Verified Monadic Programming

Robin Green

This thesis is submitted to UCD Dublin

for the degree of **Master of Science**

in the College of Engineering, Mathematical and Physical Sciences

Submitted: **March 2008**

This research was conducted

in the School of Computer Science and Informatics.

The Head of School is **Dr Joe Carthy**.

This research was supervised by **Dr Joseph Kiniry**.

February 8, 2010

Contents

List of Figures

Abstract

Avoiding introducing bugs, and limiting their impact when they are introduced, are key challenges for the software industry today. Formal verification offers hope in this endeavour, but at a high cost. This work aims to facilitate formal verification, in the program-extraction tradition, of functional programs, and thus to lower the cost of this process. It aims to do this by providing a framework for the verification of key building blocks in functional programming, such as monads and monad transformers. In addition, a theory of parametricity for the Coq proof assistant is developed, to assist in proving common theorems, and a set of implementation guidelines are developed for implementing type classes and their instances in Coq.

Acknowledgements

Thanks to my two supervisors, Joe Kiniry and Paddy Nixon, and the CSI PhD programme coordinator Padraig Cunningham, for pushing me to work harder. I am grateful to Adam Chlipala, Russell O'Connor and Matthieu Sozeau for their technical assistance, and to Conor McBride and Thorsten Altenkirch for their illuminating replies to queries about their work.

Thanks to Arthur Hughes for asking a question about my work which made me realise a limitation in it, and led me to develop the notion of semi-uniform types as a result.

I am grateful to my internal examiner, Simon Dobson, for his comments on a previous version.

Last, but by no means least, I am most grateful to my wife Nataliya for her support, and for her helpful suggestions.

Chapter 1

Introduction

The vast majority of software projects and software products written today do not have proofs of correctness associated with them. Furthermore, most widely-used pieces of software are not accompanied by a fully machine-readable specification of what their correct behaviour is supposed to be.

This situation is understandable. Many practitioners are unaware of the existence of practical techniques and software tools for proving that a piece of code adheres to its specification. Even amongst those who are aware of such techniques, formal verification with mathematical proofs has been widely perceived (to some extent, correctly) as too complicated and time-consuming to be used in practice – for the vast majority of software projects, at least. Consequently, there is less of an incentive to even develop and maintain rigorous and accurate *specifications* – and this is especially true for applications (as opposed to libraries and APIs, which often have some API documentation that should be kept up-to-date).

Nevertheless, many researchers are exploring the creation, development and use of various different software tools, reusable abstractions, automated proof techniques, and/or interactive proof strategies, in an attempt to bring down the time and complexity costs of formal verification. This thesis is intended as a contribution to that effort.

1.1 A note on definitions

This Introduction does not seek to define all the technical terms it uses straight away: that would make it rather verbose and circumlocutory. Instead, the task of providing accurate definitions is generally delegated to the relevant substantive Chapters, which are signposted below.

1.2 Detailed summary of contributions

The design and implementation work described in this thesis is intended to support formal verification of code, at points in the code where certain category-theoretic structures, such as monads and other functors, are defined or used. It does this by supporting the *proving* of the laws that make something, for example, a monad (at the point where it is *defined*); and *referring* to those proofs in other proofs (relating to points where it is *used*).

Below, "monads, and other functors, and so on" is abbreviated to "monads and functors" for simplicity's sake.

"Verified monadic programming", the title of this thesis, is the umbrella term for this activity, i.e. development with and of monads and functors, and verifying those developments.

This work provides both:

- idiomatic support – in other words, implementation guidelines;
- and library support – that is, a theorem and tactic library.

As noted in Chapter 6, where they are introduced and explained, monads and functors are used widely in pure functional programming (at least, in the Haskell tradition, which is substantial). Thus, this work is potentially broadly applicable – insofar as it could be used to aid the verification of a wide variety of functional code.

The work consists of:

- A library which formalises monads and related category-theoretic abstract structures, for the proof assistant Coq (Chapter 10)

- A key example *instance* of one of those abstractions – namely, the standard Reader monad transformer – from which other entities, such as the Reader monad, can trivially be generated. This is a fully-fledged instance which can be directly used in verified programming projects (Section 10.5).

- A set of guidelines for implementing type classes, and instances of those type classes, in Coq (Chapter 8). Monads and functors are deliberately implemented using type classes in this work – as in Haskell 98. Since Coq at the time did not have type classes built in, it was necessary to provide a way of implementing them (given this design choice). Type classes are also useful for a range of purposes other than monads and functors, however. Thus, these guidelines could potentially be reused for other verification work involving type classes.

- The beginnings of a theory of parametricity of Coq (Chapter 9), including some work on definite descriptions. This is not essential for proving results about monads and functors. However, it is intended as a convenient aid to proving some such results: namely, those which, in some sense, "fall out" easily from the structure of Coq's Calculus of Inductive Constructions (in the context of no, or only a few, axioms). Although the parametricity principles developed here are geared towards being used in proving theorems about the category-theoretic structures dealt with in this work, this is also a line of work which is potentially more broadly applicable.

Thus, the library formalising monads and functors is the "top-level" contribution of this work, with the other contributions being focused on supporting that library – but also intended to be useful in their own right.

1.3 Outline of the thesis

The Chapters in this thesis are generally relatively self-contained units (although they have not previously been published, either separately or together). There is some dependence of later chapters on earlier ones. However, due to the varying backgrounds which readers of this thesis might have, it probably does not make sense, for most of the chapters, to regard them as *essential* prerequisites for other chapters. Thus, this section is not called anything so presumptive as "How to read this thesis".

There is extensive use of cross-references within this thesis (to other Chapters and sections in this thesis); both forward and backward cross-references are employed. Again, these cross-references should not be read as necessarily constituting prerequisites, but sometimes just as helpful pointers should the reader wish to read more – or refresh their memory – on a particular point. The extensive Index on page 145 – an oft-neglected book section in this online age – is also offered up in the same vein.

1.3.1 Part I

Part I of this thesis is concerned solely with providing background information to motivate and explain the work described in Part II of the thesis – and the terminology used throughout the thesis.

For a wide variety of software producers and users, it could be very beneficial, as argued in Chapters 3 and 4, if the previously-mentioned reduction in cost of formal verification came about, and resulted in formal verification gaining wider use. Individual bugs can have a big impact – especially in safety-critical systems, and especially for security bugs. Thus, techniques to reduce the number of bugs in a system can be very valuable.

However, before examining these techniques in more detail, it is worthwhile to first clarify what *precisely* is meant here by "formal verification" – and the implications for the level of rigour required. As explained in Chapter 2, formal verification essentially means mechanically checking a machine-readable proof that a piece of code satisfies a machine-readable specification; however, there are some caveats to this definition, which are explored in section 2.3. Though these caveats mostly tread familiar ground, section 2.3.4 discusses the axiom of functional extensionality, which is of key importance in this work.

Chapter 4 explores some key benefits of formal specifications and formal verification – there are more than it may seem at first. Chapter 5 then puts formal verification in the context of other quality assurance techniques, such as model checking, and outlines their respective advantages and disadvantages. These two Chapters identify the value of *verifying* code using monads and functors, over and above the partial confidence obtained by using other quality assurance techniques.

The final Chapter of Part I, Chapter 6, introduces monads in the context of pure functional programming. Pure functional programming is the approach taken by the Haskell programming language, in which side-effects must be rigorously "partitioned off" into special structures – namely, monads. Monads are also relevant for programming directly in a proof assistant such as Coq, because the Coq programming paradigm is also one of pure functional programming.

Section 6.2 briefly mentions the category-theoretic underpinnings of monads, as used in this context. Section 6.3 motivates the use of monads with a concrete example of a problem that monads could usefully help with, and then explains what monads are and what they consist of, from a programming perspective.

1.3.2 Part II

Chapter 7 discusses the choice of implementation language for this project, with reference to the criteria of full-blown verification, general-purpose applicability, termination checking, linguistic consistency, and maturity. Cayenne, Dependent ML, Concoqtion, Epigram and Coq are all considered, and Coq emerges the clear winner.

Chapter 8 explains why type classes were chosen for this work (as opposed to Java-style classes or ML-style modules), the abstract design of the type class idiom, and then the nuts and bolts of applying the type class idiom – with snippets of Coq code from the implementation. A dependency diagram of all the Coq source code files in this project is also introduced.

Chapter 9 explains the notions of parametricity, free theorems, and definite descriptions, and how they are used in this work, with extensive source code snippets. Lastly, Chapter 10 walks through how category-theoretic concepts such as categories and monads were formalised in this work – again with extensive code samples.

1.3.3 Part III

Chapter 11 makes some observations on the proof techniques used. In the first section, various key axioms – and whether they could be dispensed with – are discussed. In the second section, infelicities and limitations encountered during the implementation are listed, along with suggestions for improvement.

Finally, Chapter 12 sums up progress made on each of the key project goals, touches on related work, concludes and looks to the future.

Part I

Background and motivations

Chapter 2

Proofs of correctness for software

2.1 Correctness defined

For the purposes of this thesis, the following definition is staked out:

proof of correctness A rigorous proof that some given chunk of code behaves according to a given specification. The chunk of code can be of any size: for example, one line of code, or an entire software package.

Development of a proof of correctness presupposes:

1. some valid source code or object code[1]
2. a specification of what the correct behaviour of that code is supposed to be. This specification has to be, for the proof to make sense, at least as formal and precise as the proof itself.

Proofs of correctness may be – in principle – either written out by hand, generated automatically by an automated prover, or constructed by a human user in an interactive prover.

[1]Here, intermediate languages typically not read directly by human beings, such as Java bytecode, are classified – along with machine code – as object code.

2.1.1 Formal proofs

This thesis is only concerned with *formal* proofs of correctness. For the purposes of computer science, these are proofs, written in a formal language, consisting of a finite sequence of reasoning steps. Each reasoning step must be just an application of an entirely *algorithmic* rule of deduction, yielding a new conclusion. The conclusion of the last reasoning step must be syntactically equivalent to the statement to be proved (that is, they must yield equivalent abstract syntax trees when parsed). Any parameters to which the rules of deduction are applied must consist solely of:

- references to definitions that have previously been defined
- references to previously-proved lemmas
- assumptions, which must be explicitly declared as such, either as part of the conclusion resulting from the reasoning step, or else in the context of the statement to be proved (also known as its *environment*). Each environmental assumption may be viewed either as an *axiom* (a statement which is simply accepted as true without proof) or as an additional assumption which needs to be established to be true, with proof or exhaustive evidence (a form of proof), in the context(s) in which the correctness lemma is later used.[2]That is, each environmental assumption is either to be accepted as true without proof, or it requires proof.
- references to conclusions of prior reasoning steps in the same proof
- auxillary parameters, which must only be used to specify which parts of the foregoing to operate on, or how to operate on them, not to sneak in undeclared assumptions

[2]It can sometimes be useful to consider every assumption referenced in the proof that appears in its environment to be implicitly included in the statement to be proved. In the logic to be used in this thesis, a transformation from environmental assumption to explicit assumption is available as an algorithmic rule of deduction – as one would expect to find in a typical formal logic.

Technically, this thesis considers scripts (*proof scripts*) which generate such formal proofs. For simplicity's sake, and following common practice in the field of computer-assisted theorem proving, this thesis regards a proof script as in some sense equivalent to the proof that it generates. The language in which the generated proof is expressed (*not* the language in which the proof script is written!), together with the deduction rules, forms a formal logic – or in some cases, the generated proof is expressed in a type theory that is *isomorphic to* a formal logic (see Appendix A). In particular, the Calculus of Impredicative Constructions, which is the language generated by the proof scripts in this thesis, is isomorphic to a formal logic.

The approach to quality assurance for software which uses software to verify formal proofs of correctness, is, in this thesis, termed "full-blown verification" – or simply "formal verification".

2.1.2 Logical soundness

It can be important to distinguish between "logical truth" – that is, statements which are provable within the formal system – and "factual truth" – statements which have a true *meaning* (after all defined terms have been replaced with their definitions). The two sets of statements are not necessarily coextensive – indeed, it can be that neither is a subset of the other. Clearly, it is essential that the rules of deduction be *truth-preserving* – that is, when given factually true assumptions and/or lemmas as parameters, they should only allow factually true conclusions to be generated. If this is not the case, the corresponding logic is said to be unsound. With an unsound logic, there is a risk (indeed, a certainty, in unsound logics of typical construction) of being able to "prove" incorrect code correct, and proofs expressed in it should not be trusted.

The other potential source of unsoundness in a logic is the possible presence of factually incorrect axioms. Clearly, merely requiring axioms to be declared is not sufficient to prevent error. Thus, the following definition is stipulated for our purposes:

rigorous formal proof a formal proof which only relies on an axiom if the axiom is both intuitively true, and the logic without that axiom is not powerful enough to be capable of expressing a proof of the statement of the axiom (both of these claims would typically be argued *informally*).

Thus, doubt over the unprovability of an axiom used in a proof (or, obviously, doubt over its factual truth) casts doubt on the rigour of a formal proof. Some argue that no axioms whatsoever should be used [37]. There are two possible ways to try and avoid even axioms believed to be unprovable within the logic: to improve the logic to make it capable of proving them, or to find alternative proof techniques which do not require such axioms. This thesis, however, does not adhere to that philosophy, because it seeks to introduce certain time-saving metatheoretical axioms, whilst improving the Calculus of Inductive Constructions is outside its scope.

2.2 Proofs about code, not merely about models

Importantly, the definition of "proof of correctness" above implies that the code *itself* must be proven to adhere to the specification. This can take place in one of two ways:

1. The proof can be written to directly reason about the code in its source code or object code representation.

2. Alternatively, the code can first be translated into some other representation, and a proof of correctness (as defined above) can be provided that the translation preserved all the relevant semantics of the code. Subsequently, the results of the translation can then be reasoned about in the "proof proper". The developer[3] can do this safe in the knowledge that the translation was such that - *assuming* the translation proof was correct - it could not have allowed incorrect code to masquerade as correct code.

[3]To avoid the confusion that might be generated by the word "prover" in this context, this thesis employs the word "developer" to encompass people writing proofs, as well as people writing code.

The second approach is actually just a special case of the first. In the second approach, a structure-preserving transformation (i.e. a homomorphism) is first used to transform the code into a more convenient form for reasoning about. The composition of the verified transformation followed by the proof can be trivially converted into a valid proof.

This equivalence is analogous to the way in which the intermediate code transformations performed by a compiler can be ignored by its users. Whether a compiler uses one or two intermediate representations – or even none – it is still a compiler. Likewise, no matter how many transformations a proof involves, it is still a valid proof – as long as those transformations are themselves valid, i.e. relevant-semantics-preserving.

Proofs of translation validity clearly have to rely on some sort of formalised semantics of the language (and in particular, the dialect or version of the language) in which the code is written, such as a denotational semantics with a deep embedding [14]. There might also be a need to parse source code represented as strings into the representation of code used by the formal semantics, and any such parsing would be part of the translation process and therefore would also need to be proved correct with respect to some formal specification. Such a syntactic specification would cover not only of the set of syntactically well-formed strings but also what they should be transformed into. However, if both the parsing and the semantic translation (or just one of them, if only one were needed) were to be specified as functions within the logic, and those specifications were to be used as implementations (which would be easy to do for e.g. a denotational semantics), the proof of correctness for the translation would be completely trivial. It would just be a matter of showing the aforementioned functions to be equal to themselves (which can be proven by reflexivity of equality).

Nevertheless, regardless of how such a translation were to be implemented in practice, the *specification* of the grammar and/or semantics of a programming language can be complicated and could contain errors. Any such errors – like any other specification bugs – would not technically make a proof incorrect, but it would mean that the proof would be about something other than what was intended – in this case, about some other programming language than the one intended (see subsection 2.3.1).

Sometimes, higher-level entities which are processed to generate conventional source code are not termed "code", but something else, such as "specifications", or "executable specifications". However, in this thesis, they are called "code", if (and *only* if) a chain of one or more pieces of functioning software exists to translate them into executable code.

Formal approaches to software development which involve a translation, that has *not* itself been formally proven correct, from source or object code into some sort of abstract modelling notation – here termed "semi-formal" approaches – may still be useful. They could provide a trade-off of greater convenience, but less theoretical rigour, which may be an appropriate trade-off for some software producers. Nevertheless, they cannot yield proofs of correctness, by this demanding definition. Clearly, with such semi-formal proofs, even if the *model* of the code is proven to satisfy the specification – or is deemed "correct by construction" – the code itself may not be: there might have been some error in translating between the code and the model.

Similarly, a formal approach which involves translation *by hand* in the reverse direction – from model to code, because no software yet exists to convert the model into running code, cannot be termed "formal verification of code" either for the purposes of this thesis. This is because, as stipulated above, a model that is not machine-executable cannot be termed "code".

It's not surprising, therefore, that a number of proof assistants – including Coq, the proof assistant used in this thesis – have implemented support for full-blown verification, so that users do not have to rely on semi-formal verification for code that they write.

2.3 The limits of correctness proofs

Although the definition of a proof of correctness given in section 2.1 might look watertight at first glance, there are some caveats to be made. One caveat has already been made in subsection 2.1.2: the possibility of the logic in use being discovered to be unsound. Some other kinds of errors are not inherently ruled out by this definition, either, as shown below.

2.3.1 Specification deficiencies

A specification, like code, may contain bugs. That is to say, it may:

- fail to accurately express stated requirements
- omit requirements that, in hindsight, should have been included but were not thought of at the time
- include requirements that are unnecessarily stringent, or were only included due to confusion or an absent-minded mistake such as excessive copying and pasting

The author once took the view that the possibility of such bugs made formal verification a waste of time, because it does not, even ignoring all its other limitations, eliminate bugs. However, that is an excessively pessimistic view. It fails to take into account the possibility that the number of bugs present in the specification of a formally-verified program might be notably smaller than the number of bugs that would be expected in a version of the program developed without formal verification. There are some reasons to believe this might be so in many programs:

- While some specifications (such as "return true if and only if input is between 3 and 10") can be trivially converted into efficient implementations of themselves, not all can be. The specifications of some tasks are simpler than efficient implementations of those tasks, and hence are easier to intuitively grasp, and are less likely to contain errors.

- The specifications of some tasks, such as sorting a list, can be implemented in many different ways, for reasons of performance, readability and even elegance. The same specification can thus be written once, debugged if necessary in the light of experience, and then reused.
- If the specification of a task is too strong to allow it to be implemented efficiently, or at all, this is likely to become evident when trying to implement it.
- If the specification of a subtask is too weak to allow the correctness of certain client code to be proved without reference to the subtask's particular implementation, this fact is likely to become evident when trying to use that specification in a subsequent proof. It is not strictly necessary for private, internal subtasks to have specifications, let alone correct ones, as long as the publicly-exposed functionality for which formal verification is desired (such as the main function) has a correct specification. However, there are some advantages of having specifications for subtasks (see Chapter 4), and they can aid in subsequent proofs if they are present and state properties that those subsequent proofs require.

An incorrect specification does not, strictly speaking, make a proof that some code fulfils that specification, incorrect – unless the specification is incorrect by virtue of being unsatisfiable, in which case the logic must be unsound! In other cases, in general, it just means that proof is still valid, but the *wrong thing* has been proven.

2.3.2 Translation bugs

Because of the ubiquity of assemblers, interpreters and compilers for all platforms today, some translation from source code or assembly language code, into object code, is almost always used[4] – a translation which could, in principle,

[4]This sentence is not strictly true. Let us say that any interpreter *metaphorically* translates source code into object code, even if it does not literally work like that.

be erroneous. Furthermore, there might be bugs in the operating system or runtime environment.[5]

The possibility of such bugs means that code with a proof of correctness (as defined above), is not *necessarily* more trustworthy than code which has a formally-checked semi-formal proof (see above). Either could exhibit bugs when executed, due to translation bugs in both cases. Thus, the distinction between a "full" proof of correctness and a semi-formal one can be misleading, in some cases.

Fortunately, compilers and runtime systems can themselves be subject to formal verification. However, even for hypothetical future verified compilers that have *themselves* been compiled by verified compilers, there will always be at least one grandparent in the compilation/interpretation ancestry chain that has not been formally verified.

Still, in practical terms, that situation would probably inspire more confidence than contemporary compilers, which are not typically verified: they typically can, and do, exhibit code generation bugs, in which the compiler generates object code that does not do what the definition of the programming language mandates that it should do.

2.3.3 Verification bugs

When developing proofs of correctness for any non-trivial software, it makes a lot of sense to use software specifically designed for creating and checking proofs on a computer. This software can take the form of an interactive proof assistant, an automated prover, or an integrated combination of the two.[6] These software packages should be preferred over paper-and-pencil (or computer typesetting software such as TEX, which supports writing a proof but not checking it) for

[5] The same bug could conceivably appear as a translation bug in one implementation of a programming language, and as a library bug in another. For example, a typical C compiler often implements floating-point division using floating point machine language instructions; however, some other C compilers, targetting embedded systems without instruction set support for floating point, would implement the same operation using a call to a C library.

[6] A development environment for a dependently-typed programming language such as Epigram [27] can be viewed as a kind of interactive proof assistant.

non-trivial proofs about programs [47]. There are two key reasons for this:

First, because they can automate some tedious parts of proofs – or even whole proofs, in some cases.

Second, because they *may* improve confidence in a proof – compared to a human-checked proof – by allowing it to be constructed in an entirely rigorous, machine-checked manner.

Nonetheless, again, this formal proof checking will be done by a checker which either has not been formally verified – or which has, but its verifier has not been – or which has, and its verifier has been, but *its* verifier has not been, and so on. At some point in the chain, it is necessary to trust that expert code review, and/or the test of time, have rendered a proof checker mature and safe enough to use for a particular application, in order to have confidence in the result of the proof checking process.

For this reason, some mathematicians have expressed skepticism about proofs checked with machine assistance. For example, the first valid proof of the Four Colour Theorem attracted criticism – partly because, as it involved checking billions of cases by computer, a subtle bug in the computer software could have rendered the proof invalid without anyone noticing [19]. However, in a software development context, a formal proof will often be a formalisation of:

- either, an existing uncontroversial mathematical result (such as many of the results proven in the present work);
- or, an informal explanation in the developer's mind of how a piece of code is supposed to work.

In both of these instances, it seems likely that formalising the proof will provide an *added* assurance of correctness, over relying on a hand-checked proof.

Also, fortunately, the amount of code that a software producer or user has to trust can be made relatively small, by making the input language of the kernel of the proof checker comparatively small and simple. Furthermore, proof checkers can, in principle, be checked using existing independently-developed proof checkers, to provide added confidence (although the second proof checker might not have enough expressive power to faithfully represent the *entire* theory of the first, so a partial check might be necessary).

2.3.4 Resource limitations

Code which requires infeasibly large amounts of time or memory to operate on realistic input values may be mathematically "correct" – but it would be misleading, at best, to describe it as "meeting requirements". Every requirements specification (outside the realms of science fiction) must be read as implying a limit on resources if it does not state one explicitly, because no real application can afford arbitrary amounts of memory.

In the remainder of this thesis, though, such resource limits are ignored. In this respect, this thesis follows much other research on functional correctness (for example, [3, 12, 17]). In effect, it makes the simplifying assumption that unlimited amounts of time and memory are available. This assumption allows for cleanly separating functional correctness issues from resource consumption issues.

The assumption is a simplifying one because proofs of functional correctness properties can consider any two *side-effect-free* functions which produce the same output as each other for every input, to be equal (this is known as *functional extensionality*). Proofs regarding time or memory consumption, by contrast, cannot do this. Or rather, they can – but only under a much more expansive definition of what a *side effect* actually is – that is, only by treating memory allocation and/or the passage of time as side effects.

Thus, the assumption of unlimited resources allows any proof to safely rely on the functional extensionality principle, without any preconditions – because by assumption, there are no resource limit requirements to be proven. This is very useful for the work described in this thesis, which relies on the functional extensionality principle (in the form of an axiom) in a number of crucial proofs (section 11.1.1 discusses the feasibility and cost of removing this axiom from the work).

Consequently, for the purposes of this work, “a valid proof that code C fulfills its specification” is an abbreviation for “a valid proof that code C fulfills its specification given sufficient – but finite – amounts of time and memory”.[7] This approach is not perfect, because – for example – it allows the erroneous development of programs which would take longer than the universe’s current age to complete. Nonetheless, it does at least facilitate catching a very large class of functional-correctness bugs at development time, rather than in the field.

2.3.5 Hardware faults

Finally, clearly no mathematical proof of correctness can establish with certainty that the computer on which the software is running will not suffer some kind of hardware fault. Therefore, proofs of correctness for software implicitly rely, to some extent, on an assumption that the platform on which it is running (both in terms of hardware and in terms of software) behaves in certain well-defined ways. It is possible – and in many cases, desirable – to prove that a system will do something sensible even in the event of a partial hardware failure, such as a sensor failure, or even one of the CPUs in a supercomputer completely locking up. Yet, obviously, without at least the assumption that some part of the hardware will work as expected some of the time, nothing can be proven.

[7] *Infinite* amounts of time or memory are ruled out by a sufficiently rigorous formalism, as will be discussed later.

Chapter 3

The ubiquity and cost of software bugs

3.1 Bugs – still a problem

Bugs are a pervasive problem in software development today – as they have been for decades. This is so despite significant improvements in contributory factors such as programming language design and development tool technology, and despite an accretion of knowledge and heuristics within the software development community regarding the avoidance of common types of bugs. A quick query at the bug database for a popular piece of software such as Firefox (`http://bugzilla.mozilla.org/`) illustrates this. For instance, in February 2007, at least 110 bugs were reported by Firefox users that were subsequently confirmed and fixed.[1]

Firefox is by no means unique in this regard. Open source and closed source software development practices alike are vulnerable to the accidental introduction of bugs by developers, ranging from relatively innocuous user interface

[1]This conservative estimate does not include 84 bugs reported in that month that have been confirmed, but have not been marked as fixed at the time of writing. These bugs were excluded because sometimes such bugs are inappropriately confirmed, when they should have been either marked as "duplicate", or as "not a bug".

glitches, to serious security vulnerabilities. For example, Microsoft recently disclosed that Windows Vista, the most recent version of the Windows operating system, was vulnerable to a "critical" vulnerability in its TCP/IP code, which allowed taking complete control over an affected system over a network [30]. Also in recent months, several release versions of the stable Linux kernel were discovered to be vulnerable to three local root exploits in `vmsplice(2)` [31, 32, 33]. Such exploits allow elevating a local user account – or a remote code execution exploit – to a position of complete control over the system (that is, root access). Meanwhile, many wireless embedded systems such as mobile phones and wireless routers have been found to contain security vulnerabilities in their handling of wireless protocols. Some of these vulnerabilities have been known to result in the device being "bricked", requiring nothing short of a firmware re-flash to get it working again [39].

3.2 Counting bugs is problematic

Moreover, it is important to note that merely counting *publicly acknowledged* bugs in a product might only uncover an arbitrarily small proportion of the actual number of bugs, for the following reasons:

- Many commercial software companies do not publish complete bug databases or even bug statistics for their products.
- Many bugs might not (yet) have been entered into the bug database – especially if many of the end-users are non-technical, or if they regard a bug as not worth going to the effort of reporting.
- Frequently, bugs which are encountered during the development process, before release, are not entered into a bug database. This is especially likely when a bug is encountered shortly after it was introduced, or noticed by the same developer who introduced it – because, in such cases, developers often feel it would be unnecessarily bureaucratic to use the bug tracker.

Nevertheless, such “informally acknowledged” bugs still have an impact, in terms of slowing down development – occasionally quite dramatically.

- Some bugs might not be noticed by any users for months or years.
- Indeed, some bugs may never be noticed by users. That does not mean they are not problematic, as discussed in section 3.3.

Last but not least, a software producer can refuse to even recognise a problem as a bug, even when the user insists it *is* a bug. This is a difference of opinion – and clearly, who is right depends on the facts of the case, and the definition of “bug” in use for that particular product. Such disagreements can happen anywhere, including in open source projects.[2] However, in expensive contract work, in particular, if the contract does not tightly specify the project requirements, such disagreements can prove expensive for users.

This thesis does not propose to define the word “bug” once and for all. However, it is assumed that problems occuring as a result of a failure to identify a requirement, would, in at least some cases, count as a bug. An example is the requirement “do not allow the user to change the price to whatever amount they like” in an online shopping system, which was an overlooked requirement in at least one real-life system[3]. In hindsight, the requirement appears to be necessary for the system to be minimally fit for purpose, so the omission arguably counts as a bug. This would be so even if it was not entailed by any explicitly stated requirement – so it is too restrictive to define bugs only as failures to meet explicitly stated requirements.

3.3 Costs to users

Even if a bug is never noticed, it can still represent a potential future risk in terms of deficiencies in security, safety or performance. For example, a never-

[2]Indeed, this problem is far from specific to software projects – it affects other engineering projects, as well.

[3]The price could be changed by modifying some HTML on the client side, and the server accepted the manipulated price without checking it.

noticed security bug could be used to inflict enormous financial damage on a company (by stealing valuable trade secrets) or individual (by identity theft).

Many bugs are discovered and fixed early enough that they never appear in an official release of a piece of software. However, a sizeable number are not. Additionally, test releases are sometimes used for real work. A notable example of this is the "eat your own dogfood" philosophy of software development, in which a company makes it a policy to use recent versions of software developed by the company, for real work. This is done to improve the incentives for – and timeliness of – software quality assurance. Bugs in test releases can therefore have some economic impact, beyond just their cost to fix.

Moreover, serious security bugs can have an economic impact which is disproportionate to their quantity: an Internet worm only needs *one* remote code execution bug in order to spread to hundreds of thousands of hosts very rapidly [35].

Serious security bugs are an example of grave bugs which can appear in a range of software of different kinds. Software for particular application domains may exhibit other types of bugs whose severity demands special attention, due to the nature of the domain. For example:

- In a safety-critical medical system, what in other domains might seem like a simple usability bug can have grave consequences. An error display bug has, at least once, contributed to a fatal dose of radiation therapy being administered [9].

- In automated securities trading, in addition to any market risks faced even if the software functions correctly, users face further risks if the software has not been verified to be bug-free. A software bug could, in principle, trigger a huge buy, or an unnecessary "unrolling" of multiple trading positions. Depending on the severity of the results, this could cause significant financial and reputational losses – not just for the company in question, but perhaps also for other indirectly involved actors in the market.

3.4 Software project failures

In addition to the direct risks to users, the cumulative impact of diagnosing, fixing, and testing the fixes[4] for both informally acknowledged and formally acknowledged bugs, put together, can be a very significant drain on the resources of a software project. In many cases, this heavy overhead – or an unwillingness to pay for such a heavy overhead – can cause the entire project to fail.

Indeed, software project failures, and large cost overruns, are relatively common occurrences. A failed project can be defined expansively as one which is cancelled, or largely fails to meet its original expectations, or overruns in cost or time to such an extent that it turns out to be very poor value for money. Frequently, a project is a failure on more than one of those grounds.

Documented examples abound on the Internet (for example, at `http://catless.ncl.ac.uk/Risks`), and in the archives of trade publications, such as the UK magazine *Computer Weekly*.

Unfortunately, surveys conducted by interested parties, to establish the extent and impact of these phenomena, can be problematic. The oft-cited annual *Standish CHAOS Report* on software project failures has been criticised, because Standish has been unwilling to publicly detail even their survey methodology, much less their raw data – and what little they disclose suggests systematic bias [18]. This is not to say that the Standish reports are *necessarily* inaccurate in their conclusions – merely that the means by which they arrived at those conclusions are questionable.

While it understates the size of the problem, as discussed above, to collate publicly acknowledged and documented high-profile project failures, it does at least provide a solid lower bound. Large, multi-million-dollar project failures occur more than once a year [11]. Project failures occur in both in-house and subcontracted projects, and in both commercial and governmental contexts. They occur in a wide variety of application domains, including supposedly pedestrian

[4]Although it is important to test fixes thoroughly to check that they do indeed fix the bug, and that they do not cause regressions, this testing is not always done in practice. So the above is an optimistic description of the bug-fixing cycle, which is only sometimes accurate.

and well-understood – albeit not uncomplicated – domains like payroll.

3.5 Opportunity costs

In economics, if there is a situation where there are only two alternatives to choose between, and those alternatives are mutually exclusive, the *opportunity cost* of either alternative is defined as the value forgone by not pursuing the other. For example, if two relevant trade conferences are being held on the same day, on opposite sides of the globe, and a salesman attends one of them, he forgoes the opportunity to win clients and orders from the other conference. His expected[5] opportunity cost is the expected economic value of those forgone clients and orders, to his employer (or to him and his sales commissions, depending on from whose point of view it is being calculated).

Cost-effective methods for reducing the number of bugs in a system and/or their impact, can free up resources to be spent on other goals, such as additional product features. Thus, *not* using these methods implies an opportunity cost, in terms of less project resources being available to devote to new features. Of course, the cost savings might be instead ploughed into non-IT divisions of an organisation, but the possibility of more resources is there. This can be very significant: it can make the difference between a project which delivers hardly any of the promised benefits, and one which delivers many or all of them. The former case, unfortunately, happens surprisingly often.

3.5.1 Human errors and malfeaseance

In particular, if more resources are available, it may be deemed desirable to invest more of them in reducing the frequency and/or impact of certain types of human errors and malfeaseance (as opposed to developer errors). The former can themselves be very costly, and even if not directly technology-related, technology might still be used to mitigate them. Medical errors, such as incorrect

[5]This word is used in this paragraph in its statistical sense: the sum of each possible value multiplied by its probability.

dosages, surgical errors, and failure to observe best-practice hygiene procedures, have been blamed for hundreds of thousands of deaths and hospital stays in the United States alone. A US presidential task-force in 2000 estimated that they cost the US as much as $29 billion annually.

In the financial securities domain, the cases of Barings Bank (bankrupted by rogue trader Nick Leeson) and more recently, Société Générale (which alleges that it has been defrauded by an employee, resulting in billions of dollars in losses) and Bear Stearns (hit hard by overexposure to risky securities backed by subprime mortgages) have merited international media attention. These cases highlight the need for careful oversight, both regulatory and managerial – in a financial system whose complexities impel computerised assistance.

The opportunity costs, therefore, of not taking more effective action to reduce bugs, may be very large in themselves.

Note that it can sometimes be difficult to cleanly separate opportunity costs from "direct" costs – because bugs can make human error more likely, or magnify its impact. Regardless, however, it is clear that both the direct costs and the opportunity costs of software bugs are very significant.

3.6 Quantifying the costs?

Aggregate economic losses due to bugs are difficult to quantify with any degree of accuracy, because of:

- the difficult of counting bugs accurately (section 3.2);
- very significant and hard-to-quantify opportunity costs (section 3.5);
- the sheer scale and variability of the problem accross the economy;
- the fact that it is unknown just how many bugs can be avoided at a feasible cost.

However, in 2002 the US government's National Institute of Standards and Technology made a crude impact assessment, not including opportunity costs, and based on an extrapolation from a case study of one industry sector. NIST concluded that “inadequate software testing infrastructure” was responsible for costing US software producers and users \$60bn/year, of which \$22bn/year could be saved by improved testing (that is, following existing known techniques in testing). [46]

If these figures are correct, then, if a regime involving both testing and formal verification could be put into practice and made cost-effective, it could potentially save even more than \$22bn/year accross the US economy – since formal verification can find bugs that testing fails to reveal.

Chapter 4

Benefits of specifications and verification

A key motivation – perhaps the most important motivation – for formal verification is that it helps to catch bugs earlier in the development cycle. The sooner a bug is caught, the easier – and therefore cheaper – it is to fix. However, it provides other benefits as well.

4.1 Benefits of precise specifications

Having precise specifications of what a method or function does, is something which has benefits even in the absence of formal verification. These benefits are discussed below.

4.1.1 Exposing requirements deficiencies

Programming more rigorously can bring to light ambiguities or possible deficiencies in the project requirements at an earlier stage [8]. Suppose, for example, that transaction line items and totals in an accounting system are supposed to be representable as 32-bit unsigned integers. However, if there are two transactions of 4 billion yen each, that will cause an integer overflow in the total. Intuitively this should be a bug, but the specification might be written in such a way that it does not classify an overflow as a bug – in which case, the specification is probably deficient.

4.1.2 Clarifying APIs

API documentation does not always clearly state all the preconditions required, and guarantees offered, by the API. For example: does a method accept a null argument? Is it guaranteed to lock a particular object? If such questions are left unanswered – or if they are answered somewhere in the documentation, but overlooked by some users of the API – this can cause several problems:

- A user of the API may fail to prevent illegal arguments being passed in – not being aware that, for example, empty strings were not allowed. This could cause an unexpected exception to be thrown at runtime, or even undefined behaviour. The error might not come to light until an empty string is actually entered – or perhaps not until a later implementation of the API is used.

- A user of the API may engage in defensive programming, trying to second-guess plausible or key ways in which some current *or future* implementation of an API could fail, and trying to avoid such failures. This approach could lead to runtime inefficiency – for example, through excessive locking, or through doing data validation twice. More importantly, though, it is also likely to lead to decreased developer efficiency, as developers waste time re-implementing functionality, and/or worrying about whether they have to do certain things.

- A user of the API may examine the source code of the particular implementation (of that API) that he or she is currently developing with, in an effort to form a mental model of the preconditions and postconditions of a method. Alternatively, he or she may do informal black-box testing to achieve the same end – or a combination of the two. These techniques can be attractive because they can enable API users to better understand the particular API implementation they are working with. Unfortunately, relying on undocumented features of an API in this way risks breaking the code, if a later implementation, or a different vendor's implementation, of the same API is later used.[1] Note that the same effect can also occur if documentation exposes details (deliberately or not) which differ between versions, or between vendors.

- The cumulative effect of many (witting or unwitting) uses of undocumented features is "API lock-in" and "protocol lock-in". Lock-in is a phenomenon which is common in the software industry, and comes in various different forms. In the API and protocol forms, it becomes expensive for a customer to move away from one vendor, because they rely on applications which use a vendor's poorly-specified or undocumented features, which are difficult to precisely replicate by anyone else. Key players in the software industry – and EU Commission regulators examining Microsoft's behaviour – have therefore come to recognise the value of unambiguous specifications, available on so-called "reasonable and non-discriminatory terms", as an aid to interoperability [16].

4.1.3 Clarifying user interfaces

All of the problems in the above section can, *in principle*, also occur in relation to user interfaces and patterns of user interaction – analogously to APIs and code using those APIs, respectively. For example, a user might not trust that an Undo function in a particular program actually undoes *everything* correctly – because, indeed it demonstrably does not, let's say. In the absence of accurate documentation as to what *can* be safely undone using the Undo function, the user might perform numerous costly "Save As" operations, to ensure than their

[1]This was a well-known problem with applications for MS-DOS and Windows in the 1980s and 1990s, which resulted in Microsoft maintaining "bug-for-bug compatibility" between versions, for selected bugs, features and idiosyncracies.

operations can be undone – reducing productivity (both mental productivity and total productivity).

4.2 Side benefits of rigorous proofs

Formal verification provides additional side benefits over simply writing specifications. ("Side benefits" are those which are additional to the main desired benefit of formal verification – catching bugs earlier.) Some of these side benefits are discussed below.

4.2.1 Clear thinking

In thinking about how to implement an algorithm, or how to apply it, a developer can become confused. While pencil-and-paper reasoning, diagrams, and prototypes can all be useful aids from time to time, building the software in a formal verification environment helps the developer to be more precise, and to make conjectures and prove them rigorously.

Admittedly, there is also a risk that a formal verification environment might overwhelm the developer with details, causing confusion and making it harder to focus on the essence of the algorithm. In some cases, reusable libraries of theorems can help to reduce the level of detail, but not always. Alleviating this problem is a topic of current research.

4.2.2 Freedom to safely maintain and refactor

Many developers enjoy working on new code – and computer science education, with its tendency to focus on starting projects from scratch, tends to give students the impression that this is a developer's ordinary occupation. Yet, in reality, substantial amounts of time have to be spent maintaining existing projects – which can be a frustratingly slow, error-prone, and even stressful process. Maintenance is made more difficult when the components being maintained, or their interactions with other parts of the system, are hard to understand.

Indeed, users, developers or managers may be very cautious about modifying production code, for fear of introducing regressions, or confusing users. As a result, they may place limits on the nature, scope, or invasiveness of changes.

Formal verification cannot entirely assuage these fears – because, for example, a developer might still introduce a regression by erroneously modifying a specification (there will always be times when requirements change, so modifying specifications cannot be prohibited in general). What formal verification can do, though, is document – in a mathematically proven way – what the current system does. This has a lot of potential to make maintenance both safer, and more efficient.

In particular, refactoring can be made safer and more efficient. Refactoring is itself carried out for the express purpose of improving the readability and maintainability of code.

4.2.3 Freedom to safely optimise

Moreover, formal verification permits more flexible and safe optimisation. Suppose that a verified main function `main` calls – directly or indirectly – a verified function `f`. Suppose, further, that neither the specification nor the correctness proof of `main`, nor the proofs it depends on, reference the body of the function `f`. Then, the body of the `f` can be replaced with any other implementation which satisfies the type and specification of `f` – without affecting the functional correctness of `main`. A compiler writer, or library writer, or optimisation rule writer, can choose to replace data structures and algorithms with ones that are expected to be more efficient, under these conditions.

For example, suppose that `f` is a function which is specified to sort a collection of comparable items into ascending order – but whether `f` removes duplicates is unspecified. That is, an item is in the output if and only if it is in the input, but if there are n copies in the input there might be n copies in the output, or only one copy in the output. *Hypothetically*, suppose it is simpler to write proofs in terms of sets (since mathematically, $\{x, x\} = \{x\}$), but faster in some situations to use an implementation involving lists (because duplicates do not have to be found and removed) – or perhaps vice-versa. Formal verification allows the data structure to be chosen for convenience of proving (or programming), and then automatically (or manually) optimised from a set to a list, or a tree, or whatever is appropriate.

Chapter 5

Formal verification in context

Full-blown verification is one approach for reducing the number of functional correctness bugs in a piece of software. It is not the only possible approach, however. The following additional approaches can be used instead of formal verification.

It is important to note, however, that they can also be used alongside formal verification, and in conjunction with each other: that is, these choices are not mutually exclusive. For example – as mentioned below – testing can be used to do a rough check on a piece of code before subjecting it to verification. Exhaustive testing and/or model checking can even be integrated into formal verification tools as a tactic (see section 5.1.1).

5.1 Testing

Testing is one of the most basic techniques for finding bugs. Unit tests test individual functions or components of a system, and integration tests check that a whole system works. Both types of test can be facilitated by specifications: for example, software like QuickCheck automatically generates random unit tests from specifications [13], while JMLC inserts runtime assertion checks into the code of a system [8].

Specification-derived tests can be used to weed out bugs from a unit or system, before (or instead of) putting in the effort or time needed for formal verification [8].

5.1.1 Exhaustive and non-exhaustive testing

It is occassionally feasible to exhaustively test a piece of code. For example, suppose a given pure function only has five allowed input values, the function takes a negligible amount of time to evaluate, and the property of the output to be checked can be ascertained in less than a second. Clearly, then, a brute-force exhaustive test (which just tries each input in turn) can be performed in less than five seconds.

A valid exhaustive test, if it passes, exhibits a valid "proof by cases" of the desired property. Thus, in this limiting case, testing is equivalent to formal verification – and in particular, to automated proving. Consequently, a proving tool could, in principle, provide an "automated test" tactic, to be applied in these sorts of situations.

However, typically code processes data with a very large permissible range, such as 32-bit integers, which generally makes brute-force exhaustive testing impractical. Testing, therefore, generally relies for its effectiveness on the following vague and informal assumption:

Small sample assumption If a piece of code works for "a few" extreme values[1] and "a few" non-extreme values – and rejects "a few" values it is required to reject (if any) – it is likely to be correct.

What counts as "a few" for the purposes of this assumption varies quite widely – some developers generate a large number of random test cases (but still do not manage to exhaustively test their code). Regardless of that, this assumption is often essentially valid. It, in turn, partly relies on a deeper assumption:

[1]"Extreme value" here means a bound of some relevant partial order.

State space collapse In many situations, a large class of states "behave in the same way", and so even though the theoretical state-space is large, those classes of equivalent states can each be collapsed into one *effective state*. (In terms of formal reasoning, that likewise means the number of cases that need to be considered is small.)

Again, this latter assumption is often true, when dealing with realistic systems. However, it can be problematic, as discussed below.

5.1.2 Testing polymorphic higher-order functions

The work described by this thesis inherently involves verifying various properties of certain *polymorphic higher-order functions* – as is made evident in Chapter 10. *Higher-order functions* are functions, one or more of whose arguments are themselves mandated to be functions (or perhaps data structures containing functions that may be invoked). If a function is *polymorphic* in a functional argument `f`, that means that it accepts functions of several different types[2] at `f`'s argument position. Note that this meaning of the word "polymorphic" is used in functional programming, and is distinct from the meaning which the word polymorphism takes on in the context of object-oriented programming (for example, in Java).

It would be remiss not to mention, for the sake of comparison, how polymorphic higher-order functions can be tested (as opposed to verified).

If an argument of a function ranges over all possible one-argument functions, whatever their type, it is possible to simply pick one-argument functions arbitrarily, and supply them as the test values. Arbitrary test cases are better than nothing, even though the human selecting the test cases will probably be biased. Happily, though, if the programming language in use enjoys parametricity properties (see Chapter 9), it may be possible to exhaustively test the function. This seemingly-impossible feat can sometimes be performed in part by supply-

[2]Function types are of the form "A to B" (written `A -> B`) where A is the domain type and B is the codomain type.

ing the identity function on a particular type as the functional argument – if the language's parametricity properties entail that if the function works correctly for that argument, it must work correctly for all possible arguments at that argument position [15]. Note, however, that is still necessary to supply test values for any *other* arguments that the function may have (because partially-applied functions yield functions, which must themselves be tested). Furthermore, it may not be feasible to exhaustively test *those* arguments, even if the functional arguments can be "tested" with just one test function each.

If an argument of a function has a *constrained* function type, on the other hand (such as "`a -> m a` where `m` is some monad"), then the language's parametricity properties may not be applicable, and it may be necesary for the tester to supply some arbitrary functions meeting the relevant constraint(s).

5.1.3 Limitations of testing

As with all alternatives to verification, there is a trade-off involved between correctness and up-front investment. Formal verification requires more up-front investment than testing, in the form of developer training, specification development and proof development. Nevertheless, it is a more powerful way of catching bugs. A test can usually only verify a miniscule fraction of the total number of possible cases, whereas a valid proof verifies *all* cases (subject to the caveats in section 2.3). In practice, the small sample assumption ameliorates this problem somewhat; however...

- Testing, in practice, tends to miss concurrency-related bugs sometimes – for example, bugs due to race conditions. This is an instance of a more general problem: non-deterministic code can be hard to test reliably – especially if the source(s) of non-determinism are not under the complete control of the tester. (See also section 5.3.1.)

- In the absence of a formal specification, there is no guarantee, in general, that the tester has considered all valid inputs. It is impossible to automatically determine whether a failure to test a class of inputs is due to those inputs being forbidden (by a precondition), or whether it is due to an oversight by the tester. This particular problem can be "defined away", however, by using automated test case generation software such as QuickCheck, and specifying that the test case generators *constitute* formal specifications.

- The judgement of whether a particular state space collapse actually occurs, or whether there are actually more possible states than it appears, is a necessary *assumption* made by a tester. In black-box testing, the code cannot be looked at, so the effective state space can only be guessed at. In white-box testing, by contrast, the code can be read, but the tester could be misled by it – after all, if the tester was 100% accurate at understanding the code, there would be no need to test it! Even under conditions where the tester is under very little time pressure, testers are human and make mistakes. Moreover, in a commercial environment, such conditions are rare: typically the tester *will* be under significant time pressure.

- Automated, random generation of test cases [13] and sequential test generation both have the potential to reduce the risk of leaving a relevant effective state untested – compared to merely employing a handful of test cases dreamt up by the tester. However, neither can entirely eliminate this risk, in general. For example, it would be unwise to use test query strings *entirely* generated by a random string generator to check that a website was free of SQL injection vulnerabilities in HTTP GET requests, because the probability of finding an exploit within a reasonable time frame would be small.

It is therefore risky to rely on the testers always making the right assumptions – especially for safety-critical systems.

5.2 Code inspection

Code inspection can be seen as an informal analogue of verification, with varying degrees of informality. The original developer – or, preferably, a different developer, to provide a "fresh pair of eyes" – reviews the code, and reasons informally about what the code appears to do, what the code ought to do, and whether the former matches the latter. This can be done with or without specifications – although specifications are clearly of use in determining the intent of the code.

Various practices which involve some degree of code inspection (code review, pair programing, and refactoring) have been incorporated into some agile development methodologies. An example is the Extreme Programming methodology [5], which combines practices such as pair programming and refactoring with testing, and other practices, to improve software quality and development efficiency.

Code inspection can catch bugs that testing misses, and vice-versa. As with testing, code inspection can be used as a sanity check prior to, or instead of, verification. Like testing, it is not sufficient to assure correctness.

5.3 Model checking

Model checking is a field of study which encompasses all algorithms for checking whether given predicates are satisfied by mathematical models of systems[3]. For example, given a model which takes an integer as input, and outputs an integer, the predicate to be checked could be "the output is always the input multiplied by two".

In general, predicates can refer to any state of a model, not just values that are designated as inputs or outputs. However, in the context of checking functional programming structures such as monads – the subject of this thesis – all predicates are formulated exclusively in terms of inputs and outputs.

[3]Some authors instead refer to the predicates themselves as the model.

If a model does not have inputs, this is equivalent to a model which takes an input but then ignores it. Likewise, if a model simply accepts or rejects an input, acceptance is equivalent to outputting True, and rejection is equivalent to outputting False. Thus, it can be assumed, without loss of generality, that the model has both input(s) and output(s).

A trivial algorithm for model checking *deterministic* models just iterates through all possible inputs, in some well-defined order, and tries to execute the model applied to each input, in turn. If an input disproving the property is found, the algorithm immediately stops and returns "disproved". If the algorithm exhausts the set of allowed inputs, it returns "proved". Otherwise, it continues executing indefinitely. Users of a model checking tool can, of course, choose to stop the model checking after a certain period of time, if they believe enough cases have been checked to provide sufficient confidence that the property holds.

More sophisticated algorithms may apply optimisations, such as parallelisation on multiprocessor systems, removing redundancy in the model, and symbolic evaluation of the model (in the context of model checking imperative programs, symbolic evaluation is termed abstract interpretation).

Non-deterministic models can be – at least for illustrative purposes – translated into deterministic ones, by converting the random variables feeding the non-deterministic state transitions into auxillary inputs.

As with proof-oriented tools, in order for the model checking to be valid, any necessary translation from code into the language of the model checker needs to preserve all the relevant semantics. For reasons of efficiency and correctness, this translation should be automated.

5.3.1 Model checking as a form of testing

Model checking of software can be seen (approximately!) as a form of automated testing, in which the model checker generates test cases automatically, and tries to optimise the testing process by use of techniques such as those just mentioned.

The model checker and its translator may have bugs – but so may the runtime environment and the compiler in a traditional testing scenario. However, the risk of bugs in the model checking tools causing bogus test results may be higher – especially if they are immature.

Another key difference with automated testing is apparent when model checking concurrent code. The model checker can control the order of interleaving – whereas this control is typically not easy to achieve in an ordinary testing framework, such as JUnit or QuickCheck.

However, taking into account these reliability and concurrency-related differences, similar comments which apply to testing in sections 5.1.1 and 5.1.2, and to automated testing in particular in section 5.1.3, apply to model checking. For example, for problems where it is practical for the model checker to be allowed to run until it covers all possible cases, model checking is essentially equivalent to an exhaustive automated test – which itself is equivalent to an automated proof.

In addition, model-checking tools can sometimes make it practical to exhaustively check a model of a piece of code, even when that code cannot practically be exhaustively tested. In particular, they can apply analyses to the model (such as symbolic analyses) to reduce its complexity. In the ideal case in which the model checker is bug-free, this model reduction avoids making erroneous assumptions about effective states – which, as stated above, is a problem when using human testers.

However, an exhaustive model check is not always practical – in which case, again, formal verification can provide greater confidence than model checking.

5.4 Static type systems

Static type systems can be used to express and verify certain properties of the inputs and outputs of a function. For example, in Java 5 and upwards [20], the type `List<String>` expresses two constraints about an object of that type:

1. the object's class must implement the `List` interface
2. the list represented by the object should only contain `String`s.[4]

5.4.1 Dependent types

The more expressive the static type system, the wider the set of properties that can be verified at compile-time. An example of a programming language with a very expressive type system is Epigram [2]. Epigram has *dependent types* – types which can depend on values. Some proof assistants, such as Coq [1, 7], also have dependent types. For example, in Coq, the type `forall n:nat, list n -> list (S n)` is a dependent function type (also known as a *dependent product*), which entails that:

- a function of that type has two arguments;
- the second argument must be a list whose length (the argument to the list type) is given by the first argument – so the *type* of the second argument depends on the *value* of the first argument, and so this a dependent type;
- the function must yield a list one element longer than the second argument (`S` is the successor constructor for natural numbers in Coq).

In principle, dependently-typed languages, such as Epigram, are isomorphic to sufficiently powerful logics for formally verifying code. That is to say, both kinds of languages can state the same class of theorems, and express the same class of proofs (in practice, however, technical limitations may cause this correspondence to be less than complete).

The well-known isomorphism which establishes this correspondence is known as the **Curry-Howard isomorphism** (see Appendix A).

However, dependent types can be convenient for expressing non-trivial constraints on types. Without dependent types, a formalisation of category theory

[4] In Java 5, this constraint cannot be guaranteed to be true at compile-time, because of the possible presence of legacy (non-generic) code at runtime – so it is checked at runtime.

(such as that developed in this work) can allow awkward nonsensical objects to be described, which would have been ruled out by suitable dependent types [38].

5.5 Separation of concerns

Separation of concerns techniques, such as modular programming, object-oriented programming, aspect-oriented programming and functional programming can be used to reduce the effective complexity of a piece of code (see page 42). The basic idea of separation of concerns is that it can be helpful to separate code which deals with distinct concerns or goals into separate units (classes, functions, aspects, etc.), rather than leaving it unnecessarily "tangled up" and intertwined with other code (which is what novice programmers often do). As a result, the code often becomes easier to maintain – and more relevantly, it can become easier to test, model check or verify as well.

Chapter 6

Verified programming with monads and functors

6.1 Non-strict languages and pure functions

Monads are abstractions that are frequently used in *non-strict* functional programming as a means of cleanly separating concerns (see section 5.5 above). Before explaining what a monad is, it may be useful to give some context.

A *strict* language is one which guarantees to always (or, always unless otherwise specified) evaluate the arguments to a function call before evaluating the function call itself. By contrast, implementations of a *non-strict* language are free to evaluate the arguments of a function call subsequently to evaluating the function call itself. However, they do not, in general, *have* to do so – this allows for flexible optimisation by compilers. All mainstream programming languages today are strict; Haskell is the most prominent of the non-strict programming languages.

The notion of a non-strict language is closely bound up with the notion of *purity* – the idea that functions should not have any observable side-effects, beyond CPU and memory usage. *Impure* functions are, in general, more complicated to reason about [47], for several reasons:

- They might call other functions which might modify non-local variables that they are using (if any);
- Other threads or processes might modify those variables as well – unless the function is suitably synchronised.
- Aliasing (two references or pointers pointing to the same memory location) can affect the results of impure functions, but not pure functions. This is because impure functions can write to variables, but pure functions cannot (that would be an observable side-effect).
- The output of an impure function may depend on any value reachable by a global variable, some user input, or the results of a call to a database or the operating system.
- In the context of formal verification, if an impure function were to be invoked within a specification for another function, this could create ugly paradoxes. For example, the impure function could change the system state such that the other function could no longer work – but this system state change would only happen if it was actually invoked (for example, by a runtime assertion checker). In other words, the very act of checking the function's correctness would cause it to behave incorrectly, even though it behaved correctly previously! For reasons such as this, JML [8] requires functions invoked in specifications to be declared to be pure.

Requiring functions to be pure thus allows non-strict evaluation to proceed safely. If a function were not pure, a change in evaluation order could cause a change in its behaviour, which would mean its semantics would be undefined in a non-strict language.

Nonetheless, side-effects such as I/O are obviously necessary in real systems. Monads are one solution to the challenge of marrying pure functional programming with side-effects. Monads are abstractions which allow a sequence of "instructions" – which may have side effects – to be bundled up into a single value (which may or may not be seen as atomic by the rest of the program, depending on what sort of monad the monad is). In section 6.3.2, monads are explained in more detail – including why they do not break purity.

Side-effects such as I/O [41], foreign function calls [10], and mutable variables [23], have all been implemented using monads. However, monads are not solely used for implementing side-effects – for example, they have been used for passing around configuration data[1].

In principle, monads are never the only possible implementation choice, even in non-strict languages [4]. However, they often facilitate separating a concern to some degree, so that it is less entangled with unrelated code – which often makes the code easier to read, and simpler to reason about.

Furthermore, by programming directly in a proof assistant such as Coq, a developer is able to program with, and reason about, pure functions directly. This contrasts with the approach of writing code and specifications in an impure language, such as Java with JML, and then automatically translating that into verification conditions (VC-generation) in the language of a prover such as Coq. VC-generation must take into account all of the complications that an impure language entails – because Coq's Calculus of Inductive Constructions is a pure language. The developer does not have direct control over how the side-effects, aliasing, etc. are represented, and the verification conditions often become large and unwieldy – precisely because of the complications of impure functions.

However, the VC-generation approach does tend to result in verification conditions corresponding to individual "steps" in the code, rather than entire functions. These smaller steps are arguably more convenient to work with, so recent work has attempted to combine the benefits of both approaches within Coq [43].

[1]See, for example, the HEAD revision of the gitit wiki software, which can be found at http://tinyurl.com/naue5v (`http://github.com/jgm/gitit/tree/master`)

6.2 Monads and category theory

Monads were first used in a practical programming context [49] in the language Haskell [40]. They derive from *category theory*, a branch of mathematics. Category theory is a theory about rigorously-defined commonalities between many different kinds of mathematical structures, which allows certain valid results from one field of mathematical enquiry to be translated into valid results in another. This can facilitate proofs, by making it possible to convert a problem in one field into a easily-solvable problem in another field. Category theory is also a field of enquiry in its own right.

A *category* consists of a collection of *objects*, and for every pair of objects (a, b), a collection of *morphisms* from a to b; it must also satisfy certain laws, as detailed on page 83. Mathematically, monads can be constructed in many different categories, but existing work on using monads in functional programming always constructs monads in just one category:

LANG The category whose objects are precisely all the possible[2] *types* in the functional language in question, and whose morphisms for the pair of objects – that is, the pair of types – (a, b) are all the possible *functions* from a to b in that language.

Other concepts grounded in category theory – functors, applicative functors, monad transformers, and the author's own invention, applicative functor transformers – are formalised in this work. All of these are useful, and indeed all but the latter *have* been used, in practical functional programming – they are not merely for mathematical modelling or proofs. The next section presents an example of where monads can be useful, and then explains what a monad is from a programming perspective.

[2]That is, predefined by the language, or definable in the language.

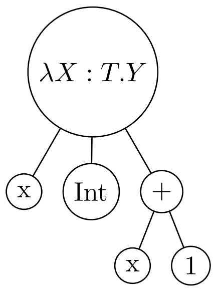

Figure 6.1: A simple Haskell expression represented as an AST

6.3 Monadic programming

6.3.1 A motivating example

An example of applying monadic programming to solve a programming problem in Coq may be helpful to illustrate the use of monads. Suppose that an abstract syntax tree (AST) data structure has been defined in Coq, to represent the parsed form of some particular language with bindings, such as Haskell. Suppose that this AST data structure admits values like the one shown in Figure 6.1, which represents the Haskell expression `(\(x :: Int) -> x + 1)`.[3] This expression will be used as a running example in the next paragraphs.

Now suppose that a generic AST walker function `g` has been written (or generated) which assists a programmer in doing some well-defined traversal, such as an inorder traversal, of all the subexpressions in a given parsed expression. It could, for example, be a "one-level recursor" that simply applies a function (supplied as an argument) to every direct subexpression, and yields the resulting expression. Such simple one-level recursor "building blocks" could be applied in a recursive manner to do things like removing unwanted parts of the tree (such as type signatures – see Figure 6.2[4]) or replacing specific nodes with other nodes (for example, as part of the process of performing β-reduction).[5]

[3]The inner parentheses are required to indicate that `Int` is the type of `x`, rather than the type of the return value of the λ-expression. This notation uses an extension to Haskell called PatternSignatures.

[4]Transformations on leaf nodes are omitted in this Figure.

[5]The recursor functions automatically generated by Coq (i.e. those with the suffixes `_rec` and `_rect`) are less suitable for these kind of constructor-insensitive tasks, as they require separate functions to be supplied for each individual constructor. Even when these recursor

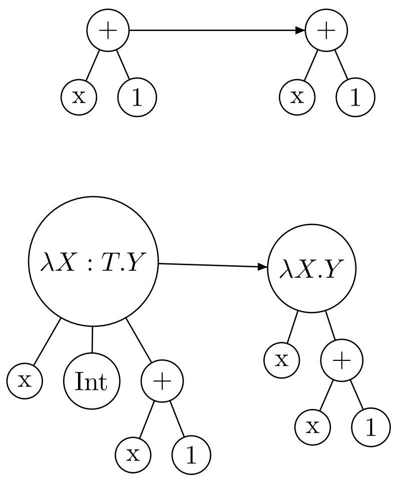

Figure 6.2: An inorder traversal which removes type signatures

However, such simple building blocks would not be suitable for all kinds of traversals. Whilst it would be possible to implement various different support functions for other kinds of tree traversals, it would be desirable to avoid needlessly duplicating functionality implemented by `g`. (It is a generally sound principle in programming to avoid duplication of code: duplication introduces more possibility for error, and means that more code has to be changed if it becomes necessary to modify the duplicated elements.)

In particular, if `g` is written to take an expression argument and a function argument of type `Subexpr -> Subexpr`, it cannot be reused to do a computation that needs information about the bindings in scope, without redundantly reimplementing part of the functionality of `g` (the part that deconstructs typed binding expressions and *reconstruct*s them after applying the function argument). This is because, as Coq functions are pure, it is not possible to use a mutable global variable to keep track of which bindings are in scope. An example of such a computation that needs scope information is transformation from a name-based representation into a representation which uses de Bruijn indices to represent bindings.

functions are appropriate, they could also in principle benefit from being used in conjunction with monads, for example for I/O.

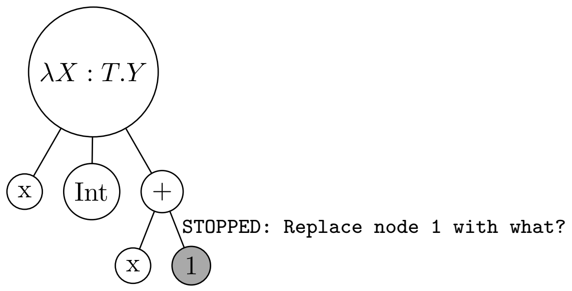

Figure 6.3: Stopping during a search-and-replace tree traversal to ask the user what to do next

If `g` is rewritten to take instead a function argument of type `Subexpr -> Bindings -> Subexpr`, it still cannot serve other needs. One example of a need that cannot be served is computing some value (such as a count of the number of lambda expressions in the tree) *separate* from the tree itself as the tree is traversed. Another is interactively asking the user what to do next during the traversal, and doing different things depending on the user's response (Figure 6.3). A flexible solution which allows `g` to act as a foundation for various different needs to be served is to make `g` monadic – that is, to make it yield a monadic *computation*.[6]

6.3.2 Monads dissected

A monadic computation can be viewed, conceptually, as an imperative program. The particular monad chosen has a set of functions which can be called, which each yield a representation of a permissible action in that monad. For the I/O monad, these actions include "write a line to standard output" and "open a file". However, actions do not have to have externally-visible effects – the Reader class of monads (see below) have an action "get the encapsulated value".

When a monadic function is evaluated, it does not *perform* an action, but rather yields a computation *representing* the action to be performed – conceptually, a little imperative program. At some later point in time, the computation may be performed – either by some other function, if the computation can be

[6]This would mean making the function passed to `g` monadic as well.

performed in a pure way, or by the runtime system itself, if it cannot.[7] Continuing with the imperative program analogy, this corresponds to executing the program. However, note that the computation might never be performed at all. For example, in Haskell, a computation `comp` might be referred to only once, by a piece of code reading `when (x == y) comp`, which essentially means "perform `comp` *only* if `x` is equal to `y`". If `x` is never equal to `y`, `comp` will never be performed.

Monadic computations can be sequenced together using the `fmap` and `join` functions – or equivalently, just using the `bind` function (written `>>=` or `=<<` in Haskell). The latter three operations are available for every monad, and allow information to be passed from one subcomputation to another. Other common functions exist that can also be used for sequencing, but they are effectively convenience functions that can be defined in terms of the previous three. Once again, sequencing two monadic computations together in this way does not perform either of the actions that they represent; it merely creates a new computation whose effect is the first one followed by the second. In the imperative programming analogy, this roughly corresponds to generating a new program which runs the first program, takes the "return value" yielded by that program, and finally passes it to the second program as an argument. (Technically, this is not quite accurate – computations do not take arguments; the argument is taken by a function, which then yields a computation.)

Haskell also has a special notation for monads – the `do` notation – which can be easier to read, as it allows the programmer to avoid explicit `bind` invocations.

The main concrete example of a monad used in the remainder of this thesis is the Reader monad (also known as the Environment monad). The nature of a reader computation is very simple: all subcomputations optionally read from the same constant (i.e. immutable) store and then return some value. The

[7]In Haskell, programs can also use "unsafe" functions such as `unsafePerformIO` to run impure monadic computations from a pure function, but these functions are marked as unsafe in their names, and should be used with caution. Faithful analogs of these functions cannot be defined in Coq – without radically redefining the representation of a pure function such that all pure functions have to be re-encoded into the new representation, which is outside the scope of this work.

constant store in question might be, for example, an application configuration data structure which must be read by many different parts of the application.

The Haskell `IO` monad is another example of a monad, this time for I/O. It has a broad range of methods which allow operations like reading to and writing from files, interacting with the user, etc. `IO` computations optionally perform such actions and then yield some value.

This pattern generalises: computations in all monads first optionally perform some monad-specific action and then return some value. However, for some computations, the value returned can be the dummy value `tt`, the sole value of the `unit` type (`tt` and `unit` are Coq terms, and are both written `()` in Haskell). This allows effectively nothing or (in C/Java parlance) "void" to be returned.

Monads have been widely regarded – including by the author – as one of the most difficult concepts to understand when learning the Haskell approach to functional programming. Understanding monads is often a stumbling block for new Haskell programmers. It is not a goal of this thesis to provide a tutorial-style introduction to monads, but fortunately many such introductions are available free online. These include the Haskell Wikibook chapter on monads, which begins at http://tinyurl.com/ewuzt (`http://en.wikibooks.org/wiki/Haskell/Understanding_monads`) , and Jeff Newbern's comprehensive monad tutorial at http://tinyurl.com/3az8zd (`http://www.haskell.org/all_about_monads/html/index.html`).

Part II

Design and implementation

Chapter 7

Selection of implementation language

This chapter summarises some of the factors which were looked at when choosing which implementation language to use.

This work has as its goal full-blown verification – as defined in section 2.1 – of category-theoretic entities such as monads. Thus, languages which require particular rules to be observed, without verifying those rules, do not qualify as suitable languages for implementing this work. In particular, the dependently-typed programming language Cayenne [3] does not qualify, because its terms only translate into valid proofs under the Curry-Howard isomorphism if their evaluation always terminates – a condition which Cayenne does not verify [17].

Furthermore, systems which are intended for the verification of specific classes of properties are probably not the most suitable for this work. This is because they are not sufficiently general-purpose to support the extraordinarily wide scope of use of monads in programming. An example of such a system is Dependent ML, which only supports properties written in terms of integers (such as the length of a list).

7.1 Concoqtion

Concoqtion [17] is a hybrid system based on *indexed types* (which have also aptly been called *simulated dependent types* [26]) instead of dependent types. Concoqtion fares better than Cayenne in the present analysis, because it uses different syntaxes for (roughly speaking) proofs and programs – Coq for proofs, and an extension of MetaOCaml for programs – so it inherits Coq's (hopefully) sound proof-checking kernel. However, it is still a partial correctness checker, because a limitation of indexed types is that termination proofs of code in the programming language (as opposed to the proof language) are not supported. To create such a proof, it would be necessary to model all of the MetaOCaml Concoqtion code, and its evaluation order, in a prover. Concoqtion does not do anything like this: it uses Coq merely to reason about the *indices* in indexed types.

Nevertheless, partial correctness checkers like Concoqtion can be used for full-blown verification of partial correctness properties – that is, properties that must be satisfied only upon termination. In principle, a separate heuristic termination checker could then be used to deal with common terminating programming styles, for example – if code termination was deemed important enough to verify, and not just test for.

A more serious limitation, however, is that developers using indexed types must "write twice" – in two different syntaxes – any types or functions that they want to refer to at both the code and the specification level. While such duplications could be automated, at least to some extent, that is not a very clean solution. As argued above, in a different context, having two distinct syntaxes in a system is not ideal, for readability reasons.

7.2 Epigram

Epigram [27, 28] is another dependently-typed programming language. Like the groundbreaking object-oriented programming language Smalltalk, it has an unusual and novel user interface for programming in, which is intended to be considered integral to the language. However, its user interface is very different from Smalltalk's. Epigram's user interface uses dependent type information to assist the developer, by – for example – supplying skeleton code for pattern matching. This can be seen as another kind of tactical progamming, in which the code artifact that is saved and edited is Epigram code [36], rather than a proof script consisting of tactic applications, as in Coq.

However, Epigram 2 has not yet been implemented, and Epigram 1 is an experimental, incompletely-documented language with few libraries. Thus, it was not investigated any further: it would have been too risky to create a lengthy development in either version of this language. In particular, there would have been a risk of encountering some fatal limitation or bug at a late stage, when it might have been too late to learn a new system and port the development over to that system.

7.3 Coq

Coq, by contrast, is a mature proof assistant with several libraries of theories and tactics, in which significant theories have been developed. For instance, a refinement of the proof of the Four Colour Theorem has been formalised in Coq [19]. Coq's underlying language for proof terms, the Calculus of Inductive Constructions, is mature and well-studied. All known soundness issues that have been discovered in CIC, or its implementation in Coq, have thus far been fixed, at the time of writing.

Chlipala [12] opines that "In the not-so-distant past, Coq was clunky to use and infeasible for real programming. Today, it is mature and reasonable to use for carrying out non-trivial certified programming projects."

Moreover, the approach of operating entirely in Coq, and using program extraction [6, 1] to generate working source code in OCaml or Haskell, offers an intriguing and radical possibility which Concoqtion, at least, does not support.

That is, generating code (in the language of Coq) using proof tactics. This technique – called here *tactical programming* – is used in the present work, and a technique for verifying some types of code generated in this way is given in section 9.2.

Taking into consideration these and other (more minor) issues, the decision was made to use Coq for this project.

Chapter 8

An idiom for type classes in Coq

8.1 Classes, type classes, modules and records

"Monad", "Functor", and other category-theoretic entities formalised in this work, are abstractions, which each have multiple instances *with different behaviours.* For example, the monad abstraction has instances such as the Reader monad and the Writer monad, which behave differently. Importantly, "Monad", "Functor" and so on, each also possess a set of *laws* which each of their instances are required to satisfy.

Rather than simply defining these behaviours and laws, and proving the laws, in an ad-hoc way, it is advantageous to declare these abstractions using some kind of coherent structure. Putting the laws to one side for a moment, several ways of structuring such abstractions and their instances in a computer program have been developed in computer science, including:

1. Object-oriented programming (OOP), as implemented in Java [20]. In Java, the abstractions would be represented as either *abstract classes* or *interfaces,* and the instances would be represented as non-abstract *sub-*

classes of those classes. Note that because the instances have different behaviours, they should not in general be represented as objects, but rather as classes.

2. Type classes [50], as implemented in Haskell [40], in which the abstractions are represented as *type classes* (which have methods, possibly with a default implementation) and the instances are represented as *instances* of those type classes (which provide implementations of any methods which were not given a default implementation);

3. ML-style modules [51] (also known as structures), as implemented in OCaml and Coq, in which the abstractions are represented as *module types* and the instances are represented as *modules*;

4. Dependent record types – which in Coq are just dependent inductive types with one constructor and some fields. These fields are eliminators corresponding to each constructor argument. In this approach, abstractions are represented as *dependent record types*, and their instances are represented as *dependent records* (that is, values inhabiting a dependent record type).

In this work, dependent record types are used to simulate type classes: that is, the fourth approach is used to implement an analogue of the second approach. In fact, another feature of Coq – implicit coercions – is used in conjunction with dependent record types, to implement a notion of inheritance – as will be explained below.

This work also extends the notion of type classes to include *laws* as well as methods.

Below, the reasons for the choice of type classes as the semantic building-block of this work are discussed.

8.1.1 Object-oriented programming versus type classes

Object-oriented programming – at least, in its Java implementation – does not support dispatching a method call based on its *return* type, rather than the type of its argument(s). However, type classes, as implemented in Haskell, do. This is a useful feature to have when working with monads – especially in conjunction with type inference – and it is a feature which programmers familiar with monads and coming from a Haskell background would expect.

8.1.2 ML-style modules versus type classes

Note: Nothing in this subsection should be construed as a criticism of Coq *top-level* modules, which are simply a way to structure developments into multiple files.

ML-style modules, as implemented in Coq, are essentially a form of generative programming. This means that they are implemented using inlining rather than dictionaries (which are closely analogous to virtual method tables or vtables in the OOP world). Any resulting object code duplication risks hurting more than it helps performance on modern hardware, due to cache misses and (potentially) increased swapping – although of course, some applications will benefit.

When using multiple monads in the same scope (which is not uncommon in monadic programming), monads implemented as modules would have to be manually disambiguated from each other, using explicit qualification or renaming, which is awkward and verbose [51].

Modules use structural matching, rather than nominal matching, to determine whether a module is an instance of a module type [51]. This means that they add a possible source of error: if two module types have identical signatures, one can be used where the other was intended. Although such errors seem unlikely to happen very often – and should be caught in at least some cases by verification – in a formally verified development it is unwise to open the door for errors to creep in.

In the author's view, the most serious problem with modules in Coq is that type variables from the context are not allowed to be passed in as module parameters, except if those type variables are *themselves* declared by a module.

This means that outside module declarations, parameterised modules can only be instantiated at particular concrete types (which makes sense, considering that Coq modules are generative).

Lastly, choosing type classes rather than modules to implement monads, means that it should be (hopefully) slightly easier for developers who have already programmed with monads to understand the present work.

8.2 Design of the type class idiom

Coq 8.1 (which was the latest release when this work was done) provides modules, but not type classes. Therefore, this work develops an *idiom* [29] for abstractions in Coq that is somewhat similar to type classes in Haskell, by using dependent record types and implicit coercions. This idiom may be considered a *prototype* of a design pattern, since design patterns are supposed to be validated by usage in multiple production projects [21], and this idiom has not been. Currently, it exists merely in the form of some implementation guidelines, given in section 8.3, since there was no time to implement a Coq language extension for it.

In this section, the features offered (and not offered) by this type class system are discussed at an abstract conceptual level, without very much emphasis on the translating of type class concepts into the language of dependent records. The translation is detailed with in section 8.3.

As some code samples are introduced in this section, Figure 8.1, which shows exactly which source files depend on each source file, may be useful to get an idea of the overall structure of the project. Each arrow from A to B indicates “A imports B”.

However, unfortunately, the only examples of the use of this type class idiom that have been actually *implemented* in this development are category-theoretic ones. Thus, there is almost a cyclic dependency between the present Chapter and Chapter 10, because in order to understand the Coq code referred to and excerpted here, it may be necessary to understand the type classes in question – but in order to understand them, it is necessary to first understand the type class idiom!

Future work might again use this type class idiom for different and perhaps simpler type classes, which might make the idiom easier to understand. Another possibility for future work is to reify this type class idiom as some kind of Coq language extension, which might make code written using it a little easier to understand.

8.2.1 Type classes and instances in the abstract

Conceptually, in this system, a type class consists of:

- one or more formal type parameters. One of these is typically used to store the "primary" values on which the instance operates. In the case of monads, this is the type used to represent the monadic computation. These type parameters consist of:
  - a name
  - a type for the type – for example, type `Set`, which means the actual parameter must be a type with no arguments, or type `Set -> Set`, which means it must be a *type constructor* with one argument. In Haskell these restrictions on types are called *kinds*, but Coq has a more expressive infinite hierarchy of *type universes*.
- one or more method declarations, which consist of:
  - a name
  - an optional coercion declaration (see section 8.2.4)
  - a type, which can be a function type;
- zero or more laws which are required to hold for those methods. Recall that Coq's Calculus of Inductive Constructions is a pure functional language,

so functions in Coq cannot have side effects. Thus, the laws are always mathematical relations or predicates on the results of applying methods and/or other functions to zero or more arguments.

An instance of a type class conceptually consists of:

- a type argument for each of the type parameters of the type class;
- implementations of each method declared in the type class;
- proofs of each law declared in the type class, for this instance.

WARNING: It is vitally important not to confuse these instances with instances in object-oriented programming – also known as objects. Objects represent a bunch of data about a particular thing, whereas type class instances are more like non-abstract classes in OOP. For example, in the type class approach, there is a type class for all monads, an instance for each kind of monad (reader, writer, etc.), and then there are *values* representing each monadic computation. These latter *values* are analogous to object instances in OOP.

8.2.2 Type inference and overlapping instances

Coq allows function formal parameters to be marked as implicit, which means that it will try to infer them automatically using type inference. However, the inference does not always succeed. This feature can be used to sometimes infer instances automatically – like in Haskell – and sometimes specify them explicitly – which Haskell does not support.

Unlike Haskell 98, this system does not seek to disallow overlapping instances – that is, two instances of the same type class with the same, or overlapping, type class parameters. This is because in this system it is always possible to specify an instance explicitly if Coq is not able to infer the desired one, because in this system type classes and instances are first-class values. Also, this avoids the developer having to tediously wrap values in other types, and then unwrap them, in order to avoid overlapping instances. The main argument for wrapping

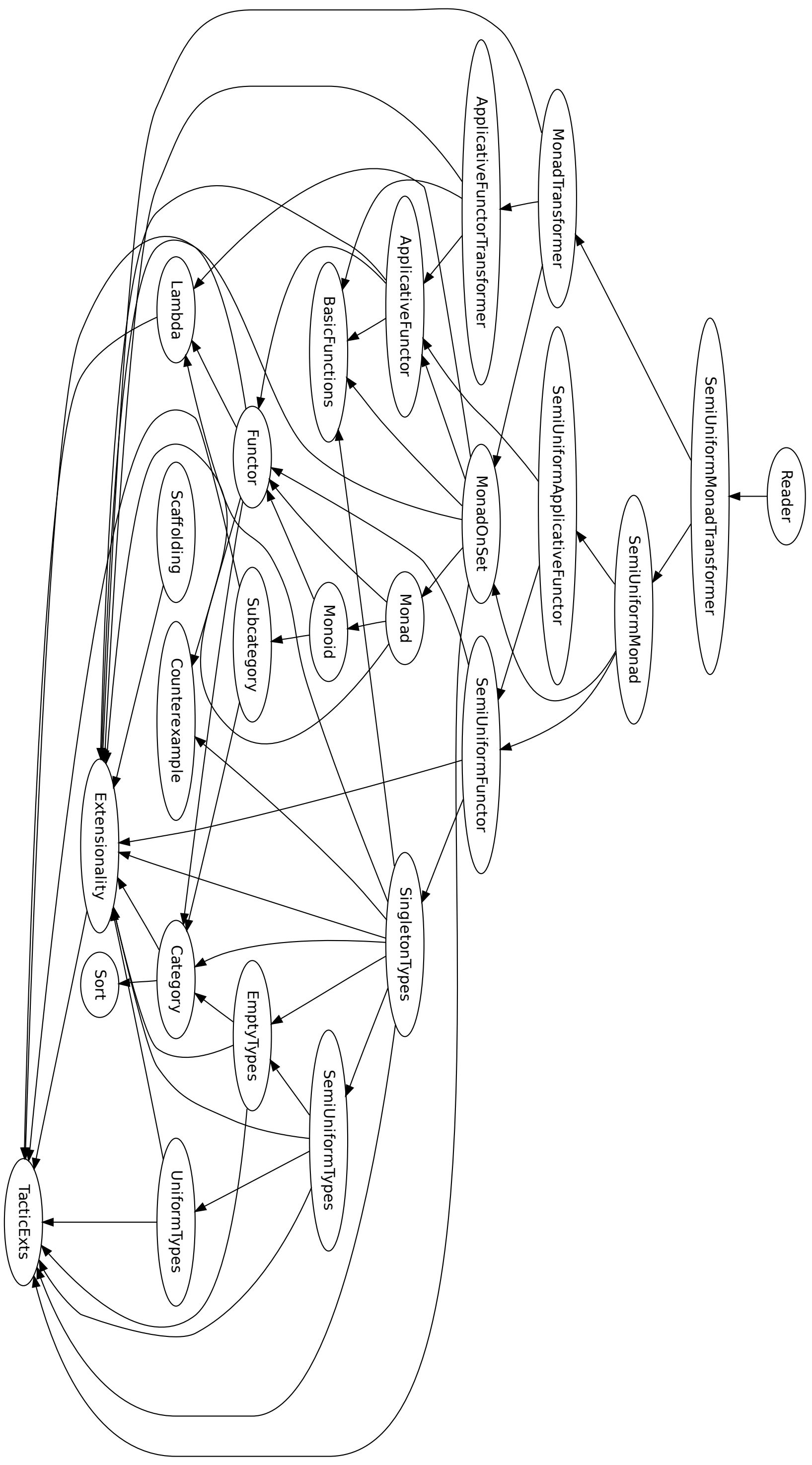

Figure 8.1: Dependency diagram for the Coq source code

is that it avoids ambiguity; however, again, explicit instances can be used to disambiguate. Explicit instances, however, can also make code substantially longer; it is a case-by-case question as to whether wrapping or explicit instances are more appropriate. Wrapping is not used in this work; however, there is no problem with using it in conjunction with this type class design.

8.2.3 Late binding

A key difference between type classes, as implemented here, and Java-style classes, is that while the former do provide a form of *late binding*, they do not provide *late-bound self reference*s (terminology due to [45]). *Late binding*, in object-oriented programming, means that a class may override a method declared in an ancestor class, so that which implementation of the method is called depends on the run-time type of an object. With type classes, this corresponds to instances implementing methods declared in type classes – again, which implementation is chosen depends on the run-time type of the value(s) involved.

The `this` pseudo-variables in Java are *late-bound self references*: that is, they allow late binding to be used from within a class, on the current object – whatever runtime type the current object happens to be, the correct implementation will be chosen. (Invocations of non-static methods in the same class are implicitly prefixed with "`this`." by the compiler, if no prefix is supplied by the programmer, so these invocations are treated in the same way.) Late binding without late-bound self references has no analogue in Java or C++.

Type classes in this work do not provide late-bound self references as standard, however, because there is no implicit or explicit "self" argument. If this is really necessary, it would be necessary to pass in such an argument on an ad-hoc, manual basis.

However, it is sometimes possible to use other techniques in place of late-bound self references. For example, one use for late-bound self references is optimisation – one method might have a shared implementation, but it might call another method which needs to be overridden for performance reasons. In the Coq context, one way to implement optimisations would be to postprocess extracted programs to make them use more efficient behaviours for certain instances (see section 4.2.3); this would separate optimisation concerns from basic functionality concerns. Also, if a method needs to call other methods with late binding, but does not need to be late bound itself, it can be implemented as an ordinary function instead – that is, a function that is not part of the type class, but which merely refers to it. Optimisations are outside the scope of this thesis, but the latter technique is used in this work.

In this system, Haskell's overridable default methods [40] are not supported: that is, methods having "default" implementations are not overridable. This is not a problem for the work in this thesis. Yet, multiple implementation inheritance is still supported.[1] In the next section, it is explained how those two design goals can come into conflict, and how this conflict is avoided here.

8.2.4 Inheritance

Instantiation

From the object-oriented perspective on inheritance, the most basic form of inheritance in this type class idiom is an instance inheriting from – that is, instantiating – the type class of which it is an instance. For example, a `List` class could be instantiated by an instance implemented in terms of the standard Coq list type, or by an instance implemented in terms of an array type (modelled as a function from natural number indices to values). This corresponds, in OOP, to a non-abstract class inheriting from an abstract class or interface, and implementing its abstract methods – and *not* to instantiating an object of that class (see the warning above).

Essentially, a type class is implemented as at least two dependent record types (one for the methods, one for the laws which must hold for those methods,

[1]The notion of inheritance used here is weaker than the definition given in [45], because late-bound self references are not supported.

and optionally others to bundle the previous two together). An instance is implemented as at least two dependent records (one record of each of the former record types). If there is only one method, a type synonym may be used for bookkeeping purposes, instead of a dependent record type, or a type may just be reused.

The same holds if there is only one law. Of course, any set of laws can be represented as one law by repeated conjunction; however, the ability to refer to each law separately by name is very convenient in proofs, so such bundling is not recommended.

Implicit coercions

Other forms of inheritance are possible in this system, however. In Coq, a function may be declared to be an *implicit coercion.* Roughly speaking, this means that whenever an expression fails to typecheck, Coq will try to insert calls to any applicable coercion, in order to make it typecheck [1]. For example, there is, in this development (see Appendix B), a coercion called `SemiUniformFunctor_Functor`. This converts records containing proofs of the laws for `SemiUniformFunctor`s for some functor, into records containing proofs of the laws for `Functor`s for that same functor:

```
Coercion SemiUniformFunctor_Functor :
   D_SemiUniformFunctor >-> D_Functor.
```

(The proof of `SemiUniformFunctor_Functor` is omitted here because the necessary background to understand it has not been covered yet at this point in the thesis.) Because this is declared to be a coercion, it means that the former can be supplied wherever the latter is required. This is the essence of inheritance. This inheritance relationship is intended to be understood as meaning: a semi-uniform functor *is* a functor. A simpler example would be a `List` type class inheriting from a `Collection` class, which would mean that all Lists would be Collections, and would satisfy any laws for Collections.

There are two unusual concepts here. One is that there are two parallel inheritance hierarchies – one for the methods, and one for the corresponding laws. The reasons for this split are:

- An instance can be defined and used, without necessarily having to prove the corresponding laws until later. This could be helpful to avoid duplicate definitions, because the proof for an instance might depend on a definition which itself depends on the original instance.

- Law records, because they depend on the method records they constrain, act as convenient predicates. For example, the type of `SemiUniform-Functor_Functor` (outside of its containing section) expresses the constraint that if something is a `SemiUniformFunctor`, that very semi-uniform functor must also be a `Functor` – it does not merely take some `SemiUniform-Functor` and yield some *arbitrary* `Functor`. It expresses this constraint by using a law record *type* as a *predicate* (see Appendix A).

- Proofs of laws are separated from method definitions. This means that one can use a tactic-driven approach to create proofs, but a traditional programming approach to write method bodies, if that is deemed more suitable.

The second unusual concept is the idea that a function can establish an inheritance relationship by converting one thing into another. This means that a subclass can have a completely different set of methods that need to be implemented by instances, than its superclass.

When converting a record of methods into a record of methods from a superclass, this allows simulating *overriding* a method within a type class, instead of implementing it within an instance. (It is only simulated because in practice, the method is still implemented in an instance, but it does allow the same method implementation to be shared among all instances of a type class.) In Java terms, that would translate into "overriding an abstract method from a superclass, in an abstract class". So, this is simply an unusual way of implementing a well-known feature. It is necessary to implement it this way because Coq's dependent records do not have any notion of overriding.

When converting a record of proofs into a record of proofs from a superclass, on the other hand, this amounts to proving a logical implication. As such an implication needs to be proved anyway to establish behavioural compatibility [45], this use of the coercion feature should not be too surprising, either.

Aggregation inheritance

Coercions are always considered to establish an inheritance relationship in this system, and they can be used for three purposes:

1. To state that A IS_A B, and separately implement one or more of B's methods for all values of A. This is done by creating a conversion function A_B, and declaring it as a coercion `A_B : A >-> B`, as in the example above.

2. To state that A IS_A B, without having to separately implement any of B's methods for all values of A. This is done by creating a method A_B in A, and declaring it as a coercion. That method must then, of course, be implemented in all instances of A.

3. To state that A HAS_A B, without having to separately implement any of B's methods for all values of A. This is done in the same way.

It is controversial [45] to use inheritance for aggregation (HAS_A) relationships, but it is convenient. The only restriction on all of these uses of inheritance is that there should not be a truly ambiguous coercion path – as defined below – between any two types. Coq does not currently enforce such a restriction: it is up to the developer to follow it (and notice any ambiguity warnings Coq gives).

A simple example of using inheritance for aggregation would be an inheritance relationship between a `Vehicle` type class and an `Engine` type class in a simulation. Clearly a vehicle is not an engine, but a vehicle has an engine, and it might be useful to be able to supply a vehicle where an engine is required without having to insert an explicit `getEngine` invocation or similar.

Diamond inheritance

If a (type) class inherits from two other (type) classes, and they in turn each inherit from the same (type) class, this is known as *diamond inheritance* [45]. A simple example of diamond inheritance would be a `AmphibiousVehicle` type class which inherited from two other type classes, `LandVehicle` and `Boat`, both of which inherited from a `Vehicle` type class.

In this system there is a formally ambiguous coercion path [1] whenever diamond inheritance occurs. This formal ambiguity may or may not be a problem: if it makes no difference which of the two paths is followed, it is only a formal ambiguity; otherwise, it is a *truly ambiguous coercion path.*

To avoid true ambiguities in diamond inheritance, it is possible to ensure that both paths result in the same superclass instance (not just the same superclass, but the same superclass *instance*). This is done at the apex of the diamond, by making the direct superclasses constrained by their types to have the same superclass instance.

An example of this technique applied in this work is `SemiUniformApplicat-iveFunctor`, which inherits from both `SemiUniformFunctor` and `Applicative-Functor`. Each of these in turn inherit from `Functor`. So the constraint on the apex of the diamond in this case is that both the `SemiUniformFunctor` and the `ApplicativeFunctor` must have the same “fmap” method implementation (which many Haskell programmers will recognise as the name of the sole method in the Functor typeclass in Haskell). Note that having the same “fmap” implementation automatically implies having a compatibly-typed “fmap” implementation, because if two things are the same they must have the same type.

Superparameters

Superparameters are type parameters which are also inherited from. Whilst this might sound like a contradiction in terms, the trick which achieves this unusual feat is simple. An extra record type is created, consisting of a reference to each superparameter, and a reference to the original methods record type – all of which are declared as coercions. A value of this new record type is thus coercible to any superparameter type. For example, a `Category_Fields` value can be used wherever a `Type` or a `Category_ob -> Category_ob -> hom_Sort` is required.

In practical terms, this means that a category variable `c` can be used to represent either the category itself, the type of its objects, or the type constructor for its morphisms, *depending on the context.* This is very convenient, and indeed category variables have been used in this way in this work – as shown below (comments have been added to highlight the different meanings of the `c` and `d` `Category` variables). Like all uses of coercions, it acts to abbreviate code, which can make it easier to read (although some might find it confusing). It is not necessary to fully understand the other lines of code here in order to understand how coercions are being used here, although knowledge of Coq syntax [1] and category theory is recommended.

```
Section WithCategories.

  Variables c d : Category s.

  Let cast := @Sort_Type_Coercion s.

  Section Functor_defs.

    Variable F : c -> d.  (* Here, c and d refer to
        the Type of objects in c and the Type of
```

```
   objects in d, respectively. *)

Definition fmap_t  := forall a b : c, cast (c a b)
    -> cast (d (F a) (F b)).  (* Here, c and d
   refer to their respective morphism type
   constructors.  Ignore the "cast" invocations,
   which are for technical reasons and are
   discussed later. *)

Variable fmap : fmap_t.

Definition Functor_1st_law_t  := forall a : c,
   fmap (id c (x := a)) = id d.  (* Here, c and d
   just refer to their own Category_Fields
   instances. *)

Definition Functor_2nd_law_t
  := forall (x y z : c) (f : cast (c y z)) (g :
     cast (c x y)),
     fmap (comp c g f) = comp d (fmap g) (fmap f).
          (* This is a combination of different
        meanings of c and d in the same law! *)

Record D_Functor : Prop :=
  { Functor_1st_law :> Functor_1st_law_t
  ; Functor_2nd_law :> Functor_2nd_law_t }.

Notation "x <$> y" := (fmap x y) (at level 59,
   left associativity).
  End Functor_defs.
(* ... *)
```

Strict and non-strict inheritance

The underlying notion of inheritance adopted here is based on *type compatibility.* As a consequence of this fact, and the separation between methods and laws, the system supports both strict inheritance and non-strict inheritance [45], while rigorously distinguishing between the two. A simple example of strict inheritance would be the example, already given previously, of a `List` type class inheriting from a `Collection` type class: a `List` would be required to follow all the laws of a `Collection`.

If a record of methods can be implicitly coerced to another record of methods, and its corresponding record of laws can be implicitly coerced to the respective record of laws, this establishes behavioural compatibility (strict inheritance). Coercion to a "simple" value (that is, not an instance of a typeclass) such as a natural number, is also regarded as strict inheritance. This is because it is assumed that there are no constraints, other than the destination type itself, to satisfy.

If, on the other hand, a record of methods can be implicitly coerced to another record of methods, but there is no corresponding coercion for their laws, behavioural compatibility is not guaranteed, but it is still counted as a form of inheritance (non-strict inheritance). It is unclear at this stage whether non-strict inheritance is useful in a formal verification context, but it is supported.

8.3 Implementation guidelines for type classes and instances

To illustrate these guidelines, this section includes snippets from the Category type class from Appendix B. These guidelines presuppose that the type class or instance to be implemented has already been designed, at the abstract level of the definitions in subsection 8.2.1.

It may be advisable – especially when working with complicated concepts like the category-theoretic concepts formalised here – to start every file with at least the following commands:

```
Set Implicit Arguments.
Unset Strict Implicit.
Set Printing Implicit.
```

The first two commands, roughly speaking, tell Coq to mark parameters that are moderately easy to infer as implicit parameters, automatically. This means (hopefully) that theories will be slightly less verbose. The third command instructs Coq to display all implicit parameters whenever it outputs a term – this can sometimes be helpful to see what is going on in detail in a proof.

In what follows, any extra commands or declarations in the Category module[2] that are not part of these guidelines are glossed over. This module will be explained in more detail in section 10.1.

8.3.1 Implementing type classes

The following guidelines should be applicable to a wide range of type classes. Sometimes, however, it may be appropriate to deviate from them (for example, if the type class is very simple, and following all of these steps would be overly bureaucratic).

1. Decide which of the type parameters, if any, should be superparameters (see page 77).
2. If there are any type parameters which are *not* superparameters:
   (a) Start a new section. A section is a useful construct for structuring theories: it allows variables to be declared just once, in the environment, and then adds them as new parameters to each declaration

[2]This is not a typographical error. In Coq, every file is automatically a module.

which uses them when the section is closed [1]. In the Category example, because the section construct is vaguely reminiscent of the `with` keyword in the Pascal programming language, and the parameter in question is a `Sort`, the section name chosen is `WithSort`:

    ```
    Section WithSort.
    ```

   (b) Declare the non-superparameter type parameters, using the `Variable` command:

    ```
    Variable hom_Sort : Sort.
    ```

3. Start a new section, with name [type class name]`_defs`:

   ```
   Section Category_defs.
   ```

4. Declare the superparameters, if any, using the `Variable` command:

   ```
   Variable ob : Type.
   Variable hom : ob -> ob -> hom_Sort.
   ```

5. If one or more non-trivial method types are to be referenced again in subclasses, for each of them, define a definition with name [method name]_t, to avoid code duplication. This has not been done for the Category module, but for an example, see `fmap_t` in the Functor module.

6. Declare the record type of methods, prefixing the name with `D_` to indicate this record depends on all the type parameters. Some notes about the example below:

   - Methods are referred to as "fields" instead of "methods" throughout this implementation, for historical reasons. This is a purely terminological difference.
   - Normally, each method should be prefixed with the name of the type class, followed by an underscore. However, in this case, because of

the expected frequency of use of these particular methods, that is not done.

- `hom_Sort_coercion` is actually effectively the identity function, but it is needed to convince Coq that, for example, `(hom x y)` yields a valid type. This is because of the use of sort polymorphism (see section 10.1).

```
Record D_Category_Fields : Type :=
  { comp : forall x y z : ob,
      hom_Sort_coercion (hom x y) ->
      hom_Sort_coercion (hom y z) ->
      hom_Sort_coercion (hom x z)
  ; id : forall x : ob, hom_Sort_coercion (
      hom x x)
  }.
```

7. Declare a variable of the type just created:

```
Variable cf : D_Category_Fields.
```

8. If one or more non-trivial laws are to be referenced again in subclasses, for each of them, define a definition with name [law name]_t, to avoid copy and pasting. For an example of this, see `Functor_1st_law_t` in the Functor module.

9. Declare the record type of laws, prefixing the name again with `D_`. Prefix the names of the laws with the name of the type class followed by an underscore:

```
Record D_Category : Prop :=
  { Category_associativity_law : forall (a b c
      d : ob) (f : hom_Sort_coercion (hom a b)
    ) (g : hom_Sort_coercion (hom b c)) (h :
        hom_Sort_coercion (hom c d)), comp cf (
        comp cf f g) h = comp cf f (comp cf g h)
    ; Category_left_identity_law : forall (a b :
         ob) (f : hom_Sort_coercion (hom a b)), f
         = comp cf (id cf) f
    ; Category_right_identity_law : forall (a b
        : ob) (f : hom_Sort_coercion (hom a b)),
        f = comp cf f (id cf)
    }.
```

10. End the section:

    ```
    End Category_defs.
    ```

11. If there are superparameters, declare the extra record type for them. (The `:>` symbol indicates that a method is also a coercion, and could have been used above, as well, but was not.)

    ```
      Record Category_Fields : Type :=
        { Category_ob :> Type
        ; Category_hom :> Category_ob -> Category_ob
           -> hom_Sort
        ; Category_Fields_d :> D_Category_Fields
           Category_hom
        }.
    ```

12. Optionally, declare a convenience type bundling together the methods with the proofs of their laws. Usually, this is declared as a `sig` type (`sig` types are similar to refinement types), using the curly braces notation (for example, in the Functor module: `Definition Functor := { f : Functor_Fields | D_Functor f }`). However, in the case of Category, which is a “large” inductive type, such a type cannot be used, because it

would lead to a universe inconsistency error. (Coq's type system has a hierarchy of universes to avoid paradoxes similar to Russell's Paradox – a universe inconsistency error means that the rules governing the universe hierarchy have been broken, and Coq cannot guarantee that there would not be a paradox, or inconsistency, as a result.)

```
Record Category : Type :=
  { Category_fields :> Category_Fields
  ; Category_proj2 :> D_Category Category_fields
  }.
```

13. End any other section started above:

```
End WithSort.
```

14. Optionally, prove independence results for the laws. This acts as a "sanity check" on the laws, ensuring non-redundancy. Ideally, laws which are consequences of other laws should be stated and proved as separate lemmas – not included in the laws for every instance to redundantly prove. However, it may not always be feasible to prove independence results.

8.3.2 Implementing instances

Implementing instances is a much simpler process. It simply involves creating records for each record type (with appropriate type parameters supplied). In the example below, the final command concisely creates two records in one go, so only three commands are needed to create four records.

```
Canonical Structure Set_D_Category_Fields :
   @D_Category_Fields Set_Sort Set (fun a b => a -> b)
   . Proof.   auto. Defined.
Canonical Structure Set_D_Category : D_Category
   Set_D_Category_Fields. Proof.   auto. Qed.
```

```
Definition Set_Category := @Build_Category Set_Sort (
    Build_Category_Fields _) Set_D_Category.
```

For an example of invoking methods on instances, refer back to the code listing on page 77.

Chapter 9

Parametricity in the Calculus of Inductive Constructions

Parametricity results are given and used in this work in order to try and make it easier to prove certain laws for monads, functors, etc. This is done by showing that for a certain class of types, the desired results in a sense "fall out" of the structure of the types in Coq (i.e. they are "free theorems").

This Chapter is quite technical and involved, so readers who are not very interested in the technical details of parametricity may skip ahead to Chapter 10, without impairing their understanding of the material in that Chapter.

9.1 Free theorems for dependent types

9.1.1 Free theorems and parametricity theorems

A so-called "free theorem" in a typed language is a theorem which holds for all terms of a particular type, and is derivable from a parametricity theorem (also known as an abstraction theorem) for the language in question [48]. The term “free theorem” is a little misleading, because there is still theoretical work involved, even though it may not involve reasoning about particular terms.

A parametricity theorem is a metatheoretical result – it is a theorem about the language – and relates to quantification over types. From the point of view of proving free theorems, the basic idea of parametricity is that “a polymorphic term that gives outputs in the same type for all input types, must be constant.” (Term here means function.) In System F augmented with the latter axiom (called System Fc), the following theorem holds:

Genericity Theorem “If two second-order terms coincide on an input type, then they are, in fact, the same function.” [25]

Theorem 9.1 The Genericity Theorem does not hold for Coq’s Calculus of Inductive Constructions.

Proof This proof consists in exhibiting a counterexample. A sized list type is a well-known dependent type which is parameterised by its length and its element type (in that order). Suppose this is converted into an indexed type – that is, the length is represented as a type-level natural number, instead of an ordinary natural number. Take the two terms in question to be the polymorphic identity on sized lists, and the function of the same type which ignores its arguments and yields an empty sized list. These terms coincide on an input type of `Zero`, but are not the same function. □

The key difference compared to System Fc is that in Coq, indexed types can be constructed – and not all constructors of an indexed type are necessarily available for all type arguments. In the counterexample above, only the empty list constructor is available when the length type argument is `Zero`.

9.1.2 Uniform types

It is still possible to obtain free theorems in Coq, however. Note that all of the definitions, lemmas and so on referred to below, may be found in Appendix B.

A uniform type (which is really a uniform type constructor) is defined as one for which the Genericity Theorem holds, in a restricted setting. Quoting from the UniformTypes module:

```
Section Uniform_Defs.
  Variable t : Type.
  Definition uniform (u : t -> Type)
    := forall (r : Set) (f g : forall s1, u s1 -> r),
        (exists s2, inhabited (u s2) /\ f s2 = g s2) ->
         f = g.
```

(This definition is not very interesting unless `t` is `Set`, so let us assume that it is.)

Here, `u` is the type constructor which is being tested for uniformity. The restrictions – which appear above – are that the functions must be of type `forall s1, u s1 -> r` for some fixed `r` that does not depend on `s1` (this Coq syntax guarantees that `r` cannot depend on `s1`), and the type `u` must be inhabited at the input type. If the habitation restriction were not included, even very simple dependent type constructors such as `(fun s => s * bool)` would fail to be uniform. This is because if `s2 * bool` is uninhabited (which occurs iff `s2` is uninhabited) *any* such `f` and `g` will be vacuously equal at `s2`.

The lemma `uniform_leaks_no_info` (also in the UniformTypes module) proves that this definition is logically equivalent to saying that the type "leaks no information" about its type parameter, in a specific technical sense (below, `t` is the same `t` from above):

```
 Section Information_Defs.
    Variable b : t -> Type.
    Variable P : t -> Prop.

    Definition informationAboutP
      := exists sn, exists sp, inhabited (b sn) /\ (~
         P sn)
             /\ exists g : forall s, b s -> bool,
                (exists x, g sp x = true) /\ forall s
                   x, g s x = true -> P s.

    (* If we don't want to know about a specific
       predicate, but simply whether there exists any
       predicate for which there is information,
     we can use this marginally shorter definition. *)
    Definition informationAbout
      := exists sn, exists sp, inhabited (b sn) /\
         exists g : forall s, b s -> bool, (exists x,
         g sp x = true) /\ forall y, g sn y = false.

  End Information_Defs.

  Hint Extern 2 (~(@ex ?x ?p)) => refine (ex_ind _).
  Hint Unfold informationAboutP.

  Require Import Bool.

  Lemma simplification b : (exists P,
     informationAboutP b P) <-> informationAbout b.
  Proof with eauto 12 with bool.
    unfold informationAbout.
    split; intro H.
      Hint Resolve not_true_is_false : bool.
      destruct H as (P, (sn, (sp, (ibsn, (npsn, (g, (
         H1))))))) ...
      destruct H as (sn, (sp, (ibsn, (g, (H1, H2))))).
      exists (fun a => exists x : b a, g a x = true)
        ...
  Qed.

  Axiom discrimination : forall r r2 (f1 f2 : forall (
     s:t), r s -> r2) sd rd,
    f1 sd rd <> f2 sd rd -> exists g : forall s, r s
       -> bool, g sd rd = true /\ forall s y, g s y =
       true -> f1 s <> f2 s.

  Lemma uniform_leaks_no_info u : uniform u <-> ~
     informationAbout u.
  Proof.
    lazy beta iota zeta delta.
    split; intros.
      destruct H0 as (sn, (sp, (iusn, (g, ((x, H0), H1
         ))))).
      replace g with (fun (s1:t) (_:u s1) => false) in
         H0; eauto 7 with bool.

      apply extensionality2.
      proof_by_contradiction H.
      destruct H0 as (sn, (H2, H3)).
      exists sn.
      destruct (not_all_ex_not _ _ H1).
      exists x.
      chopoff.
      destruct (not_all_ex_not _ _ H).
      destruct (discrimination (r := u) H0) as (g0, (
        H5, H6)).
      exists g0.
      split; [ eauto | ].
 (* used because it's much faster than chopoff at this
     point in the proof. *)
      iapplyFwd not_true_is_false.
      intro.
      applyFwd H6; eassumption.
  Qed.
```

This may be a more intuitive way to understand the meaning of a uniform type.

The proof for `uniform_leaks_no_info` relies on the `discrimination` axiom (see above) – which should not be confused with Coq's built-in `discrimination` tactic. This axiom states that if two polymorphic functions `f1` and `f2` have the same type, but differ when applied to the particular type `sd` and then the particular value `rd`, then there exists a 2-argument predicate-function `g` (definable in Coq) such that `g sd rd` is true, and `g` yields true *only if* `f1` and `f2` differ at the first argument of `g`.

Like many of the metatheoretical results introduced as axioms in this part of the work, this is not directly provable in Coq itself (without relying on further axioms), but can be justified with an informal proof. If the functions simply yield natural numbers, then simply take the predicate-function `g` to be the one which applies its arguments to both `f1` and `f2` and yields true iff the results are unequal. If, on the other hand, the functions yield more functions, instead of natural numbers, equality of functions is not decidable. However, there must exist an `rd2` (whose type does not depend on anything) where the results of `f1 sd rd` and `f2 sd rd` disagree, so now `g` should be the function which simply applies its arguments, and then `rd2`, to both functions – thus reducing the problem to an (almost) equivalent problem with one fewer argument. Glossing over a technical point, by induction (functions cannot have an infinite number of arguments), this means that any number of arguments can be accomodated. This argument can be extended to a structurally recursive argument that covers all possible output types.

9.1.3 Semi-uniform types

It turns out, however, that the definition of uniform types above is too stringent, in that it excludes some Coq types about which free theorems can be stated – such as the `option` type (called Maybe in Haskell). Thus, a notion of semi-uniform types is defined (in the SemiUniformTypes module). This differs from the notion of uniform types essentially in that the input type where the two functions are equal, is itself required to be inhabited, in order for the two functions being equal at that type to count.

```
Section SemiUniform.
  Variable su : Set -> Type.
  Definition semiUniform
    := forall (r : Set) (f g : forall s1, su s1 -> r),
         (exists s2, inhabited (su s2) /\ inhabited s2
       /\ f s2 = g s2) -> forall s3, f s3 = g s3.
```

Again, a similar lemma `semiUniform_leaks_no_other_info` establishes a logical equivalence between semi-uniform types and the intuitive notion of information leaking. Roughly speaking, this lemma states that the only information that can be leaked by a semi-uniform type is whether its type argument is inhabited.

```
  Lemma semiUniform_leaks_no_other_info
    : (forall (Q : Set -> Prop), informationAboutP su
        Q
         ->  forall t, inhabited (su t) -> (Q t <->
            inhabited t))
    <-> semiUniform.
Proof with eauto 6 with bool classical.
  lazy beta iota zeta delta.
  split; intros.
    apply extensionality.
    apply not_ex_not_all.
    intros (x, H1).
    destruct H0 as (x0, (H0, (H2, H3))).
    generalize (H (fun s => f s <> g s)).
    apply nonImplication.
      destruct (discrimination (r := su) H1) as (d,
         (H5, H6)).
      eauto 10.
      intro.
      destruct (H4 x0 H0).
      unfold not in H6...

    destruct H0 as (sn, (sp, (isusn, (H0, (d, ((x,
       H2), H3)))))).
    apply NNPP; intro; apply diff_true_false.
    symmetry in H2.
    eapply trans_eq.
    eassumption.
    pose (cf := fun t (_ : su t) => false).
    replace false with (cf sp x) by reflexivity.
    replace (cf sp) with (d sp)...
    apply H.
    subst cf.
    destruct (not_and_or _ _ H4).
      exists sn.
```

```
        chopoff.
        chopoff.
        proof_by_contradiction H0.
        replace sn with t in *.
          eapply proj1...

          Require Import EmptyTypes.
          apply empty_sets_eq; auto.

        exists t.
        chopoff.
        chopoff.
        eapply proj1...
Qed.
```

Generally, in this theory, types are asserted to be uniform, or semi-uniform, using axioms, or proved to be so using other axioms. It seems difficult to prove the uniformity of a number of different types based on a single axiom – and impossible to do without axioms about uniformity, since these are metatheoretically-justified properties. Each axiom asserted should be fairly easy to justify informally, however.

9.1.4 (Semi-)uniform target types

In order to prove useful results to make defining functor and monad instances more convenient, it is necessary to deal with a wider class of functions than the semi-uniform type definition deals with. In particular, it is necessary to deal with functions whose output type depends on their type argument.

For this purpose, the notions of uniform target and semi-uniform target are defined (in the SingletonTypes module). A type is a (semi-)uniform target basically if it is safe to use it as the output type of a function – that is, doing so will not break the (semi-)uniformity condition.

It was thought necessary during the work to ensure that the uniform type which forms the input type of the function can only "contain" at most one value whose type is the uniform type's type argument. The definition `MonoContainer` is used for this. The reason for this constraint is that non-monocontainers, if allowed, could be used to break the intended meaning of the definition of a uniform target.

It turns out, however, that all semi-uniform types are monocontainers anyway – so the definitions of uniform target and semi-uniform target below contain a redundancy, and the definition of monocontainer is thus not needed at all. This was discovered at the last minute, so there was no time to make all the necessary changes to the development to remove this redundancy.

Proof The proof is by contradiction. If there exists a semi-uniform type that is *not* a monocontainer, create two functions which do the following, when given a type parameter `p` and a value of the semi-uniform type `s`:

- If `s` can produce no values of type `p`, yield `existT (fun T => T) bool True`
- If `s` can produce only one value of type `p`, namely `x`, yield `existT (fun T => T) p x`
- Otherwise, each function should yield a different value of type `p` obtained from `s`, wrapped in an `existT (fun T => T)`

Then these functions must agree at type `unit` (because `unit` has only one inhabitant), but not at type `nat` – which contradicts the assumption that the type is semi-uniform.□

```
Section MonoContainer_Defs.
  Variable c : Set -> Type.
  Definition destructor  := forall s, c s -> option s.
  Definition monoContainer := forall (f g : destructor
      ) s1 x, ~ (f s1 x <> None /\ g s1 x <> None /\ f
      s1 x <> g s1 x).
End MonoContainer_Defs.

(* ... *)

Section Uniform_Targets.
  Variable ut : Set -> Type.
  Definition uniform_target := forall (u : Set -> Type
      ), uniform u -> monoContainer u -> forall (f g :
      forall (s1 : Set), u s1 -> ut s1),
    (exists s2 : Set, inhabited (u s2) /\ f s2 = g s2)
        -> f = g.
  Definition semiUniform_target := forall (u : Set ->
      Type), semiUniform u -> monoContainer u
    -> forall (f g : forall (s1 : Set), u s1 -> ut s1)
        , (exists s2 : Set, inhabited (u s2) /\
        inhabited s2 /\ f s2 = g s2) -> f = g.
  Axiom uniform_target_semiUniform_target :
      uniform_target -> semiUniform_target.
End Uniform_Targets.
```

9.1.5 Free theorems for functor laws

The module SemiUniformFunctor defines an eponymous type class, which permits the first of the two functor laws to be proven, and then evidence of the functor's type being semi-uniform and a semi-uniform target to be supplied in lieu of the second functor law. It provides the proof of the second law, based on the supplied evidence. This proof uses the unimaginatively-titled user-defined tactics `uniform1`, `uniform2` and `uniform3` to perform several proof steps that are common to a few proofs in the system applying uniformity results to prove free theorems.

There are other modules whose names start with `SemiUniform`, which also shorten the number of proofs to be made in exchange for evidence of semi-uniformity.

9.2 Definite description programming

In philosophy, a *definite description* is a description which is only satisfied by one object – it is definite because it picks out a unique object. So, in formally verified programming, if a type has only one inhabitant (up to equality) it is a definite description. Such a type is here called a *definite description type*.

Recall that the functional extensionality axiom states that any two functions which yield the same results for every input are equal. So, a definite description type for a function only dictates which outputs are computed from which inputs – not how the outputs are computed.

A transparent definition in Coq is also a definite description, because it "exports" its exact definition to proofs – unlike an opaque definition, which only reveals its type. Any other definition defined to be the same thing would also be the same value.

By definition, therefore, an identifier which is either transparently defined, or whose type is a definite description type, is a definite description.

9.2.1 Benefits of definite description types

Benefits for client code

Theorem 9.2 If a function's type is a definite description, and known to be so, one of the possible sources of programmer error in section 4.1.2 is prevented: it is not possible to introduce bugs by relying on undocumented features.

Proof There are two sorts of undocumented features:

1. Features which are a logical consequence of a function's type. Relying on these cannot introduce bugs, because the type guarantees their presence.

2. Features which are *not* a logical consequence of a function's type. If a function's type is a definite description, such features do not exist. If the function had such a feature – such as "the output is always sorted in this implementation" – there would be at least one other function inhabiting the function's type, one in which the output was *not* always sorted. However, since the function's type is a definite description, this cannot happen. Hence, such features do not exist. □

Admittedly, full-blown verification of the calling function, if *its* specification were correct, would catch any such bugs anyway, if the function were not typed by a definite description. However, in a hybrid system with some parts fully verified and other parts not, this assurance could be useful. Even in a system where full-blown verification was used everywhere, it could still be useful to prevent the programmer making such errors in the first place, which might be cheaper than catching them at verification time.

Furthermore, in principle, there is no ambiguity or uncertainty about what inputs the function will accept, or what it will yield. So, leaving aside non-functional issues such as performance, the programmer should not be tempted to program over-conservatively (unless they are not sure what the type implies).

Benefits for implementors

Definite description types also benefit the implementors of functions with definite description types. Such types, by definition, cannot underspecify the output – as long as the computational part of the function's output type has been chosen correctly.[1] So, a proof that a type is a definite description is a kind of quality

[1]Coq distinguishes between proof-relevant *computational* types, which generally lie in the `Set` sort, and proof-irrelevant *propositional* types, which generally lie in the `Prop` sort. Clearly, if a function is supposed to yield a sorted list, but the computational output type chosen is a set without ordering, it will of necessity be not only underspecified, but incorrect as well.

assurance check on the specification. This gives a further level of confidence: not only is the code itself verified, but the specification is subject to a “sanity check” as well.

Another way in which a proof that a type is a definite description can be a sanity check, is if a term of that type is generated by tactical programming (see page 62), instead of by supplying an explicit term. Tactical programming, in general, carries the risk that incorrect code might be generated, if no definite description for the function exists that is of lesser complexity than the function itself. However, if such a definite description does exist, there is no risk of incorrect code when using that description to give a type to the term (again, modulo non-functional issues such as performance) because by definition there is only one possible function satisfying the description.

9.2.2 Parametricity for definite description proofs

Results about uniformity and uniform targets can be used to do proofs that certain types are definite descriptions. An example of such a proof is the proof for the `uniqueId` lemma (in the SingletonTypes module), which proves that the Set category’s identity morphism is unique.

```
Theorem uniqueId : unique' (@id _ _ _ Set_Category).
Proof.
  red.
  intros.
  Hint Resolve id_monoContainer : uniform.
  compute in *.
  refine (@id_uniform_target _ _ _ _ _ _); auto with
     uniform.
  exists unit.
  Hint Constructors unit : unique.
  auto with unique.
```

```
Qed.
```

This particular lemma may not be very useful in itself, but it is just a proof of concept. This style of reasoning has not yet been explored very much in this work, due to time constraints.

Chapter 10

Verified categories, functors, monads and transformers

Again, note that all of the definitions, lemmas and so on referred to below, may be found in Appendix B.

This Chapter has Chapter 8 as a prerequisite.

10.1 Categories

In the Category module, a `Category` type class is defined that is polymorphic in the sort of the morphism type. This allows the category `Type_Category` to be defined, with morphisms in the sort `Type`, as well as the category `Set_Category`, with morphisms in the sort `Set`.

Parameterising by a sort is done using a simple technique. In the Sort module, Coq's sorts are reified into an inductive type:

```
Inductive Sort : Set := Type_Sort | Set_Sort | Prop_Sort.
```

The reason for this reification – which literally means turning an abstract thing into a concrete thing – is that `Type`, `Set` and `Prop` cannot be distinguished between by pattern matching in Coq, whereas `Type_Sort`, `Set_Sort` and `Prop_Sort`, as completely ordinary constructors, can be.

Coq's infinite hierarchy of `Type` universes is elided, because it is not necessary to reify that for the work done in this thesis.

Definitions are then provided to:

- convert a Sort into the corresponding actual Coq type (`Sort_Coq_Type`). This is declared as a coercion, so it will be applied automatically where necessary.
- convert a type in any of these three sorts into a Type (`Sort_Type_Coercion`). Coq does not do this automatically, in this context. Note that a reference to `Sort_Type_Coercion` appears in the Category module, called `hom_Sort_coercion`.
- make a signature type appropriate for each sort (`Sort_mkSig`)
- make a product or conjunction type appropriate for each sort (`Sort_mkProd`)
- construct a product value appropriate for each sort (`Build_Prod`)
- destruct a product value for any sort (`pi1` and `pi2`)

```
Definition Sort_Coq_Type
  := match s with
       Type_Sort => Type
     | Set_Sort => Set
     | Prop_Sort => Prop
  end.

Variable t : Sort_Coq_Type.


Definition Sort_Type_Coercion : Type.
```

```
  Proof.
    red in t.
    destruct s; auto.
  Defined.

  Definition Sort_mkSig (P : Sort_Type_Coercion ->
      Prop) : Sort_Coq_Type.
  Proof.
    compute in *.
    destruct s; exact (@sig t).
  Defined.

  Definition Sort_mkProd (t2 : Sort_Coq_Type) :
      Sort_Coq_Type.
  Proof.
    intro.
    compute in *.
    destruct s.
      exact (prod t t2).
      exact (prod t t2).
      exact (t /\ t2).
  Defined.

End Sort_Defs.

Coercion Sort_Coq_Type : Sort >-> Sortclass.

Section Prod_Defs.
  Variable s : Sort.
  Variables t t2 : s.
  Let prod_type := Sort_mkProd t t2.

  Definition Build_Prod (x : Sort_Type_Coercion t) (y
     : Sort_Type_Coercion t2) : Sort_Type_Coercion
     prod_type.
  Proof.
    destruct s; firstorder.
  Defined.

  Variable p : Sort_Type_Coercion prod_type.
  Definition pi1 : Sort_Type_Coercion t.
  Proof.
    destruct s; firstorder.
  Defined.

  Definition pi2 : Sort_Type_Coercion t2.
  Proof.
    destruct s; firstorder.
  Defined.

End Prod_Defs.
```

These were all the sort-polymorphic definitions needed for these purposes.

A category type class has two superparameters: `ob`, the type used to represent objects in the category, and `hom`, the type constructor used to represent morphisms between any two objects. As mentioned previously, because these are superparameters, a category `c` can be used to stand for the object type `ob` or the morphism type constructor `hom`, which is very convenient.

In some formalisations of category theory in Coq, setoid constructs are used instead of types, in order to be able to use a custom setoid equality instead of the standard equality relation. This is not necessary for the purposes of this work, and would only overcomplicate matters.[1]

The methods of a category are `comp` (composition of morphisms) and `id` (the identity morphism). Unlike in Haskell 98, which has a composition operator for composition of functions and an identity function, these methods are applicable to any category, and the case of functions falls out as a special case (`Set_Category`, to be precise). Note also that `comp` applies the first argument first and then applies the second argument to the result – the opposite argument order to the composition operator in Haskell. This is a possible source of confusion.

The laws of a category are the associativity law, the left identity law, and the right identity law. In some presentations the left identity law and right identity law are conjuncted together into one law, but in this work laws are never conjuncted together into one. This is because this way – as previously mentioned – each law has its own individual name for ease of access.

```
Section WithSort.

  Require Export Sort.

  Variable hom_Sort : Sort.

  Section Category_defs.
    Variable ob : Type.
    Variable hom : ob -> ob -> hom_Sort.
    Let hom_Sort_coercion := @Sort_Type_Coercion
        hom_Sort.
```

[1]A novel way to avoid some uses of setoids in formal verification may be the subject of future work.

```
  Record D_Category_Fields : Type :=
    { comp : forall x y z : ob, hom_Sort_coercion (
        hom x y) -> hom_Sort_coercion (hom y z) ->
        hom_Sort_coercion (hom x z)
    ; id : forall x : ob, hom_Sort_coercion (hom x x
        )
    }.

  Variable cf : D_Category_Fields.

  Record D_Category : Prop :=
    { Category_associativity_law : forall (a b c d :
         ob) (f : hom_Sort_coercion (hom a b)) (g :
        hom_Sort_coercion (hom b c)) (h :
        hom_Sort_coercion (hom c d)), comp cf (comp c
         f f g) h = comp cf f (comp cf g h)
    ; Category_left_identity_law : forall (a b : ob)
         (f : hom_Sort_coercion (hom a b)), f = comp
        cf (id cf) f
    ; Category_right_identity_law : forall (a b : ob
        ) (f : hom_Sort_coercion (hom a b)), f = comp
         cf f (id cf)
    }.

End Category_defs.

Record Category_Fields : Type :=
  { Category_ob :> Type
  ; Category_hom :> Category_ob -> Category_ob ->
     hom_Sort
  ; Category_Fields_d :> D_Category_Fields
     Category_hom
  }.

Record Category : Type :=
  { Category_fields :> Category_Fields
  ; Category_proj2 :> D_Category Category_fields
  }.
```

The product category of two categories, the `Type_Category` of all `Type`s and non-dependent functions between them, and the `Set_Category` of all `Set`s and non-dependent functions between them, are then defined.

```
Variables c d : Category.

Definition Product_D_Category_Fields :
   @D_Category_Fields (prod c d) (fun a b =>
   Sort_mkProd (c (fst a) (fst b)) (d (snd a) (snd b
   ))).
Proof with eauto.
  split; intros; apply Build_Prod.
    eapply (comp c)...

    eapply (comp d)...

    exact (id c (x := fst x)).

    exact (id d (x := snd x)).
Defined.

Ltac solve_id f := destruct (Build_Prod_surjective (
      p := f)) as (x, (y, H)); rewrite H.

  Lemma Product_D_Category : D_Category
      Product_D_Category_Fields.
  Proof.
    Hint Resolve (@Category_proj2)
        Category_associativity_law
        Category_left_identity_law
        Category_right_identity_law.
    split; intros; [ | solve_id f | solve_id f ];
        simpl in *; f_equal; autorewrite with sort;
        auto.
  Qed.

(* ... *)

Canonical Structure Type_D_Category_Fields :
   @D_Category_Fields Type_Sort Type (fun a b => a ->
   b).
Proof.  auto. Defined.
Canonical Structure Type_D_Category : D_Category
   Type_D_Category_Fields.
Proof.  Require Import Extensionality.  auto. Qed.
Definition Type_Category := @Build_Category Type_Sort
   (Build_Category_Fields _) Type_D_Category.

Canonical Structure Set_D_Category_Fields :
   @D_Category_Fields Set_Sort Set (fun a b => a -> b)
   .
Proof.  auto. Defined.
```

```
Canonical Structure Set_D_Category : D_Category
   Set_D_Category_Fields.
Proof.  auto. Qed.
Definition Set_Category := @Build_Category Set_Sort (
   Build_Category_Fields _) Set_D_Category.
```

10.2 Functors and natural transformations

Unlike the Haskell Functor type class, which is an endofunctor on the category of Haskell types, the `Functor` type class defined in the Functor module is the full category-theoretic version. A functor from `c` to `d` consists of a function `F` taking objects in `c` to objects in `d`, and a function `fmap` taking morphisms in `c` to morphisms in `d`.

```
Definition fmap_t  := forall a b : c, cast (c a b) ->
   cast (d (F a) (F b)).
(* ... *)
  Record Functor_Fields : Type :=
      { Functor_f :> c -> d
      ; Functor_fmap :> fmap_t Functor_f }.
```

An equality lemma is then defined for functors, which simply says that if two functor's components are pairwise equal, the functors are equal. John Major equality (`JMeq`), a form of heterogenous equality provided by the Coq standard library, is used to compare the `fmap` methods. It is necessary to use heterogenous equality here because the type of `Functor_fmap` depends upon the value of `Functor_f`. It seems it ought to be possible to automate such lemmas more.

```
  Require Export JMeq.
  Lemma Functor_Fields_eq (F G : Functor_Fields) :
     Functor_f F = Functor_f G -> JMeq (Functor_fmap F
     ) (Functor_fmap G) -> F = G.
```

```
Proof.
  destruct F.
  destruct G.
  intros.
  simpl in *.
  subst.
  rewriter H0.
Qed.
```

Natural transformations – one of the central concepts of category theory – are then defined. These are used in defining monads. The identity natural transformation, and composition of natural transformations, is then defined. Then, the identity functor and composition of functors is defined. These two pairs of functions could have been defined by defining categories, but this was not necessary here.

```
Definition NaturalTransformation (F G : Functor)
  := { nt : forall x : c, cast (d (F x) (G x))
       | forall (x y : c) (f : cast (c x y)), comp d
           (Functor_fmap F f) (nt y) = comp d (nt x)
           (Functor_fmap G f)
       }.

Variable F : Functor.

Definition id_NaturalTransformation :
    NaturalTransformation F F.
Proof.
  refine (exist _ (fun z => id d (x := F z)) _).
  abstract (destruct (@Category_proj2 _ d);
     congruence 2).
  Defined.

  Definition comp_NaturalTransformation (G H : Functor
     ) : NaturalTransformation F G ->
     NaturalTransformation G H ->
     NaturalTransformation F H.
  Proof.
    intros.
    refine (exist _ (fun z => comp d (proj1_sig X z) (
       proj1_sig X0 z)) _).
    abstract (destruct X; destruct X0; simpl; destruct
        (@Category_proj2 _ d); congruence 15).
  Defined.

End WithCategories.

  Variable c : Category s.

  Section Functor_Ops.

    (* Avoids universe inconsistency  *)
    Notation "'superId'" := (fun x => x).
    Notation "'superComp'" := (fun f g x => g (f x)).
    Notation "'id_Functor_Fields'" := (
       Build_Functor_Fields (fun a b => superId)).

    Lemma id_D_Functor : D_Functor (d := c) (
       Functor_fmap id_Functor_Fields).
    Proof.
      Hint Unfold Functor_1st_law_t Functor_2nd_law_t.
      Hint Constructors D_Functor.
      auto.
    Qed.

    Variables d e : Category s.
    Variable F : Functor c d.
    Variable G : Functor d e.

    Notation "'comp_Functor_Fields'" := (
       Build_Functor_Fields (fun a b => superComp (
       Functor_fmap F (a := a) (b := b)) (Functor_fmap
        G (a := F a) (b := F b)))).

    Lemma comp_D_Functor : D_Functor (Functor_fmap
       comp_Functor_Fields).
    Proof.
      destruct F.
      destruct G.
      elim d1.
      elim d0.
      split; simpl; congruence 2.
    Qed.

    Definition id_Functor : Functor c c := exist _ _
       id_D_Functor.
    Definition comp_Functor : Functor c e := exist _ _
        comp_D_Functor.

  End Functor_Ops.
```

A “product” operation for natural transformations is then defined, for later use, and a convenient reformulation of the functor second law (not shown here) is proven as a lemma, using the ext_law tactic.

```
  Variables F F0 F1 F2 : Functor c c.
  Variable N : NaturalTransformation F F1.
  Variable N0 : NaturalTransformation F0 F2.

  Definition prod_NaturalTransformation :
     NaturalTransformation (comp_Functor F F0) (
     comp_Functor F1 F2).
  Proof.
    destruct N as (n, H).
    destruct N0 as (n0, H0).
    clear N N0.
    refine (exist _ (fun x => comp c (n0 (F x)) (
       Functor_fmap F2 (n x))) _).
    abstract (destruct F as (F', (T, L));
      destruct F2 as (F2', (T2, L2));
      destruct F1 as (F1', (T1, L1));
      destruct F0 as (F0', (T0, L0)); pose (L3 :=
         @Category_associativity_law _ _ _ _ c);
      simpl in *;
      congruence 36).
  Defined.
```

Finally, an independence result is proved: the second functor law does not imply the first.

```
Theorem Functor_1st_law_ind : ~ (forall (s : Sort) (c
    d : Category s) (f : c -> d) (ff : fmap_t f),
    Functor_2nd_law_t ff -> Functor_1st_law_t ff).
```

```
Proof.
  Require Export Counterexample.
  map counterexample (Type_Sort, (Type_Category,
     Type_Category)).
  map counterexample (fun (_:Type) => bool, (fun (a b
     : Type) (_:a->b) (_:bool) => false)).
  unfold not.
  splitI.
    reflexivity.

    splitI.
      exact unit.

      counterexample true.
      discriminate.
Qed.
```

10.3 Applicative functors

An applicative functor [29] is a kind of functor, which generalises a monad. The implementation, in the ApplicativeFunctor module, is simpler than that for Functor: it is exclusively implemented for the `Set_Category`, because applicative functors are an obscure concept in category theory, but useful in programming.

```
Section ApplicativeFunctor_defs.
  Variable af : Set -> Set.

  Record D_ApplicativeFunctor_Fields : Type := { pure
     : forall a, a -> af a; ap : forall a b, af (a ->
     b) -> af a -> af b }.
```

```
Variable af_fields : D_ApplicativeFunctor_Fields.

Notation "'s_ap'" := (let (_, ap) := af_fields in ap
   ).
Notation "'s_pure'" := (let (pure, _) := af_fields
   in pure).

Definition s_fmap : @fmap_t _ Set_Category
   Set_Category af := fun a b => comp Set_Category (
   s_pure _) (s_ap _ _).

Definition ApplicativeFunctor_hom_law_t
:= forall a b (f : a -> b),
comp Set_Category (s_pure _) (s_fmap f) = comp
   Set_Category f (s_pure _).

Definition ApplicativeFunctor_interchange_law_t
:= forall a b (u : af (a -> b)) (y : a),
s_ap _ _ u (s_pure _ y) = s_ap _ _ (s_pure _ (fun f
   => f y)) u.

Definition ApplicativeFunctor_composition_law_t
:= forall a b c w (v : af (a -> b)) (u : af (b -> c)
   ),
s_ap _ _ (s_ap _ _ (s_ap _ _ (s_pure ((b -> c) -> (a
    -> b) -> a -> c) (flip (@comp _ _ _ Set_Category
    _ _ _))) u) v) w = s_ap _ _ u (s_ap _ _ v w).

Record D_ApplicativeFunctor  : Prop :=
{ ApplicativeFunctor_functor_1st_law :>
   Functor_1st_law_t s_fmap
; ApplicativeFunctor_hom_law :>
   ApplicativeFunctor_hom_law_t
; ApplicativeFunctor_interchange_law :>
   ApplicativeFunctor_interchange_law_t
; ApplicativeFunctor_composition_law :>
   ApplicativeFunctor_composition_law_t }.
```

Here again, one of the laws is reformulated using the `ext_law` tactic for convenience reasons. Then, it is proved that an applicative functor is a functor – in particular, the functor whose `fmap` method is the composition of the `pure` method and the `ap` method of the applicative functor.

```
Variable m : D_ApplicativeFunctor.

(* The hom law with extensionality applied to it. *)
Lemma hom_law_ext : forall (a b : Set) (f : a -> b)
   (x : a), @s_fmap _ _ f (s_pure _ x) = s_pure _ (f
    x).
Proof.
  intro.
  ext_law (@ApplicativeFunctor_hom_law m) x.
Qed.

Lemma ApplicativeFunctor_Functor : D_Functor s_fmap.
Proof with auto.
  destruct m.
  constructor...
  red.
  red in ApplicativeFunctor_composition_law0.
```

```
    pose (H := @hom_law_ext).
    unfold s_fmap in *.
    simpl in *.
    Require Import Extensionality.
    iapplyFwd extensionality.
    intro.
    rewrite <- ApplicativeFunctor_composition_law0.
    unfold flip.
    f_equal.
    repeat rewrite H...
  Qed.

End ApplicativeFunctor_defs.
```

10.4 Monads

Monads, like functors, are here defined in the full category-theoretic sense, not merely for the `Set_Category`. Moreover, they are defined by way of a number of other category-theoretic notions in the Monad module, and then specialised to the `Set_Category` in the MonadOnSet module.

In the Monad module, the category of endofunctors on a category is defined. Then, this category is given a strict monoidal category structure. Monads then arise as the monoids (in the category-theoretic sense) of this strict monoidal category.

```
Section Monad_Defs.

  Variable s : Sort.
  Variable c : Category s.

  Definition endofunctor_D_Category_Fields
  := @Build_D_Category_Fields Type_Sort (Functor c c
     ) (fun F G => NaturalTransformation F G) (
     @comp_NaturalTransformation _ _ _) (
     @id_NaturalTransformation _ _ _).

Lemma endofunctor_D_Category : D_Category
   endofunctor_D_Category_Fields.
Proof.
  split; [ pose (L := @Category_associativity_law _
     _ _ _ c); destruct g; destruct h
           | pose (L := @Category_left_identity_law
               _ _ _ _ c)
           | pose (L :=
              @Category_right_identity_law _ _ _ _
              c) ]; destruct f; unfold
              NaturalTransformation; auto.
Qed.

Definition endofunctor_Category := @Build_Category _
    (Build_Category_Fields
   endofunctor_D_Category_Fields)
   endofunctor_D_Category.

Definition endofunctor_MonoidalStructure :=
   @Build_D_Category_Fields Type_Sort unit (fun _ _
   => Functor c c) (fun _ _ _ => @comp_Functor _ _ _
    _) (fun _ => id_Functor _).

Definition
   endofunctor_D_StrictMonoidalCategory_Fields :
```

```
   D_StrictMonoidalCategory_Fields
   endofunctor_Category.
Proof.
  apply (@Build_D_StrictMonoidalCategory_Fields
     endofunctor_Category
     endofunctor_MonoidalStructure).
  intros P1 P2 P3.
  simpl in *.
  exact (prod_NaturalTransformation (fst P3) (snd P3
     )).
Defined.

Ltac id_law_tac f
  := destruct f as ((F, fmap), d);
      applyFwd sig_eq';
      simpl;
      apply Functor_Fields_eq;
      [
      | replace fmap with (fun (a0 b0 : @Category_ob
          s (@Category_fields s c))
       (x : @Sort_Type_Coercion s
              (@Category_hom s (@Category_fields s
                 c) a0 b0)) =>
          fmap a0 b0 x)].

Lemma endofunctor_D_StrictMonoidalCategory :
   D_StrictMonoidalCategory
   endofunctor_D_StrictMonoidalCategory_Fields.
Proof.
  split.
```

```
    split; intros; [ simpl; unfold comp_Functor |
       id_law_tac f | id_law_tac f ]; unfold Functor
       ; auto.

    Require Import Extensionality.
    split; red; iapplyFwd sig_eq'; apply
       extensionality.
      intro.
      destruct a.
      simpl.
      rewrite (Functor_1st_law (proj2_sig c1)).
      rewriter (@Category_left_identity_law _ _ _ _
         c).
      unfold prod_rect, prod_NaturalTransformation.
      destruct f.
      destruct g.
      simpl in *.
      destruct c0.
      destruct c1.
      destruct c2.
      destruct c3.
      destruct y.
      destruct f0.
      elim d.
      pose (L1 := @Category_associativity_law _ _ _
         _ c).
      simpl in *.
      congruence 48.
Qed.
```

```
Definition endofunctor_StrictMonoidalCategory :
    StrictMonoidalCategory
  := exist _ (Build_StrictMonoidalCategory_Fields
       endofunctor_D_StrictMonoidalCategory_Fields)
       endofunctor_D_StrictMonoidalCategory.

Definition Monad := { smf : D_StrictMonoid_Fields
    endofunctor_StrictMonoidalCategory |
    D_StrictMonoid smf }.
```

In the MonadOnSet module, it is proven that monads in the `Set_Category` are applicative functors. This is quite an involved proof, so some subsidiary lemmas are proved. However, instead of writing them as lemmas outside of, and prior to, the main proof, in the normal way, a user-defined tactic `nest` is used to allow the lemmas to be stated and proved within the context of an environment generated by tactics within the main proof. This is convenient, and although the exported lemmas are not stated in terms of this environment (because it does not exist outside the proof!) they can mostly be rewritten back into their original stated forms in other proof scripts, with some judicious tactic applications.

Unfortunately, due to a bug/infelicity in Coq 8.1pl3, the exported lemmas are named lemma0, lemma1, etc. instead of something meaningful. This has since been fixed in Coq's source code repository.

This huge proof is not included here, for space reasons.

10.5 Monad transformers and applicative functor transformers

A monad transformer is a type constructor which takes a monad as an argument and returns a monad as a result.

Monad transformers [24] can be used to compose features encapsulated by monads – such as state, exception handling, and I/O – in a modular way. Typically, a monad transformer is created by generalising an existing monad. Applying the resulting monad transformer to the identity monad yields a monad which is equivalent to the original monad. This fact can be capitalised on to potentially reduce proving effort.

In the Reader module, a monad transformer is defined for the Reader monad, using the SemiUniformMonadTransformer type class (see section 9.1.5). Thanks to the structure of category theory and the proofs in other modules, the Reader monad, the Reader applicative functor, and the Reader applicative functor transformer, all follow automatically from this definition – without the need for any further proofs.

The long set of proofs in the Reader module are omitted here for space reasons.

In the MonadTransformer module, the monad transformer first law is actually not treated as a law, but as a theorem which can be proven with just the `auto` tactic. This is because, instead of defining a monad transformer in the normal Haskell way, the `lift` and `mult` methods are defined to be natural transformations. From these two methods, the created monad can be defined. The monad transformer first law follows as an easy consequence.

```
Record D_MonadTransformer_Fields : Type :=
 { MonadTransformer_lift : forall m,
    NaturalTransformation m (mt m)
 ; MonadTransformer_mult : forall m,
    NaturalTransformation (comp_Functor (mt m) (mt
    m)) (mt m) }.

 Variable mtf : D_MonadTransformer_Fields.

 Section WithM.
```

```
  Variable m : t.

  Definition MonadTransformer_Fields_Monad_Fields
      : D_StrictMonoid_Fields (
      endofunctor_StrictMonoidalCategory
      Set_Category).
  Proof.
    pose (lift := @MonadTransformer_lift mtf).
    pose (mult := @MonadTransformer_mult mtf).
    unfold Monad_Functor in *.
    econstructor; simpl.
      eauto.

      instantiate (1 := m).
      Unset Printing Implicit.
      applyFwd (@comp _ _ _ (endofunctor_Category
          Set_Category)).
        set (T := StrictMonoid_unit m).
        simpl in *.
        eexact T.

        eauto.
  Defined.

Theorem MonadTransformer_1st_law : comp (
    endofunctor_Category Set_Category) (
    @StrictMonoid_unit _ m) (@MonadTransformer_lift
     mtf _)
    = @StrictMonoid_unit _
      MonadTransformer_Fields_Monad_Fields.
  Proof.
    auto.
  Qed.
```

An applicative functor transformer is very similar to a monad transformer, but for applicative functors – indeed, monad transformers *are* applicative functor transformers. However, thanks to the compositionality of applicative functors, any applicative functor can automatically be converted into an applicative functor transformer. This is proved in the `composition_ApplicativeFunctorTransformer` lemma in the ApplicativeFunctorTransformer module. Thus, in contrast to the situation with monads, when defining an applicative functor which does *not* qualify as a monad, it is wise (to save effort) to define it as an applicative functor, and generate an applicative functor transformer from that.

The long set of proofs in the ApplicativeFunctorTransformer module are omitted here for space reasons.

Part III

Discussion

Chapter 11

General observations on proof techniques used

In this Chapter, some general observations on issues relating to proofs throughout this work are raised. The next and final Chapter then sums up progress made on each of the key components of the implementation in turn, and concludes.

11.1 Reducing reliance on axioms

It may be possible to reformulate this work to reduce its reliance on certain axioms. It is not clear, however, that removing well-known axioms like the functional extensionality axiom would *necessarily* be an improvement: they are convenient and tend to lead to shorter proofs.

On the other hand, reducing the number of axioms relied upon may lessen the chance of accidentally using an inconsistent set of axioms. A lot of thought has gone into avoiding inconsistencies in the choice of axioms in this work, but the possibility cannot be entirely ruled out.

11.1.1 Removing the unlimited resources assumption

It should be possible to modify the theorems and proofs in this work to avoid using the functional extensionality axiom entirely. However, doing so would entail at least the following:

- creating an equivalence relation to simulate functional extensionality;
- changing the relevant proofs to do rewriting using equivalences, instead of using facts about equality. This could make the proofs more verbose.

It should not be necessary to reimplement the category-theoretic work to allow categories to be parameterised over arbitrary setoids, however. The single equivalence relation just mentioned should be sufficient for this work, since it is equivalent to the notion of equality used so far.

11.1.2 Removing the assumption of injectivity of dependent equality

This work uses an axiom (`JMEq_eq`) from the Coq standard library, which states that heterogenous "John Major" equality implies ordinary equality. This axiom is necessary to use John Major equality meaningfully (without using setoids), and is equivalent to the axiom of injectivity of dependent equality. This in turn is equivalent to the axiom that equal dependent pairs are pairwise equal.

Preliminary experiments in an unrelated project have shown that in some cases, banishing all heterogenous equality types leads to the sprouting up of `eq_rect` terms all over the place. This can lead to excessively long and unreadable proof subgoals, which can be difficult to work with. It is possible to develop new tools to deal with this complexity – some of which might be useful more generally for other complex proof terms – but heterogenous equality certainly seems more convenient.

Furthermore, injectivity of dependent equality has powerful intuitive appeal. Its negation implies that there exists a pair of distinct types which share one or more inhabitants. Yet in Coq, each term is supposed to have a single, unambiguous type.

11.1.3 Restricting proofs to constructive logic

Classical logic – which involves adding a classical axiom to Coq's constructive logic foundations – has been used in this work, essentially only to prove certain metatheoretical properties. By splitting off the metatheory into a separate project and then importing it as axioms – which is a desirable goal in its own right – the dependency on classical logic could be removed, which would put the project on an intuitionistic foundation. Constructive logic is appealing for both philosophical and practical reasons: by exhibiting how to construct a given entity, constructive proofs both provide philosophical certainty that that entity exists within a given formal framework, and an effective (though not necessarily efficient) algorithm for constructing it.

11.2 Proof technology

Coq is a mature proof assistant: apart from a few minor issues relating to stack overflows in the IDE, CoqIDE, and universe inconsistencies, Coq was capable of dealing with everything this project could throw at it – both in terms of being expressive enough to represent it, powerful enough to prove it, and efficient enough to work with it effectively.

However, proving in Coq is an extremely time-consuming procedure – even when using CoqIDE:

- Layout conventions have been developed in the Coq user community to make Coq scripts easier to follow, and this project uses such a layout convention. However, CoqIDE has no support for auto-indenting new lines, or reindenting a block of lines – let alone automatically handling a layout convention. It is possible to use an external editor, however – or switch to emacs with Proof General, which does have better indentation support and is scriptable.

- Autocompletion is offered as an option in CoqIDE to speed up typing, but this can be distracting and irritating.

- Some tactics such as “auto”, “eauto” and “congruence” have optional integer arguments to limit search depth. The highly-conscientous developer thus wastes time tweaking these arguments to make their proof scripts run fastest without breaking them, when this is something the system could do for them.

- Long, complex proof terms are difficult to navigate in CoqIDE, because it has no support for matching parentheses or collapsing subterms in the subgoal pane – and even scrolling is prone to distracting overshoots. Again, Proof General has an advantage here: emacs supports outlining.

- Although this project has used the tactic language to create new tactics, there is still plenty of scope for automation. Certain tactics, such as “destruct” and “rewrite”, seem to be less convenient to use in automation. The restriction that “auto” must either solve a subgoal, or fail to do anything, seems somewhat arbitrary and limiting.

- It is probable that major gains in productivity could be achieved in practice by automatically trying tactics in the background in CoqIDE and suggesting them to the user.

- Work to integrate Coq with external provers is at an early stage and not very well-signposted.

- “simpl”, the simplification tactic, is amazing – but somewhat mysterious: its documentation does not appear to be up-to-date – and not easy to extend, for those unfamiliar with ML.

- This project involved frequent rounds of revising/refactoring proofs to make them shorter and tighter – partly for the benefit of clarity for future reference, and partly as an exercise to educate the author in writing better proofs. However, Coq offers no explicit support for these activities. Such support could range over:

- re-laying out indented proofs (as mentioned above)
- cut elimination
- automatically checking for – and removing – redundant lines in a proof
    - e.g. coalescing lines prior to an "auto" tactic invocation into that auto, where possible
- or even trying to reorder proof scripts to find hidden redundant lines

In hindsight, this project might have been slightly more productive had Proof General been used throughout, instead of CoqIDE.

Chapter 12

Conclusions and future work

12.1 Type class idiom

The type class idiom presented here is at an early stage of development. One problem with it is that the prefixed names created by the guidelines, such as `SemiUniformApplicativeFunctor_interchange_law`, can become overlong. It would have been possible to make use of the existing top-level module structure as a namespace mechanism and dispense with the prefixes. However, the current approach provides some much-needed clarity to often abtruse category-theoretic proofs and definitions. Perhaps all that is needed is some judicious abbreviation.

As previously stated, future work could include trying to use the type class idiom in other contexts, to see how well it shapes up and perhaps to provide more readily comprehensible examples of using the type class idiom. Eventually, if it proved enduringly useful, it might be worthwhile to develop a preprocessor or language extension to facilitate following the type class guidelines presented here. However, just before this thesis was submitted, Coq version 8.2, which included a different implementation of first-class type classes [44], was released.

Before developing a language extension for this type class idiom, then, it would be necessary to make a thorough comparison of the two approaches, to see if the present approach offers any major advantages over that implementation. Perhaps the use of superparameters, or the method of safely supporting diamond inheritance, would be significant advantages. Even then, it might be possible to incorporate some features into the Coq language implementation, rather than working on a separate project.

As previously discussed, the type class idiom presented here is different in many respects from the implementations of type classes in various implementations of Haskell. In particular, it is an idiom with no tool support (yet), so things like type inference and error reporting have room for improvement. However, the ability to specify an instance explicitly where necessary, without having to wrap data in wrapper types, can be viewed as a distinct advantage over the Haskell type class system.

12.2 Parametricity

The parametricity theory in this work was developed in a rather ad-hoc, "modify the definitions to fix a bug" style – and without awareness of prior work on parametricity in other languages. It was only after the implementation phase was completed that the author became aware of the related work in this area, unfortunately.

The first priority for future work would be to take advantage of the last-minute insight that semi-uniform types are all monocontainers, and reformulate the definitions and proofs in this work accordingly. The second order of business for future work would be to compare the theory built up here with the more abstract, category-theory-based theory in [22], which – like the present work – targets indexed types (in the form of Generalised Abstract Data Types).

A more complete theory of parametricity in Coq is necessary in order to fairly assess the benefits and costs of using parametricity results (and an outgrowth of this work, definite description results) to attempt to streamline proofs. At the moment, it is unclear whether the complex and axiom-heavy (and hence error-prone) approach used here is really deserving of the name "theorems for free". Due in part to the expressive power of the Coq programming language, establishing that a type is semi-uniform may be more daunting than simply proving the theorem one wanted to avoid proving in the first place! However, perhaps after being proven once, semi-uniformity results can be reused multiple times.

12.3 Formalising category theory

By defining the methods for a monad transformer, and proving the required laws, this framework supports the verification of the monad transformer, and also the automatic generation of the associated monad and applicative functor (that is, the result of applying the monad transformer to the identity monad, and the applicative functor generated by that monad). If the monad transformer is based on a semi-uniform type, the developer has to prove even less to achieve these results (assuming that the evidence that the type is semi-uniform is a given).

There have been several previous formalisations of category-theoretic concepts in proof assistants ([38] provides a review, and a formalisation of its own), but the author is not aware of any that provide all the same features. However, Schröder's work on formalising a monad-independent dynamic logic for monads in HasCasl [42] is notable because it considers side-effects and how they may be reasoned about, which is not covered at all in this thesis. His work is somewhat different, however, in several respects. For example, it focuses on monads, and it hews closer to the semantics of Haskell, by supporting partial functions. However, the presence of partial functions can (and indeed does in this case) complicate reasoning. For this reason, it was deliberately chosen not to support them in this thesis, since they can still be simulated with techniques such as the

Maybe monad – and indeed, often are, in Haskell, even though Haskell allows partial functions. Schröder's paper is also more closely related, in some sense, to Moggi's original seminal paper [34] (which introduced monads for semantics, rather than for use in programming directly). Clearly, the work in this thesis would benefit from being extended to be able to reason about the side effects of monads, and Schröder's paper is likely to offer useful insights in that direction.

The framework is rather technical, but formal verification is a very technical field, perhaps inherently so – and designing and implementing a new monad is not something that beginner programmers are likely to try very often. One area that could be improved is that the definition of the monad type class is highly category-theoretic, compared to Haskell. It would be possible with relatively little effort to derive a simpler type class for monads in the `Set_Category` only, and prove that this simpler class was equivalent to the old one.

The `Type_Category` did not really work as planned, because it led to universe inconsistencies – but this is quite a minor problem at the moment. Still, much research on applying category theory to computer programming has focused on just one category – typically the Set category. Experimenting with categorical programming in multiple categories is an intriguing possibility for future research. Many useful concepts in mathematics can be derived – and generalised – category-theoretically. For example, a set with a total order relation forms a category, with $\leq$ being the set of morphisms. Endofunctors on this category are monotonic functions on the carrier set.

Unfortunately, only one example verified monad, the Reader monad, has been developed during the course of this work, due to time constraints. Future work could obviously include defining other well-known monads (or their corresponding monad transformers), and identifying common proof problems for possible factoring out into lemmas and/or tactics.

Appendix A

Type Theory and the Curry-Howard correspondence

Constructive Type Theory (or just Type Theory) in its various incarnations is the theoretical foundation for dependent types. For our purposes, the key type-formers in Constructive Type Theory are:

Σ (dependent sum) which forms pairs in which the type of the second element can depend upon the value of the first, and

Π (dependent product) which forms functions in which the type of the result can depend on the value of the argument.

The Curry-Howard correspondence (or isomorphism) is of fundamental importance to dependent types and indexed types. For our purposes, we can express it as follows:

Curry-Howard Given a typed language L (for example, Type Theory):

1. There exists a logic C, such that:

2. There exists a computable bijection F taking each well-formed type or type constructor T in L to a well-formed proposition (or predicate, respectively) P in C, and
3. There exists a computable bijection G taking each type T and inhabitant t of type T to a *constructive* proof of $F(T)$ in C.[1]

Curry-Howard is simpler to define rigorously in a language without partial functions, because partial functions complicate the notion of inhabitance. Fortunately, Type Theory has no partial functions – although they can of course be modelled in it – and so for Curry-Howard from Type Theory to a logic, inhabitant simply means value.

The definition above just states the *existence* of correspondence functions F and G. It is also necessary to specify the correspondence rules for each type-former, which together make up F and G (which obviously are defined at the meta-level, not in L itself). For example, the two type-formers Σ and Π above correspond to existential and universal quantification, respectively. Further details will not be given here, for reasons of space.

Note that Type Theory is constructive in the sense that undecidable oracle functions are not allowed, because they would correspond to non-constructive proofs (although, as has been shown with respect to the Predicative Calculus of Inductive Constructions, it is not necessarily inconsistent to add axioms corresponding to such functions).

However, type constructors correspond to *predicates*, and type constructors are not themselves types, so it's perfectly all right if their corresponding predicates are undecidable. The same goes for certain parts of type expressions. Even types corresponding to undecidable propositions can be formed, but they will not be inhabited – even when they correspond to true propositions. Curry-Howard is about *proofs* – it does not cover all truths.

Of course (just to add to the confusion), undecidable predicates are banned in Constructive Type Theory *itself* – because there, predicates are just boolean-

[1]There is also a correspondence between β-reduction and cut elimination.

valued functions, and functions must be decidable. For clarity, therefore, I distinguish between logic-predicates (as in the previous paragraph) and function-predicates (as in this paragraph) where appropriate.

Appendix B

Electronic Appendix: Coq theory listings

This is an electronic-only appendix, available at:

`http://greenrd.pbworks.com/Thesis`

Readers are recommended to step through it in CoqIDE or Proof General as necessary, to see how it works, rather than to just try to understand it by reading it, which is an impossible task. Prefixing a tactic invocation with `info` can be very useful to see which primitive tactics are invoked by a high-level tactic.

Bibliography

[1] *Documentation of the Coq Proof Assistant (Version 8.1)*
URL `http://bit.ly/cYhfvW`

[2] Altenkirch, T; McBride, C; *et al.* Why dependent types matter, 2005
URL `http://bit.ly/csjtyY`

[3] Augustsson, Lennart. Cayenne - a language with dependent types. In *International Conference on Functional programming.* Baltimore, Maryland, United States: ACM. ISBN 1-58113-024-4, 1998 pp. 239–250
URL `http://dx.doi.org/10.1145/289423.289451`

[4] Barendsen, Erik; Smetsers, Sjaak. Uniqueness typing for functional languages with graph rewriting semantics. In Mathematical Structures in Computer Science, 1996. vol. 6, pp. 579–612

[5] Beck, Kent. Embracing change with extreme programming. In Computer, 1999. vol. 32, pp. 70–77. ISSN 0018-9162
URL `http://dx.doi.org/10.1109/2.796139`

[6] Berger, Ulrich; Schwichtenberg, Helmut; *et al.* The Warshall algorithm and Dickson's lemma: Two examples of realistic program extraction. In Journal of Automated Reasoning, Feb. 2001. vol. 26, pp. 205–221
URL `http://dx.doi.org/10.1023/A:1026748613865`

[7] Bertot, Yves; Castéran, Pierre; *et al.* *Interactive Theorem Proving and Program Development: Coq'Art : the Calculus of Inductive Constructions.* Springer, 2004. ISBN 3540208542

[8] Burdy, Lilian; Cheon, Yoonsik; *et al.* An overview of JML tools and applications. In International Journal on Software Tools for Technology Transfer (STTT), 2005. vol. 7, pp. 212–232
URL `http://dx.doi.org/10.1007/s10009-004-0167-4`

[9] Casey, Steven M. *Set Phasers on Stun: And Other True Tales of Design, Technology, and Human Error.* Aegean, 1998. ISBN 0963617885

[10] Chakravarty, Manuel; Finne, Sigbjorn; *et al.* The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report, 2002
URL `http://bit.ly/cPYM2A`

[11] Charette, Robert N. Why software fails. In IEEE Spectrum, Sep. 2005
URL `http://bit.ly/9k9gKS`

[12] Chlipala, Adam. Position paper: Thoughts on programming with proof assistants. In *Programming Languages meets Program Verification 2006*, vol. 174 of *Electronic Notes in Theoretical Computer Science.* Elsevier, Jun. 2007 pp. 17–21
URL `http://dx.doi.org/10.1016/j.entcs.2006.10.035`

[13] Claessen, Koen; Hughes, John. Quickcheck: a lightweight tool for random testing of Haskell programs. In *International conference on Functional programming.* ACM. ISBN 1-58113-202-6, 2000 pp. 268–279
URL `http://dx.doi.org/10.1145/351240.351266`

[14] Collins, Graham. Supporting reasoning about functional programs: An operational approach. In Turner, David N. (ed.), *Functional Programming*, Workshops in Computing. Springer. ISBN 3-540-14580-X, 1995 p. 3

[15] Day, Nancy; Launchbury, John; *et al.* Logical abstractions in Haskell. In *Haskell Workshop*, 1999

[16] EUROPA. Ip/07/1567: Antitrust: Commission ensures compliance with 2004 decision against microsoft. Press Release, 2007
URL `http://bit.ly/aRDp76`

[17] Fogarty, Seth; Pasalic, Emir; *et al.* Concoqtion: indexed types now! In *Partial evaluation and semantics-based program manipulation*. Nice, France: ACM. ISBN 978-1-59593-620-2, 2007 pp. 112–121
URL `http://dx.doi.org/10.1145/1244381.1244400`

[18] Glass, Robert L. The Standish report: does it really describe a software crisis? In Communications of the ACM, 2006. vol. 49, pp. 15–16. ISSN 0001-0782
URL `http://dx.doi.org/10.1145/1145287.1145301`

[19] Gonthier, Georges. A computer-checked proof of the four colour theorem, 2004
URL `http://bit.ly/czah4K`

[20] Gosling, J.; Joy, B.; *et al. The Java Language Specification*. Addison-Wesley Professional, 3rd edn., 2005. ISBN 0-321-24678-0

[21] Grand, Mark. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, vol. 1. New York: John Wiley & Sons, 2002. ISBN 0471258393

[22] Johann, Patricia; Ghani, Neil. Foundations for structured programming with GADTs. In *Principles of Programming Languages 2007*, vol. 43. New York: ACM, 2008 pp. 297–308
URL `http://dx.doi.org/10.1145/1328897.1328475`

[23] Jones, Mark P. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming: First Inter-*

national Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, vol. 925 of *Lecture Notes on Computer Science*. Springer. ISBN 3540594515, 1995 pp. 228–266

[24] Liang, Sheng; Hudak, Paul; *et al.* Monad transformers and modular interpreters. In *Principles of programming languages*. San Francisco, California, United States: ACM. ISBN 0-89791-692-1, 1995 pp. 333–343
URL `http://dx.doi.org/10.1145/199448.199528`

[25] Longo, Giuseppe; Milsted, Kathleen; *et al.* The genericity theorem and parametricity in the polymorphic lambda-calculus. In Theoretical computer science, 1993. vol. 121, pp. 323–349

[26] McBride, Conor. Faking it: Simulating dependent types in Haskell. In Journal of Functional Programming, 2003. vol. 12, pp. 375–392
URL `http://dx.doi.org/10.1017/S0956796802004355`

[27] McBride, Conor. Epigram: Practical programming with dependent types. In *Advanced Functional Programming 2004, Tartu, Estonia*, vol. 3622 of *Lecture Notes in Computer Science*. Springer, 2005 pp. 130–170

[28] McBride, Conor; McKinna, James. The view from the left. In Journal of Functional Programming, 2004. vol. 14, pp. 69–111
URL `http://dx.doi.org/10.1017/S0956796803004829`

[29] McBride, Conor; Paterson, Ross. Applicative programming with effects. In Journal of Functional Programming, 2008. vol. 18, pp. 1–13
URL `http://dx.doi.org/10.1017/S0956796807006326`

[30] Microsoft. Microsoft security bulletin MS08-001 – critical: Vulnerabilities in Windows TCP/IP could allow remote code execution (941644), 2008
URL `http://bit.ly/aBpU7A`

[31] MITRE. CVE-2008-0009 (under review), 2008
URL `http://bit.ly/d7UUnd`

[32] MITRE. CVE-2008-0010 (under review), 2008
URL `http://bit.ly/9ciDVN`

[33] MITRE. CVE-2008-0600 (under review), 2008
URL `http://bit.ly/a4uJX6`

[34] Moggi, Eugenio. Notions of computation and monads. In Information and Computation, 1991. vol. 93(1), pp. 55–92

[35] Moore, David; Shannon, Colleen; *et al.* Code-red: a case study on the spread and victims of an internet worm. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement.* New York, NY, USA: ACM. ISBN 1-58113-603-X, 2002 pp. 273–284
URL `http://dx.doi.org/10.1145/637201.637244`

[36] Morris, Peter. Ecce isn't Epigram 2. In *Dependently Typed Programming.* Nottingham, 2008 (Presentation slides)
URL `http://bit.ly/bLimz5`

[37] Nipkow, Tobias; von Oheimb, David; *et al.* μJava: Embedding a programming language in a theorem prover. In Bauer, Friedrich L.; Steinbrüggen, Ralf (eds.), *Foundations of Secure Computation*, vol. 175 of *NATO Science Series F: Computer and Systems Sciences.* IOS Press, 2000 pp. 117–144
URL `http://bit.ly/bfi7f0`

[38] O'Keefe, Greg. Towards a readable formalisation of category theory. In Electronic Notes in Theoretical Computer Science, 2004. vol. 91, pp. 212–228
URL `http://dx.doi.org/10.1016/j.entcs.2003.12.014`

[39] Petäjäsoja, Sami; Mäkilä, Tommi; *et al.* Wireless security: Past, present and future, Feb. 2008
URL `http://bit.ly/9WpU18`

[40] PEYTON JONES, SIMON L. Haskell 98 language and libraries: The revised report. In Journal of Functional Programming, 2003. vol. 13. ISSN 0956-7968
URL `http://bit.ly/9BwNmC`

[41] PEYTON JONES, SIMON L.; WADLER, PHILIP. Imperative functional programming. In *Principles of programming languages*. Charleston, South Carolina, United States: ACM. ISBN 0-89791-560-7, 1993 pp. 71–84
URL `http://dx.doi.org/10.1145/158511.158524`

[42] SCHRODER, LUTZ; MOSSAKOWSKI, TILL. Monad-independent dynamic logic in HasCasl. In J Logic Computation, 2004. vol. 14(4), pp. 571–619
URL `http://dx.doi.org/10.1093/logcom/14.4.571`

[43] SOZEAU, MATTHIEU. Program-ing finger trees in Coq. In *International conference on Functional programming*. ACM. ISBN 978-1-59593-815-2, 2007 pp. 13–24
URL `http://dx.doi.org/10.1145/1291151.1291156`

[44] SOZEAU, MATTHIEU; OURY, NICOLAS. First-class type classes. In MOHAMED, OTMANE AÏT; MUÑOZ, CÉSAR; *et al.* (eds.), *TPHOLs*, vol. 5170 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-540-71065-3, 2008 pp. 278–293
URL `http://dx.doi.org/10.1007/978-3-540-71067-7_23`

[45] TAIVALSAARI, ANTERO. On the notion of inheritance. In ACM Computing Surveys, 1996. vol. 28, pp. 438–479
URL `http://dx.doi.org/10.1145/243439.243441`

[46] TASSEY, GREGORY. The economic impacts of inadequate infrastructure for software testing, 2002
URL `http://bit.ly/94fKTm`

[47] Thompson, Simon. Formulating Haskell. In Launchbury, John; Sansom, Patrick (eds.), *Functional Programming*, Workshops in Computing. Glasgow: Springer-Verlag, 1992

[48] Wadler, Philip. Theorems for free! In *Functional programming languages and computer architecture.* Imperial College, London, United Kingdom: ACM. ISBN 0-89791-328-0, 1989 pp. 347–359
URL `http://dx.doi.org/10.1145/99370.99404`

[49] Wadler, Philip. Comprehending monads. In *LISP and functional programming, 1990.* Nice, France: ACM. ISBN 0-89791-368-X, 1990 pp. 61–78
URL `http://dx.doi.org/10.1145/91556.91592`

[50] Wadler, Philip; Blott, Stephen. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages.* Austin, Texas, United States: ACM. ISBN 0-89791-294-2, 1989 pp. 60–76
URL `http://dx.doi.org/10.1145/75277.75283`

[51] Wehr, Stefan. *ML Modules and Haskell Type Classes: A Constructive Comparison.* Master's thesis, Albert-Ludwigs-Universität, Freiburg, Germany, Nov. 2005
URL `http://bit.ly/biZRai`

Index