaviour, which is a morphism from $[\![\,\texttt{int}\,]\!]$ to $[\![\,\texttt{int}\,]\!]$. Thus, it is clear that we cannot simply take the natural numbers as the size object of the type `int -> int`.

A possible approach is to define size objects for each base type, and then to define the size of a functional type recursively by the following rule:

$$s(A \longrightarrow B) \;=\; s(A) \longrightarrow M \times s(B).$$

Thus, for example:

$$s(\texttt{int} \; -> (\texttt{int} \; -> \texttt{int})) \;=\; s(\texttt{int}) \longrightarrow M \times (s(\texttt{int}) \longrightarrow M \times s(\texttt{int})).$$

However, there are a number of difficulties, for example, it is not clear how to extend the finiteness conditions in the definition of an analysis. It should be interesting to see how this approach relates to the work of Cook and Kapron [CoKa90] as they do not appear to require this degree of complexity in their approach.

9.3.2 Timing concurrent systems

This thesis has developed semantic frameworks for modelling the complexity of sequential programming languages. In this section, we describe some work in progress on modelling the complexity of concurrent systems.

The model we have been examining is Petri nets [Rei85], first developed by Petri in 1966 and subsequently the subject of a great deal of study. Recently, several authors [BrGu90], [MeMo88], [MOMe89], [Win88] have constructed categories

of Petri nets in which the morphisms represent, variously, simulation, implementation, refinement or interpretation. Some of these authors [BrGu90], [MOMe89], [Win88] have studied the issue of modularity in concurrent systems by looking at the structure of these categories.

We have taken the following approach to timing Petri nets. Meseguer and Montanari [MeMo88] have shown how to associate a symmetric monoidal category $\mathcal{B}(N)$ with a Petri net N, in which the objects represent markings of the net, and the morphisms represent possible sequential and parallel combinations of events in the net. Thus, the category $\mathcal{B}(N)$ represents the behaviour of the net N. Our idea is to represent a timing of a net N by a map from $\mathcal{B}(N)$ to an object M of resources, which assigns a duration to each event e.

In chapter 2, we motivated the use of a monoid of resource values to model the complexity of the sequential composition of programs. Here, we can compose events both sequentially and in parallel. This motivates the definition of an object of resource values with two monoid operations. Thus, for example, we might take M to be $\mathbf{N}$ and model the time $t(\alpha;\beta)$ of a sequential composition of events by $t(\alpha) + t(\beta)$, and model the time $t(\alpha|\beta)$ of a parallel composition of events by $max\{t(\alpha), t(\beta)\}$. This suggests that we could model the durations of the events the net N by a symmetric monoidal functor from $\mathcal{B}(N)$ to a one-object symmetric monoidal category. In fact, the situation is a little more delicate, as Meseguer and Montanari assume that parallel composition distributes over sequential composition in the sense that:

$$(\alpha_0;\alpha_1)|(\beta_0;\beta_1) \;=\; (\alpha_0|\beta_0);(\alpha_1;\beta_1)$$

However, this clearly fails if we take into account the durations of the events, or even just their causal dependencies. Accordingly, we give a modified definition of $\mathcal{B}(N)$, and model the time of a time net by a lax symmetric monoidal functor from $\mathcal{B}(N)$ to a one-object, poset-enriched, lax symmetric monoidal category $\mathcal{M}$ [Gurr90]. We have developed a formal system for deriving timings of net events, and have proved that it is sound and complete with respect to interpretation in these timed nets.

As we have mentioned, several authors have constructed categories of Petri nets. Meseguer and Montanari extend the definition of $\mathcal{B}(N)$ to a category $\mathcal{B}$ of behaviours of nets, by taking as morphisms symmetric monoidal functors. However, it is not entirely clear that this is a suitable definition of morphism between behaviours. In particular, the function $\mathcal{B}(-)$ does not extend to a functor from **Net** to $\mathcal{B}$ for any of the categories **Net** of nets in the literature.

We have taken a different approach, and constructed a category $\mathcal{B}$ of behaviours of nets, such that we do obtain functors from **Net** to $\mathcal{B}$ for several of the categories **Net** of nets in the literature. The objects of $\mathcal{B}$ are the categories $\mathcal{B}(N)$ of behaviours of nets. Interestingly, the morphisms in $\mathcal{B}$ are not, in general, functors although the objects of $\mathcal{B}$ are categories. This approach seems promising, although further work is required.

Finally, we have constructed, for each category $\mathcal{B}$ of behaviours of nets, a number of categories of timed nets, culminating in a category $\mathcal{B}_t$ whose morphisms can be interpreted as implementations which satisfy specified time constraints. In addition, there is a forgetful functor $U : \mathcal{B}_t \longrightarrow \mathcal{B}$ which has a left adjoint, which constructs a free net with no time constraints.

9.3.3 A new categorical semantics of computations

In chapter 5, we argued that Moggi's categorical semantics of computations is not sufficiently general to capture all the examples of interest to us. We suggested a new framework, in the special case of complexity, based on the notion of an external datum.

An evident question is whether we could generalise or modify the definition of an external datum in order to obtain a new categorical semantics of computations which would capture all the non-examples in Moggi's categorical semantics of computations. A promising direction seems to be to concentrate on the distinction between the functional behaviour (definition 7.4.14) and the computational part of a program. We intend the functional behaviour to represent just the input-output behaviour of a program. An interesting observation is that the input-output

behaviour is, of course, a partial function from values to values. This suggests to us that non-termination should be treated as a special case and incorporated into the functional behaviour, rather than being considered as a notion of computation.

This work is still at an early stage, and several problems remain to be resolved. In particular, it seems likely that we will need to consider arbitrary monoidal categories, rather than just categories with finite products, in order for $\mathcal{C}$ to model partial functions from input to output. An interesting preliminary to this work is to extend the theory of external complexity categories to the case where $\mathcal{D}$ is an arbitrary monoidal category. It appear that this should be relatively straightforward.

9.3.4 Logics for complexity

In chapter 3, we presented a formal system, the λ_{com}-calculus, for reasoning about programs with complexity, and we gave an interpretation of a language for computations in any internal complexity category. In chapter 5, we constructed a new class of models, the external complexity categories, which strictly contains the internal complexity categories. An immediate question is whether we can extend the interpretation of the language for computations to external complexity categories or, if this is not possible, then what should we use to replace the λ_{com}-calculus. Similar questions arise for the categories constructed in chapters 7 and 8 for modelling upper bounds, input measures and non-exact complexity.

Unfortunately, the answer to this question is no. The problem is that it is not always possible to interpret a computational type $T\tau$ in an external complexity category. In the case of an internal complexity category $\mathcal{C}_M$, a type $T\tau$ is interpreted by the object $T[\![\tau]\!]$ and a term $\Gamma \vdash e : \tau$ is interpreted by a morphism from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ in $\mathcal{C}_M$, that is a morphism from $[\![\Gamma]\!]$ to $T[\![\tau]\!]$ in $\mathcal{C}$. In particular, a term $\Gamma \vdash e : T\tau$ is interpreted by a morphism from $[\![\Gamma]\!]$ to $TT[\![\tau]\!]$ in $\mathcal{C}$. In the case of an external complexity category, we would expect by analogy a base type τ to be interpreted by an object $[\![\tau]\!]$ of the category $\mathcal{C}$, and a term $\Gamma \vdash e : \tau$ to be interpreted by a morphism from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ in $\mathcal{C}^M$, that is a pair of morphisms

from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ in $\mathcal{C}$ and from $U([\![\Gamma]\!])$ to M in $\mathcal{D}$. However, we cannot interpret a type $T\tau$ by an object $T[\![\tau]\!]$ in $\mathcal{C}$ as $\mathcal{C}$ will not, in general, come equipped with an endofunctor T. It is not at all clear how else we might interpret such a type.

If we remove the type constructor T and the terms μ and [] from the language for computations, then it is straightforward to extend the interpretation of the language to external complexity categories. Furthermore, the λ_{com}-calculus without the rules for μ and [] will be sound with respect to this interpretation. However, the difficulties of modelling higher order types remain.

It is clear that more work is required on this subject. In particular, it is still unclear exactly what calculus will be complete with respect to interpretation in external complexity categories, and what calculus we require for reasoning about non-exact complexity.

Appendix A

Proofs from chapter 4

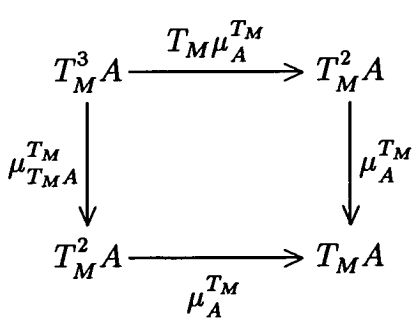
A.1 Proof of lemma 4.2.9

We present the proof of lemma 4.2.9.

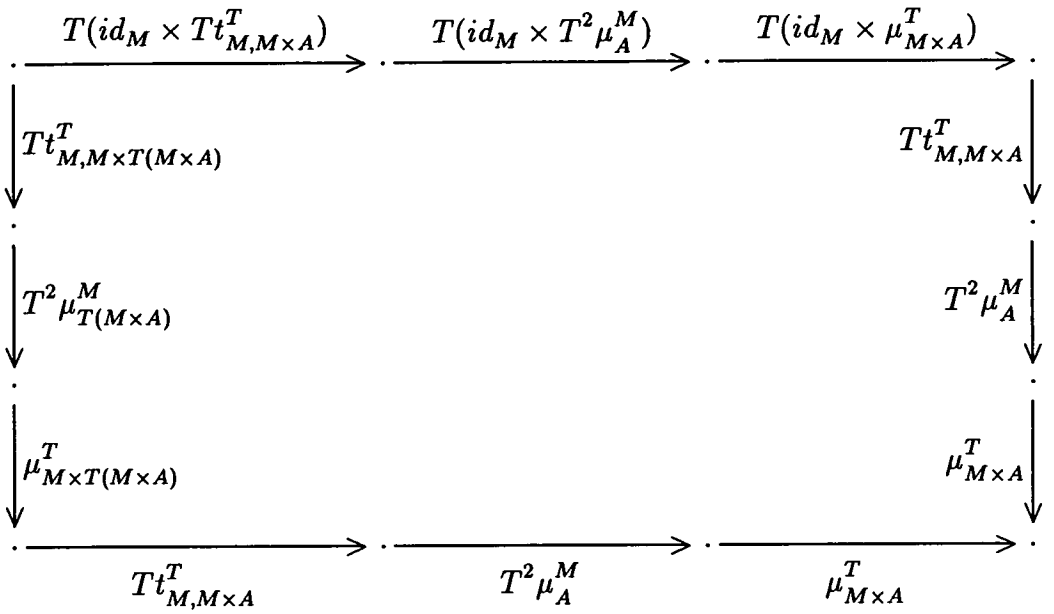
Proof: We verify that $\langle T_M, \eta^{T_M}, \mu^{T_M}, t^{T_M} \rangle$ satisfies the relevant seven diagrams in definitions 3.2.1 and 3.2.5.

Diagram 1

We require:

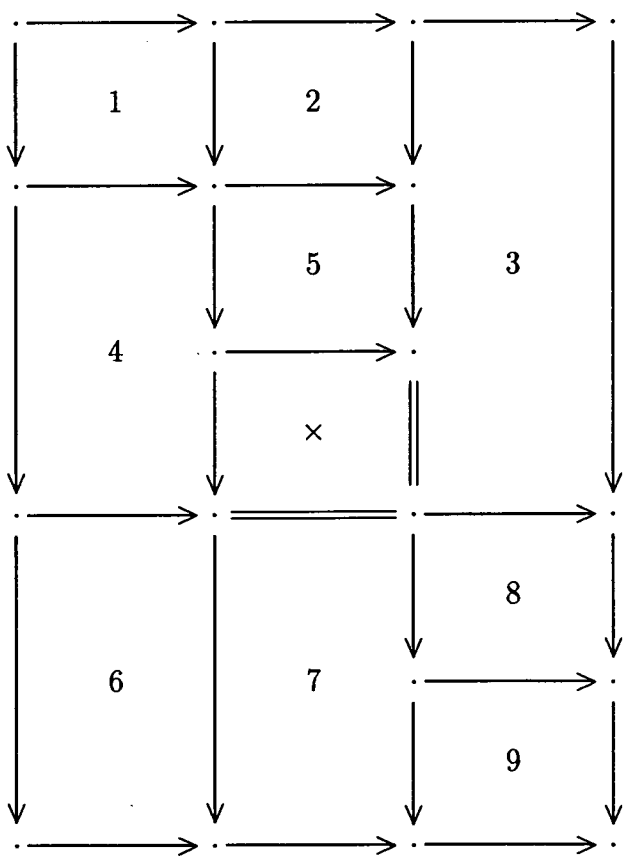


Expanding this out in terms of the data for the monads T and "com" we have:



where we recall that $T_M f = T(id_M \times f)$.

It is not immediately clear that this diagram commutes. However, we can fill in the diagram as follows:



We verify in detail each of the diagrams 1 to 9.

$$\begin{array}{ccc} T(M \times T(M \times T(M \times A))) & \xrightarrow{T(id_M \times Tt^T_{M,M\times A})} & T(M \times T^2(M \times M \times A)) \\ \Big\downarrow{\scriptstyle Tt^T_{M,M\times T(M\times A)}} & 1 & \Big\downarrow{\scriptstyle Tt^T_{M,T(M\times(M\times A))}} \\ T^2(M \times (M \times T(M \times A))) & \xrightarrow[T^2(id_M \times t^T_{M,M\times A})]{} & T^2(M \times T(M \times M \times A)) \end{array}$$

commutes because it is T applied to the diagram defining t^T as a natural transformation $- \times T(-) \to T(- \times -)$ at the morphism

$$id_M \times t^T_{M,M\times A} : \langle M, M \times T(M \times A)\rangle \longrightarrow \langle M, T(M \times (M \times A))\rangle.$$

$$\begin{array}{ccc} T(M \times T^2(M \times (M \times A))) & \xrightarrow{T(id_M \times T^2\mu^M_A)} & T(M \times T^2(M \times A)) \\ \Big\downarrow{\scriptstyle Tt^T_{M,T(M\times(M\times A))}} & 2 & \Big\downarrow{\scriptstyle Tt^T_{T(M\times A)}} \\ T^2(M \times T(M \times (M \times A))) & \xrightarrow[T^2(id_M \times T\mu^M_A)]{} & T^2(M \times T(M \times A)) \end{array}$$

commutes as it is T applied to the natural transformation diagram for t^T at

$$id_M \times T\mu^M_A : \langle M, T(M \times (M \times A))\rangle \longrightarrow \langle M, T(M \times A)\rangle.$$

$$\begin{array}{ccc} T(M \times T^2(M \times A)) & \xrightarrow{T(id_M \times \mu^T_{M\times A})} & T(M \times T(M \times A)) \\ \Big\downarrow{\scriptstyle Tt^t_{T(M\times A)}} & & \\ T^2(M \times (M \times A)) & 3 & \Big\downarrow{\scriptstyle Tt^T_{M,M\times A}} \\ \Big\downarrow{\scriptstyle T^2t^T_{M\times A}} & & \\ T^3(M \times (M \times A)) & \xrightarrow[T\mu^T_{M\times(M\times A)}]{} & T^2(M \times T(M \times A)) \end{array}$$

commutes as $\langle T, \eta^T, \mu^T, t^T\rangle$ is a strong monad and it is T applied to diagram 7 in definition 3.2.5 of a strong monad at the object $\langle M, M \times A\rangle$.

Diagram 4 is T^2 applied to b in:

$$\begin{array}{ccc}
(M \times M) \times T(M \times A) & \xrightarrow{t^T_{M\times M, M\times A}} & T((M \times M) \times (M \times A)) \\
\downarrow{\scriptstyle \alpha_{M,M,T(M\times A)}} & a & \downarrow{\scriptstyle T\alpha_{M,M,M\times A}} \\
M \times (M \times T(M \times A)) & \xrightarrow{t^T_{M,M\times(M\times A)} id \times t^T_{M,M\times A}} & T(M \times (M \times (M \times A))) \\
\downarrow{\scriptstyle \mu^M_{T(M\times A)}} & b & \downarrow{\scriptstyle T\mu^M_{M\times A}} \\
M \times T(M \times A) & \xrightarrow[t^T_{M,M\times A}]{} & T(M \times (M \times A))
\end{array}$$

a commutes because it is diagram 5 in definition 3.2.5 for T at the object

$$\langle M, M, T(M \times A)\rangle.$$

The outer square commutes because it is the natural transformation diagram for t^T at

$$\mu^M_{M\times A}\alpha : (M \times M) \times (M \times A) \longrightarrow M \times (M \times A).$$

However α is iso and so b must also commute and thus 4 commutes.

$$\begin{array}{ccc}
T^2(M \times T(M \times (M \times A))) & \xrightarrow{T^2(id_M \times T\mu^M_A)} & T^2(M \times T(M \times A)) \\
\downarrow{\scriptstyle T^2 t^T_{M,M\times A}} & 5 & \downarrow{\scriptstyle T^2 t^T_{M,A}} \\
T^3(M \times (M \times (M \times A))) & \xrightarrow[T^3(id_M \times \mu^M_A)]{} & T^3(M \times (M \times A))
\end{array}$$

commutes as it is T^2 applied to the natural transformation diagram for t^T at

$$id_M \times \mu^M_A : \langle M, M \times (M \times A)\rangle \longrightarrow \langle M, M \times A\rangle.$$

$$\begin{array}{ccc}
T^2(M \times T(M \times A)) & \xrightarrow{T^2 t^T_{M,M\times A}} & T^3(M \times (M \times A)) \\
\downarrow{\scriptstyle \mu^T_{M\times T(M\times A)}} & 6 & \downarrow{\scriptstyle \mu^T_{T(M\times(M\times A))}} \\
T(M \times T(M \times A)) & \xrightarrow[Tt^T_{M,M\times A}]{} & T^2(M \times (M \times A))
\end{array}$$

commutes as it is the natural transformation diagram for μ^T at

$$t^T_{M,M\times A} : M \times T(M \times A) \longrightarrow T(M \times (M \times A)).$$

$$\begin{array}{ccc} T^3(M\times(M\times A)) & \xrightarrow{T^3\mu^M_A} & T^3(M\times A) \\ \Big\downarrow{\scriptstyle \mu^T_{T(M\times(M\times A))}} & 7 & \Big\downarrow{\scriptstyle \mu^T_{T(M\times A)}} \\ T^2(M\times(M\times A)) & \xrightarrow[T^2\mu^M_A]{} & T^2(M\times A) \end{array}$$

commutes as it is the natural transformation diagram for μ^T at

$$T\mu^M_A : T(M \times (M \times A)) \longrightarrow T(M \times A).$$

$$\begin{array}{ccc} T^3(M\times(M\times A)) & \xrightarrow{T\mu^T_{M\times(M\times A)}} & T^2(M\times(M\times A)) \\ \Big\downarrow{\scriptstyle T^3\mu^M_A} & 8 & \Big\downarrow{\scriptstyle T^2\mu^M_A} \\ T^3(M\times A) & \xrightarrow[T\mu^T_{M\times A}]{} & T^2(M\times A) \end{array}$$

commutes as it is T applied to the natural transformation diagram for μ^T a

$$\mu^M_A : M \times (M \times A) \longrightarrow M \times A.$$

$$\begin{array}{ccc} T^3(M\times A) & \xrightarrow{T\mu^T_{M\times A}} & T^2(M\times A) \\ \Big\downarrow{\scriptstyle \mu^T_{T(M\times A)}} & 9 & \Big\downarrow{\scriptstyle \mu^T_{M\times A}} \\ T^2(M\times A) & \xrightarrow[\mu^T_{M\times A}]{} & T(M\times A) \end{array}$$

commutes because it is diagram 1 in definition 3.2.1 for T at $M \times A$.

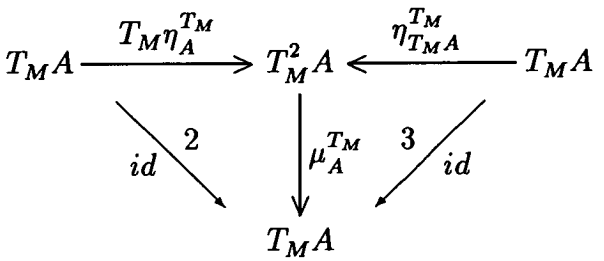
Finally,

$$T^3(M\times(M\times(M\times A))) \underset{T^3\mu^M_{M\times A}}{\overset{T^3(id_M\times\mu^M_A)}{\underset{\times}{\rightrightarrows}}} \cdot \xrightarrow{T^3\mu^M_A} T^3(M\times A)$$

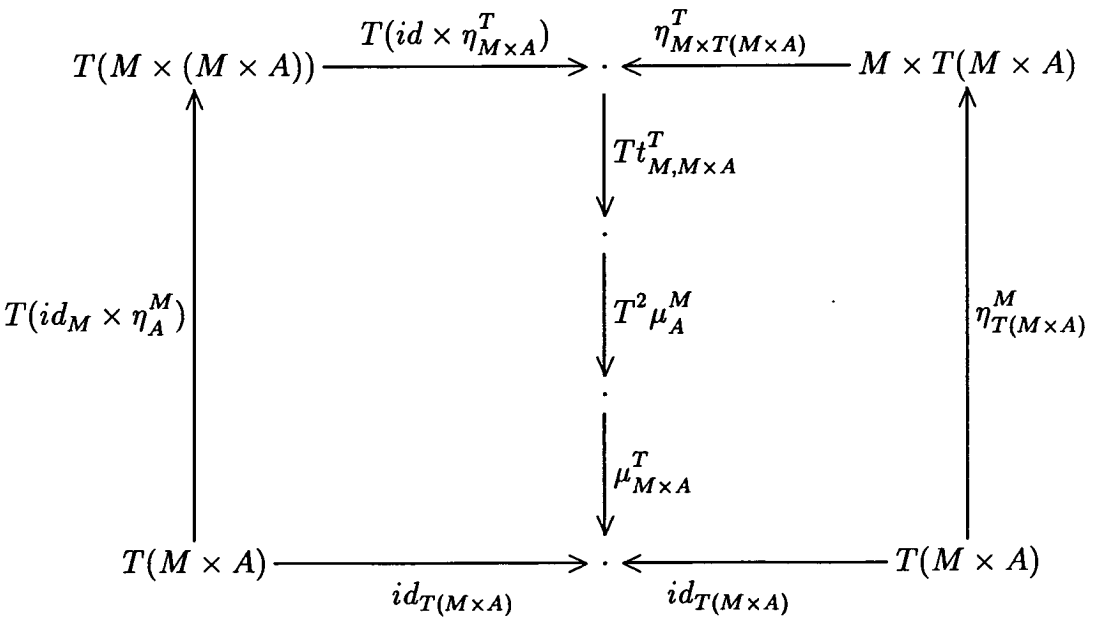
commutes because it is T^3 applied to diagram 1 in definition 3.2.1 for *com* at A. This suffices to establish that the outer square of the whole diagram commutes as required.

Diagrams 2 and 3

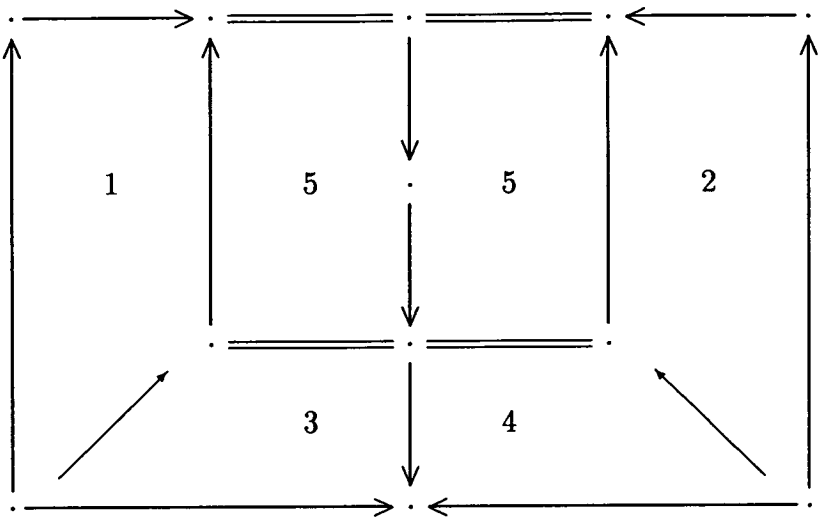
We require:



which expands to the diagram:



We can fill this in as:



$$\begin{array}{ccc} T(M \times (M \times A)) & \xrightarrow{T(id_M \times \eta^T_{M\times A})} & T(M \times T(M \times A)) \\ \uparrow{\scriptstyle T(id_M \times \eta^M_A)} & 1 & \uparrow{\scriptstyle T\eta^M_{T(M\times A)}} \\ T(M \times A) & \xrightarrow[T\eta^T_{M\times A}]{} & T^2(M \times A) \end{array}$$

commutes as it is T applied to the natural transformation diagram for η^M at the morphism

$$\eta^T_{M\times A} : M \times A \longrightarrow T(M \times A).$$

$$\begin{array}{ccc} T(M \times T(M \times A)) & \xleftarrow{\eta^T_{M\times T(M\times A)}} & M \times T(M \times A) \\ \uparrow{\scriptstyle T\eta^M_{T(M\times A)}} & 2 & \uparrow{\scriptstyle \eta^M_{T(M\times A)}} \\ T^2(M \times A) & \xleftarrow[\eta^T_{T(M\times A)}]{} & T(M \times A) \end{array}$$

commutes as it is the natural transformation diagram for η^T at the morphism:

$$\eta^M_{T(M\times A)} : T(M \times A) \longrightarrow M \times T(M \times A).$$

$$\begin{array}{ccccc}
T^2(M\times A) & = & T^2(M\times A) & = & T^2(M\times A) \\
\uparrow{\scriptstyle T\eta^T_{M\times A}} & 3 & \downarrow{\scriptstyle \mu^T_{M\times A}} & 4 & \uparrow{\scriptstyle \eta^T_{T(M\times A)}} \\
T(M\times A) & \xrightarrow[id]{} & T(M\times A) & \xleftarrow[id]{} & T(M\times A)
\end{array}$$

commute since these are diagrams 2 and 3 in definition 3.2.1 for T at the object $M \times A$. For 5 consider the following diagram:

$$\begin{array}{ccccc}
1\times T(M\times A) & \xrightarrow{r_{T(M\times A)}} & T(M\times A) & \xrightarrow{\eta^M_{T(M\times A)}} & M\times T(M\times A) \\
\downarrow{\scriptstyle t^T_{1,M\times A}} & a & \| & b & \downarrow{\scriptstyle t^T_{M,M\times A}} \\
T(1\times(M\times A)) & \xrightarrow[Tr_{M\times A}]{} & T(M\times A) & \xrightarrow[T\eta^M_{M\times A}]{} & T(M\times(M\times A))
\end{array}$$

The outer square commutes because it is the natural transformation diagram for t^T at the morphism $\eta^M_{M\times A} : \langle 1, M\times A\rangle \longrightarrow \langle M, M\times A\rangle$. Square a commutes because it is diagram 4 in definition 3.2.5 for T at $M \times A$. Square b then commutes as well because $r_{T(M\times A)}$ is iso.

Now diagram 5 above is T applied to β in:

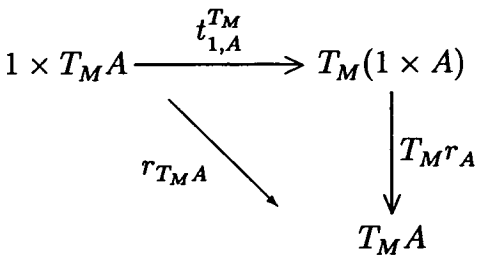
$$\begin{array}{ccc}
T(M\times A) & \xrightarrow{\eta^M_{T(M\times A)}} & M\times T(M\times A) \\
\uparrow{\scriptstyle T(id_{M\times A})} & \overset{\alpha}{\swarrow}\ \overset{\beta}{\searrow} & \downarrow{\scriptstyle t^T_{M,M\times A}} \\
T(M\times A) & \xrightarrow[T\eta^M_{M\times A}]{} & T(M\times(M\times A))
\end{array}$$

The outer square is b which commutes by the above. α commutes since it is T applied to diagram 2 in definition 3.2.1 for the monad *com* from which we can see that β is $id_{T(M\times A)}$ and so 5 is $id_{T^2(M\times A)}$.

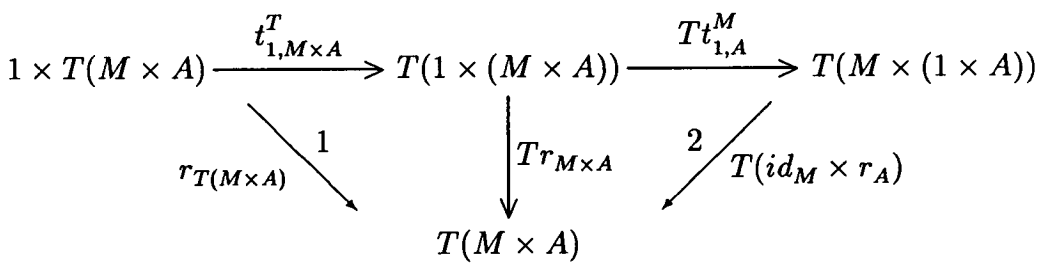
We can now see that this whole diagram commutes because 1,2,3 and 4 commute and 5 is the identity at $T^2(M \times A)$.

Diagram 4

We require:



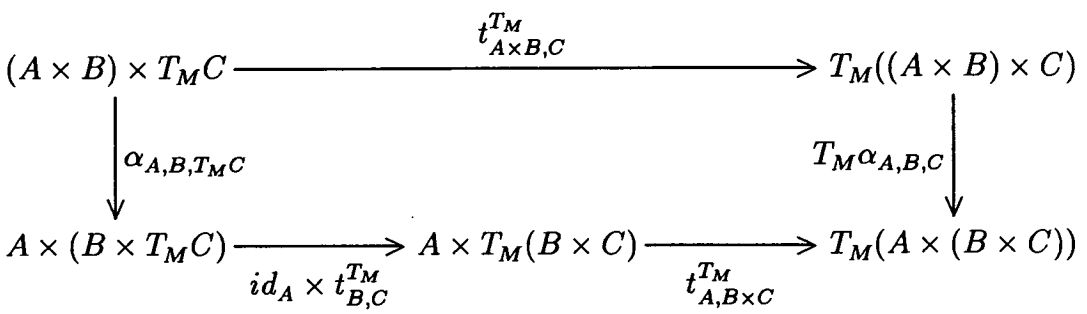
which expands to the diagram



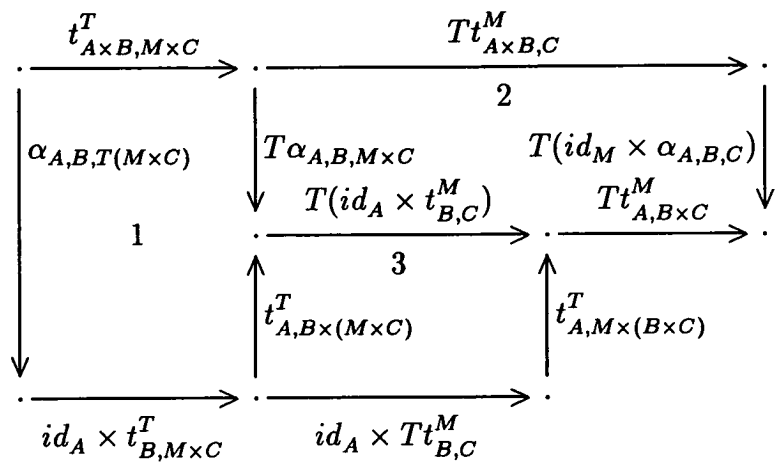
1 is diagram 4 in definition 3.2.5 for T at $M \times A$. 2 is T applied to diagram 4 in definition 3.2.5 for *com* at A. Thus the diagram commutes.

Diagram 5

We require:



which can be expanded and filled in as:



1 commutes because it is diagram 5 in definition 3.2.5 for T at the object $\langle A, B, M \times C\rangle$.

2 commutes because it is diagram 5 in definition 3.2.5 for *com* at the object $\langle A, B, C\rangle$.

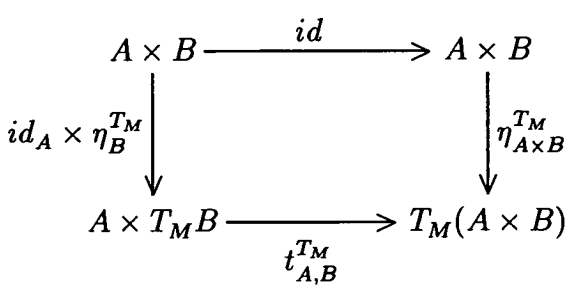
3 commutes because it is the natural transformation diagram for t^T at the morphism

$$id_A \times t^M_{B,C} : \langle A, B \times (M \times C)\rangle \longrightarrow \langle A, M \times (B \times C)\rangle.$$

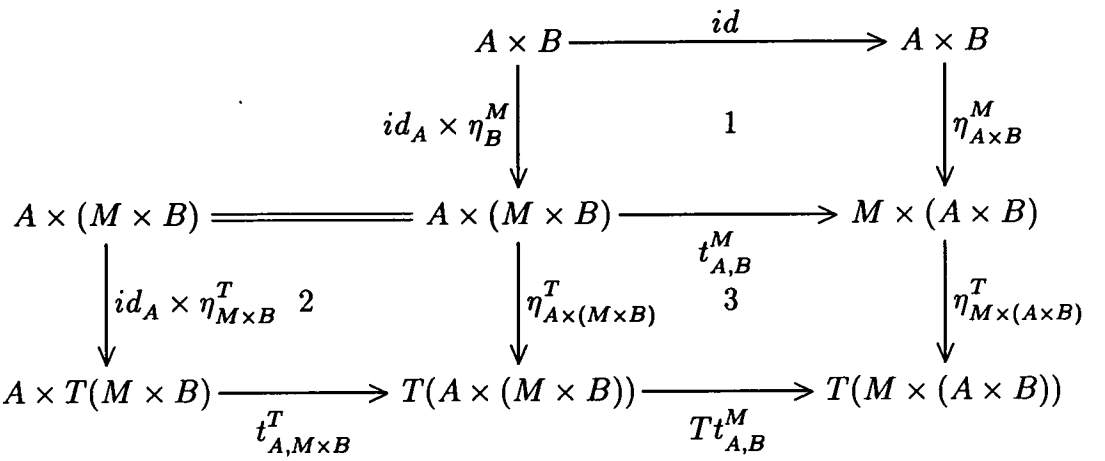
Thus the outer diagram commutes as required.

Diagram 6

We require:



We expand and fill this in as:



1 commutes because it is diagram 6 in definition 3.2.5 for *com* at $\langle A, B\rangle$.

2 commutes because it is diagram 6 in definition 3.2.5 for T at $\langle A, M \times B\rangle$.

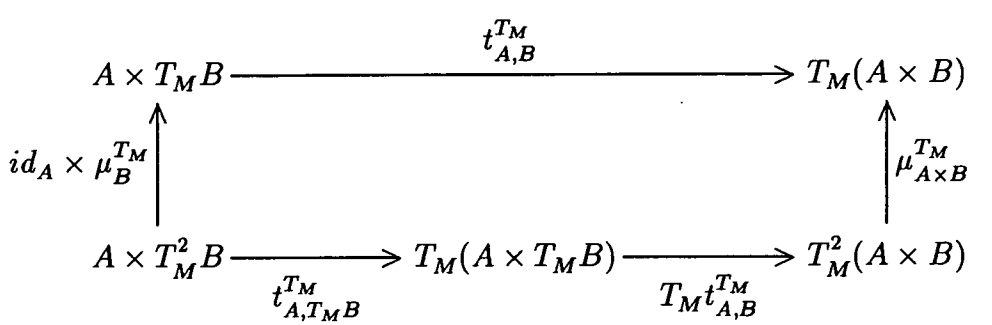
3 commutes because it is the natural transformation diagram for η^T at

$$t^M_{A,B} : A \times (M \times B) \longrightarrow M \times (A \times B).$$

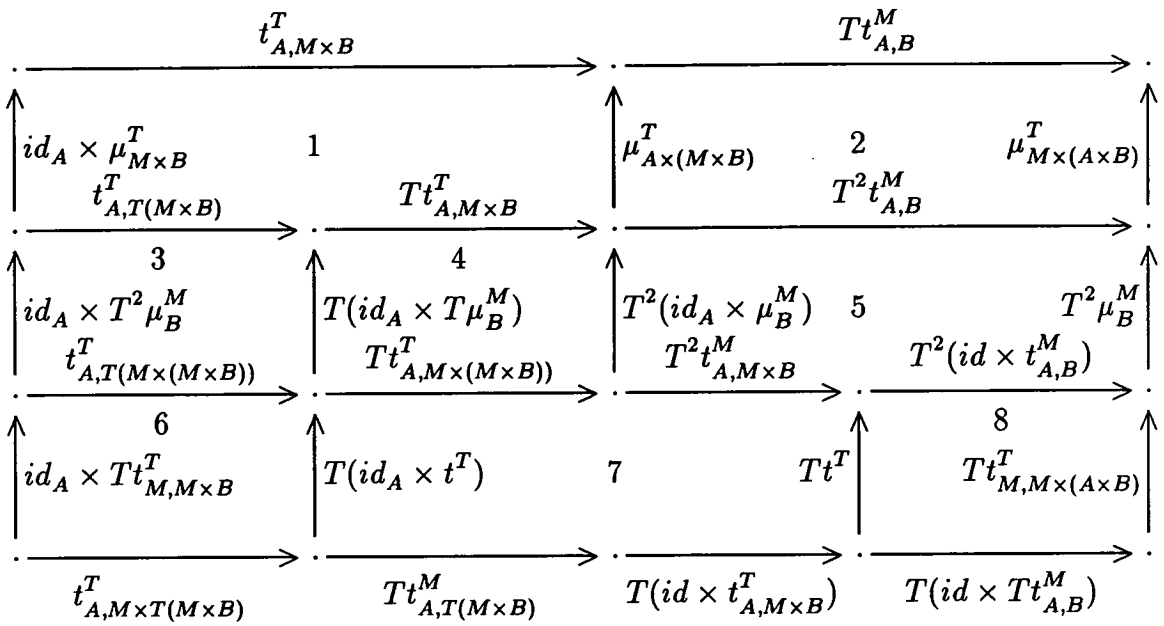
Thus the outer diagram commutes as required.

Diagram 7

Finally, we require:



We can expand and complete this as:



1 is diagram 7 in definition 3.2.5 for T at $\langle A, M \times B\rangle$.

2 is the natural transformation diagram for μ^T at the morphism

$$t^M_{A,B} : A \times (M \times B) \longrightarrow M \times (A \times B).$$

3 is the natural transformation diagram for t^T at the morphism

$$id_A \times T\mu^M_B : \langle A, T(M \times (M \times B))\rangle \longrightarrow \langle A, T(M \times B)\rangle.$$

4 is T applied to the natural transformation diagram for t^T at

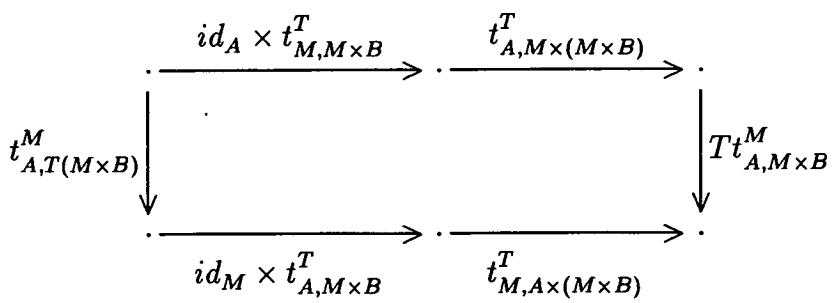
$$id_A \times \mu^M_B : \langle A, M \times (M \times B)\rangle \longrightarrow \langle A, M \times B\rangle.$$

5 is T^2 applied to diagram 7 in definition 3.2.5 for *com* at $\langle A, B\rangle$.

6 is the natural transformation diagram for t^T at the morphism

$$id_A \times t^T_{M,M\times B} : \langle A, M \times T(M \times B)\rangle \longrightarrow \langle A, T(M \times (M \times B))\rangle.$$

7 is T applied to the diagram:



t^T is a natural transformation $t^T_{A,B} : A \times TB \longrightarrow T(A \times B)$ and thus $id \times t^T$ is a natural transformation $(id \times t^T)_{A,B,C} : A \times (B \times TC) \longrightarrow T(A \times (B \times C)$. Further, $t^T_{A,B\times C}$ is a natural transformation $t^T_{A,B\times C} : A \times T(B \times C) \longrightarrow T(A \times (B \times C)$ and so $t^T_{A,B\times C}(id \times t^T)$ is a natural transformation $A \times (B \times TC) \rightarrow T(A \times (B \times C)$. Denoting this natural transformation by s, the diagram above can be seen to be the natural transformation diagram for s at the morphism

$$t^M_{A,M\times B} : \langle A, M, M \times B\rangle \longrightarrow \langle M, A, M \times B\rangle.$$

This commutes because of the naturality of s.
8 is T applied to the natural transformation diagram for t^T at

$$id_M \times t^M_{A,B} : \langle M, A \times (M \times B)\rangle \longrightarrow \langle M, M \times (A \times B)\rangle.$$

Thus the outer diagram commutes, *i.e.* T_M satisfies the seventh and final condition in the definition of a strong monad.
This completes the proof that T_M is a strong monad. □

A.2 Proof of lemma 4.2.17

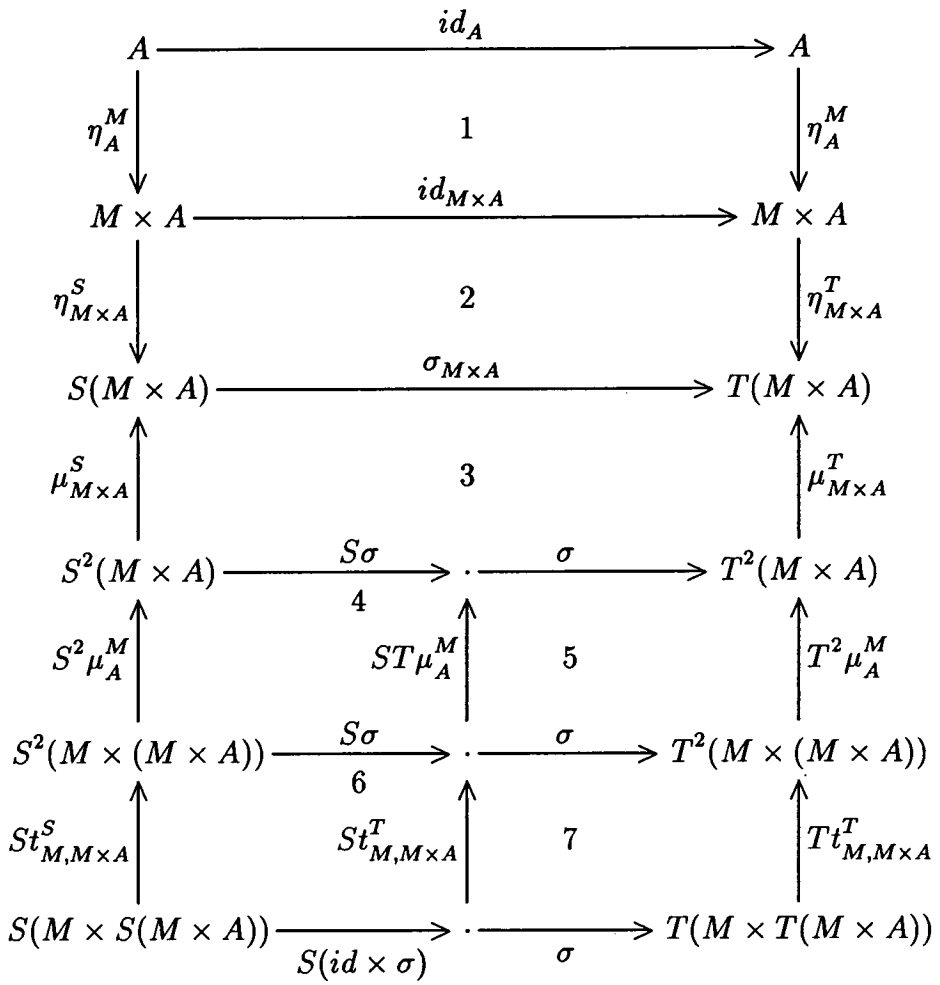
We present the proof of lemma 4.2.18.

Proof: The proof involves verifying the following diagrams:

$$\begin{array}{ccccc}
A & \xrightarrow{\eta^{S_M}_A} & S_M A & \xleftarrow{\mu^{S_M}_A} & S^2_M A \\
\downarrow{\scriptstyle id} & I & \downarrow{\scriptstyle \sigma^M_A} & II & \downarrow{\scriptstyle \sigma^M\sigma^M} \\
A & \xrightarrow[\eta^{T_M}_A]{} & T_M A & \xleftarrow[\mu^{T_M}_A]{} & T^2_M A
\end{array}
\qquad
\begin{array}{ccc}
A \times S_M B & \xrightarrow{t^{S_M}_{A,B}} & S_M(A \times B) \\
\downarrow{\scriptstyle id \times \sigma^M_A} \quad III & & \downarrow{\scriptstyle \sigma^M_{A\times B}} \\
A \times T_M B & \xrightarrow[t^{T_M}_{A,B}]{} & T_M(A \times B)
\end{array}$$

(recall that $(\sigma\sigma)_A = \sigma_{TA} S\sigma_A = T\sigma_A \sigma_{SA}$).

We can fill in I and II as:



We show that each of the squares 1 - 7 and hence the outer square commutes.

1 is clear.

2 is diagram 1 in definition 4.2.2 for σ at $M \times A$.

3 is diagram 2 in definition 4.2.2 for σ at $M \times A$.

4 is S applied to the natural transformation diagram for σ at

$$\mu_A^M : M \times (M \times A) \longrightarrow M \times A.$$

5 is the natural transformation diagram for σ at

$$T\mu_A^M : T(M \times (M \times A)) \longrightarrow T(M \times A).$$

6 is S applied to diagram 3 in definition 4.2.2 for σ at $\langle M, M \times A\rangle$.

7 is the natural transformation diagram for σ at

$$t^T_{M,M\times A} : M \times T(M \times A)) \longrightarrow T(M \times (M \times A)).$$

Thus I and II commute.

We can fill in digram III above as:

$$\begin{array}{ccccc}
A \times S(M \times B) & \xrightarrow{t^S_{A,B}} & S(A \times (M \times B)) & \xrightarrow{St^M_{A,B}} & S(M \times (A \times B)) \\
\downarrow{\scriptstyle id \times \sigma_{M \times B}} & 1 & \downarrow{\scriptstyle \sigma_{A \times (M \times B)}} & 2 & \downarrow{\scriptstyle \sigma_{M \times (A \times B)}} \\
A \times T(M \times B) & \xrightarrow[t^T_{A,B}]{} & T(A \times (M \times B)) & \xrightarrow[Tt^M_{A,B}]{} & T(M \times (A \times B))
\end{array}$$

1 is diagram 3 in definition 4.2.2 for σ at $\langle A, M \times B \rangle$.

2 is the natural transformation diagram for σ at

$$t^M_{A,B} : A \times (M \times B) \longrightarrow M \times (A \times B).$$

Thus the outer square commutes.

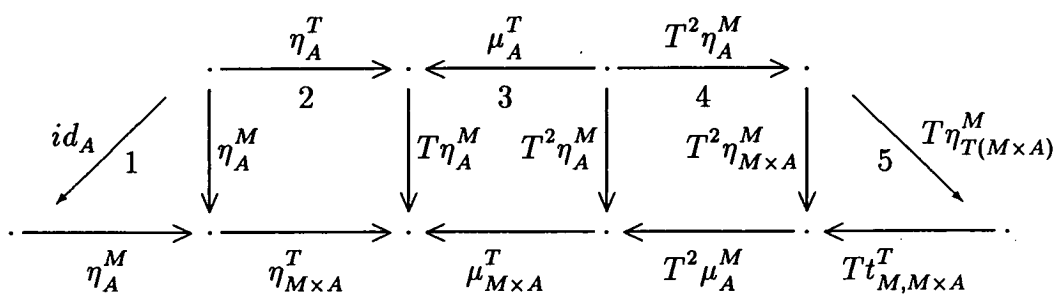
Therefore σ^M is a natural transformation $S_M \to T_M$ and satisfies diagrams I-III in definition 4.2.2. Hence σ^M is a strong monad morphism $S_M \to T_M$. □

A.3 Proof of lemma 4.2.22

We present the proof of lemma 4.2.23.

Proof: We verify the coherence conditions I to VI.

Diagrams I and II can be expanded and filled in as:

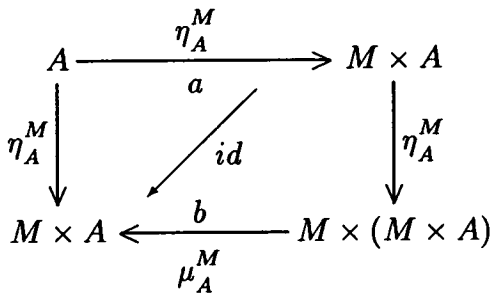


1 is clear.

2 is the natural transformation diagram for η^T at $\eta^M_A : A \longrightarrow M \times A$.

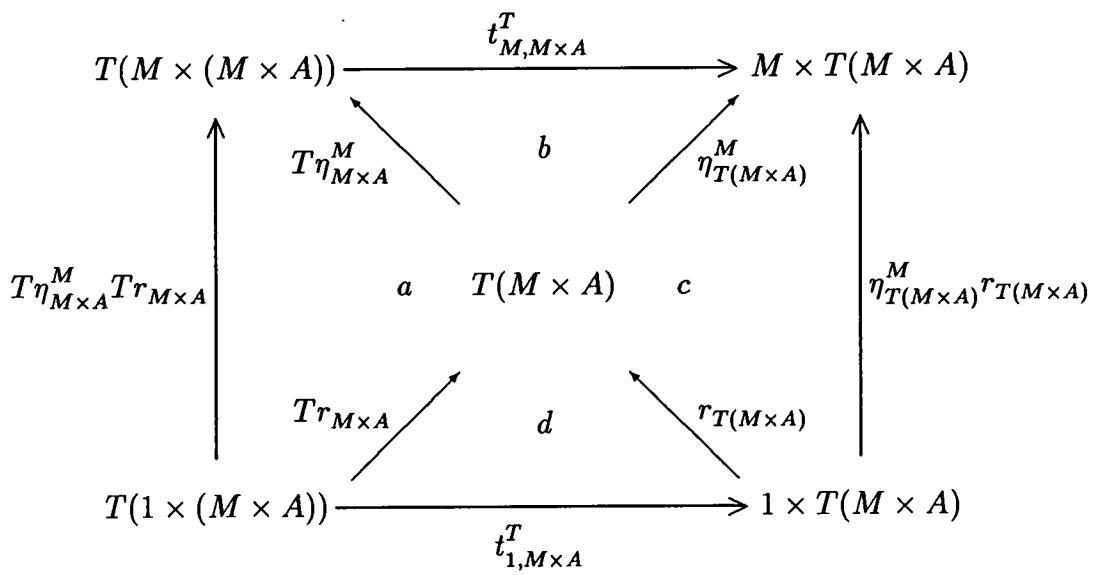
3 is the natural transformation diagram for μ^T at $\eta^M_A : A \longrightarrow M \times A$.

4 is T^2 applied to the outer square of the diagram



It is clear that a commutes. b is diagram 3 in definition 3.2.1 for *com* at A and so it commutes. Hence 4 commutes.

5 is T applied to b in:



It is clear that a and c commute.

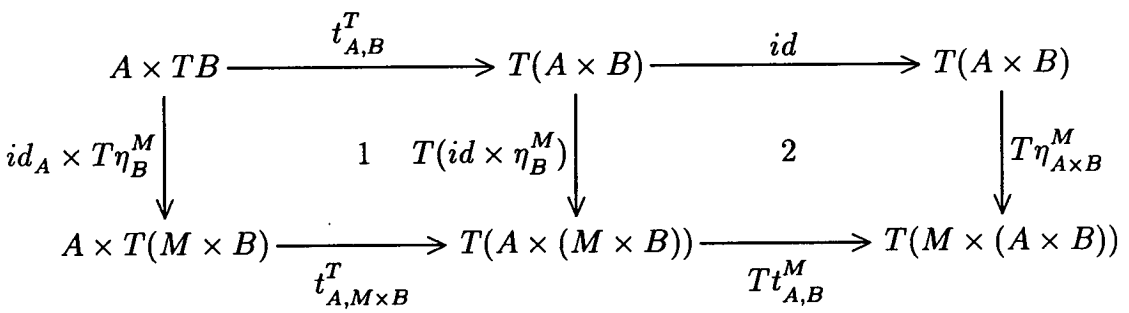
d commutes as it is diagram 4 in definition 3.2.5 of T at $M \times A$.

The outer square commutes as it is the natural transformation diagram for t^T at

$$\eta^M_{M\times A}r_{M\times A} : \langle 1, M \times A\rangle \longrightarrow \langle M, M \times A\rangle.$$

Finally $r_{T(M\times A)}$ is iso and in particular epi and so b and thus 5 commute.

Diagram *III* can be expanded and filled in as:



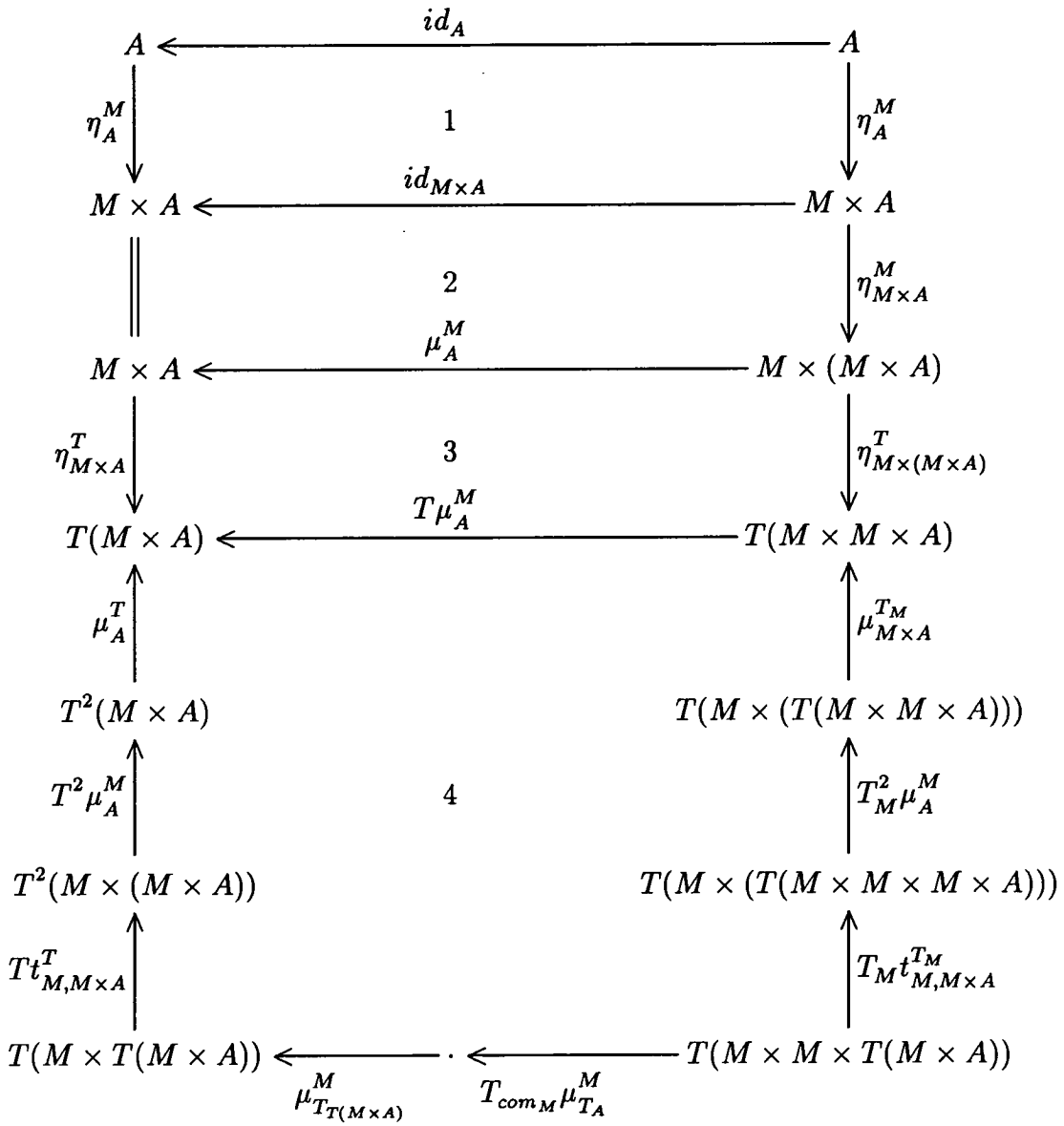
1 is the natural transformation diagram for t^T at $id_A \times \eta^M_B : \langle A, B\rangle \longrightarrow \langle A, M \times B\rangle$.

2 is T applied to diagram 6 in definition 3.2.5 for *com* at $\langle A, B\rangle$.

Diagrams *IV* and *V*

This is the point at which we require that the monoid $\mathcal{M}$ be commutative. If the monoid is not commutative, it will not in general be the case that diagram V commutes.

Given a commutative monoid, we can expand and proceed as follows:

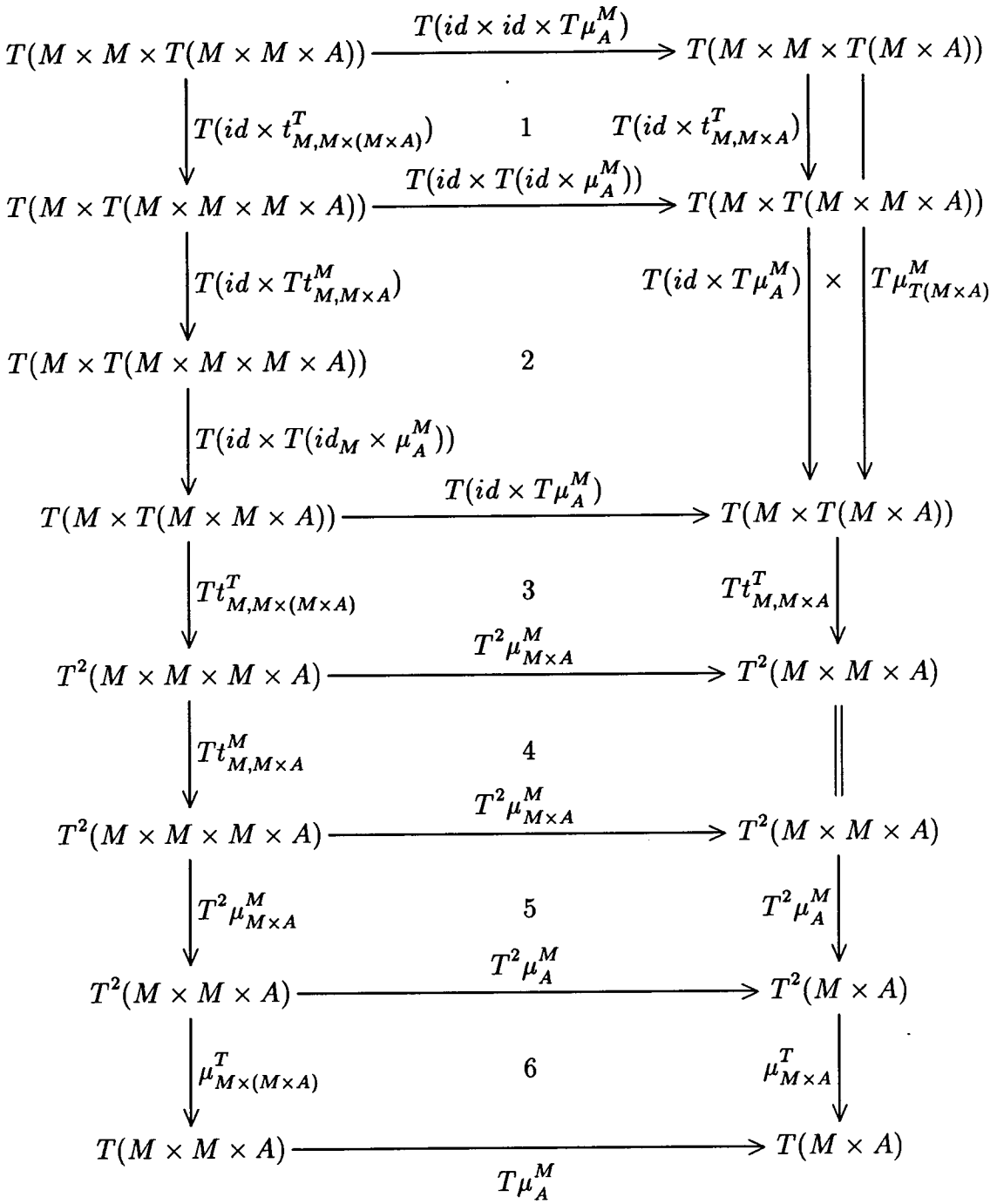


1 is clear.

2 is diagram 3 in definition 3.2.1 for *com* at A.

3 is the natural transformation diagram for η^T at $\mu_A^M : M \times (M \times A) \longrightarrow M \times A$.

Diagram 4 further expands as:



1 is T_M applied to the natural transformation diagram for t^T at

$$id_M \times \mu_A^M : \langle M, M \times (M \times A)\rangle \longrightarrow \langle M, M \times A\rangle.$$

2 is $T_M T$ applied to the diagram:

$$\begin{array}{ccc}
M \times (M \times (M \times A)) & \xrightarrow{id_M \times \mu^M_A} & M \times (M \times A) \\
\downarrow{\scriptstyle t^M_{M,M\times A}} & & \downarrow{\scriptstyle \mu^M_A} \\
M \times (M \times (M \times A)) & & \\
\downarrow{\scriptstyle id_M \times \mu^M_A} & & \\
M \times (M \times A) & \xrightarrow[\mu^M_A]{} & M \times A
\end{array}$$

which commutes since, by assumption, $\mathcal{M}$ is commutative.

3 is T applied to the natural transformation diagram for t^T at

$$id \times \mu^M_A : \langle M, M \times (M \times A)\rangle \longrightarrow \langle M, M \times A\rangle.$$

4 follows from the commutativity of $\mathcal{M}$.

5 is clear.

6 is the natural transformation diagram for μ^T at

$$\mu^M_A : M \times (M \times A) \longrightarrow M \times A.$$

Finally,

$$M \times (M \times T(M \times A))) \underset{id \times T\mu^M_A id \times t^T_{M,M\times A}}{\overset{\mu^M_{T(M\times A)}}{\overset{\longrightarrow}{\underset{\longrightarrow}{\times}}}} M \times T(M \times A)$$

does not commute, but it is co-equalized by

$$T\mu^M_A t^T_{M,M\times A} : M \times T(M \times A) \longrightarrow T(M \times A)$$

because $\mathcal{M}$ is a commutative monoid and this suffices to show that the whole outer diagram commutes.

Diagram VI can be expanded and filled in as:

$$\begin{array}{ccccc}
\cdot & \xrightarrow{t^T_{A,M\times(M\times B)}} & \cdot & \xrightarrow{Tt^M_{A,M\times B}} & \cdot \xrightarrow{T(id_M \times t^M_{A,B})} \cdot \\
\downarrow{\scriptstyle id_A \times T\mu^M_B} & 1 & \downarrow{\scriptstyle T(id_A \times \mu^M_B)} & 2 & \downarrow{\scriptstyle T\mu^M_{A\times B}} \\
\cdot & \xrightarrow[t^T_{A,M\times B}]{} & \cdot & \xrightarrow[Tt^M_{A,B}]{} & \cdot
\end{array}$$

1 is the natural transformation diagram for t^T at

$$id_A \times \mu_B^M : \langle A, M \times (M \times B)\rangle \longrightarrow \langle A, M \times B\rangle.$$

2 is T applied to diagram 7 in definition 3.2.5 for *com* at $\langle A, B\rangle$.

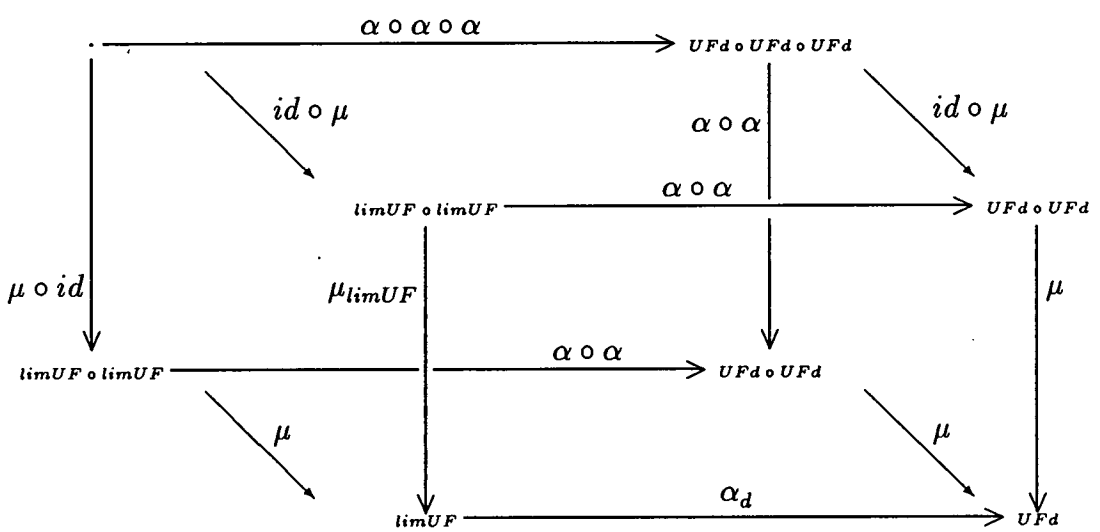
Thus η_T^M and μ_T^M are monad morphisms and thus well-defined. This completes the proof of the lemma. □

A.4 Proof of proposition 4.3.11

We present part of the proof of proposition 4.3.11.

Proof: We verify that μ_{limUF} is associative and η_{limUF} is a unit.

Associativity:



The **Front Face:** commutes by the limiting property, *i.e.* by construction.

The **Back Face:** is (front face)o*id* and so commutes because the front face does.

The **Top Face:** is *id*o(front face) and so commutes because the front face does.

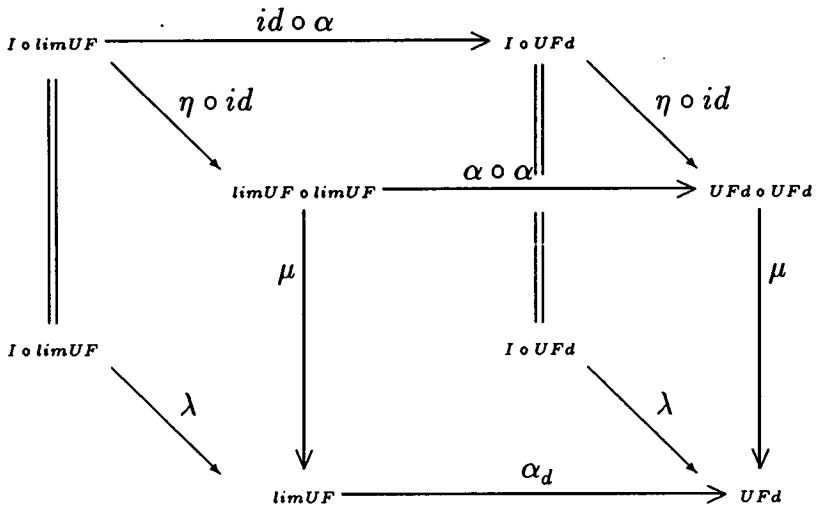
The **Bottom Face:** is the front face.

The **RH Face:** commutes because μ is associative at UFd.

The **LH Face:** composed with α_d commutes because the rest of the diagram does. However the α_d are jointly monic and so the LH face commutes.

Thus associativity holds.

Left Identity



The **Front Face:** commutes by the limiting property, *i.e.* by construction.

The **Back Face:** commutes because λ is a natural transformation and this is the natural transformation diagram for it at

$$\alpha : limUF \longrightarrow UFd.$$

The **Top Face:** commutes because η is a natural transformation.

The **RH Face:** commutes because, by assumption, UFd is a monoid in $\mathcal{K}$.

The **LH Face:** composed with α_d commutes because the rest of the diagram does. However the α_d are jointly monic and so the LH face commutes.

Thus the left identity axiom holds.

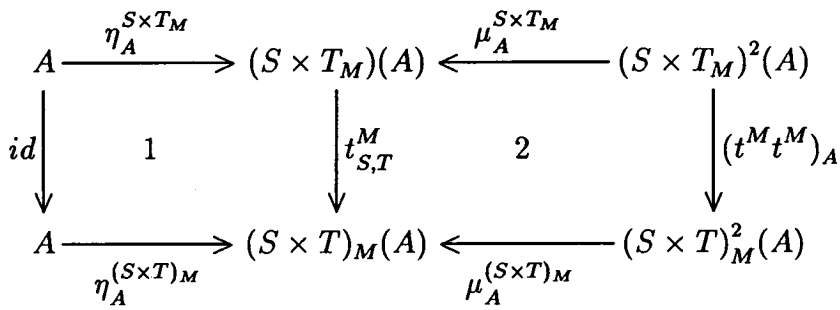
Right identity follows in the same way as left identity mutatis mutandis and this completes the proof. □

A.5 Proof of lemma 4.4.5

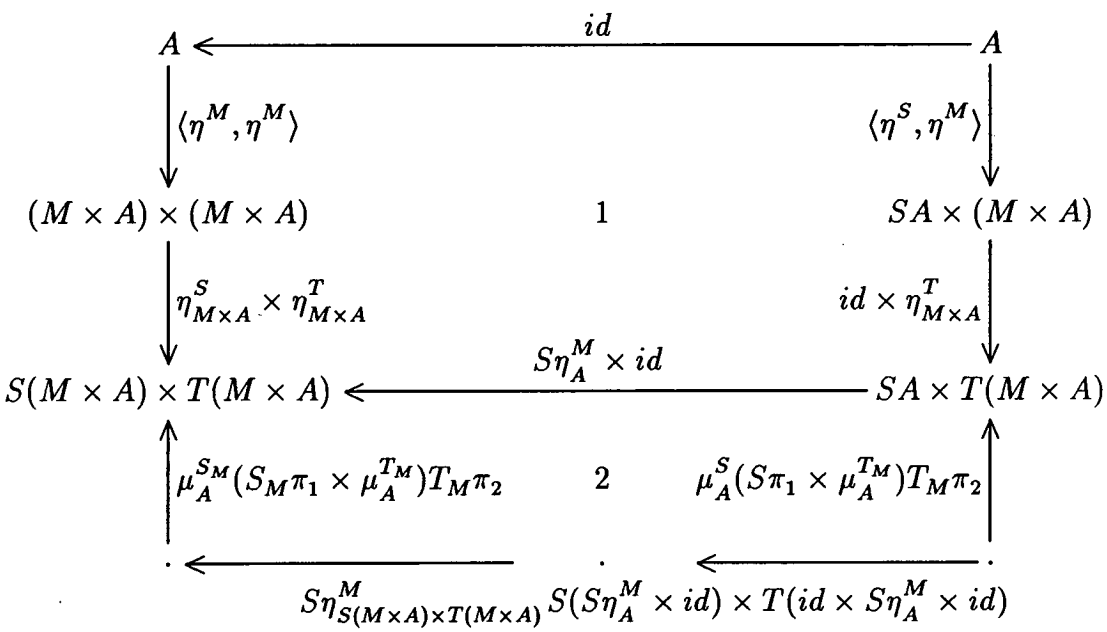
We present the proof of lemma 4.4.5.

Proof: To show that $t^M_{S,T}$ is a monad morphism we have to verify that the diagrams from definition 4.2.2 commute.

Diagrams 1 and 2 we require:



We expand this as:



1 commutes because it is the product of diagram 1 in definition 4.2.2 for η^M_S at A and an identity diagram.

2 is the product of diagram 2 in definition 4.2.2 for μ^M_S at A and an identity diagram.

Diagram 3 We require:

$$\begin{array}{ccc} A \times (S \times T_M)(B) & \xrightarrow{t^{S \times T_M}_{A,B}} & (S \times T_M)(A \times B) \\ \Big\downarrow{\scriptstyle id \times t^M_{S,T_B}} & 3 & \Big\downarrow{\scriptstyle t^M_{S,T_{A \times B}}} \\ A \times (S \times T)_M(B) & \xrightarrow[t^{(S \times T)_M}_{A,B}]{} & (S \times T)_M(A \times B) \end{array}$$

We expand this as:

$$\begin{array}{ccc} \cdot & \xrightarrow{\langle id_A \times \pi_1, id_A \times \pi_2 \rangle} & \cdot \xrightarrow{t^S_{A,B} \times t^{T_M}_{A,B}} \cdot \\ \Big\downarrow{\scriptstyle id_A \times S\eta^M_B \times id} & & \Big\downarrow{\scriptstyle S\eta^M_{A \times B} \times id} \\ \cdot & \xrightarrow[\langle id_A \times \pi_1, id_A \times \pi_2 \rangle]{} & \cdot \xrightarrow[t^{S_M}_{A,B} \times t^{T_M}_{A,B}]{} \cdot \end{array}$$

which commutes since it is the product of diagram 3 in definition 4.2.2 for η^M_S at $\langle A, B \rangle$ and an identity diagram.

Thus $t^M_{S,T}$ is a monad morphism. □

A.6 Proof of proposition 4.4.6

We present the proof of proposition 4.4.6.

Proof: To show that t^M is natural in S and T we require:

$$\begin{array}{ccc} S \times T_M & \xrightarrow{t^M_{S,T}} & (S \times T)_M \\ \Big\downarrow{\scriptstyle \sigma \times \tau^M} & & \Big\downarrow{\scriptstyle \sigma^M \times \tau^M} \\ S' \times T'_M & \xrightarrow[t^M_{S',T'}]{} & (S' \times T')_M \end{array}$$

which holds if it holds at each $A \in |\mathcal{C}|$. At A it is:

$$\begin{array}{ccc} SA \times T(M \times A) & \xrightarrow{S\eta^M_A \times id} & S(M \times A) \times T(M \times A) \\ \downarrow{\scriptstyle \sigma_A \times \tau_{M \times A}} & & \downarrow{\scriptstyle \sigma_{M \times A} \times \tau_{M \times A}} \\ S'A \times T'(M \times A) & \xrightarrow[S'\eta^M_A \times id]{} & S'(M \times A) \times T'(M \times A) \end{array}$$

which commutes by the naturality of σ and τ.

Thus t^M is natural $Id_{\mathrm{Mon}(\mathcal{C})} \times (-)_M \to (- \times -)_M$.

It now remains only to check diagrams IV to VII in definition 3.2.5.

Diagram IV We require:

$$\begin{array}{ccc} 1 \times T_M & \xrightarrow{t^M_{1,A}} & (1 \times T)_M \\ & {\scriptstyle r_{T_M}} \searrow & \downarrow{\scriptstyle (r_T)_M} \\ & & T_M \end{array}$$

where 1, the terminal object in $\mathrm{SMon}(\mathcal{C})$, is the monad $1(A) = 1$ (*c.f.* example 4.3.17).

This holds if it holds pointwise, *i.e.* if $\forall A \in |\mathcal{C}|$ the following diagram commutes:

$$\begin{array}{ccc} 1 \times T(M \times A) & \xrightarrow{id_1 \times id} & 1 \times T(M \times A) \\ & {\scriptstyle r_{T(M \times A)}} \searrow & \downarrow{\scriptstyle r_{T(M \times A)}} \\ & & T(M \times A) \end{array}$$

which is clear.

Diagram V We require:

$$\begin{array}{ccc} \cdot & \xrightarrow{(R\eta \times S\eta) \times id} & \cdot \\ {\scriptstyle \alpha}\downarrow & & \downarrow{\scriptstyle \alpha} \\ \cdot & \xrightarrow[id \times S\eta \times id]{} \cdot \xrightarrow[R\eta \times id \times id]{} & \cdot \end{array}$$

and it is clear that this commutes.

Diagram VI We require:

$$\begin{array}{ccc} SA \times TA & \xrightarrow{id} & SA \times TA \\ {\scriptstyle id \times T\eta^M_A}\downarrow & & \downarrow{\scriptstyle S\eta^M_A \times T\eta^M_A} \\ SA \times T(M \times A) & \xrightarrow[S\eta^M_A \times id]{} & S(M \times A) \times T(M \times A) \end{array}$$

which clearly commutes.

Diagram VII We require:

$$\begin{array}{ccccc} S \times T_M & & \xrightarrow{t^M_{A,B}} & & (S \times T)_M \\ {\scriptstyle id_S \times \mu^M_T}\uparrow & & & & \uparrow{\scriptstyle \mu^M_{S\times T}} \\ S \times T_{c_M} & \xrightarrow[t^M_{S,T_M}]{} & (S \times T_M)_M & \xrightarrow[(t^M_{S,T})_M]{} & (S \times T)_{c_M} \end{array}$$

At $A \in |\mathcal{C}|$ this is:

$$\begin{array}{ccc} SA \times T(M \times A) & \xrightarrow{S\eta^M_A \times id} & S(M \times A) \times T(M \times A) \\ {\scriptstyle id_{SA} \times T\mu^M_A}\uparrow & 1 & \| \\ SA \times T(M \times M \times A) & \xrightarrow{S\eta^M_A \times T\mu^M_A} & S(M \times A) \times T(M \times A) \\ {\scriptstyle S\eta^M_A \times id}\downarrow & 2 & \uparrow{\scriptstyle \mu^M_{S\times T}} \\ S(M \times A) \times T(M \times M \times A) & \xrightarrow[(t^M_{S,T})_M]{} & S(M \times M \times A) \times T(M \times M \times A) \end{array}$$

1 is clear.

2 is the product of the diagrams:

$$\begin{array}{ccc} SA & \xrightarrow{S\eta^M_A} & S(M \times A) \\ {\scriptstyle S\eta^M_A}\downarrow & a & \uparrow{\scriptstyle S\mu^M_A} \\ S(M \times A) & \xrightarrow[S(id \times \eta^M_A)]{} & S(M \times (M \times A)) \end{array} \quad \text{and}$$

$$\begin{array}{ccc} T(M \times (M \times A)) & \xrightarrow{T\mu^M_A} & T(M \times A) \\ {\scriptstyle T(id)}\downarrow & b & \uparrow{\scriptstyle T\mu^M_A} \\ T(M \times (M \times A)) & \xrightarrow[T(id)]{} & T(M \times (M \times A)) \end{array}$$

a is S applied to diagram 2 in definition 3.2.1 for *com* at A.

b is clear.

Thus t^M is well-defined, natural and satisfies diagrams IV to VII in definition 3.2.5.

This completes the proof that $\langle (-)_M, \eta^M, \mu^M, t^M \rangle$ is a strong monad. □

Bibliography

[AHU75] A. V. Aho, J. E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, (Addison Wesley, 1975).

[AsTu82] P.R.J Asveld & J.V. Tucker, *Complexity Theory and the operational structure of algebraic programming systems*, (Acta. Inf. 17, pp 451-476, 1982)

[Bar84] H. P. Barendregt, *The lambda calculus: its syntax and semantics*, (North Holland, 1984).

[BaWe85] M. Barr and C. Wells, *Toposes, triples and theories*, (Springer Verlag, New York, 1985).

[BjHo89] B. Bjerner and S. Holmström, *A compositional approach to time analysis of first order lazy functional programs*, (manuscript, presented at FPCA, London 1989).

[BrGu90] C.T. Brown and D.J. Gurr, *A Categorical Linear Framework for Petri Nets*, (5th LICS Conf. IEEE, 1990).

[BrGu91] C.T. Brown and D.J. Gurr, *A Categorical Linear Framework for Petri Nets*, (Inf. and Computation, to appear).

[BuGo81] R. Burstall and J. Goguen, *Algebras, theories and freeness: an introduction for computer scientists*, (Proc. 1981 Marktoberdorf NATO summer school, Riedel 1981 and CRS-101-82, University of Edinburgh 1982).

[BuRy88] R. Burstall and D. Rhydehead, *Computational category theory*, (Prentice Hall, 1988).

[CoKa90] S. A. Cook and B. M. Kapron, *Characterizations of the Feasible Functionals of Finite Type*, (manuscript).

[EhMa85] H. Ehrig and B. Mahr, *Fundamentals of algebraic specification I: Equations and Initial Semantics*, (Springer Verlag, 1985).

[FSZ89] P. Flajolet, B. Salvey and P. Zimmerman, *Lambda-Upsilon-Omega, The 1989 Cookbook*, (Technical Report, Institut National de Recherche en Informatiques et en Automatique, 1989).

[FrSc90] P. Freyd and A. Scedrov, *Categories and Alegories*, (North Holland, to appear).

[GaJo79] M. R. Garey and D. S. Johnson *Computers and Intractability: A guide to the theory of NP-completeness*, (Freeman, 1979).

[Gir87] J-Y. Girard, *Linear Logic*, (TCS 50, 1987).

[GiSS90] J-Y. Girard, A. Scedrov, P.J. Scott, *Bounded Linear Logic*, (manuscript).

[Gurr90] D. J. Gurr, *Timing Nets, the poset enriched, one object, lax, symmetric monoidal way*, (manuscript, presented at meeting on Linear Logic, Dialectica categories and Petri nets, Edinburgh April 1990).

[HMT87] R. Harper, R. Milner and M. Tofte, *The semantics of standard ML version 1*, (ECS-LFCS-87-36).

[Hoa69] C. A. R. Hoare, *An axiomatic basis for computer programming*, (comm. ACM 12, 1969).

[JoTi84] A. Joyal and M. Tierney, *An extension of the galois theory of Grothendieck*, (Mem. AMS 51(1984) 307-310).

[Kel82] M. Kelly, *Basic Concepts of Enriched Category Theory*, (LMS 64, CUP 1984).

[Koc72] A. Koch, *Strong functors and monoidal categories*, (Archiv der Mathematik 23, 1972).

[LaSc86] J. Lambek and P. J. Scott, *Introduction to Higher Order Categorical Logic*, (volume 7 of Cambridge Studies in Advanced Mathematics, CUP, 1986).

[Lan64] P. J. Landin, *The mechanical evaluation of expressions*, (Comp. Journal 6, 1964).

[Mac71] S.A. Mac Lane, *Categories for the working mathematician*, (Graduate Texts in Mathematics 5, Springer Verlag, 1971).

[MeMo88] J. Meseguer and U. Montanari, *Petri Nets are Monoids: A New Algebraic Foundation*, (3th LICS Conf. IEEE, 1988).

[Mil89] R. Milner, *Communication and Concurrency*, (Prentice Hall, 1989)

[Mog88a] E. Moggi, *The Partial Lambda Calculus*, (Phd Thesis, University of Edinburgh, available as ECS-LFCS-88-63).

[Mog88b] E. Moggi, *Computational lambda-calculus and monads*, (4th LICS Conf. IEEE, 1989 and ECS-LFCS-88-66, 1988).

[Mog89] E. Moggi, *An Abstract View of Programming Languages*, (Lecture Notes, Stamford University 1989, available as ECS-LFCS-90-113).

[MOMe89] N. Marti-Oliet and J. Meseguer, *From Petri Nets to Linear Logic*, (In CTCS Manchester 1989, vol. 389 of LNCS, 1989).

[NeRS89] A. Nerode, J.B. Remmel, A. Scedrov, *Polynomially Graded Logic I, A Graded Version of System T*, (4th LICS Conf. IEEE, 1989).

[Niel84] H.R. Nielson, *Hoare Logics for run-time analysis of programs*, (PhD Thesis, University of Edinburgh, 1984).

[Pit89] A. M. Pitts, *Non-trivial power types can't be subtypes of polymorphic types*, (4th LICS Conf. IEEE, 1989).

[Plo81] G. D. Plotkin, *A Structural Approach to Operational Semantics*, (Report DAIMI FN-19, Aarhus University, 1981).

[Plo87] G. D. Plotkin, *Postgraduate Lecture Notes in Domain Theory*, (Edinburgh University, incorporating the "Pisa Notes").

[PlSm82] G. D. Plotkin and M.B. Smyth, The category-theoretic solution to recursive domain equations, (SIAM J. Computing, vol. 11, no. 4, 1982).

[Rei85] W. Reisig, *Petri Nets*, (Springer Verlag, 1985).

[Ren84] J. C. Renolds, *Polymorphism is not set theoretic*, (in G. Kahn, et al., Semantics of Data Types, LNCS 173, Springer Verlag, 1984).

[RoRo90] E. Robinson and G. Rosolini, *Polymorphism, Set theory and Call-by-Value*, (5th LICS Conf. IEEE, 1990).

[Sco76] D. S. Scott, *Data Types as Lattices*, (SIAM J. Computing 5, 1976).

[See89] R.A.G. Seely, *Graded Multicategories of Polynomial-time Realizers*, (In CTCS Manchester 1989, vol. 389 of LNCS, 1989).

[Shul90] J. Shultis, *On the Complexity of Higher-Order Programs*, (manuscript).

[Sto77] J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, (The MIT Press 1977).

[Stou88] A. Stoughton, *Fully Abstract Models of Programming Languages*, (Research Notes in Computer Science, Pitman (London), Wiley (New York, Toronto), 1988).

[Str72] R. Street, *The formal theory of monads*, (J. Pure and Applied Algebra 2(1972) 149-163).

[Win88] G. Winskel, *A Category of Labelled Petri Nets and Compositional Proof System*, (3th LICS Conf. IEEE, 1988).

Index

Index

$\mathcal{V}$-category, 128
$\mathcal{V}$-functor, 129
$\mathcal{V}$-natural transformation, 130
λ_c-model, 42
Monoids($\mathcal{K}$), 98, 112
SMon($\mathcal{C}$), 85
λ_c-calculus, 51
λ_{com}-calculus, 60
Mon($\mathcal{C}$), 85

algorithm, 15
analysis, 152

closed
    term, 48
complete, 54
complexity category, 26
    external, 117
    external ordered, 142
    internal, 110
    internal ordered, 136
    non-exact, 178
complexity datum, 26
computation
    language for, 45
congruence, 162
context, 46

domain, 14

entails, 54, 70
external data, 112
    morphism of, 114
    morphism of ordered, 141
    ordered, 138

free variable, 47
functional behaviour, 21, 170
functor
    lax, 132
    strict, 132

hom object, 129

input measure, 16
internal data, 109
    morphism of, 109
    morphism of ordered, 136
    ordered, 136
interpretation, 49, 69

Kleisli category, 40

M-equivalence, 163
measure, 150
    lax, 159
    non-exact, 185
monad, 39
    constructor, 86

constructor for complexity, 87
morphism of, 84
strong, 40
monoid, 24, 97, 112
commutative, 91
morphism of, 98, 112
ordered, 135
monoidal category, 96, 112
strict, 97, 112
monomorphism requirement, 39
monotone, 134

non-exact data, 177
morphism of, 178
notion of computation, 38

object set, 129

partial order, 134

quotient
category, 162
functor, 163

relation, 133
antisymmetric, 134
equivalence, 134
reflexive, 134
symmetric, 134
transitive, 134

semantics
axiomatic, 14
denotational, 11
operational, 10, 79
signature
for computations, 45
sound, 54, 71
strong
monad constructor, 104
monad for complexity, 43
monad morphism, 85
syntactic value, 78

T-exponential, 42
tensorial strength, 41
time complexity, 15
triple rules, 60
Turing machine, 15, 158

underlying
category, 131
monoid, 135, 163

value, 54

well pointed, 25
wire
fly by, 12