

Constructs & Concepts

Language Design for Flexibility and Reliability

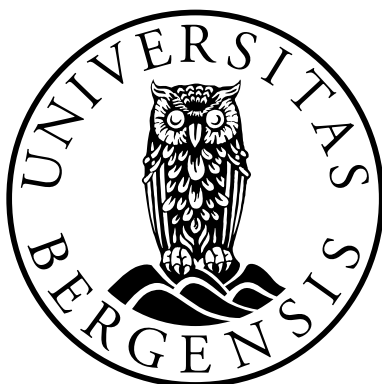
ANYA HELENE BAGGE



Constructs & Concepts

Language Design for Flexibility and Reliability

ANYA HELENE BAGGE



Dissertation for the degree of philosophiae doctor (PhD)
at the University of Bergen

November 2009

ISBN 978-82-308-0887-0

University of Bergen, Norway. Submitted 2009-07-20, final print 2009-10-23.

Papers reprinted by permission from ACM, Elsevier and Springer. All rights reserved.

All other text, illustrations and photos © 2009 Anya Helene Bagge.

Printed by AIT.

Prepared with \LaTeX and set in *Nofret* by Gudrun Zapf von Hesse, with program code in `txtt`.

Frontispiece: Magnolia flower made up of the entire source code of the Magnolia compiler.



To my parents

Contents

Preface	ix
1 Introduction	1
1.1 The Software Quality Toolbox	2
1.2 Domain, Background & Motivation	8
1.3 Magnolia	9
1.4 Method and Outline	14
2 Playing with Signatures	17
2.1 Introduction	20
2.2 The Magnolia Language	23
2.3 Relating Functions and Procedures	24
2.4 Discussion	29
2.5 Conclusion	33
3 Working with Axioms	35
3.1 Introduction	37
3.2 Concepts and Axioms	39
3.3 Rewriting with Axioms	41
3.4 Axioms for Testing	46
3.5 Implementation Issues	48
3.6 Discussion	49
3.7 Conclusion	51
4 Axiom-Based Testing	53
4.1 Introduction	55
4.2 Concepts	58
4.3 From Axioms to Test Code	59
4.4 Generating Test Data	65
4.5 Discussion	70
4.6 Conclusion	75
5 Handling Failure and Exceptions	77
5.1 Introduction	80

5.2	Problem	82
5.3	Separation of Concerns	86
5.4	Alert Language Extension	87
5.5	Implementation	99
5.6	Related Work	101
5.7	Conclusion	103
6	The Magnolia Programming Language	107
6.1	Introduction	107
6.2	Signatures, Concepts and Implementations	108
6.3	Procedural Abstractions	108
6.4	Type System	116
6.5	Data Abstraction	119
6.6	Expressions and Statements	125
6.7	Concepts and Axioms	125
6.8	Genericity	127
6.9	Language Implementation	128
6.10	Status of the Magnolia Implementation	131
7	Supporting Language Extensions	133
7.1	Introduction	135
7.2	The Magnolia Language	136
7.3	Extending Magnolia	137
7.4	Conclusion	145
8	Discussion	147
8.1	Related Work	147
8.2	Evaluation	155
8.3	On Language Design	156
8.4	Future Work	160
9	Conclusion	163
	Summary	167
	References	169
	Citation Index	187
	Index	191

Preface

Most PhD dissertations end up being read by very few people, which is kind of frustrating, considering the amount of time that usually goes into writing one. So if you have received a nice, printed copy of this one, I hope you appreciate it, and that it will look good on your bookshelf.

If you are actually going to read it, I hope you enjoy it and find it interesting – it is a result of years of hard work, stress, anguish, intense discussions, as well as newspaper-reading, hanging around, not hanging around and general procrastination. Oh, and – of course – struggles to make the typesetting look *just right* within (well, mostly outside) the rather narrow constraints set by the University style.

Bergen, 2009-07-20

ACKNOWLEDGEMENTS

In addition to all the effort by the candidate, a PhD dissertation is also a result of (less) hard work, stress, anguish and intense discussions on the part of the supervisor(s). Mine is no different, and my main supervisor Magne Haveraaen deserves much of the credit for the finished product. Thanks! Thanks also to co-supervisor Eelco Visser, for teaching me more or less everything I know about program transformation, for providing a good counter-point to Magne's research style and for taking care of me during my stay in Utrecht.

Many other thanks are due after a project of this size. First and foremost, thanks to May-Lill Sande for all her love, support, and friendship throughout the past twelve years of my life. Thanks to WALL-E and EVE for being bright and cheerful every morning. Thanks also to my family – to my parents for all their support, to my brother Jon Bagge for getting me interested in computers, to my sister Elin Rotevatn for getting me interested in everything else, and to my nieces Emilie and Mathilde for endless vacation fun.

At the University, many of my research plans and schemes have been hatched together with Karl Trygve Kalleberg – if he can just get over his fascination with medicine, maybe we can revisit some of them. Valentin

David has been a walking C++ standards document, and provided invaluable help with the intricacies of C++ and of SDF2 parsing. Eva Burrows has been a good friend and travel mate, and given me endless hours of discussion about almost anything and many welcome breaks from tedious work. She has also provided useful feedback on this dissertation. Joseph Young always has helpful advice, and has given me insight into semantics and type theory. Ida Holen has provided invaluable help and encouragement, first as study adviser and then as my boss. I have also had much fun discussing study-related matters with the new advisers Inger Nilsen and Siv Hovland Erstad. Daniele Jesus has challenged me on Cell-related matters, and has also been a fun travel companion. Many welcome breaks have been spent in Paul Simon Svanberg's office, or hanging at his door while waiting for print-outs. He has also provided invaluable help in organising my defence dinner. Travelling with Alessandro Rossini and Adrian Rutle was a lot of fun, though I doubt my liver can take much more of it... Thanks to my students as well, for giving me fun and interesting challenges.

Internationally, Jan Heering helped get me started in this direction in the first place, and Jurgen Vinju has been a welcome friend at various conferences. Sriram Sankaranarayanan has provided valuable hints for my trial lecture.

The Department of Computer Science at the University of Bergen has given me a place to work, financial support and lots of opportunity for travelling. My stay in Utrecht would have been impossible without the financial support of the Research Council of Norway.

All my research would be worth nothing without good friends to share my spare time with. Eli Kristine Hausken has been an invaluable source of fun, and has also enabled me to (partly) realise my secret desire to be an Egyptologist. She is also much to blame for making me show up at work, by letting me ride with her in the mornings. Eskil and Toril Saatvedt and Tone Storemark has been good friends over the past years, with countless hours spent playing Buffy the Vampire Slayer (sometimes also with Aleksander Gjøen and Eli Kristine) and other games.

Thanks, finally, to my opponents Don Sannella and Russel Winder, for all the time they have spent reviewing this work – it is much appreciated; and to Marc Bezem for leading the committee.

This dissertation would have been a typographic mess, were it not for Robert Bringhurst's *The Elements of Typographic Style* – any remaining messes are entirely of my own doing. My final thanks go to Donald E. Knuth for getting me interested in design and typesetting through his books and software. Let's hope I won't lose as many years to it as he did...

Scientific Environment

The research presented in this dissertation has been conducted in the Programming Theory group of the Department of Informatics at the University of Bergen, and while visiting the Software Technology group of the Department of Information and Computing Sciences at Utrecht University.

ICT

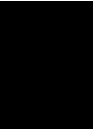
**Research School in
Information and Communication Technology**

UNIVERSITY OF BERGEN
Department of Informatics





Texas State Capitol
Austin, TX (Supercomputing 2008)



Introduction

Developing high-quality software is a major challenge, particularly as project size grows. Ultimately, the burden rests on the software engineer, but over time, programming languages and development tools have evolved to aid in constructing quality software.

We may measure quality along many axes. For example, reliability, usability, flexibility, maintainability, portability, efficiency, security are commonly listed as desired software traits. We shall take a closer look at a few of them.

SOFTWARE
QUALITY

Reliability implies that the program gives consistently correct results. In particular, if given valid inputs, the program should arrive at a correct result, and furthermore, the program should not present false results as if they were correct.

Robustness is sometimes considered part of reliability, and implies that a program deals appropriately with errors and unexpected situations. If given invalid data, the program should report it properly, and handle it in some appropriate way. A program should avoid giving seemingly valid results for invalid input. Any unexpected error conditions (network outage, disk error, etc) should be detected and handled.

Flexibility is to what degree the program may be adapted to changing circumstances without having to rewrite large parts of the code. Flexibility is of particular importance for libraries, which are typically meant for heavy reuse – but flexibility in application code is useful as well, to keep up with new user demands, changing specifications (tax rules, for example) and changing hardware architectures (for numerical software).

Maintainability is how easy it is to find and remove bugs, and do other large and small modifications over the lifetime of the software. Program code is *brittle* if a change in one place is likely to break the code in other – often unexpected – places. Brittleness can make modifications so

BRITTLINESS

difficult that scrapping the old code and writing new may be easier than adapting existing code. The other quality factors mentioned above have an impact on maintainability, but also other factors such as code clarity and documentation can also be important for maintainable code.

Software quality isn't an either-or proposition, and we may be able to live with a less-than-perfect score on the various traits. Most users have learnt to live with some degree of imperfection in software. Over time, software engineering methodologies have been developed to ease the task of creating quality software. Unfortunately, project sizes grow ever larger, making the task more complicated again. In this dissertation we will look at language features to support a development method aimed at creating quality software, with particular focus on flexibility, reliability, robustness and maintainability.

1.1 The Software Quality Toolbox

Let's start by looking at a few techniques commonly used in developing quality software. Then, in the following sections, we'll see how we shall expand on these techniques in the rest of this dissertation.

1.1.1 ABSTRACTION

Abstraction is a crucial mechanism in the development of software. By breaking programs down into small, independent pieces, we make the programming task more manageable, and gain the ability to independently test and verify individual components as well as the possibility of replacing or updating parts without disturbing the whole program. This insight is fundamental to our approach.

HIDING DETAIL

Abstraction hides or generalises from unnecessary detail, giving a higher-level overview that is easier to understand and reason about. Throughout this dissertation we will exploit this to the advantage of the computer as well as the programmer, as the compiler and development tools can take advantage of the increased ease of reasoning of abstract code.

We may abstract through *parametrisation* – abstracting away from single concrete cases, making code work on a wider range of cases – and by *specification* – defining the *what* rather than the *how*. There are several kinds of abstraction that may be useful:

FUNCTIONS &
PROCEDURES

Procedural or control abstraction is present in basically all languages, in the form of subroutines, functions, procedures, methods or similar concepts. Procedural abstraction has two purposes: to abstract away the details of *how* something is done, so we may focus on *what* is done (or *why*), and to *parametrise* operations, so we can easily reuse the same operation on different values. For example, we can use a

gcd function without caring how it computes the greatest common divisor, and we would definitely like to be able to compute gcd of arbitrary numbers and not just, say, 4.

In Chapter 2 we shall look at a way to make definition and use of procedural abstractions more flexible.

Data abstraction separates the abstract properties of a data type from its concrete representation and implementation. An *abstract data type* (ADT) provides an interface of operations for manipulating objects of that type, and all information about data representation is hidden behind the interface.

ABSTRACT DATA
TYPES

In object orientation, data abstraction is achieved through *classes*, which bundle a single type together with operations (methods) for manipulating objects. Abstract classes or interfaces achieve another layer of abstraction, where multiple classes that implement the same interface can be used interchangeably.

Error abstraction abstracts over the way errors and exceptional situations are handled in a program. *Exceptions* are the most common form of error abstraction – an exception is thrown when an error occurs, propagates up the call stack until it is caught by some part of the program that is capable of handling the exception. Further error abstraction may be provided using *aspect orientation*, for example by declaring common exception behaviour for particular classes or operations in an *aspect*, separated from the normal control flow.

EXCEPTIONS

In Chapter 5, we introduce *alerts*, which provide a new form of error abstraction, where exceptional behaviour is declared as part of the interface of an operation.

1.1.2 SEPARATION OF CONCERNS

The term *separation of concerns* was coined by Dijkstra [1982], and refers to focusing one's attention on one aspect of a problem – without ignoring the existence of other aspects, though. It is a scientific thought process, and it also becomes a programming practise when the time comes to formulate one's thoughts into software.

Though separation of concerns is related to abstraction, it is perhaps more fundamental. The abstraction mechanisms mentioned above offer some degree of separation of concerns – by separating the *what* from the *how*, but we can go further. A *cross-cutting concern* [Douence et al., 2001] touches many parts of a program, irrespective of module and abstraction boundaries. Error handling, logging and resource access / locking are typical cross-cutting concerns; they must be dealt with everywhere and cannot easily be modularised and hidden away in a separate part of the program.

CROSS-CUTTING
CONCERNS

ASPECT
ORIENTATION Aspect-oriented programming [Kiczales et al., 2001; Steimann, 2006] is a programming model for dealing with cross-cutting concerns, where the concerns are separated into *advice*, which describes how the concern is to implemented; and *pointcuts*, which describe the *join points* – where the advice should be applied. The join point model of systems like AspectJ [Kiczales et al., 2001] allow selection of join points based on both static and dynamic criteria.

TANGLING Lack of separation of concerns leads to *tangled code*; code which deals with many concerns at the same time. Tangled code may be cluttered and difficult to read (though, some degree of tangling may make it easier to see everything that's going on), but is first and foremost a flexibility and maintainability problem – making it difficult to adapt the code to changing circumstances. For example, if error handling is tangled with normal code, switching error handling policies will require touching large parts of a program.

We will deal with separation of concerns in a wider sense than aspect orientation. In the following chapters, we'll look at separating concerns along the following axes:

Specification, Implementation and Use: For example, implementing an in-place sorting operation, but using or specifying it like a function returning a fresh value. This is of particular interest in numerical applications, where the most efficient implementation style doesn't necessarily match the most natural way of using mathematical abstractions. This is the subject of Chapter 2. Another example is separation of an implementation deals with errors from how code that uses the implementation deals with errors (Chapter 5).

Normality and Exceptionality: Separating code dealing with the normal computation (e.g., solving the actual problem) from code dealing with errors and other unexpected occurrences.

Implementation and Optimisation: Separating code optimisations from the basic implementation code, so that we may quickly adapt the code to different architectures, and also avoid code clutter and subsequent maintainability issues. According to Knuth [1974], '*premature optimization is the root of all evil*'; with various hardware architectures placing different demands on software, and software systems growing ever more complex, this is even more true today. The desire for separating out optimisation lies behind axiom-based transformation, discussed in Chapter 3 and Section 7.3.2.

1.1.3 SPECIFICATION

A *specification* states the requirements, behaviour and effects of an abstraction. Specification may be done informally, through documentation

written in a natural language, or formally, using a formal specification language such as CASL [Astesiano et al., 2002]. The specification may appear together with the program code, as comments or embedded specifications – or as a separate document, or a combination, as in Javadoc comments which are present both in the code and extracted for use in a separate document.

FORMAL &
INFORMAL

Specification has multiple purposes:

- It defines the behaviour of an abstraction, and through that, guides the process of forming the abstraction. For example, if a procedure abstraction is difficult to specify, that may indicate that the abstraction boundary has been poorly chosen.
- It may be used for formal verification of the correctness of an implementation of an abstraction, and in the verification of code that uses the abstraction.
- It can be used as a basis for testing that code behaves according to the specification.
- It serves as documentation of how the code is expected to behave.

The ability to verify or test code is important not just while the code is being developed, but perhaps even more so to determine that later modifications don't break the expected behaviour.

The Eiffel programming language has pioneered *design by contract* [Meyer, 1992], where requirements, effects and invariants are stated directly in the code. This gives a specification of code interfaces, where the implementation and clients of an abstraction enter into a sort of *contract* – provided that a client fulfils the requirements of the contract, we can be assured that the implementation will uphold the postconditions stated in the contract. The compiler can insert automatic checks so that pre- and postconditions are checked at run-time.

DESIGN BY
CONTRACT

Systems like Larch [Guttag et al., 1985] allow for complete specification of programs. Specifications are developed separately, and linked to the program code through interface languages tailored to particular languages. With the aid of a theorem prover, an experienced programmer can make a formal proof of an implementation. This is usually considered too complicated and time-consuming for mainstream software however, and is more useful for critical applications or in limited domains, such as hardware circuits, OS kernels or secure systems [Wing and Gong, 1990].

SPECIFICATION &
VERIFICATION

Concepts provide a way to specify component interfaces together with axioms describing the semantics. Unlike Larch, the specification is tied directly to the abstractions in the program code. We shall discuss the use of concepts and axioms in more detail below, and in Chapters 2–4. Similar ideas are realised as *type classes* in Haskell [Bernardy et al., 2008; Hall et al., 1996], where types that support a given set of operations belong to a type class. Function parameters may then be typed by classes and be generic in all the types belonging to a type.

CONCEPTS

1.1.4 TESTING

Software testing is a way to check reliability and robustness empirically. *Unit testing* is the testing of individual components independently of the rest of the program and is currently standard practise in software development. Other types of testing include functionality testing and integration testing. Unit testing checks that a component performs correctly according to its specification, integration testing checks that components work together (e.g., even if all components work correctly according to their specification, they may not work correctly together, because the specifications may be faulty), and functionality testing checks that a program actually performs the tasks it should perform. We will limit our discussion to unit testing.

In test-driven development [Beck, 2002] and agile methods like extreme programming (XP) [Beck, 1998], program development is centred around tests, particularly unit tests. For each new feature to be implemented, tests are written *before* the implementation code – reducing the temptation to skip testing after implementation, and providing an easy way to check whether the new implementation is acceptable. In this way, unit tests serve as a sort of program specification. Tests are written with the help of unit testing frameworks like JUnit [Beck and Gamma, 2009] and integrated into the build process and development tools.

Another way to approach testing is specification-based testing, where tests are derived either manually or automatically from a program specification. In the approach of Antoy and Hamlet [2000], an initial specification is used alongside the implementation. A *representation mapping* [Hoare, 1972] translates between the abstractions of the specification and the concrete data structures of the implementation. All objects in the system contain both a concrete value and an abstract value (in the form of a normalised term over constructors in the specification), and the equations from the specification can be evaluated by treating them as rewrite rules on the abstract value terms. Self-checking functions are made by doing an additional abstract evaluation according to the specification, and – using the representation mapping – comparing the result of normal execution and evaluating the specification. In this way, a whole program can be described and evaluated in two distinct ways – using program code and algebraic specification – providing good protection against programming errors. This is also the disadvantage of the approach – the implementation work must basically be done twice. The overhead of the abstract evaluation and comparison can probably be lowered by running the testing code in a separate thread on a multicore system.

The approach of DAISTS [Gannon et al., 1981] and similar, later systems like Daistish [Hughes and Stotts, 1996], QuickCheck [Claessen and Hughes, 2000] and JAX [Stotts et al., 2002] is to use program specification for unit testing. Axioms from the specification are used as test oracles, which are fed with test data. The oracles are evaluated using the test data and

KINDS OF TESTING

TEST-DRIVEN
DEVELOPMENTAn initial specification is
one in which objects
are equal iff they can
be proven equal using
the equations of the
specification. We will
deal mostly with loose
specifications, however.SELF-CHECKING
CODEAXIOM-BASED
TESTING

the implementation to be tested – e.g., executing the two sides of an equation and comparing the results. This is the approach we will explore in later chapters of this dissertation. Unlike the self-checking approach, this works with loose specifications which may not give a full description of program behaviour. Self-checking gives only as good test coverage as the program being tested, which means library testing has to be done with carefully written test programs. With the DAISTS approach, coverage depends on the test data and the completeness of the specification. Using random test data seems to be a reasonable approach [Hamlet, 1994] – perhaps combined with some carefully chosen boundary values. Test coverage – how many axioms are actually tested, and how much of the implementation code is exercised by the tests – can be measured by a profiling tool. Ideally, information from the coverage analysis can be fed back into the test data generation, so that coverage can be maximised.

Formal specification is also useful in other forms of testing not discussed here – for instance, in protocol testing [Lai, 2002].

1.1.5 TOOL SUPPORT

Today's programmers expect considerable tool support in their programming and maintenance work. Integrated development environments (IDEs), such as Eclipse (IBM) and NetBeans (Sun) typically provide language sensitive editors, compilers, unit testing frameworks, debuggers, code inspectors, refactoring tools and code generators for boilerplate code. Any new language focusing on maintenance and development convenience will have to provide some form of integrated environment with support tools. Stroustrup and Reis [2005] claims that development of new programming languages will be too costly, and that the way to do language development instead should be done through extensions and restrictions of existing, well-supported languages. Though, with modern language tools, such as GLR parsers [van den Brand et al., 2001; Visser, 1997], transformation tools [Bravenboer et al., 2006], and IDE building/extension tools [Kalleberg and Visser, 2007; Klint et al., 2008] this may not be so much of an issue.

IDEs

Integrated editors typically support – as a minimum – syntax highlighting and automatic indentation. Other desirable features are semantics-aware search and replace / rename, tool-tips with declaration information, help system integrated with documentation comments, auto-completion of names, and auto-generation of boilerplate code such as `hashCode` and `equals` in Java. Refactoring support may include features such as moving code between classes and abstracting code into functions. The editor will also typically integrate with debuggers, testing frameworks, version control, modelling tools and so on, forming a fully integrated development environment.

INTEGRATED
EDITORS

GRADUAL
PROGRAMMING

We will not discuss IDEs much here, development of such tools to support our development methods is future work. An IDE idea which we may explore in the future is interactive refinement of code, where the programmer may give a fairly loose, prototype-like high-level implementation at first, and then gradually refine it into an efficient implementation by filling in details such as which data structures to use [Chang et al., 2009]. For example, a program may initially be developed using only library modules specified as concepts. One may even be able to derive missing implementation code from a specification. The IDE may then present various choices to the programmer that fit with the selected concepts, and the programmer may explore the possible configuration space in interaction with the development environment, integrating feedback from trial runs.

1.2 Domain, Background & Motivation

HIGH-
PERFORMANCE
COMPUTING

Although the techniques discussed here are applicable to software development in many domains, our choices have been influenced by the problems of developing high-performance, numerical software. The scientific programming and high-performance computing (HPC) communities are faced with particular problems that make the adoption of modern programming techniques especially difficult.

First and foremost, performance is *everything*. Scientific programmers will go to great lengths to squeeze every last bit of performance out of a supercomputer. The reasons for this are clear; while computers are now quite cheap, high-end supercomputers are still expensive and are typically shared by many people. And, when computations take many days, a few percent speedup can translate into many hours of saved time – reducing both waiting time and resource usage. Furthermore, lower resource use can be applied towards solving ever larger problems.

While today’s supercomputers often use mainstream processors, the hardware architecture is still quite different from a standard desktop computer, with sophisticated communication networks between processors, and multi-layered memory hierarchies. Some supercomputers employ special hardware in addition to or instead of regular processors. For example, the IBM RoadRunner is equipped with Cell processors (also found in the Sony PlayStation 3), in addition to normal AMD Opteron processors. Cray’s XT5 series of supercomputers can be equipped with vector processing units and reprogrammable chips – field-programmable grid arrays (FPGAs).

Each vendor will usually supply high-performance compilers specially adapted to their systems. Even so, different systems need careful programming in order to make full use of them, and porting numerical applications from platform to platform can be a tedious task.

SUPERCOMPUTERS

The work described here fits in to a larger, long-term project aimed at developing better software methodologies and tools for programming numerical software. In particular, a numerical library has been developed, Sophus, in which the underlying mathematical principles of seismic simulation and partial differential equations have been mapped to high-level abstractions [Haveraaen et al., 1999], specified using algebraic specification. The library has been implemented in C++, and has been relying on ad hoc tools to support its programming style.

SOPHUS

This is the reasoning behind our choice of C++ as a basis for much of this research, even though languages like Haskell, Scheme or OCaml might have been more convenient. C, C++ and Fortran are the main languages supported by supercomputer vendors and high-performance compilers, and of these, C++ has the best support for user-defined abstractions.

In some cases, such as for the Cell processor, support for other languages has been poor to non-existent.

Unfortunately, while C++ is great for developing abstractions within the language, it is a poor choice for experiments with language extension, as it has no extension facility itself, and building or extending a C++ frontend is in itself a huge project [David, 2009]. This led to the decision to design a new language, Magnolia, both as a basis for this research, and as a flexible basis for further research projects. By giving Magnolia semantics fairly close to C++, we can keep full control over resource usage and experiment with HPC-oriented optimisations (which would be difficult in a functional language), and by having the compiler emit C or C++ code, we can remain compatible with existing high-performance compilers.

Developing a new language does require some justification though, as it is a huge project, not to be undertaken lightly. For us, designing a new language is not our main purpose, but rather a means to the end of exploring novel language features and development methods. To some degree, the language you're working with will always influence your development method – if we continued with C++, we would likely end up focusing on generative template programming and object orientation; if we worked with Haskell, we would seek functional solutions. With research on programming methodology, language design follows hand in hand, and fitting the ideas to an existing language may hamper the research. The same observation was made by Liskov [1993] when the language CLU sprung out of her research on data abstraction in the early seventies. As the research matures, though, we may be able to apply the principles in other languages.

WHEN TO DEVELOP
A NEW LANGUAGE

1.3 *Magnolia*

In this dissertation we shall start developing the Magnolia programming language, to support a development method based on abstraction, specification, testing and tools. As discussed in the last section, the need for a

new language – rather than building upon an existing one – grows out of a desire to have a language that is easily processed by tools, and is extensible so that we can experiment with new language features. Some of the features we’re interested in are so fundamental to the language design that adapting an existing language would be hard without rebuilding much of the infrastructure – and we would then have to struggle with language features and design decisions which may not be appropriate for us anyway. Some of our experiments have been done in C++ or C, but processing these languages (particularly C++) is hard, and integrating new language constructs with the old sometimes gives less than elegant results.

Magnolia programs are organised around the following fundamentals:

- A *signature* is a set of declarations of operations (procedural abstractions) and types (data abstractions).
- An *implementation* is a signature together with definitions for its declarations (operation bodies, data structures).
- A *concept* is a signature together with *axioms* specifying the behaviour of the types and operations. A concept can also provide default definitions for some or all of its operations.

An implementation *models* a concept if it shares the same signature (or a superset of it) and contains definitions consistent with the axioms of the concept. A single concept may have multiple independent and interchangeable implementations. A *signature morphism* maps between signature differences in concepts and implementations.

Operations can be either functions, which are pure and may not perform parameter updates – or procedures, which may update parameters. Procedures may be either pure or impure – an impure procedure is allowed to access state outside of its parameters – such as doing I/O or user interaction. Through *functionalisation* and *mutification* we can treat procedures as functions and vice versa – this is the subject of Chapter 2.

Data abstraction comes in the form of abstract types (in signatures and concepts) and data structures (in implementations). Types and data structures may be parametrised, and parameters may be constrained according to concepts. An implemented type has a data structure, a data invariant and an equivalence relation. Access to data structures is only available to implementation code upholding the data invariant and equivalence relation – normally, this is code defined in the same module. This corresponds to the data hiding / encapsulation idea in object orientation – though we make data invariants and equivalence tangible entities that should be defined for all types.

1.3.1 THE MAGNOLIA DEVELOPMENT METHOD

Program development with Magnolia should start with domain engineering – exploring which concepts are fundamental to the problem

being solved. We may then select appropriate abstractions for dealing with those concepts, giving us a specification, consisting of concept declarations with associated abstract data types, operation signatures and axioms.

DOMAIN
ENGINEERING &
SPECIFICATION

The specification tells us which abstract data types are needed – we then need to provide data representations and operations for them; i.e., provide implementation code that models the concepts. Code should be written using concepts rather than concrete implementations in dependencies – the idea being that different modules implementing the same concept should be interchangeable. For each implementation we must choose the most appropriate way of modelling the concepts – this may be different for different implementation of the same concept, and the programmer has a fair degree of freedom in this choice. Axiom-based testing is used to check the implementation, and program development proceeds iteratively.

IMPLEMENTATION

TESTING

Concerns such as error handling and optimisation should be kept separate in both design and implementation, and dealt with by appropriate abstractions and language features. In particular, premature optimisation, and optimisation by breaking abstraction barriers should be avoided.

SEPARATION OF
CONCERNS

Program configuration consists of picking the right set of implementation modules among those that model a program's concepts – for example, choosing between sequential and parallel implementations depending on available hardware. Since the implementations are known to satisfy certain axioms, the compiler is free to apply high-level optimisations consistent with the axioms. Such optimisation opportunities are typically missed in traditional compilers, which rely on program analysis to uncover program properties.

CONFIGURATION

The Magnolia language itself is discussed in more detail in Chapter 6. Here we will just give quick overview of the most important features, to provide some background for the following chapters.

1.3.2 CONCEPTS

Concepts and *axioms* make the integration of specification and implementation in program development easier. A concept consists of types, operations and axioms. For example, the following concept specifies a *monoid*:

```
concept Monoid(type T) extends Semigroup(T) {
  // function T binop(T, T); -- from Semigroup
  function T neutral();
  axiom Identity(T x) {
    assert binop(x, neutral()) == x;
    assert binop(neutral(), x) == x;
  } }
```

It inherits all the properties of the *semigroup* concept, – including the binary operation *binop* – and specifies that the constant operation *neutral* should be available on the type *T*. It also gives a single axiom *Identity*, stating that $\text{binop}(x, \text{neutral}) = x$ and $\text{binop}(\text{neutral}, x) = x$ for all $x \in T$. The full expressive power of the language is available to specify axioms and axiom conditions – but some use cases require simpler axioms, like conditional equations used in generating rewrite rules.

MODELS

We may provide an implementation of rationals:

```
type rational = struct { int num; int denom; };  
define rational _+(rational a, rational b) = ...;  
define rational zero = ...;  
...
```

and state that our implementation models the *Monoid* concept, with $+$ as the binary operation and *zero* as the neutral element:

```
model Monoid(rational) {  
  define rational binop(rational a, rational b) = a + b;  
  define rational neutral = zero;  
}
```

The body of the model declaration gives renamings – a *signature morphism* mapping between names used in the implementation and those used in the concept.

Stating that the implementation *Rational* models *Monoid* implies that it satisfies the *Identity* axiom, and any axioms from the *Semigroup* concept. Once we extend our selection of concepts with other algebraic classes, we may instead state that *Rational* models *Ring*, which has more operations, and also gives us laws like distributivity and commutativity, etc. We can also create more implementations of *Ring*, and whenever we need a data type with ring operations, we can use *Ring.T* and obtain generic code that will work with any ring implementation.

1.3.3 AXIOM-BASED TESTING

By having a full-featured implementation language and allowing unrestricted expressions in axioms, full formal verification of program correctness may not be possible. Instead, we fall back to using the program specification as a basis for testing.

Testing based on algebraic specification has been known since the DAISTS system [Gannon et al., 1981] in the early eighties, in which conditional equations over the functions defined in a program were used as test oracles. Combined with a simple test data description language, DAISTS provided a tool for specification based unit testing. Experiments with DAISTS were promising, and in recent years similar ideas have been explored; for instance, Daistish for C++ [Hughes and Stotts, 1996],

QuickCheck [Claessen and Hughes, 2000] for Haskell, JAX [Stotts et al., 2002] and JAXT [Haveraaen and Kalleberg, 2008] for Java.

Axiom-based testing is based on the idea that axioms are Boolean expressions that will evaluate to true for all possible values of the axiom variables, if the axiom holds for the current implementation. Generating a test oracle from an axiom is a straight-forward conversion of the axiom into a Boolean function. For example, for the *Identity* axiom:

TEST ORACLES

```
forall type T where Monoid(T)
function bool Identity_oracle(T x) =
  binop(x, neutral()) == x && binop(neutral(), x) == x;
```

The above oracle function is a template that will work for any type *T* in an implementation that models *Monoid*. The names from the concept (*binop*, *neutral*) are replaced with names from the model declaration (e.g., *+* and *zero* for *Rational*) when the code is processed. This oracle returns a *bool* – a more sensible version would return one of three possible values: *true/success*, *false/failure* and *unknown* (in case of conditional axioms when the condition fails and the axiom is never tested).

1.3.4 PARTIALITY, FAILURE AND GUARDING

A *partial operation* is one that is undefined for some input values. For example, division by zero is undefined, making division a partial operation. Similarly, some operations can fail unexpectedly, for example, due to disk or network error. How we wish to deal with partiality can differ from case to case, even with the same operation. With division by zero, we may sometimes be able to sensibly approximate the result by dividing the derivatives instead. In other cases we may want to abort the program, or ask the user to supply a corrected value.

PARTIAL
OPERATIONS

In program code, partiality and failure conditions are typically dealt with either by specifying that an operation should never be called with inappropriate arguments, or by signalling an error using some mechanism like exceptions or an error return code. In robust code, improper arguments should be reported as an error, or undefined behaviour may result. However, error checking and reporting can lead to an excess of code for dealing with errors, obscuring the code doing actual work, and leading to maintainability issues.

On the specification side, partiality can be similarly dealt with by making operations total, and having them yield a special error result on failure. This also leads to clutter and makes specifications more complicated. Another approach is *guarded algebras* [Haveraaen and Wagner, 2000], where partial operations can be *guarded* by some condition that ensures that the operation is total as long as the condition holds. Axioms are then implicitly guarded by the assumption that the guard conditions hold.

GUARDS &
PRECONDITIONS

We can model this approach in implementations by using *preconditions*. Precondition checks can be automatically inserted in the code, to ensure that all arguments are properly checked. We are still left with the problem of what to do if a precondition check fails – in the specification world, we may be able to pretend that the guards always hold, but in the real world, things go wrong from time to time. The most common approaches are exceptions [Goodenough, 1975] and error return values. Exceptions have the advantage that the default behaviour is to propagate errors, making it unlikely that an error goes undetected. Error handling code can still be quite intrusive, though. With return values, the default is to ignore the error, which is in most cases not a very robust choice.

ALERTS

In Chapter 5, we will introduce *alerts*, which provides a way to specify how errors should be handled regardless of how they are reported. Preconditions can trigger alerts directly, and handlers can deal with the error in a sensible way, for example by substituting a default value.

1.4 Method and Outline

The results in this dissertation have been arrived at through the following research method:

1. Identify and explore a problem or problem area
2. Propose a solution
3. Implement the solution to verify its feasibility
4. Evaluate the solution to determine its effectiveness. With a bit of luck we may also find that the solution solves other problems.
5. Iterate, with feedback from earlier stages.

The first and second steps may be switched, particularly in the case where have a solution to one problem, and would like to explore its usefulness. The work on axiom-based testing and optimisation is (at least partly) based on this inverted method.

Note, though, that as this work forms part of a larger, long term research project, providing a proper evaluation at this point is premature. In particular, any complete evaluation of a development method or a programming language should include experience gained through real-world development and user experiences, both of which are beyond the scope of this work.

OUTLINE

We will finish this introduction with an overview of the remaining chapters. As the papers in this dissertation are a result of cooperation, I will point out what I am responsible for.

Chapter 2 is about how procedures and functions can be related through the use of functionalisation and mutification. By mapping between

algebraic and imperative signatures, we can easily link implementation code with algebraic specifications in the form of concepts. This work stems from the problem of how to combine an algebraic programming style with the efficiency demands of high-performance computing; though it also neatly provides increased separation of concerns and is useful in generic programming.

The chapter is a reprint of *Interfacing Concepts: Why Declaration Style Shouldn't Matter* (LDTA'09), co-authored with Magne Haveraaen. I am the main author with the work being supervised by Magne Haveraaen, who also came up with the original idea. The work builds on earlier work on mutification, by Dinesh et al. [2000].

Chapter 3 is a reprint of *Axiom-Based Transformations: Optimisation and Testing* (LDTA'08), also co-authored / supervised by Magne Haveraaen. The chapter explains how axioms, part of the concept proposal for C++0x, can be used as a basis for code transformation, optimisation, and testing.

To some degree, this paper is an answer to the problem of 'we have a nice formal specification – now, what can we do with it?'. The work on optimisation deals with the problem of achieving high performance from code that makes extensive use of abstraction. The work on testing is a continuation of initial work by Haveraaen and Brkic [2005], and is further expanded in Chapter 4 (testing) and Chapter 7 (transformation).

Chapter 4 takes a closer look at axiom-based testing for C++. The chapter is a reprint of *The Axioms Strike Back: Testing with Concepts and Axioms in C++* (GPCE'09), written together with Valentin David and Magne Haveraaen. The paper is a team effort, with me being responsible for design work and writing, Valentin doing implementation and C++ details, and Magne supervising and working on data generation.

Chapter 5 introduces the *alert* and *alert handler* constructs for dealing with failure and partiality. The paper was presented at GPCE'06 as *Stayin' Alert: Moulding Failure and Exceptions to Your Needs*, and was largely a team effort with design and planning done in together in meetings. I focused on design, Karl Trygve Kalleberg on related work, Valentin David did the prototype implementation and Magne Haveraaen supervising and supplying ideas.

The remaining chapters (and this introduction) are solely authored by me:

Chapter 6 describes Magnolia and the Magnolia implementation as it was when this dissertation was finished. The implementation is mainly my work, and I am also responsible for much of the design, with Magne Haveraaen and Valentin David as the main

other contributors (especially when it comes to concepts). Further implementation and design is a team effort.

Magnolia is a spin-off from the other research in this dissertation, motivated by the need to implement and experiment with new language constructs.

Chapter 7 describes an experimental extension to Magnolia for creating language extensions and controlling the compilation process. The chapter is a preprint of *Yet Another Language Extension Scheme* (SLE '09), and may differ from the final published version.

Chapter 8 puts the dissertation into context by reviewing selected related work in the areas of language development, abstraction and specification; as well as discussing issues of language design, and outlining directions for future work. More detailed related work discussions are spread out in the various chapters.

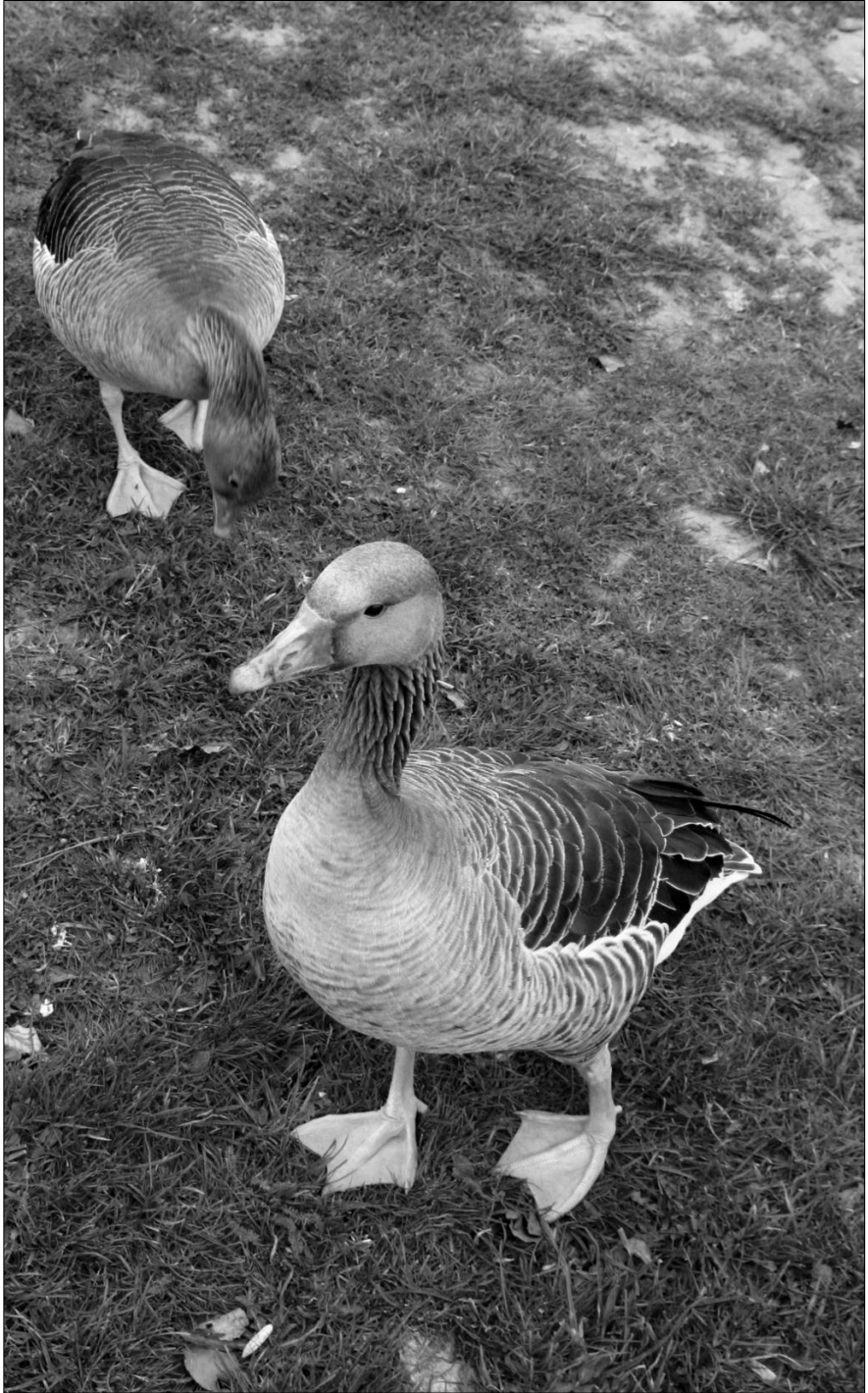
Chapter 9 is the conclusion.

Playing with Signatures

The same algorithm may often be realised in several different ways. For example, sorting can be seen as reordering a sequence, or as producing a new sorted sequence from an input sequence. The former maps naturally to an imperative procedure, while the latter maps naturally to how we would specify sorting in an algebraic specification. In this chapter, we introduce *mutification* and *functionalisation*, which allow us to map back and forth between imperative and algebraic code.

The imperative style is often more efficient, especially with numerical software that deals with large amounts of data. Our technique allows algorithms to be implemented imperatively, but used algebraically in both program specification and code. The algebraic style code can then be translated – or *mutified* – automatically to imperative style code. The ability to link imperative-style code to an algebraic specification will prove quite important in the coming chapters.

This paper was presented at LDTA'09, although the basic idea dates back more than ten years, originally envisaged as a way to write math-intensive code in familiar notation while still achieving good performance [40].



Greylag geese at the University of York. The campus is rumoured to have the largest
goose-to-student ratio of any UK university.
York, UK (ETAPS / LDIA '09)

Interfacing Concepts

Why Declaration Style Shouldn't Matter

ANYA HELENE BAGGE

MAGNE HAVERAAEN

Department of Informatics, University of Bergen, Norway

ABSTRACT

A *concept* (or *signature*) describes the interface of a set of abstract types by listing the operations that should be supported for those types. When implementing a generic operation, such as sorting, we may then specify requirements such as 'elements must be comparable' by requiring that the element type models the Comparable concept. We may also use axioms to describe behaviour that should be common to all models of a concept.

However, the operations specified by the concept are not always the ones that are best suited for the implementation. For example, numbers and matrices may both be addable, but adding two numbers is conveniently done by using a return value, whereas adding a sparse and a dense matrix is probably best achieved by modifying the dense matrix. In both cases, though, we may want to pretend we're using a simple function with a return value, as this most closely matches the notation we know from mathematics.

This paper presents two simple concepts to break the notational tie between implementation and use of an operation: *functionalisation*, which derives a set of canonical pure functions from a procedure; and *mutification*, which translates calls using the functionalised declarations into calls to the implemented procedure.

2.1 Introduction

CONCEPTS

Concepts is a useful feature for generic programming, allowing programmers to specify the interface and behaviour of abstract data types. The concept feature was introduced to C++0x (the upcoming C++ standard revision) in order to give clearer error messages in templated code, and to provide a way to more easily glue together generic code. Similar features are available in other programming languages.

In this paper we will look at concepts in the context of the Magnolia programming language, an experimental language loosely based on C++. Concepts form an integral part of Magnolia, and the programmer is encouraged to specify code dependencies in terms of concepts, and use concepts to specify the interface of implementation modules. Concepts are used to hide implementation details, so that one module may be replaced by another modelling the same concept. Even 'standard' data types like numbers and strings are handled through concepts, allowing for the possibility of replacing or using different implementations of basic data types.

PROGRAMMING STYLES

If we are to hide away different implementations behind a common interface, we must consider that different situations call for different programming styles. Operations on primitive data types, for example, are conveniently handled using return values, whereas the corresponding operation on a big data structure may be better handled by updating it.

Also, the needs of a library implementation may differ from the needs of the code that uses it. For example, a mathematical problem may be easily expressed with function and operator calls, whereas a communication protocol could be better expressed as a sequence of procedure calls. As a library user, however, you are locked to the style the library writer chooses to support – and your preferred style may be less convenient to implement, or less efficient, and thus not supported.

C++ is an example of a language where the proliferation of notational variants is especially bad (or good, depending on your point of view) – with both member and non-member functions, operator overloading, and a multitude of parameter passing modes (value, reference, `const` reference, pointers). Functional languages like ML and Haskell are less problematic since the languages are already restricted to functional-style declaration forms.

DECLARATION FORMS

Generic and generative techniques are hampered by a proliferation of declaration forms (prototypes, in C++ terminology) for implementations of the same abstract idea. For instance, implementing generic algorithms, such as a generic `map` operation for arrays, is made more difficult when the declaration forms of the element operations vary – so we end up with a multitude of different `map` implementations, e.g., one for arrays of numbers and one arrays of matrices.

User refers to the programmer who is using a library, and the word *implementer* to the programmer who has implemented it. The *end user* who runs the finished software product is outside the scope of our discussion.

For C++0x, the problem of having many different declaration forms for what is essentially the same operation is solved by allowing the implementer to add explicit glue code in the *concept map* (a declaration that some given types model a given concept). In constrained template code, the user will use the declarations from the concept and need not know anything about how the implementation is declared. Note, however that this convenience is available in constrained template code only.

In this paper, we show how this is solved in Magnolia by separating the *implementation signature* from the *use signature*, allowing different call styles to be used independently of the implementation style. By letting the compiler translate between different call styles, we gain increased flexibility for program processors – of both the human and software variety. We call these translations *functionalisation* and *mutification*.

We will start out by giving an intuition about our method, before we define it more formally later in the paper. We illustrate our points with examples in Magnolia, which has functionalisation and mutification as basic, built-in features.

Magnolia supports imperative *procedures*, which are allowed to update their arguments, and pure *functions*, which are not. A sample Magnolia procedure declaration looks like this:

```
procedure fib(upd int n);
```

The procedure `fib` takes a single parameter, an *updatable integer*, which is used both for input and for returning a result.

Functionalisation takes a procedure declaration, and turns it into a function declaration. The functionalisation of `fib` is:

```
function int fib(int n);
```

– a function with an integer parameter, returning an integer. This is the declaration that would typically be used in a concept. A different declaration of the `fib` procedure – like this, for example,

```
procedure fib(obs int n, out int r);
```

– which *observes* the parameter `n` and *outputs* the parameter `r`, yields the same functionalisation.

Functionalising the procedure declaration gives us a function we can call from Magnolia expressions, but no implementation of that function. This is where *mutification* comes into the picture. Mutification takes an expression using a functionalised procedure, and turns it into a statement that calls the procedure directly. Here we show a call for each of the declarations above, the functional form (in the middle) can be transformed to either of the procedural calls shown.

IMPLEMENTATION
& USE
SIGNATURES

MAGNOLIA

FUNCTIONALISATION

MUTIFICATION

```

y = x;
call fib(y);      ←  y = fib(x);      →  call fib(x,y);

```

THE ALGEBRAIC STYLE

Why would we want to implement an algorithm with one declaration form and use it with another? As explained above, the implementation side is often dictated by what fits with the problem or algorithm – in-place sorting and updating matrix operations, for instance, will require less memory and may run faster than versions that construct new objects. There are several reasons why an algebraic style is useful on the user side:

Flexibility: Multiple implementations with different characteristics can be hidden behind the same interface. This is particularly useful in generic code and when building a basis of interchangeable components.

Ease of reasoning: Not just for humans, but also for compilers and tools. For example, axiom-based rewrite rules [7; 90; 147] allow programmers to aid the compiler’s optimisation by embedding simplification rules within a program. Such rules are much harder to express with statements and updating procedures.

Notational clarity: certain problems are most clearly expressed in an algebraic or functional style, which is close to the notational style of mathematics, a notation developed over the centuries for clarity of expressions. Many formal specification languages, e.g., the Larch Shared Language [68] and CASL [115] use the functional style due to its clarity. A program written in an algebraic style is easy to relate to a formal specification.

Using a single declaration form makes it possible to state rules such as “+ and × are commutative for algebraic commutative rings”, and have it apply to all types satisfying the properties of commutative rings (i.e., all types modelling the concept *CommutativeRing*) – independently of whether the implementation uses return values or argument updates.

We may in principle choose any declaration form as the common, canonical style – we have chosen the algebraic style for the reasons above, and because it is less influenced by implementation considerations (such as the choice of which argument(s) should output the result). As we shall see, this choice does not mean that we must enforce an algebraic programming style.

We will now proceed with a deeper discussion of the matter. The rest of this paper is organised as follows: First, we introduce the necessary features of the Magnolia language (Section 2.2). Then we define mutification and functionalisation, and explain how they are applied to Magnolia programs (Section 2.3). We continue by discussing some limitations and pragmatic considerations, and the benefits and possibilities of the approach (Section 2.4).

2.2 The Magnolia Language

This section gives a brief overview of the Magnolia Programming Language. Magnolia is based on C++, with some features removed, some features added, and some changes to the syntax. We have designed it to be easier to process and write tools for than C++ (this is actually our main reason for using a new language, rather than working with C++), while being similar enough that we can easily compile to C++ code and use high-performance C++ compilers.

The following features are the ones that are relevant for this paper:

FEATURES

Procedures have explicit control over their inputs and outputs: they are allowed to modify their parameters, according to the *parameter modes* given in the procedure declaration. The available modes are *observe* for input-only arguments, *update* for arguments that can be both read and written to, and *output* for output-only arguments. These modes describe the data-flow characteristics of the procedure – which values the result may depend on, and which variables may be changed by the procedure. The parameter passing mode (e.g., by value, by reference, by copying in/out) is left undefined, so the compiler is free to use the most efficient passing mode for a given data type. Procedures have no return values, the result is given by writing to one or more arguments. Procedure calls use the `call` keyword, so they are easy to distinguish from function calls.

Functions have a single return value, which depends solely on the arguments. They are not allowed to modify arguments. Operators can be overloaded, and are just fancy syntax for function calls.

Functions and procedures are known collectively as *operations*.

Data types are similar to C++ structs and classes, though there are no member operations – everything is treated as non-members. There is no dynamic dispatch or inheritance yet, and we won't consider that in this paper.

Concepts describe the interfaces of types by listing some required operations, axioms on the operations and possibly other requirements. A set of types is said to model a concept if the operations are defined for those types and the requirements are satisfied. For example, the following defines a simple 'Indexable' concept:

```
concept Indexable(A,I,E) {
  E getElt(A, I);
  A setElt(A, I, E);

  axiom getset(A a, I i, E e) {
    assert getElt(setElt(a, i, e), i) == e;
  }
}
```

We have avoided using operators in the example to keep the syntax simple

Indexable has three types, and array-like type **A**, an index type **I** and an element type **E**. The concept defines two functions, `getElt` and `setElt`, and a simple axiom relating the functions.

Generics: Generic programming is done through a template facility similar to C++'s. Type parameters may be constrained using concepts, so that only types modelling a given concept are acceptable as arguments.

No Aliasing: Aliasing makes it difficult to reason about code, because it destroys the basic assumption that assigning to or updating one variable will not change the value of another. Functional languages avoid this problem by simply banning the whole idea of modifying variables. We feel disallowing modification is too high a price to pay, particularly when working with numerical software and large data structures, so Magnolia has a set of rules designed to prevent aliasing while still having most of the freedom of imperative-style code:

- No pointers or references. Any data structures that need such features must be hidden behind a 'clean' interface. The programmer must take responsibility for ensuring that the implementation is safe.
- There is no way to refer to a part of a data structure. For example, you can't pass an element of an array as a procedure parameter – you must either pass the entire array, or the *value* of the element. Thus, changing an object field or an array element is an operation on the object or array itself (unlike in C++, where fields and elements are typically l-values and can be assigned to directly). This is why the `setElt` operation in the *Indexable* concept above is declared as a function returning an array, and not returning a reference to an element as is typical in C++.
- If a variable is passed as an `upd` or `out` argument to a procedure, that variable cannot be used in any other argument position in the same call.

2.3 *Relating Functions and Procedures*

We will now establish a relationship between Magnolia functions and procedures, so that each procedure declaration has a set of corresponding function declarations given by functionalisation (Definition 1), and every expression has a corresponding sequence of procedure calls given by mutification (Definition 2).

2.3.1 FUNCTIONALISATION OF DECLARATIONS

Definition 1 Functionalisation, \mathcal{F} , maps a procedure declaration to one or more function declarations. This makes procedures accessible from expressions, at the signature level. Since a procedure can have multiple output parameters, and a function can only have one return value, we get one function for each output parameter of the procedure (numbered 1 to i):

$$\mathcal{F}_i(\text{PROC}(n, \mathbf{q})) = \text{FUN}(n_i, \text{Out}(\mathbf{q})_i, \text{In}(\mathbf{q})) \quad (2.1)$$

For clarity, we use abstract syntax in the definitions, with **PROC** (name, proc-parameter-list) being a procedure declaration, and **FUN** (name, return-type, fun-parameter-list) being a function declaration. **In** and **Out** gives the input and output parameters of a procedure, respectively:

$$\begin{aligned} \text{In}(\mathbf{q}) &= [t \mid \langle m, t \rangle \leftarrow \mathbf{q}, m \in \{\text{obs}, \text{upd}\}] \\ \text{Out}(\mathbf{q}) &= [t \mid \langle m, t \rangle \leftarrow \mathbf{q}, m \in \{\text{out}, \text{upd}\}] \end{aligned}$$

We're using list comprehension notation (similar to set notation), as in Haskell or Python.

where m is the parameter mode and t is the parameter type.

We can then obtain the list of functions corresponding to a procedure:

$$\mathcal{F}(\text{PROC}(n, \mathbf{q})) = [\mathcal{F}_i(\text{PROC}(n, \mathbf{q})) \mid i = 1 \dots \text{len}(\text{Out}(\mathbf{q}))] \quad (2.2)$$

Note the similarity between functionalisation and standard techniques for describing the semantics of a procedure with multiple return values. This is the link between the semantics of the procedure and the functionalised version, and the key in maintaining semantic correctness between the two programming notations.

For example, the following procedures:

```
procedure plus(upd dense x, obs sparse y);
procedure plus(obs int x, obs int y, out int z);
procedure copy(obs T x, out T y);
```

functionalise to the following functions:

```
function dense plus(dense x, sparse y);
function int plus(int x, int y);
function T copy(T x);
```

which is what would be used in a concept declaration. We keep the namespaces of functions and procedures, as well as their usage notations (expressions versus calls), distinct, thus avoiding overloading conflicts. For multi-output procedures, the functions get numbered names – Magnolia also allows the programmer to choose the function names, if desired.

Note that the inverse operation – obtaining a procedure from a function – is not straight-forward, since there are many different mode combinations for the same function declaration. However, we could

define a *canonical proceduralisation*, \mathcal{P} , in which every function is mapped to a procedure with one **out** parameter for the return value and one **obs** parameter for each parameter of the function:

$$\mathcal{P}(\text{FUN}(n, t', [t_1, \dots, t_n])) = \text{PROC}(n, [\langle \text{obs}, t_1 \rangle, \dots, \langle \text{obs}, t_n \rangle, \langle \text{out}, t' \rangle]) \quad (2.3)$$

2.3.2 MUTIFICATION

MUTIFICATION

Definition 2 (Mutification of Assignment) *Mutification turns a sequence of function calls and assignments into a procedure call. Given a procedure $p = \text{PROC}(n, \mathbf{q})$:*

$$\begin{aligned} \mathcal{M}(\overline{\text{ASSIGN}(\mathbf{y}, \text{APPLY}(\mathbf{f}, \mathbf{x})))} &= \mathbf{i} ; \text{CALL}(\text{PROC}(n, \mathbf{q}), \mathbf{x}'), \text{ where } \mathcal{F}(\text{PROC}(n, \mathbf{q})) = \mathbf{f}, \\ \text{and } \langle \mathbf{x}', \mathbf{i} \rangle &= \text{unzip} \left(\mathbf{q} \begin{bmatrix} \langle \text{obs}, s \rangle \rightarrow \langle x, \text{NOP} \rangle \text{ if } x \notin \mathbf{y} \\ \text{where } x = \downarrow \mathbf{x} \\ \langle \text{obs}, s \rangle \rightarrow \langle t, \text{TMPVAR}(t, x) \rangle \text{ if } x \in \mathbf{y} \\ \text{where } x = \downarrow \mathbf{x} \\ \langle \text{out}, s \rangle \rightarrow \langle \downarrow \mathbf{y}, \text{NOP} \rangle \\ \langle \text{upd}, s \rangle \rightarrow \langle \gamma, \text{ASSIGN}(\gamma, \downarrow \mathbf{x}) \rangle, \\ \text{where } \gamma = \downarrow \mathbf{y} \end{bmatrix} \right) \\ &\text{and } \gamma_j \equiv \gamma_k \iff j = k \text{ for all } \gamma_j, \gamma_k \in \mathbf{y} \end{aligned} \quad (2.4)$$

The pattern $\overline{\text{ASSIGN}(\mathbf{y}, \text{APPLY}(\mathbf{f}, \mathbf{x}))}$ recognises a sequence of assignments, one for each **upd** or **out** argument of p . The list of functions called, \mathbf{f} , must match the functionalisation of p , in sequence. Dummy assignments can be inserted to accomplish this, if no suitable instructions can be moved from elsewhere. All variables (\mathbf{y}) assigned to must be distinct.

We then construct a new argument list \mathbf{x}' and a list of setup statements \mathbf{i} by examining the formal parameter list of p , picking (\downarrow) an argument from \mathbf{x} in case of **obs**, picking from \mathbf{y} in case of **out**, and picking from \mathbf{y} and generating an assignment $\text{ASSIGN}(\gamma, x)$ in case of **upd**. **NOP** denotes an empty instruction, **TMPVAR** creates a temporary variable, and **ASSIGN** denotes an assignment. To avoid aliasing problems, a temporary is needed to store the value of an **obs** argument which is also used for output.

Generating assignments for **upd** arguments is necessary, since the variables \mathbf{y} may have different values than the corresponding variables in the original argument list. This may generate redundant $\text{ASSIGN}(\gamma, \gamma)$

instructions when γ is already in an update position – these can trivially be eliminated at a later stage.

For example, if we have a procedure $p(\text{out } t_1, \text{upd } t_2, \text{obs } t_3)$ and $f_1, f_2 = \mathcal{F}(p)$:

$$\begin{array}{ll} a = f_1(3, 2); & \rightarrow \quad b = 3; \\ b = f_2(3, 2); & \text{call } p(a, b, 2); \end{array}$$

2.3.3 FUNCTIONALISATION OF PROCEDURE CALLS

Given a canonical mapping of function declarations to procedure declarations (i.e., *proceduralisation*, as sketched in Section 2.3.1), we can functionalise procedure calls – transform them to function calls and assignments.

This means we lose control over some aspects of the program, such as the creation and deletion of intermediate variables. However, since we argue that expression-based programs are much easier to analyse and process by tools, it would be useful to obtain something as close to a pure expression-based program as possible – even if we will eventually mutify it again to obtain imperative code. Thus, the choice of language form to work on becomes one of convenience. Our experience is mostly with high-level optimisations that take advantage of algebraic laws – which is a perfect fit for algebraic-style Magnolia. The same can be seen in modern optimising compilers, which will typically transform low-level code to and from Static Single Assignment (SSA) form depending on which optimisation is performed.

TRANSLATING TO
EXPRESSIONS

Definition 3 (Functionalisation of Procedure Calls) For a procedure $p = \text{PROC}(n, \mathbf{q})$:

$$\mathcal{F}(\text{CALL}(p, \mathbf{x})) = [\mathcal{F}_i(\text{CALL}(p, \mathbf{x})) | i = 1 \dots \text{len}(\text{Out}(\mathbf{q}))] \quad (2.5)$$

$$\mathcal{F}_i(\text{CALL}(p, \mathbf{x})) = \text{ASSIGN}(\mathbf{y}_i, \text{APPLY}(\mathcal{F}_i(p), \mathbf{x}')) \quad (2.6)$$

where $\mathbf{y} = \text{Out}_{\mathbf{q}}(\mathbf{x})$ and $\mathbf{x}' = \text{In}_{\mathbf{q}}(\mathbf{x})$. The ordering of assignments is immaterial, since the requirements on procedure call arguments ensures that the variables \mathbf{y} are distinct from \mathbf{x}' .

To functionalise a procedure call $\text{CALL}(n, \mathbf{q})$, we build a new argument list \mathbf{x}' of all the input arguments, then generate one assignment for each output argument. $\text{In}_{\mathbf{q}}$ and $\text{Out}_{\mathbf{q}}$ select the input and output arguments of an actual argument list with respect to a formal parameter list \mathbf{q} :

$$\begin{aligned} \text{In}_{\mathbf{q}}(\mathbf{x}) &= [x \mid \langle x, \langle m, t \rangle \rangle \leftarrow \langle \mathbf{x}, \mathbf{q} \rangle \text{ if } m \in \{\text{obs}, \text{upd}\}] \\ \text{Out}_{\mathbf{q}}(\mathbf{x}) &= [x \mid \langle x, \langle m, t \rangle \rangle \leftarrow \langle \mathbf{x}, \mathbf{q} \rangle \text{ if } m \in \{\text{out}, \text{upd}\}] \end{aligned}$$

For example, given a function `int f(int)`, we may use the canonical proceduralisation (2.3) to obtain and use a procedure `p(obs int,`

out int). Since (2.3) only gives us single-output procedures, we will only get a single assignment for each call to p . Using (2.5) above, we can functionalise the following statements:

$$\begin{array}{ll} \text{call } p(5, x); & \rightarrow x = f(5); \\ \text{call } p(x, y); & y = f(x); \end{array}$$

Data-flow analysis will allow us to transform the code to $y = f(f(5))$, possibly eliminating the x if it is not needed.

2.3.4 MUTIFICATION OF WHOLE PROGRAMS

Mutification has strict assumptions about the instructions it operates on, so we may need to perform instruction reordering and manipulation to make a program suitable for mutification:

Nested expressions must be broken up. This is done by moving a sub-expression out of its containing expression and assigning it to a temporary variable. When functionalising a program, the reverse operation (expression inlining) can be applied to make as deeply nested expressions as possible, thus enabling easy application of high-level transformation rules.

Calls in control-flow statements must be moved outside. Mutification can't be applied directly to calls in conditions, **return**-statements and so on. Introducing a temporary, as above, we replace the call with a reference to the variable (taking care to recompute the value of the variable for each loop iteration in the case of loops).

Multi-valued procedures should be specialised. If we can't fill up all the outputs of a multi-valued procedure, we create a *sliced* version, with only the needed outputs, and with any unnecessary computations removed.

Instructions should be reordered to take advantage of mutification to a multi-valued procedure call. In general, if the instruction i_2 does not depend on the result of i_1 , and does not change variables in i_1 , it can be moved in front of i_1 .

If reordering fails and slicing is also impossible – for example, if the procedure implementation isn't available – we must insert a dummy call with a throw-away result.

Instructions may be made independent of each other by introducing a temporary. For example, if we want to move the second instruction in front of the first, we can store the value of y in a temporary variable:

<code>x = f(y);</code>		<code>int t = y;</code>		<code>int t = y;</code>
<code>y = g(3);</code>	\rightarrow	<code>x = f(t);</code>	\rightarrow	<code>y = g(3);</code>
		<code>y = g(3);</code>		<code>x = f(t);</code>

Dig up your old compiler book! Instruction reordering and scheduling is well-known from compiler construction, and similar techniques can be applied here. For example, reordering assignments to take advantage of a multi-valued procedure call is not unlike reordering instructions to take advantage of parallel execution paths in a processor.

Many familiar optimisation techniques like constant propagation, value numbering, common sub-expression elimination, dead variable/code elimination and so on can readily be applied to Magnolia programs, either by first converting to a SSA (static single assignment) form, or by using simple data-flow analysis [118]. Pure functions and explicit information about input and outputs of procedures makes it a lot easier to apply optimisations.

Figure 2.1 shows an example program before and after mutification, using the actual intermediate output of the mutification stage of the compiler.

2.4 Discussion

2.4.1 WORKING WITH CONCEPTS AND AXIOMS

The main benefit we have seen so far (apart from the somewhat fuzzily defined notational clarity) is in the relationship between code and specification, particularly with the concept feature.

- Most operations defined in a concept will have an algebraic-style declaration (with the exception of certain things like I/O). There is no need for deciding whether to define a functional-style interface, or an OO-style interface or an imperative-style interface.
- Axioms defined in concepts map directly to algebraic specifications. In C++, axioms for imperative-style concepts are typically written using the comma operator!in C++ (sequential composition of expressions):

```
axiom getset(A a, I i, E e) {
    setElt(a,i,e), getElt(a,i) == e;
}
```

- In most cases, there is a built-in well-defined mapping between procedure declarations and the function declarations used in a concept. As long as program code is written against the concept interfaces, axioms are easily related to program code. In C++, small

ALGEBRAIC
SPECIFICATION &
STYLE

```

procedure fib(out int f, obs int n)
{
  if(n < 2)
    f = n;
  else
    f = fib(n-1) + fib(n-2);
}

procedure fib2(upd int n)
{
  if(n < 2)
    ;
  else
    n = fib2(n-1) + fib2(n-2);
}

procedure fib (out int f, obs int n) {
  bool a_0;
  call _<_(a_0, n, 2);
  if(a_0)
    f = n;
  else
    int c_0;
    c_0 = n;
    call _-_(c_0, 2);
    int b_0;
    call fibonacci::fib(b_0, c_0);
    int d_0;
    d_0 = n;
    call _-_(d_0, 1);
    call fibonacci::fib(f, d_0);
    call _+_(f, b_0);
}

procedure fib2 (upd int n) {
  bool e_0;
  call _<_(e_0, n, 2);
  if(e_0)
    ;
  else
    int f_0;
    f_0 = n;
    call _-_(f_0, 2);
    call fibonacci::fib2(f_0);
    call _-_(n, 1);
    call fibonacci::fib2(n);
    call _+_(n, f_0);
}

```

FIGURE 2.1: Left: Two different variants of a recursive Fibonacci procedure. Right: Results of mutifying the two procedures (actual intermediate output from the compiler). Note that **fib2** – using an **upd** parameter – requires fewer temporaries than **fib**, which uses an **out/obs** combination. The notation **_+_(a,b)** is simply a desugared function-call variant of an operator call **a + b**, and **call _+_(a,b)** is an updating procedure call, similar to **a += b** in C++.

wrappers are often required to translate between the declaration style in the concept and that of the implementation. This indirection makes it much harder to relate axioms to actual program code, for example for use as rewrite rules [147].

Axiom-based rewriting [7] is a convenient way to do high-level optimisation by using equational axioms as rewrite rules – similar to how algebraic laws are used to simplify arithmetic expressions. With code based on pure, well-behaved functions and no aliasing, this is very simple to implement, and rules can be applied without any of the complicated alias analysis that are part of modern compilers.

Note that while this work complements the concept feature, it is by no means dependent on it, and will work equally well without.

2.4.2 LIMITATIONS OF MUTIFICATION

Mutification and functionalisation have some limitations. We are basically hiding imperative code behind an interface of pure functions, and if we are to do this without costly overhead, what we're hiding must be reasonably pure. This means that:

- Procedure results can only be influenced by input arguments.
- A procedure can have no other effect on program state than its effect on its output arguments.
- Objects used as function arguments must be *clonable* – i.e., it must be possible to save and restore the complete state of the object.

The two former limitations rule out procedures operating on global variables. The latter rules out things like stream I/O and user interaction.

Global variables are problematic because they interfere with reasoning – introducing unexpected dependencies or causing unexpected side-effects. This breaks the simple algebraic reasoning model that allows us to move code around and modify it without complicated analysis. Global variables can be handled in some cases by passing them as arguments, either explicitly or implicitly (having the compiler add global variables to declarations and calls). At some point we will reach an outermost procedure in the call tree which will have to either get all globals as arguments from the run-time system, or be allowed to access globals directly. If the global state is large, passing it around can be impractical, particularly since mutification will require that the state can be cloned when necessary. However, global variables are generally frowned upon anyway, and create problems with reentrancy and multi-threading, so this may not be a problem in practise.

GLOBAL VARIABLES

The clonability requirement comes from the need to sometimes introduce temporaries while applying mutification, which is necessary to protect against aliasing, and also useful to avoid unnecessary recomputations. Some objects are however unclonable, e.g., because they represent

CLONABILITY

some kind of interaction with the real world (reading and writing to a file or terminal, for example). Such objects are considered *impure*. They can only be used as `upd` arguments, and transformation on code involving impure objects must preserve the order of and conditions under which the objects are accessed. This is similar to how a C compiler would treat `volatile` variables in low-level code like device drivers.

IMPURITY

Impure objects are best handled using procedures, though in principle we can mutify function calls involving impure objects as long as no copying is required.

2.4.3 PERFORMANCE

We have no performance data on Magnolia yet, but we have experimented with a simple form of mutification to reduce overhead in the Sophus numerical software library [40; 79]. Sophus is implemented in C++ in an algebraic coding style – i.e., preferring pure function calls over argument updates. Mutifying the code of the Seismod seismic simulation application, we saw a speedup factor of 1.8 (large data sets) to 2.0 (small data sets) compared to the original non-mutified algebraic-style code. Memory usage was reduced to 60%.

2.4.4 RELATED WORK

FLUENT
LANGUAGES

Fluent languages [53] combine functional and imperative styles by separating the language into sub-languages, with `PROCEDURES` which are fully imperative, `OBSERVERS`, which do not cause side-effects, `FUNCTIONS`, which do not cause and are not affected by side-effects, and `PURES` which are referentially transparent (no side-effects, and return the same value on every evaluation). The invariants are maintained by forbidding calls to subroutines with more relaxed restrictions. Our mutification, on the other hand, takes responsibility for protecting against harmful side-effects, allowing calls to procedures from functions.

EUCLID

The Euclid language [122] is designed with verification in mind, and has the same distinction between procedures and side-effect free functions as Magnolia, and also forbids aliasing. The aliasing rules are similar to Magnolia, but also deal with pointers and passing array components and dereferenced pointers as arguments, which is forbidden in Magnolia.

Mutification bears some resemblance to the translation to three-address code present in compilers [1]. Our approach is more high-level and general, dealing with arbitrary procedures instead of a predetermined set of assembly/intermediate-language instructions.

FUNCTIONAL
LANGUAGES

Copy elimination [58] is a method used in compilation of functional languages to avoid unnecessary temporaries. By finding an appropriate target for each expression, evaluation can store the results directly in the

place where it is needed, thus eliminating (at least some) intermediate copies.

Expression templates [153] avoid intermediates and provide further optimisation by using template meta-programming in C++ to compile expressions (particularly large-size numerical expressions) directly into an assignment function. This allows the user to write algebraic-style expressions, but an extra burden is placed on the implementer who must manually code all operators into the expression template system.

C++ TEMPLATES

2.5 Conclusion

We have presented a programming method and formal tool, which decouples the usage and implementation of operations. Through *functionalisation* and *mutification*, we make imperative procedures callable as algebraic-style functions and provide a translation between code using function calls and code using procedure calls. This decoupling brings us the following benefits:

DECOUPLING USE
&
IMPLEMENTATION

- Reuse – having a unified call style simplifies the interfacing of generic code with a wide range of implementations. Signature manipulation may also help integrate code from different sources. This is by no means a complete solution to the problem of reuse, but a small piece in the puzzle which may make things easier.
- Flexibility – multiple implementations with different performance characteristics can be accessed through the same interface.
- Notational clarity of functional style – algebraic notation is similar to mathematics and well suited to express mathematical problems.
- Link to axioms and algebraic specification. The algebraic notation can be directly related to the notation used in specifications, thus bridging the gap between practical programming and formal specifications. This enables the use of axioms for high-level optimisations and for automated testing [7].
- The procedural imperative style with *in situ* updating of variables provide better space and time efficiency, particularly for large data structures.

The notational and reasoning benefits are similar to what is offered by functional programming languages, without requiring immutable variables.

Our initial experiments with mutification were done on C++, because the excellent, high-performance compilers available for C++ make it a good choice for performance-critical code. Languages like C++ are however notoriously difficult to analyse and process by tools. The idea behind Magnolia is to cut away the parts of C++ that are difficult to

process or that interfere with our ability to reason about the code, and then add the features necessary to support our method. The Magnolia compiler will produce C++ code, allowing us to leverage both good compilers and good programming methodology.

This paper expands on earlier work [40] on mutification by (1) allowing the imperative forms to update an arbitrary number of arguments (previously limited to one), (2) allowing functionalisation of procedure calls, (3) providing a formal treatment of functionalisation and mutification, and (4) doing this independently of the numerical application domain which was the core of [40]. Functionalisation and mutification was originally motivated by code clarity and efficiency concerns, with the benefits to generic programming becoming apparent later on.

More work remains on determining the performance improvement that can be expected when using imperative vs. algebraic styles, and the productivity improvement that can be expected when using algebraic vs. imperative style. Both these aspects are software engineering considerations. Further work on formally proving the effectiveness and correctness of mutification is also needed.

A prototype implementation of Magnolia, supporting functionalisation of declarations and mutification of calls, is available at:

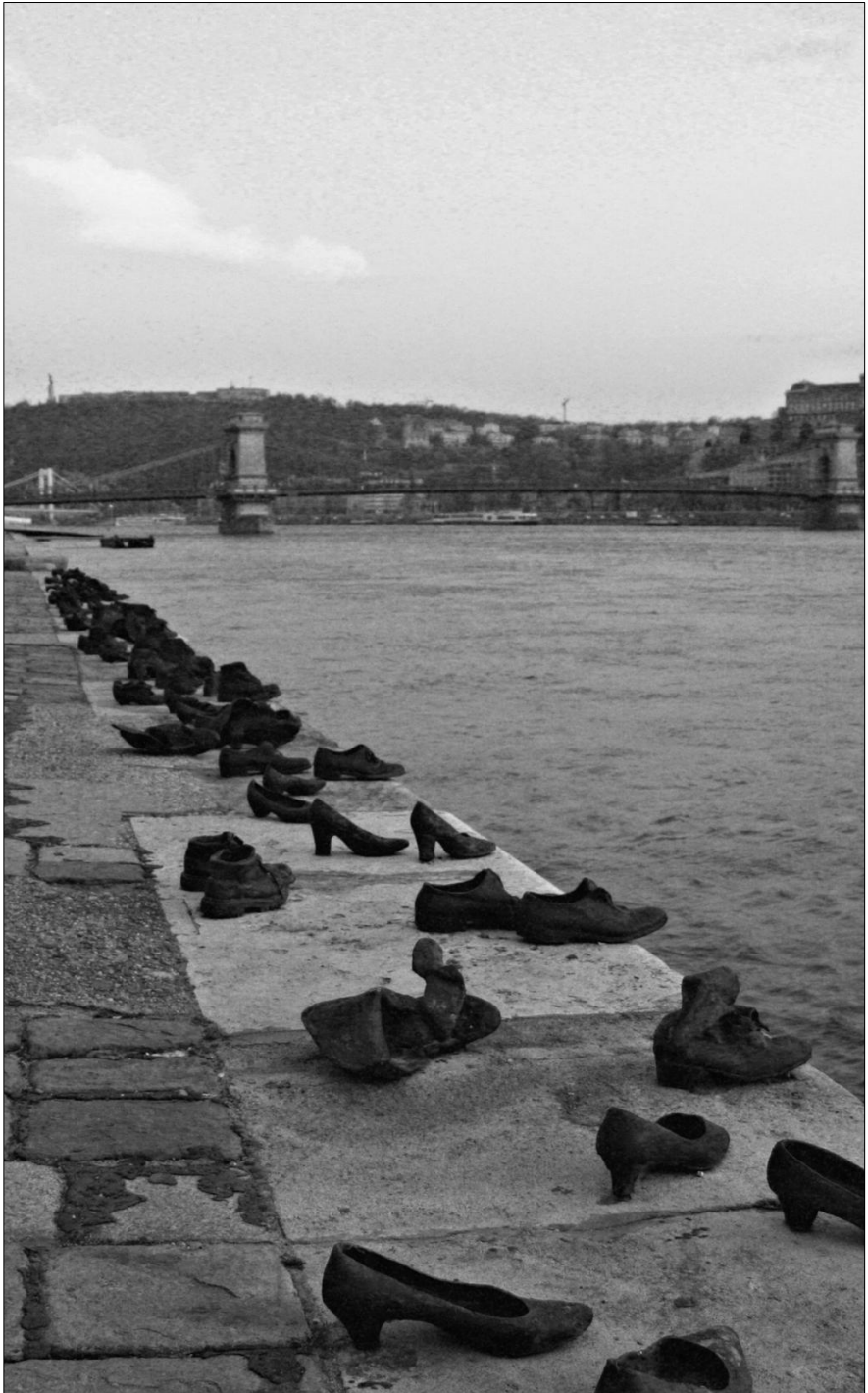
<http://magnolia-lang.org>

Working with Axioms

Bringing specification together with implementation creates interesting possibilities. Axioms give meaning to operations in a program in a way that is easier to reason about than implementation code, providing the compiler with information that enables high-level transformation of code. Also, as axioms are typically formulated in an entirely different way from implementation code, we get some redundancy in the semantic description of a program, which we can exploit for testing purposes.

In the previous chapter, we saw how we could use an algebraic signature style, even for procedures which update their parameters. This is particularly handy when specifying operation behaviour using axioms. The paper in this chapter, presented at LDIA 2008, is based on the concept extension to C++. Unfortunately, C++ does not support functionalisation and mutification, nor does it guarantee side-effect free expressions. Axioms for operations with side effects may be expressed using C++'s control-flow sensitive operators, such as the comma operator. Although we can still do rewriting and testing with axioms under these conditions, we may need more analysis when doing rewrites. This chapter assumes mostly algebraic-style C++, which will of course work nicely with a realisation of the same idea in Magnolia.

In the next chapter, we will explore the use of axioms for testing in more detail. In Magnolia, axiom-based transformations can be implemented by mapping axioms and axiom classes to *transforms* in the compiler – this is discussed later, in Section Chapter 7.



Memorial to Jews shot into the Danube by the Arrow Cross Militia during 1944-45.
Danube Promenade along Széchenyi Street, Budapest (ETAPS'o8 / LDTA'o8)

Axiom-Based Transformations

Optimisation and Testing

ANYA HELENE BAGGE

MAGNE HAVERAAEN

Department of Informatics, University of Bergen, Norway

ABSTRACT

Programmers typically have knowledge about properties of their programs that aren't explicitly expressed in the code – properties that may be very useful for, e.g., compiler optimisation and automated testing. Although such information is sometimes written down in a formal or informal specification, it is generally not accessible to compilers and other tools. However, using the idea of *concepts* and *axioms* in the upcoming C++ standard, we may embed axioms with program code. In this paper, we sketch how such axioms can be interpreted as rewrite rules and test oracles. Rewrite rules together with user-defined transformation strategies allow us to implement program or library-specific optimisations.

3.1 Introduction

In the coming C++0x standard [64], it is proposed that axioms be part of the new *concept* construct. The idea of concepts is to let programmers place restrictions on template parameters. For instance, a generic sorting function may specify that its argument should be an *Indexable* object with *LessThanComparable* elements. Without concepts, one would have to just go ahead and use the indexing and less-than operators, and then the compiler would give an incomprehensible error message if someone tried to use the sorting function with an unsuitable data structure.

When we refer to 'C++' in this paper, we refer to the concept-enabled proposed standard. *Note that concepts have recently been removed from the proposed final standard.*

This is a reprint of: Bagge, A.H., and Haveraaen, M. 2008. Axiom-Based Transformations: Optimisation and Testing. In Proceedings of the Eighth Workshop on Language Descriptions, Tools and Applications (Budapest, Hungary, April 5, 2008). LDIA '08. Electronic Notes in Theoretical Computer Science **238**/5 (2009). DOI: 10.1016/j.entcs.2009.09.038
www.elsevier.nl/locate/entcs, © 2009 Elsevier B.V. Printed by permission.

Concepts can also be organised in a hierarchy – allowing well known algebraic concepts to be mapped to C++ concepts [60] in a structured way.

CONCEPT MAPS

Using a *concept map*, one can specify that a class or group of classes satisfies a given concept, and possibly also map between the names and signatures of the concept and those of the class. For example, “my class is *LessThanComparable*, with ‘ $!(x \geq y)$ ’ as the less than operator”. The idea of axioms in the proposed standard is in the early stages. There is a defined syntax for it, and a few examples, but the specification of behaviour is fairly limited.

The compiler is allowed, but not required, to replace expressions with equivalent expressions according to axioms – for optimisation purposes, for example. Earlier, the compiler had little opportunity to make assumptions about user code, for instance, it could not apply the many simplifications available for built-in operators to expressions with user-defined operators. Such rules may now be stated as concepts, but there is still no way to give hints to the compiler as to which axioms may be useful, which side of the equality is preferable, how rewrite rules should be applied, etc.

USING AXIOMS

Axioms may serve many purposes in a software system:

1. Recording knowledge about template arguments for generic classes and methods.
2. Proving that one set of axioms implies another set of axioms – for example, that a class belonging to a concept C_1 may be used where a concept C_2 (with the same or a smaller signature) is required – and also program verification – proving that a class satisfies its stated properties.
3. Semantics-preserving rewrites of the code, e.g., for optimisation purposes.
4. Testing that a class satisfies its stated properties.

Item (1) is more or less where the proposed standard stands today; with axioms as a form of structured documentation of the requirements on concepts. Item (2) may not be applicable to C++, partly because C++ syntax and semantics is very difficult to analyse, partly because many requirements cannot be expressed as axioms. We will focus on items (3) and (4) in this paper, based on our previous experience with user-defined rewrite rules for C++ [11] and axiom-based testing [74; 75].

The rest of this paper is organised as follows. We start by introducing axioms, then discuss the use of axioms for rewriting (Section 3.3) and testing (Section 3.4). We will then sketch some implementation issues (Section 3.5), and finish with a discussion and conclusion (Section 3.6).

```

RequiresClause? "axiom" Identifier "(" ParamDecls ")" AxiomBody
                                     -> AxiomDef
"{" Axiom* "}"                      -> AxiomBody
ExpressionStmt                      -> Axiom
"if" "(" Condition ")" ExpressionStmt -> Axiom

```

FIGURE 3.1: Proposed C++0x Standard Syntax for Axioms (in SDF2 notation). The *RequiresClause* allows for additional concept constraints on axiom parameters – we will ignore it in this paper.

```

RequiresClause? "axiom" Identifier "(" ParamDecls ")"
    GroupClause? AxiomBody                -> Declaration
":" {GroupName " " "+"                  -> GroupClause
"{" (Axiom|Statement)* "}"              -> AxiomBody
"assert" "(" Expression ")" ";"          -> Axiom

```

FIGURE 3.2: Our syntax for Axioms (in SDF2 notation).

3.2 Concepts and Axioms

The proposed C++ standard syntax for axioms is shown in Figure 3.1. Each axiom definition can contain multiple axioms, and the axioms themselves are expression statements with the ‘==’ operator. For example, here’s a concept *Monoid* with an *Identity* axiom (taken from Gregor et al. [64]):

```

concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
    T identity_element(Op);
    axiom Identity(Op op, T x) {
        op(x, identity_element(op)) == x;
        op(identity_element(op), x) == x;
    } }

```

We may then specify that a class *Vector* satisfies the *Monoid* concept, with *Vector::plus* as the operation and *Vector::zero* as the identity element:

```

concept_map Monoid<Vector::plus, Vector>{
    Vector identity_element(Vector::plus) {
        return Vector::zero;
    } }

```

For our work, we have chosen a slightly different syntax (see Figure 3.2). Our syntax extends the original syntax by allowing *axiom groups* (see Section 3.2.1), by allowing any statement to be used within the axiom body (used in testing, see Section 3.4), and by marking the actual axiom with the keyword *assert*. Also, we allow axioms to be declared outside

Vector::plus is a class wrapper around the plus operation, necessary to use it conveniently as a template argument. *Vector::plus()* is an object of this class, usable as a function as it has an overloaded *()*-operator. This is a usual way of doing things in C++, and we shall refrain from commenting on the intuitiveness of it...

GroupDef	-> Declaration
"axiom_group" GroupName (GroupBody ";")	-> GroupDef
"{" GroupDecl* "}"	-> GroupBody
"using" Name ";"	-> GroupDecl
AxiomDef	-> GroupDecl
TemplateSpec GroupDecl	-> GroupDecl

FIGURE 3.3: Syntax for Axiom Groups. A *GroupDef* defines or extends a named axiom group. The *GroupName* is an identifier or a qualified (nested) name. The *GroupBody* lists the axioms or axiom groups that form the group. *Name* should be the name of an axiom, group or concept.

CONSTRAINED
TYPES

of concepts, so that one may attach simple axioms directly to a class, instead of having to declare concepts and concept maps for it.

A standard C++ concept map is only ‘active’ when the mapped class is used as a template argument that has been constrained to a concept. E.g., *Vector* will only be considered a *Monoid* when it is used as a *Monoid* in generic code. This restricts the use of axioms, and we will instead assume that the existence of a concept map means we can use concept axioms for the classes and operations in the map.

Using the *assert* keyword makes it easier to allow a wider variety of statements in the axiom body, and to allow for other kinds of axioms than just equality. It also simplifies making the axioms executable as test code (Section 3.4).

In our syntax, the above monoid concept becomes:

```
concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
  T identity_element(Op);
  axiom Identity(Op op, T x) : simplify {
    assert(op(x, identity_element(op)) == x);
    assert(op(identity_element(op), x) == x);
  } }
```

The *axiom group* ‘*simplify*’ identifies the *Identity* axiom as usable for a simplification rule, where the right-hand side is assumed to be simpler (less resource-intensive) than the left-hand side.

3.2.1 AXIOM GROUPS

Axiom groups are used to identify which axioms are useful as rewrite rules, and to distinguish between different types of rules – e.g., simplification rules, reordering rules like associativity/commutativity, or rules that should be applied early or late in the transformation process.

Axiom groups are defined using the *axiom_group* construct (Figure 3.3), or by listing the group name in the *group clause* of an axiom

Such tagging is also
important in
specification and proof
systems like CASL
[115].

definition. The `axiom_group` definition is open-ended and can be extended later on. Here's a sample axiom group *simplify*, containing the *Identity* axiom from the *Monoid* concept (having the same effect as in the example above):

```
axiom_group simplify {
    template<typename Op, typename T>
        using Monoid<Op,T>::Identity;
}
```

The template declaration has the effect of universally quantifying the operator and type for the monoid, adding the identity axioms from all monoids.

The `using` directive allows us to add a single axiom, all axioms from a given concept (e.g., *using Monoid*), or all axioms from another axiom group (*using my_simplify*). Axioms may also be defined directly in the axiom group.

Listing an axiom in a group definition is equivalent to listing the group name in the axiom definition. We allow both possibilities to cut down on the amount of code that needs to be written for a simple axiom. We recommend using the axiom definition for fairly standard groupings that follow straight-forwardly from the axioms – commutativity for example – and using separate axiom group definitions for transformation system-related grouping – ordering rewrites in stages, for example. A few suggested axiom groups are:

AXIOM GROUPING

- ac* – associative-commutative – a (possibly non-terminating) reordering of an expression
- simplify* – right-hand side is preferred over left-hand side; repeated application should terminate
- propagate* – introduction and propagation of properties across expressions

3.3 Rewriting with Axioms

The jump from having axioms to using them for optimisations isn't far. The proposed C++ standard already suggests that compilers may use axioms for simplifying code, and there already exists a proof-of-concept for axiom-based optimisation [147], based on ConceptGCC. Similar ideas have been used successfully in systems like TAMPR [22] and CodeBoost [11].

3.3.1 BASIC REWRITING

Let's start with a brief explanation of how rewriting works, for those unfamiliar with the concept. A conditional rewrite rule consists of a match pattern, a replacement pattern and a condition. If the condition is always

REWRITE RULES

true, it can be omitted. The patterns may have variables (sometimes called meta variables, to distinguish them from C++ variables).

For example, consider the following rewrite rule:

$$g(f(x)) \rightarrow h(x)$$

$g(f(x))$ is the match pattern, $h(x)$ is the replacement pattern, and x is a variable. If the rule is applied to the expression $g(f(42))$, the result will be $h(42)$. Variables in the replacement pattern should be a subset of the variables in the match pattern and condition.

We may derive a rewrite rule from any axiom formed from a conditional equation, simply by choosing one side of the equation as the match pattern, and the other as the replacement pattern. If we choose the other way around, we get the inverse rule. In case of a simplification rule, going one way is preferred, but in the case of a commutativity rule, either rewrite direction can be useful. The axiom's parameter list defines the variables of the rule.

Rules are applied to expressions. If we wish to apply a rule to a whole program (i.e., all expressions in the program), we must use a rewriting strategy (discussed below). A rule may either *succeed*, if it matches and its condition is true, or it may *fail*. If it succeeds, the rewrite is performed and we are done. If it fails, we may try other rules if we are applying a group of rules, until we find one which succeeds. A typical rewrite strategy would visit all expressions in a program, and repeatedly apply rules until no rules succeed.

It is important to note that rewriting is not merely syntactic, it also takes into account the types and signatures of the expressions. For example, a rule derived from this axiom

```
axiom Commute(int a, int b) : ac {
    assert(a + b == b + a);
}
```

will apply to an expression $5 + 4$, but not to $4.2 + 6.9$, which uses the floating-point plus operator.

Rewrite rules are typically given names, in our system the name follows from the axiom name. Rule names share the same name space as axiom group names. Rules are allowed to have the same name – in that case they form a group and will be applied together (i.e., tried in an arbitrary order until one succeeds or all have failed).

We can expect a rewrite system based on C++ axiom to be both non-confluent (i.e., applying rules in a different order gives a different result) and non-terminating. Rewrite strategies together with axiom groups provide a pragmatic solution to this, and allow us to carefully control the application of rules.

SUCCESS &
FAILURE

SEMANTIC
MATCHING

CONFLUENCE &
TERMINATION

```

"strategy" Identifier "(" ParamDecls ")" ";" -> StrategyDecl
"strategy" Identifier "(" ParamDecls ")" StrategyBody
                                                -> StrategyDef
"{ " Statement* "}"                               -> StrategyBody

```

FIGURE 3.4: Syntax for strategy definitions – this may be parsed as a definition of a function returning a value of type ‘strategy’. A limited set of statements and operators (Figure 3.5) for the strategy language usable in the *Stratego*Body.

3.3.2 REWRITE STRATEGIES

We can get pretty far basing an optimisation tool on just one or a few fixed rewriting strategies, and linking each strategy to a particular axiom group. For example, a possible default optimisation strategy would be to do a bottom-up simplification (using the *simplify* group) of the program tree, modulo *ac* rules. Axiom-based optimisation could then be turned on by a compiler option like *-frewrite-rules*.

We can do better, of course. User-defined strategies allow detailed control over when and how various rewrites are applied. Boyle showed in the TAMPR system [22] that such control was needed to obtain many of the goals of program optimisation from rewrite rules.

Basing ourselves upon an existing program transformation language like Stratego [25], we can make its strategy-building constructs available within C++-like syntax (see Figure 3.4), and then either compile our rewrite rules and strategies to Stratego code, or execute them using a Stratego interpreter. The above default strategy may be encoded like this:

STRATEGO

```

strategy simple_opt() {
  bottomup(repeat(simplify || (ac && simplify)));
}

```

where *bottomup* does a bottom-up traversal of the program tree, and *repeat* applies its argument until it fails. The choice combinator *||* tries its left argument, then its right argument if the left fails (corresponds to *<+* in Stratego), and the sequence combinator *&&* applies its arguments in sequence and succeeds only if both succeed (corresponds to *;* in Stratego). The two group applications *simplify* and *ac* will apply the actual rewrite rules at the current position in the program tree. See Figure 3.5 for an overview of strategy combinators.

TRAVERSALS &
COMBINATORS

Strategies like *simple_opt* may then be made available to the user as a compiler or transformation tool option, e.g. *-frewrite-using=simple_opt*.

3.3.3 INTEGRATING WITH OTHER OPTIMISATIONS

Often an optimisation rule may only be applied at a particular level of abstraction. Inlining, for example, will commonly expose some oppor-

<code>true</code>	do nothing – always succeeds
<code>false</code>	do nothing – always fails
<code>all(s)</code>	apply <code>s</code> to all children of the current node.
<code>s1 s2</code>	apply either <code>s1</code> or <code>s2</code> , trying <code>s1</code> first
<code>s1 && s2</code>	apply <code>s1</code> then <code>s2</code>
<code>a</code>	apply rewrite rules name <code>a</code>
<code>G</code>	apply any rule from axiom group <code>G</code>
<code>G[a]</code>	apply any rule named <code>a</code> from axiom group <code>G</code>
<code>repeat(s)</code>	apply <code>s</code> repeatedly until it fails.

FIGURE 3.5: Some suggested strategy combinators and builtin strategies.

tunities for rule application, while hiding others. For example, in an expression $a + f(b)$ we may be able to inline or do partial evaluation of f and figure out that it returns a zero – thus allowing us to eliminate the $+$ (using a rule derived from *Monoid::Identity*). But if we inline the plus (e.g., in the case of vector addition), our rule would no longer match.

INLINING

Exposing an inliner interface to the strategy language would be quite useful. In our earlier work on user-defined rules [11] in C++, we added inlining rules for simple functions to open up optimisation opportunities that may otherwise have been lost. For example, we might have a rule

```
axiom GetElement(T a, T::index_type i) : inline {
    assert(a[i] == a.data[int(i)]);
}
```

inlining the code of the user-defined `[]`-operator. Since this is just a simple duplication of the implementation of `[]`, getting the compiler's inliner to do the job would be more general and preferable.

Substitution of expression assignments may also be helpful. For example, consider the rule $a * x + y == \text{axpy}(a, x, y)$ that combines a multiply and an addition into a single operation. We may not be able to tell that the rule can be applied if the operations occur in different statements (possibly far apart in the code):

```
Vector a, b;
a = 5 * a;
b = a + b;
```

But if the expression assigned to a is substituted in the last statement,

```
b = 5 * a + b;
```

we may be able to transform to a more efficient

```
b = axpy(5, a, b);
```

Combined operations like *axpy* are common in numerical libraries like BLAS, when working on large data structures like vectors and matrices.

Supplying such rules together with a library saves the programmer from having to remember all the optimised special-case forms – and also means that new optimisations can be added later without having to rewrite existing code.

3.3.4 PROPERTIES AND PROPAGATION

Certain axioms may depend on certain properties being fulfilled. For example,

```
axiom SortSort(T a) : simplify {
  if(sorted(a))
    assert(sort(a) == a);
}
```

This axiom can be used to eliminate unnecessary sorting of an already sorted data structure. It is perhaps not very likely that a programmer will ask for an array to be sorted again just after it was sorted, but by using data-flow analysis, we may track this kind of information throughout the program.

To do this, we need to figure out when an array is sorted (we'll simply call it an 'array', even though it might as well be any ordered data structure). A just-sorted array is sorted:

PROPAGATION
AXIOMS

```
axiom Sorted(T a) : propagate {
  assert(sorted(sort(a)));
}
```

Furthermore, removing an element from an array results in an array that is still sorted (well, at least it does for our kind of array):

```
axiom SortedRemove(T a, T::index_type i) : propagate {
  if(sorted(a))
    assert(sorted(a.remove(i)));
}
```

The *propagate* group is used to identify axioms that may be suitable for data-flow propagation of properties. Any Boolean predicate like *sorted* above is usable as a propagated property. Note that without propagation axioms we are unable to assume that any modification (e.g, call to a non-const function) of an object preserves its properties.

Properties may be used as a basis for choosing a more efficient algorithm. For example, the axiom

```
axiom SortedSearch(T a, T::value_type e) : speedup {
  if(sorted(a))
    assert(linsearch(a, e) == binsearch(a, e));
}
```

allows us to choose binary search over linear search of a sorted array. We have chosen the axiom group *speedup* for this axiom, as it's a slightly different concept from expression simplification. Conceivably, our optimisation strategy may put more work into proving that a speedup rule can be applied, than it needs for a simplification rule.

Tracing properties of objects is often useful in numerical programming, where certain operations can be drastically faster if it is known that the operands have special properties, like symmetry in matrices, for example.

3.4 Axioms for Testing

Once we have axioms and rewrite rules based on them, we'll want to check that our implementation satisfies the axioms. In particular, before we apply optimisation rules to a program, it is a good idea to check that the rules won't change the meaning of the program. While axioms may be used for formal program verification, this is difficult to achieve in a general-purpose language. We can however take a more pragmatic approach, and use the axioms as a basis for testing.

A test oracle is something that tells you the correct result for an operation you want to test.

Using axioms as test oracles is straight-forward – fill in test data for the free variables, and see if the axiom evaluates to true [50; 74; 75]. It is a pity this kind of specification-based testing isn't made more apparent in the upcoming standard, as it would be a good motivation for actually writing axioms in programs.

Providing basic support for testing is quite simple – we only need to make the instantiated (after concept mapping) axiom code available as callable functions. The testing code may then be called from a test program, or from a testing framework (like JUnit [105] for Java).

For example, referring to the *Identity* axiom for *Monoid*, we may test that it holds for integers, by calling it with a few different integer values:

```
for(int i = -2; i <= 2; i++) {  
    Monoid::Identity(std::multiplies<int>(), i);  
    Monoid::Identity(std::plus<int>(), i);  
}
```

Here, *std::multiplies* and *std::plus* refers to predefined class wrappers for built-in operators. They are necessary because of how the monoid concept is defined (with the operation as a template parameter) – the operator parameter is not really a free variable in the axiom.

3.4.1 AXIOMS WITH COMPLEX TESTING CODE

Axioms used for testing can be written in any (computable) logic. For testing purposes, it therefore makes sense to allow arbitrary C++ code

inside an axiom definition – though this is not allowed in the proposed C++ standard. For instance, we may state that two arrays are equal if they have the same number of elements, and that the elements are equal

```
axiom ArrayEqual(Array a, Array b) {
    bool eq = a.size() == b.size();
    if(eq)
        for(int i=0; i<a.size; ++i)
            eq &= a[i] == b[i];
    assert(eq == (a==b));
}
```

The first lines are needed to iterate through the data set and accumulate information about its components. The `assert` keyword helps to identify which part of this statement sequence is actually the test which defines the axiom. It could be given its C library meaning – abort the program if the test fails – or we could make a more elaborate implementation that counts the number of failures and successes and records the axioms that fail. It may even be useful to allow `assert` to have additional parameters, e.g., for adding extra information about the test.

Although just allowing simple expressions in axioms is nice, being able to write support code for an axiom provides us with more expressive power. It is also possible to live without “helper” code – then we would need to encapsulate the helper code as a Boolean function, possibly making simple axioms more complicated.

HELPERS

Note that while the above axiom may seem trivial, properly testing the implementation of equality is important in order to be sure that other axiom tests relying on it work correctly.

3.4.2 TESTING EXCEPTION BEHAVIOUR

It is also useful to state as an axiom that a method should throw an exception under specific conditions:

```
axiom DivZeroThrows(T x, T y) {
    if(y.iszero())
        try { div(x, y); assert(false);}
        catch(DivisionByZero) {assert(true);}
}
```

The first assertion tests the lack of an exception being thrown. The second confirms the expected catching of an exception. If helper statements are not allowed, this axiom also needs to be encapsulated.

3.4.3 LIMITATIONS ON TESTING

The tests derived in this fashion are clearly only as good as the axioms they are based on. If the axioms are wrong, the tests will also be wrong

(though one is likely to discover this if the implementation one is testing is correct).

Tests based on the equalities and other comparisons rely on the comparison being correctly implemented. Also, if the two sides of an equality are faulty, but give equal results, this will go unnoticed. In practise, though, comprehensive testing with varied test data and multiple axioms is likely to uncover that something is wrong, even if circumstances conspire against some of the axiom tests.

TEST DATA

Effective testing requires good test data. This is something we haven't considered here – we have relied on the programmer supplying appropriate test data. The danger here is that a programmer's assumption of what constitutes good test data is often wrong – the bugs are hiding where one least expects them to be. We will revisit this subject later (Section 3.6).

3.5 Implementation Issues

The biggest hurdle facing any implementation of a C++ extension is implementing support for the base language. Existing C++ frontends are either far away from supporting the full C++, or are difficult to extend, making the cost of prototyping new language features high.

Support for concepts and axioms has been implemented in the experimental ConceptGCC compiler [63]. At least one attempt to implement axiom-based rewriting (based on ConceptGCC and the C++0x proposal) has been made [147] (and they note that rule application and pattern-matching on the internal GCC representation is indeed quite challenging).

We have not implemented the features described in this paper, but we do have previous experience implementing axiom-based rewriting for C++ [11] and deriving C++ tests from axioms [75]. In this section we will briefly sketch how a prototype may be implemented for C++.

3.5.1 TRANSLATING AXIOMS TO RULES

The syntax used in axioms differs from actual C++ code – people write $a + o$, not $op(a, identity_element(op))$. This means that to apply rules to user code, we must first translate the rules according to concept map mappings, inlining any concept map functions (like *identity_element*). For example, the *Identity* axiom

```
op(x, identity_element(op)) == x;
```

instantiated for *Vector::plus* and *Vector* becomes,

```
Vector::plus()(x, identity_element(Vector::plus())) == x;
```

then if we inline *Vector::plus()* and *identity_element()*, we get

```
x + Vector::zero == x;
```

An implementation of axiom-based testing for C++ is discussed in the next chapter.

SYNTACTIC ISSUES

which is what we would expect to see in user code.

Overload resolution from the C++ frontend can be used to figure out the signature of operations in the rule (e.g., integer addition as opposed to floating-point addition) – this information is then used in building the match pattern for the rule. Axiom parameters become variables in the match pattern. We used this idea in our previous implementation of user-defined rules.

Note that rewrite rules can be applied to generic template code when the template parameters are constrained by concepts. Without concepts, you'd have to instantiate the template to see which classes are used before you could apply rewriting. Template code making use of concepts will already be written in the same terms as the axioms (e.g., using *op* and *identity_element*), so less work has to be done to adapt the rules.

3.5.2 OVERALL TRANSFORMATION PROCESS

The overall processing of axioms as rules may proceed as follows. First, we need to parse and perform semantic analysis on the program, giving an abstract syntax tree (AST) annotated with type and signature information. The frontend's overload resolution should also be applied to axioms (either now, or after concept mapping) so that the full signature of functions etc. in calls are available for matching in rules.

The axioms and strategy definitions are then picked out from the program tree. Axioms may then be compiled to rules that apply at the concept level, and after applying concept maps, to rules at the class/user level. Testing code is generated after applying concept maps.

Once the rules are available, we apply them according to our built-in or user-defined optimisation strategy. Rule application should be combined with inlining and data-flow analysis for maximum benefit.

3.6 Discussion

Embedding optimisation rules in programs is not a new idea. User-defined rules in CodeBoost [11] were inspired by the rewrite rules in the Glasgow Haskell Compiler [90]. The CodeBoost implementation was more advanced, however, and supported both conditions and multiple strategies (through a simpler version of the axiom groups introduced in this paper). Conditional rewrite rules is well known from transformation languages such as Stratego [25] and ELAN [21], both of which also support strategies, and from term rewriting in general.

The CodeBoost user-defined rules were limited to rewriting, and although we derived optimisation rules from a formal algebraic specification, the actual coding of the rules was done by hand and related directly to C++ classes. Concepts in C++ let us bridge the gap between

TRANSFORMATION
LANGUAGES

CODEBOOST

the specification (written in terms of algebraic ideas like monoids, fields, rings etc.) and the concrete implementation as C++ classes. Writing the relevant parts of our specifications in concept syntax, with axioms, will make the axioms automatically available for use by tools. There is still a piece missing in the puzzle, though – tying the information from concepts and axioms together with some form of structured documentation. No doubt someone is already working on this problem.

CONCEPTGCC

Support for axiom-based rewriting for ConceptGCC [147] follows the C++ standard proposal more closely than we do here. Rule application in [147] is for now restricted to contexts explicitly constrained by concepts (see Section 3.2), and transformation is restricted to a single strategy (leftmost-outermost reduction). The goal is to have a framework for concept based optimisation that may eventually replace type-specific built-in simplifications in the compiler. We do not aim to compete with this implementation, though we hope that some of our ideas will be useful for their work.

EXPRESSIONS &
STATEMENTS

Axiom-based rules as described here is limited to operating on expressions, and will not be fully effective if the transformation system isn't able to combine expressions from multiple statements (e.g., nesting expressions as much as possible). There is a trade off here between rule-based optimisations, and optimisations like common sub-expression elimination – duplicating some common sub-expressions may open up for rule applications but can also lead to duplicated work.

SOPHUS

Our initial experiments with rules had quite promising results. A small number of rules from the specification of the Sophus numerical library was used (together with general simplification rules) to optimise the C++ implementation of Sophus, giving 5–10 times speedup (the latter after the source code was simplified to take advantage of the rules) as well as reduced memory use. Some of the speedup was due to other optimisations in the system – running without rule-based optimisations gave up to 2 times speedup.

Applying axiom-based rules to existing program code may not give quite as good results, since programmers often have done many of the same optimisations by hand already. We expect the full benefit to be more apparent with previously unoptimised programs written in a high-level style, and when delivered together with a performance sensitive library, such as for numerical software – an idea also known as *active libraries* [36].

Using axioms as a basis for testing is known from systems like DAISTS [50]. Compared to unit-testing frameworks like JUnit [105] – and the xUnit family of frameworks for other languages – the advantage of axioms is the separation of test code from test data. This means that one may easily test one axiom with many different data values, and use the same values to test many different axioms. Comprehensive testing can be

done by building libraries of axioms and test data, and testing all axioms against all suitable data. *Theories*, available in JUnit 4.4 allow universal quantification [126], and may be called from testing code like axioms in this paper. JUnit 4.4 also allows automatic application of available test data to tests.

QuickCheck [34] is a testing system for Haskell where a programmer can state *laws* (like C++ axioms) as Haskell functions. The *quickcheck* testing function generates random test values, and tests that a law holds. Conditional laws are also allowed – in that case more test values will be generated as necessary to test the law sufficiently. The algebraic data types in Haskell makes generating random data structures fairly simple, and the programmer can supply generator functions to fine-tune the data generation. There is no link between QuickCheck and rewrite rules in the Glasgow Haskell Compiler (mentioned above).

QUICKCHECK FOR
HASKELL

DATA GENERATION

Partition testing is another testing approach where one attempts to divide the input domain into regions and test only one value from each region (and some boundary values). This does not necessarily give better results than random testing, though [72]. Both partitioning and random data generation will likely require some help from the C++ programmer. This is an area that should be explored further in order to make full use of axiom-based testing.

3.7 Conclusion

Concepts and axioms, which are being introduced in the upcoming C++0x standard, are language features that may prove quite useful for both documentation, automated testing and program optimisation. To reap the full benefit of axioms, we have introduced a few additional language features:

PROPOSED
FEATURES

- *Axiom groups* for classifying axioms according to how they may be used
- *Strategies* for specifying how and when axiom-based rules should be used
- *Callable axioms / axiom groups* to make axioms usable for testing
- *Properties* which may be propagated according to axioms, and used in axiom conditions

Allowing a wider range of statements in axiom bodies, and allowing axioms to occur outside concepts, gives us some added usability benefits over the proposed standard.

Our main contribution over our own and others' previous work in this area, is setting it in the context of the upcoming C++ standard, and tying together the ideas of specification-based optimisation and specification-based testing.

Acknowledgements

Thanks to Valentin David for useful comments and help with the intricacies of C++ and grammars, and thanks to the referees for many useful comments and tips.

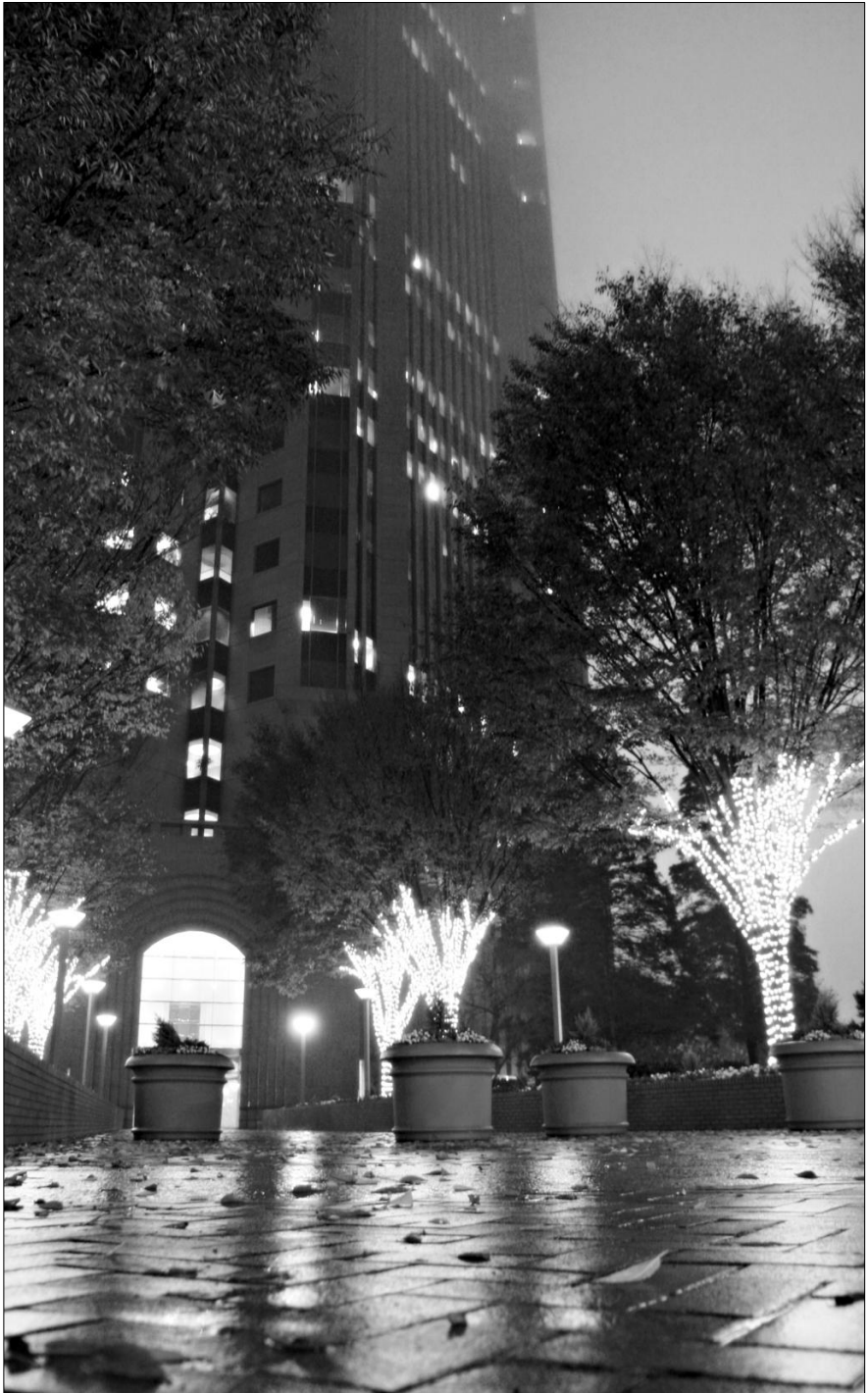
Axiom-Based Testing

The previous chapter introduced concepts and axioms, and also discussed the idea of generating automated tests from axioms. This chapter discusses axiom-based testing in more detail. As before, the discussion is in the context of concept-enabled C++. Axioms from the concepts are used to generate test oracle code, and then test cases are generated for each set of types that model a concept.

Generating test code is surprisingly straightforward, and our tool can even generate code for non-concept C++. But, there is a disconnect between the usual object-oriented or imperative C++ coding style, and the expression-oriented style suitable for equational axioms. Our proposed extensions from the previous chapter might alleviate this, by allowing arbitrary code to be placed inside axioms, at the cost of making axiom-based rewriting more difficult. There is also the problem that C++ functions can have arbitrary side effects, which may interfere with testing and hinder reuse of generated test data.

Both these problems can be avoided in Magnolia, since we can easily write axioms as pure expressions without side effects, and map them to imperative-style implementation code using mutification.

As discussed in this chapter, there are several systems that do unit testing based on algebraic specification. discussing this in the context of C++ concept is particularly useful as there has been much debate about the usefulness of axioms in concepts. A fruitful idea for continued research would be to apply functionalisation to solve issues with side effects, and to use guarding to deal with undefinedness and exceptions.



Bank of America Plaza, the 30th tallest building in the world, and tallest in Georgia.
Atlanta, GA (Workshop on Exception Handling '08)

The Axioms Strike Back

Testing with Concepts and Axioms in C++

ANYA HELENE BAGGE

VALENTIN DAVID

MAGNE HAVERAAEN

Department of Informatics, University of Bergen, Norway

ABSTRACT

Modern development practises encourage extensive testing of code while it is still under development, using unit tests to check individual code units in isolation. Such tests are typically case-based, checking a likely error scenario or an error that has previously been identified and fixed. Coming up with good test cases is challenging, and focusing on individual tests can distract from creating tests that cover the full functionality.

Axioms, known from program specification, allow for an alternative way of generating test cases, where the intended functionality is described as rules or equations that can be checked automatically. Axioms are proposed as part of the *concept* feature of the upcoming C++0x standard.

In this paper, we describe how tests may be generated automatically from axioms in C++ concepts, and supplied with appropriate test data to form effective automated unit tests.

4.1 Introduction

Modern software engineering practises encourage the use of unit testing to increase software reliability. Test-driven development (TDD) [13] dic-

UNIT TESTING

This is a reprint of: Bagge, A. H., David, V., and Haveraaen, M. 2009. The Axioms Strike Back: Testing with Concepts and Axioms in C++. In Proceedings of the 8th international Conference on Generative Programming and Component Engineering (Denver, Colorado, USA, October 4 – 5, 2009). GPCE '09. ACM, New York, NY.

© 2009 Association for Computing Machinery, Inc. Reprinted by permission.

tates that software should be extended by writing tests for a new feature *first*, before implementing the feature. The tests provide a specification of the behaviour of the new feature, and provide an easy way to check the implementation throughout development and refactoring.

Less extreme methods call for tests for all program units, and for regression tests to be written to ward off the reappearance of known bugs. Such methods may be practised rigorously, or in an ad hoc manner. Common to all is that they rely on the programmer to invent good test cases that cover both likely, unlikely and even ‘impossible’ errors. The programmer must also be careful that the tests exercise the full expected feature set, or the implementation will seem OK when all it does is implement the bare minimum to pass the tests (as is common in TDD, where the tests are the actual specification of expected behaviour).

4.1.1 AXIOM-BASED TESTING

We suggest writing tests based on *axioms* that formally specify expected behaviour, rather than relying on ad hoc test cases. Axiom-based testing was introduced in the early eighties in the DAISTS [50] system, which used formal algebraic specifications as a basis for unit testing. In DAISTS, a test consists of axioms in the form of conditional equations, which serve as a test oracle, an implementation with an equality operator; and a set of test data. A simple coverage analysis is done of the test runs to ensure that all the axioms and program code are exercised by the tests.

ASTOOT [44] applied the ideas of axiom-based testing to object orientation, with automated testing for Eiffel. Axioms were specified in an OO-like style, rather than the functional notation used in DAISTS.

The Daistish system [86] brought these ideas to C++. Unlike ASTOOT, Daistish used a functional notation for axioms, giving a notational gap between the conditional equational form of the axioms and the methods of C++.

Traditional unit testing, as popularised by agile methods in the last decades, is practically oriented, and does not rely on formal methods. Mainstream software engineers have focused on development methods like TDD and extreme programming [14], while much formal methods research has focused on formal specification and verification – which have been difficult to apply to mainstream languages and mainstream development.

Research on axiom-based testing has continued, however, and axioms have been introduced as part of the new *concept* proposal for the upcoming C++ standard [16; 63] – giving a mainstream language built-in syntactic support for axioms. The CASCAT system [162] provides a tool for testing Java components based on algebraic specification. Axiom-based testing has been employed in the Sophus numerical C++ library [75],

EARLY WORK

CONCEPTS

and also in the JAX [142] (Java Axioms) testing approach and JAXT [77] tool. Axiom-like features have also been added to recent versions of JUnit [126]. Gaudel and Gall [52] provide a survey of the use of algebraic specification in testing.

Axiom-based testing from concepts has two main parts that instrument the implementation being tested:

- axioms, in the form of conditional equations, and
- suitable test data points.

AXIOM-BASED
TESTING

Running an axiom-based test consists of evaluating the condition and (if it succeeds) the two sides of the equation using the test data, and comparing the results, typically with the help of the equality operator. For example, to test the following commutativity axiom $x + y = y + x$, we may substitute 4 for x and 5 for y , evaluate $4 + 5$ and $5 + 4$, and then verify that $9 = 9$. A good test data set for this case would also include negative numbers and zero.

If the results are to be reliable, the axiom must correctly express the desired feature. Earlier it was considered crucial that the code for the equality operator had to be correct [51]. We have discussed this previously [75], concluding that with testing the equivalence and congruence properties of the equality operator, it can be treated alongside any other function being tested. Another problem appears if the equality operator used in a concept axiom is not implemented. This is known as the oracle problem, and can be handled by techniques based on behavioural equivalence [32; 51], i.e., two values are considered equal if they cannot be distinguished by any operation in the system. The `ASTOOT` [44] system is based on behavioural equivalence, though in practise the user must still define equivalence, either through an axiom, or by an equals operator in the implementation. Chen et al. [32] describe a system for testing object-oriented programs, and provide a technique for determining behavioural equivalence based on white-box heuristics. In this paper, though, we will assume that an equality operator has been implemented for every type that occurs in the left- or right-hand side of an axiom.

EQUALITY

Concepts and axioms are still a work in progress as far as C++ standardisation is concerned. Previous work on C++ axioms has mainly focused on their use for optimisation [7; 147]. Our contributions in this article include:

C++ concepts have recently been dropped from the final proposal.

- a technique for using C++ axioms for testing, and
- a tool to support this technique.

The rest of the paper is organised as follows. In the next two sections we introduce C++ concepts and axioms, and show how to generate test oracles and test code from them. In Section 4.4, we discuss how to generate test data, both random and user-selected. We finish off with a discussion and conclusion in Sections 4.5 and 4.6.

OUTLINE

4.2 Concepts

Concepts [16; 63] allow restrictions to be placed on template arguments. A concept describes a specification for types. It lists the members (functions, associated types, etc.) that are required for some types to model the concept, and the axioms that apply to those members. For example, the following *Monoid* concept requires the existence of an `identity_element` and an operator, and gives an *Identity* axiom (adapted from [16]):

```
concept Monoid<typename T>
    : Semigroup<T> {
    T op(T, T);
    T identity_element();
    axiom Identity(T x) {
        op(x, identity_element()) == x;
        op(identity_element(), x) == x;
    } }
```

A *monoid* is an algebraic class with an operator \oplus and an identity element e , such that $x \oplus e = e \oplus x = x$.

For example, $\langle \text{int}, +, 0 \rangle$ and $\langle \text{int}, *, 1 \rangle$ are monoids.

AXIOMS

Axioms are simple conditional equations (or inequalities), universally quantified over the axiom parameters. Multiple equations may be given inside an axiom – they are combined by logical *and*. More complicated axioms, e.g., with existential quantification cannot, be expressed directly. The sides of the equations are full C++ expressions, allowing use of things like the comma operator and calls to any accessible function.

CONCEPT MAPS

To state that a set of types model a concept, we use a *concept map*. The concept map can specify a mapping between the implementation names (from the class) and the names used in the concept, and can also be used to add extra code necessary to model the concept. Any functions not mentioned explicitly in the concept map is taken from the context – in many cases the body of a concept map is quite short, or empty. In the concept map below, we state that the `FiniteInt` class of bounded integers models the `Monoid` concept, and give an operator that returns the addition of elements and an `identity_element` function that returns the `FiniteInt::zero` identity element.

```
template<int size>
concept_map Monoid<FiniteInt<size> >{
    FiniteInt<size> op(const FiniteInt<size>& a,
                     const FiniteInt<size>& b) {
        return a+b;
    }
    FiniteInt<size> identity_element() {
        return FiniteInt<size>(0);
    } }
```

Without the concept map body, we would have to provide `op` and `identity_element` for `FiniteInts` directly.

```

concept Indexable<typename A, typename I,
                typename E>
: std::EqualityComparable<A,A>,
  std::EqualityComparable<E,E> {
requires SameShape<A, I>;
const E& operator[](const A&, const I&);
E& operator[](A&, const I&);

axiom ArrayEqual(A a, A b, I i) {
  if (a == b)
    a[i] == b[i];
}
}

```

FIGURE 4.1: The concept *Indexable* has indexing operators and an axiom *ArrayEqual* that states that two if Indexables are equal, then their elements are equal. **A** is an indexable type, **I** is the index type, and **E** is the element type. **A** and **I** are required to be of the same shape, i.e., the values of type **I** are the allowable indices for the type **A**. *SameShape* is a trivial concept used to state that the indexable and index type are of compatible shapes/dimensions.

Concepts may also be declared **auto**, in which case an implicit concept map is provided for any set of types that have the relevant functions declared. We feel it is best to avoid axioms in auto concepts – since they may end up specifying behaviour for functions without the programmer being aware of it (though, a few standard cases like having equality, comparison or assignment operators can probably safely be made auto). We will therefore only generate tests for the cases where the programmer has explicitly used a concept map to declare that the implementation models a concept.

AUTO CONCEPTS

4.3 From Axioms to Test Code

There are two steps involved in generating tests from concepts. First, we generate a *test oracle* for each axiom in each concept. The test oracle is a function having the same parameters as an axiom, and returning true or false depending on whether the axiom holds for the given arguments.

For example, consider the *Indexable* concept in Figure 4.1, intended for data structures such as arrays. It has the usual indexing operators you would expect, and an axiom *ArrayEqual*. The axiom can be transformed into callable code by creating a normal C++ template class for the concept (*Indexable_oracle*), and making the axiom a Boolean function within that class – see Figure 4.2.

```

template <typename A,typename I,typename E>
requires Indexable<A, I, E>
struct Indexable_oracle
{
    static bool ArrayEqual(A a, A b, I i)
    {
        if (a == b)
            if (!(a[i] == b[i]))
                return false;
        return true;
    }
};

```

FIGURE 4.2: Oracle code from the ArrayEqual axiom. The oracle returns immediately upon failure, otherwise we continue, as there may be more than one equation in the axiom.

GENERATING TEST CASES

The second step is to generate test cases for each type that models a concept. This is done by finding all the concept maps within the program, and generating code for each of them. The test case will use data iterators (see Section 4.4) to iterate through a set of data values for each argument to the axioms, and then call the test oracle for each combination of data values. Success or failure of the oracle test is then reported to the testing framework.

For example, consider an `ArrayFI` class – an array indexed by finite (bounded) integers. A simplified version of the class is shown in Figure 4.4. It is supplied with two concept maps, relating the implementation to the *SameShape* and *Indexable* concepts. The first stating that any `ArrayFI` of size `size` has the same shape as a `FiniteInt` of size `size` – this is needed to fulfil the *SameShape* requirement of the *Indexable* concept. The second states that `ArrayFI` is *Indexable* with index type `FiniteInt` and element type `E`. Note that the concept maps are templated, working on any integer `size` and element type `E`.

The test case (seen in Figure 4.3) consists of an `Indexable_testCase` class specialised for `ArrayFI<size, E>`, `FiniteInt<size>` and `E`. The class contains a test function, `ArrayEqual`, which iterates over the data generators and calls the generic test oracle derived from the axiom. The two outer loops generate arrays (`*a_0` and `*c_0`), while the inner loop generates indexes (`*d_0`). The test oracle (from Figure 4.2) will check that the array code behaves as expected for an *Indexable* structure. The `HasDataSet` provides a mapping from a type to a data generator for that type (reasonable defaults for this are generated automatically – see Section 4.4).

```

template <int size, typename E>
requires Indexable<ArrayFI<size, E>, FiniteInt<size>, E>
struct Indexable_testCase<ArrayFI<size, E>, FiniteInt<size>, E>
{
    static void ArrayEqual() {
        typedef HasDataSet<ArrayFI<size, E>>::dataset_type dt_0;
        dt_0 b_0 = HasDataSet<ArrayFI<size, E>>::get_dataset();
        for (DataSet<dt_0>::iterator_type a_0 = DataSet<dt_0>:: begin(b_0)
            ; a_0 != DataSet<dt_0>::end(b_0); ++a_0) {
            typedef HasDataSet<ArrayFI<size, E>>::dataset_type dt_1;
            dt_1 d_0 = HasDataSet<ArrayFI<size, E>>::get_dataset();
            for (DataSet<dt_1>::iterator_type c_0 = DataSet<dt_1>:: begin(d_0)
                ; c_0 != DataSet<dt_1>::end(d_0); ++c_0) {
                typedef HasDataSet<FiniteInt<size>>::dataset_type dt_2;
                dt_2 f_0 = HasDataSet<FiniteInt<size>>::get_dataset();
                for (DataSet<dt_2>::iterator_type e_0 = DataSet<dt_2>:: begin(f_0)
                    ; e_0 != DataSet<dt_2>::end(f_0); ++e_0)
                    check(Indexable_oracle<ArrayFI<size, E>,
                        FiniteInt<size>,
                        E::ArrayEqual(*a_0,*c_0, *e_0),
                        "Indexable", "ArrayEqual");
            }
        }
    }
};

```

FIGURE 4.3: Concrete test code generated from a concept map. *HasDataSet* is used to select an appropriate data set for each data type. **check** is a hook for reporting results to a testing framework.

4.3.1 REUSABLE TESTS

A convenient effect of having concepts and their axioms separate from the classes that implement them is that they can be freely reused for testing new types that model the same concepts. If you already have a *Stack* concept with carefully selected axioms, you get the tests for free when you implement a new stack class.

Having libraries of standard concepts for things such as algebraic classes [60] (including *monoid*, *ring*, *group* and others that apply to numeric data types), containers (indexable, searchable, sorted, ...) as well as common type behaviours [62] (*CopyAssignable*, *EqualityComparable*, ...) cuts down on the work needed to implement tests. A well thought-out library is also far less likely to have flawed or too-weak axioms than axioms or tests written by a programmer in the middle of a busy project.

4.3.2 CONCEPT COMBINATIONS

Some combinations of classes can create interesting interactions between concepts. For example, the *FiniteInt* type we used in the implemen-

```

template<int size, typename E>
class ArrayFI {
private:
    E data[size];
public:
    E& operator[](const FiniteInt<size>& i) {
        return data[i];
    }
    bool operator==(const ArrayFI& a){
        for(int i = 0; i<size; ++i)
            if (data[i] != a.data[i])
                return false;
        return true;
    }
    int getSize() const {
        return size;
    }
};

template<int size, typename E>
concept_map SameShape<ArrayFI<size, E>,
                    FiniteInt<size> >{
}

template<int size, typename E>
concept_map Indexable<ArrayFI<size, E>,
                    FiniteInt<size>, E>{
}

```

FIGURE 4.4: The *ArrayFI* class, parameterised with a size and an element type.

tation of *ArrayFI* satisfies the *Monoid* concept from Section 4.2 (as well as several other algebraic concepts that are too lengthy to include in this paper). If we extend our *ArrayFI* with element-wise operations, an instance *ArrayFI*<*FiniteInt*> can ‘inherit’ the *Monoid* concept from the *FiniteInt*. For this to work, we need to provide a concept map

```

template<typename A>
requires DefaultIndexable<A>,
        Monoid<DefaultIndexable<A>
            ::element_type>,
        std::CopyConstructible<A>
concept_map Monoid<A> {
    A op(const A& a, const A& b) {
        return Shape<A>::map(

```

```

    Monoid<DefaultIndexable<A>
        ::element_type>::op,
    a, b);
}
A identity_element() {
    return Shape<A>::build(
        Monoid<DefaultIndexable<A>
            ::element_type>
            ::identity_element());
}
}

```

The *Indexable* concept may be used in several different ways on the same array type with different index and element types. As we want the compilation process to automatically deduce which way to index our data structure, we need to provide a default pair of index type and element type to each *Indexable* through the following concept *DefaultIndexable*:

```

concept DefaultIndexable<typename A> {
    typename index_type;
    typename element_type;
    requires Indexable<A, index_type,
        element_type>;
}

```

Then, for example, *ArrayFI* would only need a small concept map like the one below to inherit all the axioms.

```

template <int size, typename E>
concept_map DefaultIndexable<
    ArrayFI<size, E>> {
    typedef FiniteInt<size>
        index_type;
    typedef E element_type;
}

```

Based on the above concepts and the concept maps, an *ArrayFI*<size, *FiniteInt*> would have test code for the *ArrayEqual* axiom (instantiated from the template code in Figure 4.3), and for the *Monoid::Identity* axiom. And, as *ArrayFI*<size, *FiniteInt*> is itself a *Monoid*, we can use it as the element type for a new *Indexable Monoid* *ArrayFI*<size1, *ArrayFI*<size2, *FiniteInt*>, and so on. Such constructions are important in some problem domains [80] and allow us to do some simple integration testing with axioms as well.

4.3.3 TEST DRIVERS / SUITES

So far we have generated test oracles from axioms, and test cases that generate test data and call the oracles. To actually perform the testing, we need to call the test cases as well. There are three ways to do this: we may call the code manually, we may generate code that calls all known test functions, or we may use a combined approach.

By default, our tool will generate a `main` function filled with calls to all non-template test functions. Guessing at sensible template parameters is difficult in the case of unconstrained template parameters and when there is a large or infinite number of choices (as in the case of the nested arrays above). We therefore rely on the user to choose which templated tests to run, as explained in Section 4.4.

If we want fully automatic test program generation, we could analyse existing application code and find suitable template instantiation arguments there. Or, in cases where template parameters are constrained by concepts, we could generate calls with all classes that fulfil the concept constraint (with a cut-off in place to avoid infinite nesting). This would allow quite exhaustive exercising of code, including combinations that a programmer would likely never think of.

Even if the tool does not automatically generate full test suites, it could help the programmer by generating code templates. With integration into an IDE – such as Eclipse – test suite building can be done in a guided manner.

4.3.4 AXIOMS FOR OBJECT-ORIENTED CODE

The axiom examples we have used so far have mostly been in a functional style where results are returned and there are no side-effects on arguments. Realistic C++ code will often be written in a more object-oriented style.

Object-oriented style favours side-effects on the first argument. To capture side-effects in concepts, some functions will have to have reference type arguments. If the first argument is a reference, then the function can be defined in the concept map as a method defined in a class. If the first argument is not a reference, or it is a `const` reference, then the function is defined as `const` method. Non-`const` methods – methods that may change the object – is the norm in C++ programming, which means that function declarations in concepts will often have reference parameters. The side effect of those functions have to be captured somehow by the axiom. The comma operator can be useful for testing side effects. An example is the following axiom:

```
axiom CopyPreservation(T x, U y) {  
    (x = y, x) == y;  
}
```

OO-STYLE
CONCEPTS

This axiom states that after assigning y to x , the value of x should be equal to y . The comma operator has the effect of first assigning y to x , and then yielding the value of x .

Figure 4.5 shows the traditional bounded stack example also used for DAISTS [50], Daistish [86] and JAX [142]. The *BoundedStack* concept is written in a functional syntactic style, but the reference of the first parameter on `pop` and `push` captures the object-oriented style. These two stack operations modify the current object, rather than return a new modified stack.

BOUNDED STACK
EXAMPLE

In our stack axioms, we have intentionally not specified what happens when we attempt to push onto a full stack or pop an empty stack. In a traditional bounded stack, pushing onto a full stack has no effect. By leaving this behaviour undefined, we leave the door open for alternative solutions (handled by non-axiom test cases, for example).

However, if we wish to specify that an exception should be thrown when attempting to push onto a full stack, we would need a small helper function to do the push, catch the exception and return true or false – see Figure 4.6. With some small changes [7] to the proposed C++ syntax, we could avoid the use of the helper function.

This state-modifying style of axioms has some consequences for test code generation, since the test oracles will modify the test data. For this reason, the test oracles avoid reference arguments, ensuring that the data is copied into the oracle function. This may not be sufficient for all data structures, though. We are still unsure of the best way to handle this, as we would like to keep data generation as simple and efficient as possible. Fortunately the `const`/`non-const` status of parameters will give a clue as to when this may be a problem – for example, the `equals` operator is safe, since *EqualityComparable* specifies that it has `const` arguments. We could then try to force copying of test data which is passed as `non-const` arguments in axioms. Alternatively, one could simply expect the test driver to generate fresh data for every oracle invocation.

DATA
MODIFICATION IN
ORACLES

4.4 Generating Test Data

Creating a test oracle from the concept axioms and a concept map is straightforward, as described in the previous section. Such a test oracle will normally have parameter variables (free variables) that need to be instantiated by suitable values in order to actually perform testing.

We have three cases to consider when we want to provide data for a free variable:

- The parameter has a known, primitive C++ type.
- The parameter has a known, user-defined type. In this exposition we will not investigate the issues arising if the known type can be subclassed.

```
concept BoundedStack<typename S> {
  requires std::DefaultConstructible<S>,
    std::EqualityComparable<S>;
  std::EqualityComparable E;
  E top(S);
  E pop(S&);
  void push(S&, E);
  bool full(S);
  bool empty(S);

  axiom PushTop(S s, E e) {
    if(!full(s))
      (push(s,e), top(s)) == e; }
  axiom PushPop(S s, E e) {
    if(!full(s))
      (push(s,e), pop(s)) == e;
    if(full(s))
      (push(s,e), pop(s), s) == s; }
  axiom Empty1() {
    empty(S()) == true; }
  axiom Empty2(S s, E e) {
    if(!full(s))
      (push(s, e), empty(s)) == false; }
  axiom Equal1() {
    S() == S(); }
  axiom Equal2(S s, E e) {
    if(!full(s))
      (push(s, e), s) != S(); }
  axiom Equal3(S s1, S s2) {
    if(!empty(s1) && (s1 == s2))
      s1.top() == s2.top();
    if(!empty(s1) && (s1 == s2))
      (pop(s1), s1) == (pop(s2), s2);
  }
}
```

FIGURE 4.5: An example of a bounded stack concept capturing side effect of OO programming style, with a selection of axioms. The comma operator (,) is used to first evaluate a call for the side effect (left side), then choosing the value we're interested in (right side). `S()` constructs a new stack.

```

...
bool PushFull_help(S s, E e) {
    try { push(s, e); return false; }
    catch(...) { return true; }
}
axiom PushFull(S s, E e) {
    if(full(s))
        PushFull_help(s, e) == true;
}

```

FIGURE 4.6: Checking for exceptions. The function **PushFull_help** will return true if pushing onto a full stack throws an exception, and false otherwise.

- The parameter type is a template argument to the test oracle. In this case, the template may have additional constraints, e.g., that a parameter models a given concept, see Section 4.3.3.

For the last case we will rely on concept maps to identify candidate types. Though some authors [34] claim that fixing the test data set for one such candidate will be sufficient, we believe test data sets should exercise several of these in order to check that the stated requirements are sufficient constraints on the template arguments.

We provide test data through associating test data generators with each class. For the primitive types, we can use a random generator library, to obtain an arbitrarily large selection of test data. User defined classes should provide a test data generation interface, allowing our testing tool to feed generated test data to the test oracles. A test data generator for a class template may call upon the data generators for the argument classes.

For a known type, whether primitive or user-defined, we see several strategies for providing test data.

1. User selected data sets.
2. Randomly chosen generator terms.
3. Randomly chosen data structure values.

The first is the classical approach to testing and the one (implicitly) favoured by test driven development. Here the tester decides, e.g., that integer values -1 , 0 , 1 and 3 are of prime importance, or that stacks `SC`, `SC.push(1)` and `SC.push(1).pop()` are specifically important. Such selected data sets are useful for regression testing, where specific data sets that have exposed problems in the past are rechecked with each revision of the code. The data sets can also be targeted for other purposes, e.g., path coverage of the implemented algorithms.

The second is favoured by Claessen & Hughes in their QuickCheck

USER SELECTED
DATA

RANDOM TERMS

system [34] and by Prasetya et al. for their Java-based testing system [123]. The idea is to let random expressions or sequences of (public) methods compute data values of the appropriate type. By choosing a suitable enumeration of terms this will always be possible and give good data coverage. For example, testing integer-like types (with axioms such as associativity, commutativity, distributivity) we may use expressions $0, 0 + 1, (-1 + 0) * 2, \dots$ and for stacks sequences like `S s(); s.push(1+-2); s.push(3|4); s.pop();` may be used.

RANDOM
STRUCTURES

The third approach requires the tester to have access to the data representation (data field attributes) of a type. For primitive types such as floats, this means setting the bit patterns of a floating point number directly. For a user-defined class this implies that each data field is given a random value of the appropriate type, subject to the constraints of the implementation. For instance, having a rational number class where we represent rational numbers as pairs of integers (a nominator and a denominator, the denominator different from zero), we may choose random pairs of integers for the attributes, discarding any pair where the denominator part would be equal to 0. Such direct setting of attribute values may give access to a larger range of test values than allowed by method 2, and is needed if all or some of the data fields are publicly available. Setting data attributes directly requires a filtering mechanism that identifies all bad data combinations, i.e., a complete data (class) invariant. If the data invariant has narrow requirements on the data, e.g., that the stack has a length field required to be equal to the length of the linked list representing the data on the stack, independently generating random integers and random linked lists will probably turn up too few good combinations for this technique to be worthwhile.

DATA HARVESTING

Harvesting the data produced by an application program is related to the second method, in that it provides values computed by the public methods of the classes, though harvesting ensures a statistical distribution of data much closer to those that appear in practise. One way of harvesting application data would be to insert the test oracles directly as assertions into an application, using the available data values as parameter arguments to the oracle. This would only be safe for stateless data types or copy-assignable data types, otherwise we risk that the oracle itself modifies the state of the application.

RANDOM DATA

Currently, random test data generation seems to be favoured by the literature [65; 72; 73]. Studies of testing efficiency seem to indicate that random testing outperforms most other test set designs. For any fixed data set size, a carefully chosen data set will normally be better than a random data set, but a slightly larger, often cited as 20% larger, random data set is often just as good [73]. Random data generation offers an easy route to expand the data set to any reasonable size.

Similarly to the data invariant, a conditional axiom itself represents a filtering mechanism. A conditional axiom contains an if-statement, and

only those data combinations that satisfy the condition will really be tried. Assume that we want to test the transitivity axiom for equality on a user-defined rational number type.

```
if (a == b && b == c) a == c;
```

With the representation of rationals as pairs of integers sketched above, we may compute the equality of $\frac{n_1}{d_1}$ and $\frac{n_2}{d_2}$ by the Boolean expression `n1*d2 == n2*d1` involving integer equality. Choosing arbitrary combinations of integers for nominator and denominators, chances are rather slim we ever will get to the truth part in the transitivity axiom. As in QuickCheck we will provide a warning in such cases, encouraging the user to provide data sets where a significant amount of data reaches the body of the condition. On the other hand, only choosing obviously equal nominator and denominator pairs, skews the data set towards trivially satisfying an axiom, and not providing good tests for the algorithms in general.

Claessen & Hughes also point out that different uses of a data type may benefit from different data distributions. The observation being that the data set of integers which best checks that the integers form a monoid, may not be the ideal data set for array sizes when generating finite array test sets. We see this observation on targeted generation of data sets as very important, and expect the locality we have by associating the data generators with each class will provide this flexibility.

DATA
DISTRIBUTION

Once the test oracles and the test data machinery are in place, it is easy to run the tests by iterating through the corresponding data set for each of the free variables of each test oracle. However, this easily leads to a combinatorial explosion in the testing size. A test set of 100 elements is quite reasonable, but when we test axioms with several free variables this may become a problem. Take the transitivity axiom. It has three free variables, hence we will test it for one million elements altogether. This may be OK for integers, but what about one million finite arrays? We can deal with this by providing the data generators with a parameter related to the number of arguments in an axiom. Our test generator tool can then fill in this parameter automatically based on the number of free variables in an axiom.

4.4.1 ASSOCIATING A DATA SET WITH A TYPE

Any type that is part of a universal quantification on an axiom needs to have a test data set associated with it. Our tool expects the user to configure the data generation with a concept map, where relevant types model a concept *HasDataSet*.

For any type, the user must then provide a concept map for *HasDataSet* with a function `get_dataset`, which provides the iterators needed

to obtain values of the type. The `for`-loops in Figure 4.3 show how `get_dataset` is used to obtain test data. The exact mechanics of iterators and data source (predefined values, random or generated by some other scheme) is up to the user, but the library provided with the testing tool provides a general implementation which can be used as a basis for generating predefined and random values.

4.5 Discussion

AXIOM
DEBUGGING

There is no reason to believe that writing axioms (or test cases) is any less error-prone than programming in general. Failure of a test can just as well indicate a problem with the axioms or the equals operator as a problem in the implementation. It is important to be aware of this while programming, so that bug-hunting is not exclusively focused on implementation code. The same issue arises with hand-written tests, though, so this is not specific to axiom-based testing. Also, since axioms have a different form than implementation code (equation versus algorithm), it is unlikely that a bug in an axiom and in the implementation will ‘cover’ for each other so that neither are detected. It is still possible, though; having several axioms covering related behaviour will make this less likely.

AXIOM LIBRARIES

Building libraries of well-tested concepts with axioms will increase confidence in the completeness and correctness of the axioms, and reduces the training needed to make effective use of axioms. Not everyone can be expected to know all the laws governing integer arithmetic – but using an existing axiom library and simply stating that “my class should behave like an integer” is easy.

4.5.1 EQUALITY TESTING

Axiom-based testing (at least with equations) relies on a correct implementation of equality. In many cases, problems with equality will be uncovered in testing, but it is possible to write an implementation of equality that tries to hide most errors – for example, by simply returning true for all arguments (which may be detected when testing inequalities, unless a `!=` operator has been provided with the same problem).

CONGRUENCES

We expect the equals operator to be a *congruence relation* – an equality relation that is preserved by all functions. This means that it has the usual reflexivity, symmetry and transitivity expected of an equivalence relation, with the additional requirement that all equal objects are treated the same by all functions, i.e. $f(a) = f(b)$ if $a = b$ for all a, b , and f . A straightforward bitwise comparison of two objects will often lack this property. In some cases, such as with floating-point numbers, a usable equals operator will not be truly transitive (due to a small amount of ‘fuzz’

Informally speaking, since C++ functions may have side-effects or rely on global data.

when comparing, to cover up round-off errors) – this has little impact on our use, however.

The *EqualityComparable* concept in the standard library provides axioms for the equivalence relation of the equality operator and also ensures that inequality operator is the negation of the equality.

It may not always be desirable that the equality operator is a congruence. In the cases we want this property, the relevant axioms should be tool generated, since they will involve every method belonging to the class being tested.

A ‘bad’ equality operator, returning arbitrary results, will almost certainly be caught during testing since it is basically tested by every axiom in the system relevant for the particular type. Trivial cases like equality always returning true is easily caught by testing based on equality axioms, while more subtle bugs may only show up in general testing, and will be more difficult to trace to the equality operator.

BAD EQUALITY

Note that having an equality operator is not strictly necessary. Any type that is *EqualityComparable* is observable in our test oracles, i.e., can be tested on equality. But any type that can be projected on an observable type becomes observable. A projection or context is a term with placeholder for a variable [164]. This kind of test oracle generation has not been developed in our tool yet and we for the moment require tested types to be *EqualityComparable*.

NO EQUALITY

Note, though, that even if equality is not generally available for a type, it can be provided in a concept map, thus making it available in any template context where the type is constrained to *EqualityComparable*.

4.5.2 ALGEBRAIC AXIOMS AND IMPERATIVE CODE

As discussed in Section 4.3.4, a particular problem occurs for code written in an object-oriented or imperative style, relying on side-effects on arguments. Although this is a poor fit for algebraic-style axioms, side-effects can be captured by using the comma operator. Another issue is that the concept itself must specify whether side-effects occur or not, through the use of non-`const` reference arguments. If an implementation has chosen a different approach, a mapping between the two styles may be given in a concept map, possibly at the expense of an extra temporary. A solution to this problem is provided by *mutification* [8], which automatically maps between algebraic and imperative/OO-style code.

In the `ASTOOT` [44] system, algebraic specification of object-oriented programs is done in the `IOBAS` formalism which supports OO syntax. Each axiom relates object states or values that are computed through a sequence of method calls; optionally, observer functions may be called at the end each sequence to inspect the objects. The system is purely

ASTOOT

algebraic, allowing no side-effects in operations, except for modifying object state in methods – though a relaxation of this is described by Doong and Frankl [42; 43]. `ASTOOT` will automatically generate test drivers from class interfaces, and also generates test cases from a `LOBAS` algebraic specification. Automated tests can be augmented by manual test generation.

As the C++ axiom proposal allows arbitrary expressions, the `ASTOOT` / `LOBAS`-style can easily be used with C++ axioms; though, without disciplined use within same restrictions, there is a danger that side-effects will interfere with testing, as discussed in Section 4.3.4.

The ideas of `ASTOOT` have been developed further by Chen et al., and applied to axiom-based testing of object-oriented code at the level of class clusters and components [32; 33].

4.5.3 AXIOM SELECTION AND ALGEBRAIC SPECIFICATION

Early work by Liskov and Zilles [102] discuss techniques for formal specification of abstract data types. They point out that specification should be done by relating the various operations of the abstract data type, rather than directly specifying the input / output of each operation. The latter leads to over-specification, providing many unnecessary details and hiding the essential properties of the data type – for example, by enforcing some order on the elements of an unordered set. Specifying operations in terms of each other avoids bias towards particular representations or implementations. In traditional unit testing, there is always a temptation to over-specify by focusing on testing the input and output of every operation, though a disciplined developer can still avoid over-specification.

In the context of C++ concepts, the concept is separate from the implementation and should avoid putting undue constraints on how the concept may be implemented. Hence, axiom expressions should be limited to using the operations provided in the concept (together with C++'s primitive operations – on Booleans, for example – these can be considered implicitly defined in every concept).

Among the techniques discussed by Liskov and Zilles, algebraic specification [54; 67; 69; 102] shows the most promise in terms of usability and in avoiding over-specification. An algebraic specification consists of a syntax description and a set of axioms; this maps to the C++ idea of concepts, which provide axioms together with a syntax description in the form of associated types and operations.

To ensure that the behaviour of the abstract data type is fully specified (or *sufficiently complete*) one can divide the operations into *constructors* (the set of which can generate all possible values), *transformers* (which can be defined in terms of constructors) and *observers* (which yield values of

AVOIDING OVER-
SPECIFICATION

SUFFICIENTLY
COMPLETE
SPECIFICATIONS

another type). Left-hand sides for the axioms of a sufficiently complete specification can then be constructed from the combination of each constructor with every non-constructor. Further guidelines for constructing specifications are discussed by Guttag [66] and Antoy [4].

Many of the existing axiom based testing approaches, such as JAX and Daistish, rely on sufficiently complete specifications, provided by complete axiomatisations or initial specifications. This gives extra properties on which to base tools. For example, the approach of Antoy and Hamlet [5] uses initial specifications, which are evaluated alongside the implementation, as a *direct implementation* [69] of the specification. All objects in the system contain both a concrete value and an abstract value (in the form of a normalised term over constructors in the specification), and the equations from the specification can be evaluated by treating them as rewrite rules on the abstract value terms. A *representation mapping* translates between the abstractions of the specification and the concrete data structures of the implementation. Self-checking functions are made by doing an additional abstract evaluation according to the specification, and – using the representation mapping – comparing the result of normal execution and evaluating the specification. In this way, a whole program can be described and evaluated in two distinct ways – using program code and algebraic specification – providing good protection against programming errors. This is also the disadvantage of the approach – the implementation work must basically be done twice. The overhead of the abstract evaluation and comparison can probably be lowered by running the testing code in a separate thread on a multicore system.

TESTING WITH
INITIAL
SPECIFICATIONS

Axioms written in C++ concepts will normally be loose and incomplete, making many of these testing techniques void. The approach described in this paper will work equally well with an incomplete specification (though, it will of course not be able to test unspecified behaviour). Our experience with developing and testing Sophus [75; 80] shows that such axioms are very useful.

4.5.4 EXPERIENCES WITH AXIOM-BASED TESTING

There is currently no large body of code around that uses C++ axioms, since the standard proposal is not yet finished and compiler support is still not mature. A version of the Matrix Template Library [136] (MTL) with concepts and axioms is in development and we plan to apply our tool to it as soon as it is ready.

The prototype ConceptGCC compiler works well in some cases, but is not complete yet.

We have experience with axiom-based testing from the Sophus numerical software library [75]. This predates C++ axioms, so the tests were written by hand, based on a formal algebraic specification. In our experience, the tests have been useful in uncovering flaws in both the implementation and the specification, though we expect to be able to do more rigorous testing with tool support.

JAxT The JAxT tool [77; 92] provides axiom-based testing for Java, by generating tests from algebraic specifications. The axioms are written as static methods and are related to implementation classes through inheritance and interfaces. For any class with axioms, the JAxT tool will generate code that calls the associated axioms. A team of undergraduate students successfully wrote JAxT axioms for parts of the Java collection classes, discovering some weaknesses in the interface specifications in the process [109].

JAX The JAX [142] method of combining axioms with the JUnit [15; 105] testing framework has provided some valuable insight into the usefulness of axiom-based testing. The JAX developers conducted several informal trials where programmers wrote code and tests using basic JUnit test cases and axiom testing, and found that the axioms uncovered a number of errors that the basic test cases did not detect.

DAISTS & DAISTISH Initial experiences with DAISTS [50] were positive and indicated that it helped users to develop effective tests, avoid weak tests, and the use of insufficient test data. With Daistish [86], the authors did trials similar to those done with JAX, with programming teams reporting that their axioms found errors in code that had already been subjected to traditional unit testing. Testing also uncovered numerous incomplete and erroneous axioms – the Daistish team note that this is to be expected since the programmers were students learning algebraic specification. This is probably a factor, but some axiom errors can be expected even from trained programmers.

Further experiences and case studies are summarised by Gaudel and Le Gall [52].

4.5.5 TOOL IMPLEMENTATION

CONCEPT TRANSFORMERS Our implementation is based on the Transformers C++ parsing toolkit [19; 148] and the Stratego program transformation language [25]. We have extended Transformers with the new syntax for concepts and axioms, and written a tool, **extract-tests**, that reads C++ with concepts and generates testing code from the concepts and concept maps in the code [10].

As part of our concepts extension to Transformers, we also have an embedding of the Concept C++ grammar into Stratego, so that Stratego transformation rules can be written using concrete C++ syntax. This makes it easy to modify the code templates for the generated code, for instance, changing the test oracles to report success / failure to a testing framework. As an example we use a backend for test oracles that instead of returning a Boolean, throws an exception for the CUTE library [138] with the line number of the axiom, so we get test results reported within the Eclipse IDE.

Together with the tool, we have a utility library with basic data generation support, and hooks into a testing framework. This library

provides a concept for test data generators. Each type to be tested is expected to have an associated data generator specified through a concept map. This allows the user to specify which generator to use, to create any new kind of generator, and finally to combine streams of generated data.

Since compiling Concept C++ is usually slow, and since generating code directly for pure C++ is complex, the tool is delivered with a Concept C++ to C++ tool translation. Though this tool is not complete, it can still give a sufficient translation to be able to work on a big part of Concept C++ with a standard pure C++ compiler.

4.5.6 FUTURE WORK

We have identified several areas for improvement throughout this paper. Areas of particular research interest are:

- Perform proper trials to gauge the effectiveness of axiom-based testing and its impact on development.
- Testing of multi-threaded applications is notoriously difficult [132], and it would be interesting to see if axiom-based testing could be applied here.
- As discussed in Section 4.4, there are many open issues with data generation. These will likely only be resolved once we apply the method to realistic-sized projects (like `MTL`).

There are also much engineering work to be done (in no particular order):

- A library of common concepts with axioms should be written. There has been some work on this already [60]. Such concepts should eventually make their way into the C++ standard, for consistency and interoperability.
- Our tool is still experimental, and would need many improvements to be ready for production use. In particular, the underlying framework needs to be developed to handle the kind full-featured C++ code found in mainstream application.
- The tool should be extended with the ability to generate meta-axioms for testing, e.g., congruence axioms for the equality operator or axioms checking the preservation of class invariants in all methods.
- Generate oracles that can test equality on observable types that have no direct equality comparison operator.

4.6 Conclusion

The use of axioms and “informal formal methods” has seen a surge in popularity recently. We have presented a method for doing axiom-based

testing in the context of proposed concept and axiom features for C++, along with a tool to make generation of such tests automatic.

Both the C++ standard, and programming tools such as compilers are still in development and should be considered ‘unstable’. However, our initial experiments with simple test cases show promise, and experiences with axiom-based testing from other languages (both our own and others’) encourage us to push forward with tool development and larger-scale experiments.

Acknowledgements

This research was funded in part by the Research Council of Norway. We thank the anonymous reviewers and May-Lill Sande for providing us with useful feedback, corrections and references.

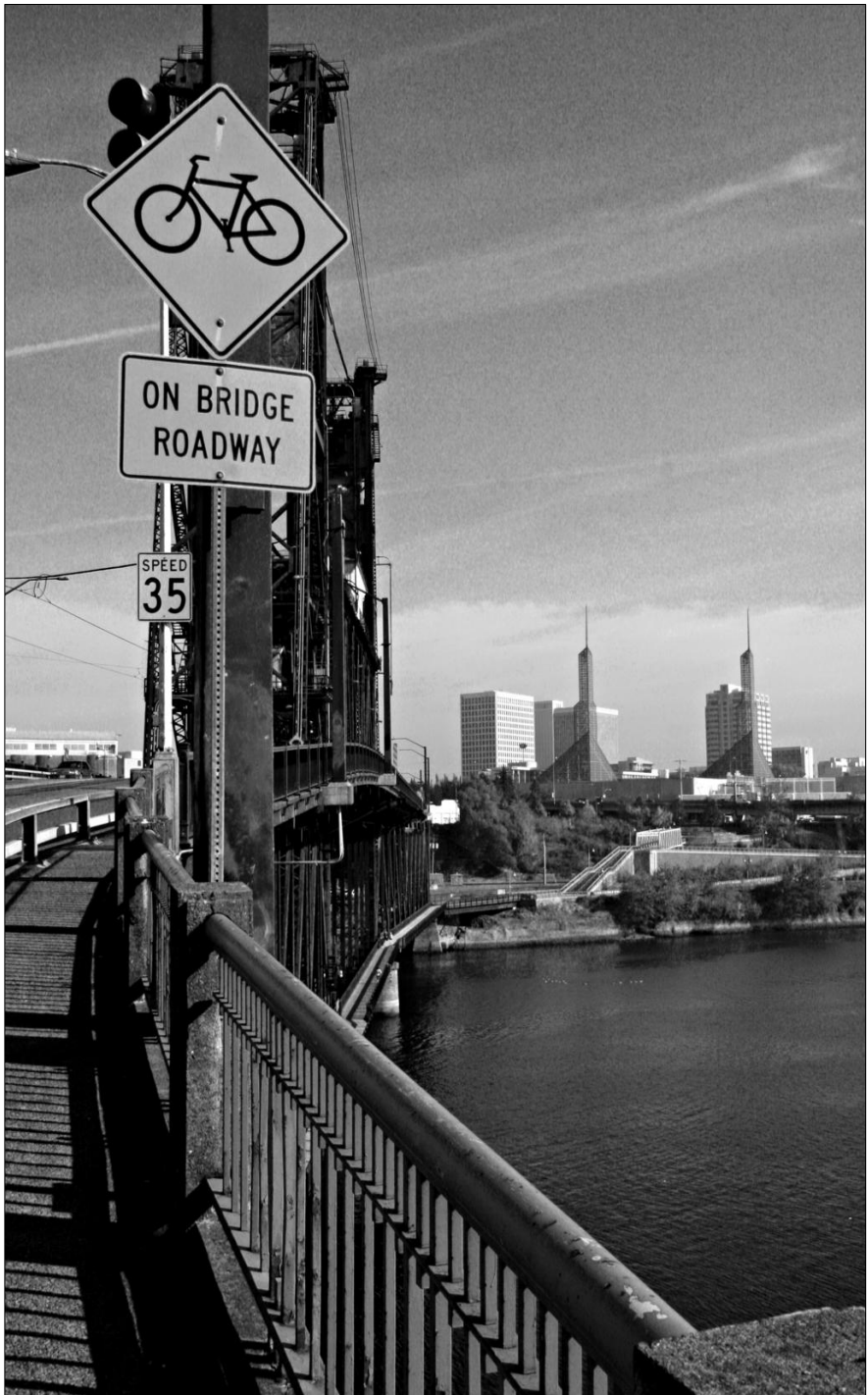
Handling Failure and Exceptions

Proper handling of failures and unexpected situations that arise is important in creating reliable software. *Alerts*, presented in this chapter, provides a unified approach to reporting and handling failure. In particular, alerts can enforce checking of return codes, and provide local or global handlers for different error situations.

Alerts can also associate errors with particular argument values, thus providing precondition checks integrated with the alert handling system. Improper checking of arguments is a significant source of bugs and security problems, so this is an important feature.

From a specification point of view, operations that can end in failure are *partial*. A convenient way of handling partiality in specification is through *guarding* [78], where preconditions – *guards* – are added to partial operations. By assuming that the guards always hold, operations can be treated as total, greatly simplifying the specification.

The paper in the chapter was presented at GPCE'06. A simple version of alerts for the minimal TIL language [155] was also presented at the First Domain-Specific Aspect Languages Workshop, as an example of how a domain-specific aspect language can be implemented using a syntax extension together with a transformation library [9].



Steel Bridge across the Willamette River, with Oregon Convention Center in the background.
Portland, OR (GPCE'06)

Stayin' Alert

Moulding Failure and Exceptions to Your Needs

ANYA HELENE BAGGE

VALENTIN DAVID

MAGNE HAVERAAEN

KARL TRYGVE KALLEBERG

Department of Informatics, University of Bergen, Norway

ABSTRACT

Dealing with failure and exceptional situations is an important but tricky part of programming, especially when reusing existing components. Traditionally, it has been up to the designer of a library to decide whether to use a language's exception mechanism, return values, or other ways to indicate exceptional circumstances. The library user has been bound by this choice, even though it may be inconvenient for a particular use. Furthermore, normal program code is often cluttered with code dealing with exceptional circumstances.

This paper introduces an alert concept which gives a uniform interface to all failure mechanisms. It separates the handling of an exceptional situation from reporting it, and allows for retro-fitting this for existing libraries. For instance, we may easily declare the error codes of the `posix C` library for file handling, and then use the library functions as if `C` had been extended with an exception mechanism for these functions – a *moulding* of failure handling to the user's needs, independently of the library designer's choices.

This is a reprint of: Bagge, A. H., David, V., Haveraaen, M., and Kalleberg, K. T. 2006. Stayin' Alert: Moulding Failure and Exceptions to Your Needs. In Proceedings of the 5th international Conference on Generative Programming and Component Engineering (Portland, Oregon, USA, October 22 – 26, 2006). GPCE '06. ACM, New York, NY, 265-274.

DOI=<http://doi.acm.org/10.1145/1173706.1173747>

© 2006 Association for Computing Machinery, Inc. Reprinted by permission.

5.1 Introduction

Wherever there is software, there are errors and , and these must always be considered when writing and maintaining programs. Programming failure handling code is a tedious and error-prone task. Dealing with every possible exceptional situation leads to cluttered and hard to read code; not dealing with errors can have costly or perhaps even fatal consequences.

Some have argued that error handling should be avoided altogether. Instead, programs should be written so that errors never occur. Algorithms should be formulated so as to remove the exceptional corner cases, as this improves both the readability and maintainability of the code. This view is fundamental to the design of SPARK Ada [12], where the Ada exception mechanism has been removed in an attempt at making validation and verification easier. This ideal advocated by such a “keep errors out” approach is certainly desirable. It is generally preferable to write algorithms with as few corner cases as possible.

In many cases, however, removing the errors altogether is simply not feasible [125]. Most modern applications run in multi-user, multi-process environments where they share resources such as storage and network with other applications. In these situations, operations on files, network connections and similar operating system resources can always fail, due to interaction with other programs on the running system or external devices.

Errors and exceptional situations need not always be caused by external factors, however. Even in situations where resource requirements are known in advance and guaranteed to be available, exceptional situations may occur, as none of the mainstream languages support resource-aware type systems [149].

As an example, consider the implementation of a simple abstract data type, say, a hash table, that is intended for other developers to reuse. In the case where the user (the *caller*) tries to look up a value for a non-existent key, an exceptional situation has occurred. Some possibilities for dealing with such a situation are:

- *Undefinedness*: this situation is outside the specified behaviour of the hash table. The caller cannot have any expectations as to what will happen.
- *Termination*: the program will terminate when this situation occurs. It is up to the caller to ensure that this does not happen.
- *Alert the caller*: report that an exceptional condition occurred. Given proper language mechanisms, alerts allow the user of the hash table to implement alert handling, such as logging, recovering from or ignoring the failure.

Undefinedness requires no language support, and termination can usually be implemented by a call to an `exit` function. In languages supporting Design by Contract (discussed in Section 5.2.1), termination is automatic if a function fails to satisfy declared conditions either before or after invocation.

Several different alert reporting mechanisms are in common use. Goodenough [57] first introduced the exception handling mechanism that is now found in most modern languages, and is currently the recommended way of handling exceptional situations. Returning a special error value, often `-1` or `null`, or setting a global variable is another common technique, often used in older code and languages. Other than exceptions, most reporting mechanisms are ad hoc, in that there is no way to declare which mechanism is used. Conventions do exist – for example, most `posix` [119] functions report errors by returning `-1` – but they are not declared explicitly in the code, making it difficult to automate alert handling. We therefore propose that each function declares its alert reporting. For example, a hash table lookup function may declare that it returns `null` if the key was not found:

```
val lookup(tbl t, key k)
  alert NotFound post(value == null);
```

Handling alerts is no easy task either. Different reporting mechanisms have different default handlers – exceptions, for example, typically terminate the program if they are not caught, whereas return values are ignored if they are not explicitly tested. Furthermore, different alert reports are checked in radically different ways; exceptions are received by a `try/catch` clause somewhere in the call hierarchy, return values must be checked after each return – often tedious and inconvenient. Changing the report mechanism means changing all handlers. We propose a way of declaring handlers which is independent of the alert reporting mechanism, and which can apply at various granularities, from a single expression to the whole program. For example, the following handler ensures that `NotFound` errors from the `lookup` function is handled by substituting the string `"(unknown)"`:

```
on NotFound in lookup() use "(unknown)";
```

Our contribution in this paper is a detailed discussion of failure handling mechanisms and the proposal of a language construct for alerts: Alert reporting may be declared for precondition and postcondition violations, exceptions, error flags and return codes, simplifying the task of staying alert for run-time problems. Alert handlers can be defined independently of the reporting mechanism, allowing a library implementer to alert its user in a way convenient for the library, and a library user to handle the alert in a convenient way at the call site. Alert handlers can be declared at a per-function and per-call-site basis, but it is also

The word ‘exception’ was coined as a way of emphasising that exceptions are not just for handling errors, but can be used for any kind of exceptional circumstance. However, it is easy to confuse the concept of handling exceptional circumstances and the exception handling language constructs found in many languages. We have therefore elected to use the word ‘alert’ for any reported exceptional situation, independent of the alert reporting mechanism and the alert handler, which receives the alert report and deals with it appropriately. The word ‘exception’ on its own will refer to the language construct.

possible to declare policies common to a group of functions, such as a class or a library. In this way we can relatively easily retro-fit alert declarations for legacy code, e.g., the `posix` C library, easing the burden of checking in all kinds of strange ways for relevant I/O errors. Hence we approach *mouldable programming*, a way of moulding programming to our needs, and not being forced to program in strange ways due to arbitrary choices from language and library designers, or from perceived expectations from a user community.

This paper is organised as follows. In Section 5.2, we elaborate on the problem of handling failures and exceptional situations. In Section 5.3, we discuss separation of concerns, and granularity. In Section 5.4 we introduce our alert language extension, and continue by discussing its implementation in Section 5.5. In Section 5.6 we discuss related work, leading up to a concluding discussion of our language extension in Section 5.7.

5.2 Problem

The problem we are facing, is implementing an *alert protocol* between callers and callees that can transmit status information from the callee about the validity of its computed result back to the caller. Goode-nough [57] points out that this is a way to extend an operation's domain (input space) or range (output space). The caller will declare an *alert handler* for the types of alerts it wants to handle, and the callee may *report an alert* during its computation, thus becoming the (*alert*) *reporter*.

5.2.1 ALERT REPORTING MECHANISMS

Current programming paradigms and languages provide a number of ways for dealing with failure, dating back to the earliest days of programming. Hill [81] discusses possible mechanisms, anno 1969, which includes specific return values, use of `gotos` to parametrised labels, call-backs, global error flags, and passing pointers to variables which will receive an error status.

RETURN VALUES Designating at least one value in the domain of the return type as an error marker is perhaps the most prevalent form of alerting. This technique is frequently found in operating system APIs, such as `posix` and `Win32`, in many language standard libraries, and in many frameworks. Functions returning objects often use `null` as such a marker, functions returning numeric values for file handles or indexes often use `-1`. If the return type only allows for one error marker, an additional mechanism, such as a global flag, is needed to distinguish between different kinds of errors.

The IEEE floating-point arithmetic standard [56] allows a wide range of error return values. Some of these automatically propagate through an expression, like NaN – “Not a Number”. NaN occurs as a result of $0/0$, $\sqrt{-1}$, $\log(-1)$, etc. A more interesting error value is $+\infty$ or $-\infty$, which is the result of e.g. M/m where a very large number M is divided by a very small number m , resulting in numerical overflow – a number too large to be represented as a floating point number. Infinities propagate through addition, subtraction and multiplication, but disappear after division. The expression $a + 4/(M/m)$ yields a , as 4 divided by infinity yields 0.

SPECIAL ERROR
RETURNS

If the return value for failure can also be a valid return value, for example if division by zero returns zero, we are faced with the so-called semipredicate problem: it is not possible to know if the return value signifies a failure or a valid value.

A property of the return value mechanism is that it will only propagate the alert one level, to its immediate caller. Also, it requires no alert handler setup or teardown, and thus has no overhead.

GLOBAL ERROR FLAGS Many older APIs, such as `posix` and `Win32` use global error flags, often in conjunction with special return values, to elaborate on a failed function. In `Win32`, the function `GetLastError` is used to retrieve the failure code of the previously executed system call. In `posix`, the global `errno` variable serves an identical purpose.

The use of a global error flag variable is not thread safe. Unless special consideration is taken, multiple threads in the same process will share the same error flag variable, making it impossible to know which of the previous threads’ system calls a given error belongs to. This is alleviated by having global error function, like `GetLastError`, instead.

LONG DISTANCE JUMPS In C, the functions `setjmp` and `longjmp` are used to transfer control directly from one stack frame to one which is arbitrarily higher up. This report mechanism is often used to propagate errors many levels, but can only send an integer value. This is a low-level C/Unix-specific technique, which is also found as `RtlUnwind` in `Win32`. Both alternatives rely on low-level machine-specific register set saving and compiler knowledge. Another drawback is the difficulty of freeing allocated resources properly before the handlers for such resources leave the variable scope.

EXCEPTIONS Today, the most common way of alerting is to use the exception mechanism introduced by Goodenough [57], in languages that provide this, such as `CLU` [103], `C++` [143], `Java` [59], `Ada` [146], `ML` [114] and `Python` [152].

Raising an exception consists of two parts: First, a function, say A, sets up an exception handler listening for a particular type of exception E, using a `try/catch` (Java), `handle` (SML) or `try/except` (Python) construct. It then invokes the function B, either directly or indirectly. B raises the exception E by invoking the `raise` (Python, SML) or `throw` (Java) language construct, and the search for an appropriate exception handler starts. Each stack frame is consulted in succession, until one with a handler for E is found. If no new handler for E is declared in the functions between B and A on the stack, control is transferred from B to the handler declared in A.

For the languages above, after the handler in A finishes, execution continues in A. In other languages, it is possible to either resume after the `raise` statement in B, or restart B, see Section 5.6 for details.

CHECKED EXCEPTIONS

Exceptions may be either *checked* – must be declared by all functions that may throw them, both directly and indirectly – or *unchecked* – may be thrown without being mentioned in a function's list of throwable exceptions. In Java, checked exceptions are the default, but unchecked exceptions are used for catastrophic errors, such as out-of-memory errors and disk failure.

Exception handlers need not only be declared as markers on the stack. They can also be attached to classes, statically giving each class its own handler, or to objects giving each object a specific handler. This is discussed in Section 5.6.

CONDITION SYSTEM The PL/I ON condition system, allows the programmer to attach handler blocks for pre-defined language exceptions that may occur in expressions, such as division by zero and end of file. These handlers are installed and removed dynamically. Extensions of this idea can be found in Dylan, Smalltalk and Lisp, which have systems for detecting exceptional situations based on (a restricted description of) the state space of a program. When a given failure condition is met, control is transferred to a specified error handler which can elect to try error recovery followed by resume, or terminate the function that threw the exception.

EVENT HANDLERS Event handlers and posix signal handlers both provide a callback mechanism which may be used for passing error notifications from the operating system to the application, or between parts of an application. This model requires no special language support, and is usually tied to the API or framework the application was written with, e.g. posix (signals) or Win32 GDI (events).

GUARDING A pre-condition may be declared on a function, testing beforehand whether the function will return normally with the data. Effectively, pre-conditions ensure that the input falls within a function's domain, and attempts to ascertain whether the state of the system allows the function to complete. Formulating such a pre-condition may not always be possible, e.g., during complex interaction with external resources.

CONTRACTS A significant extension to guarding is *design by contract*, described by Meyer [111]. In this technique, explicit pre- and post-conditions are declared on every function. Whenever either fails, the program terminates immediately. A contract should never be checked by the caller; contract verification must happen during the implementation phase, not at run-time. Eiffel [110] was the first language to support and enforce contracts, but also comes with a notion of exception handling. A routine may have a rescue handler declared for it, which may either provide some default return value, retry the routine, or fail. In the latter case, the failure will be propagated to the method's caller.

DESIGN BY
CONTRACT

GOTO The use of goto as an exception handling technique has almost disappeared with the introduction of various exception handling language features. In some restricted domains, such as the kernel code of operating systems, where space and performance considerations outweigh readability, gotos are still prevalent.

5.2.2 ALERT HANDLING POLICIES

Even using the same basic alert reporting, different usage policies lead to large differences in alert handling in the design of frameworks and libraries. The policy about retrying on failure, is one example. Unix leaves it to the user to retry failing operating system calls, for example if a long running kernel operation is preempted. Windows (and BSD Unix variants), on the other hand, retries preempted operations automatically.

In many languages, guiding principles exist about using the exception handling feature of the language. This is the case for Java, where the general recommendation is to use exceptions for alerting. Despite such principles, there are numerous examples in the library where error return values are used, among them in the implementation of the hash table.

GUIDING
PRINCIPLES

5.2.3 A GAME OF ANTICIPATION

The state of the art is to use design experience on a case-by-case basis to provide suitable alerts. Specifically, there exists no declarative way to select the desired error handling mechanism for part of a program. The

need for design experience comes from the fact that fundamental trade offs between the caller and callee of an abstraction must be managed. The caller is the party which will be implementing the alert handling. As the various handling techniques have different affinities with alert reporting, and every caller is potentially different, the implementer of the callee must anticipate the handling techniques that will be used by the callers.

From the callee side, the ideal reporting mechanism may depend on the implementation of an algorithm. For instance, if we are within a deeply nested data traversal, it may be more convenient to throw an exception than to use return values.

CALLER VS. CALLEE

Another consideration is who should do error checking on the input parameters. Should the callee accept erroneous input and produce garbage? Should the callee do all checking? This decision is usually coupled with significant performance trade offs.

The amount of anticipation required by the implementer of the callee is significant, perhaps especially in core language libraries. In Java, an example can be found in `java.util.Queue`, which provides pairs of identical functions, save for differences in alert reporting: `poll()` and `remove()` can both be used to remove the head of a queue. `poll()` returns `null` if there is no head, whereas `remove()` throws an exception in the same case. Similarly, `add()` throws an exception if a new element cannot be added to a queue, whereas `offer()` returns `false` if the insertion failed.

WHAT IS FAILURE?

Determining a suitable reporting mechanism when implementing a callee is compounded by another problem: the caller is the final arbiter of what is normality and what is failure. Returning `null` from a hash table lookup may in one application be completely acceptable, and not constitute a special case in the algorithm using the hash table. In another application, it will be the sign of severe data corruption and violation of crucial invariants. In the first case, returning a `null` is neither an exceptional case, nor an error, and this situation is therefore not a candidate for alert handling. In the second, it is critical that proper alerting be used.

5.3 *Separation of Concerns*

ASPECT
ORIENTATION

SEPARATION OF ALERT REPORTING AND HANDLING Although the mechanisms in Section 5.2.1 are essentially equivalent in that they all report exceptional situations (possibly with additional information), the default action taken when an error occurs differs. For return values and error flags, the default is to ignore the error. For exceptions, the default is to propagate the exception through the call hierarchy, possible leading to

termination of the program. For guarding, the default is not to guard, i.e., ignore the error.

In all cases, the callee implicitly decides the default action in case of an error, by choosing a given report mechanism. This is unfortunate, since the goal of raising an alert to the caller is to let the caller decide the appropriate course of action (otherwise, the callee could simply handle everything on its own). Additionally, the choice of alert mechanism is often based on implementation pragmatics, rather than whether the default action is likely to be appropriate for the severity of the error. If the callee is changed to use a different mechanism, all call sites must be updated. Thus, we have a *tangling* of alert mechanisms (callee implementation) and alert handling (caller implementation).

TANGLING OF
REPORTING AND
HANDLING

SEPARATION OF NORMALITY AND EXCEPTIONALITY With existing techniques for handling exceptional situations, we get a tangling of a program's normal behaviour and its exceptional behaviour. If our handling policy is to report small errors to the user and abort the program on serious errors, we have to code this into all places where errors are handled. Thus we end up with a mix of code dealing with normal circumstances, and code dealing with exceptional circumstances (alert behaviour). This leads to cluttered code and maintainability problems: if we wish to change our policy for some errors, we may need to change a lot of code in many different parts of our program.

TANGLING OF
NORMAL AND
ERROR CODE

This problem has also been observed in [100], where aspects are used to untangle the alert behaviour from the normal behaviour. The authors advocate that alert handling code be put in separate declarations – aspects – instead of being scattered around the code (see Section 5.6).

GRANULARITY In some cases, mixing normal and alert behaviour may be beneficial; for example, by taking alerts into account locally, we may compensate, e.g., by substituting a different return value. In some cases, such decisions must be made for each call site; in other cases, we may be able to provide a common policy for an entire class, a module or a set of functions.

SCOPING OF
HANDLERS

An example of this is IEEE floating-point expressions. Here a NaN is propagated through the expression and may be tested for, while an overflow ($\pm\infty$) may be propagated or consumed depending on the expression itself.

5.4 Alert Language Extension

While the existing body of research on exception handling addresses many of the concerns we have mentioned above, one area of the design

```

declaration ::= alert { alert-def* } super-alert* ;
      alert-def ::= alert-name [(parameter-list)] super-alert*
      super-alert ::= : alert-name

```

FIGURE 5.1: Grammar for the declaring new alerts.

space remains relatively unexplored: how to extract and declare separately the handling of exceptional situations. This is what we will address in the next sections.

Our mouldable abstraction of alert handling provides for separation of mechanism and handlers, separation of normal behaviour and failure behaviour, and allow decisions to be taken at the appropriate level of granularity, i.e. at the expression, statement, function, class, module or component level. Furthermore, our proposed solution allows the callee to declare what is normality and what is exceptional; allows the caller to declare the desired alert handling policies; can be applied retroactively to existing libraries; and is able to distinguish different types of errors.

A grammar for the alert extension is presented in Figures 5.3, 5.4 and 5.5. Although our prototype implementation (discussed in Section 5.5) is an extension of the C language, we will discuss the extension in terms of a C/C++/Java/C#-like language with an exception facility.

The grammars in this paper are meant for human consumption, and not for use directly in an implementation. Non-terminals written in upright font are meant as hooks into the base language. Non-terminals ending in *name*, *type* or *expr* are all names, types or expressions of the appropriate type. The notation “^{*}” and “⁺” is used for comma-separated lists.

5.4.1 DISTINGUISHING DIFFERENT ALERTS

Alerts are declared with the **alert** declaration, using syntax similar to the **enum** declaration (see grammar in Figure 5.1):

```
alert {MyAlert};
```

Multiple comma-separated alerts can be declared in the same declaration. For example, a selection of `posix` error codes is listed in Figure 5.2. These errors can occur during normal file operations, such as **open**, **read** and **write**. We can give each of them a unique alert name with the following declaration, with different (case-sensitive) names to avoid name clashes:

```
alert {eBadF, eIntr, eIO, eNoEnt, eNoMem};
```

Alert names are in a separate namespace, but error codes are commonly declared with macros in C, which ignores namespace boundaries.

EBADF	Bad file descriptor
EINTR	Interrupted system call
EIO	Input/output error
ENOENT	No such file or directory
ENOMEM	Insufficient memory available

FIGURE 5.2: A small selection of POSIX error codes, used, e.g., for **open**, **read** and **write**. The codes are set in the global **errno** variable, and should be checked whenever a function raises an error (typically by returning **-1**)

If we look at the selected error codes (and a POSIX reference), we see that they fall into roughly four categories: temporary conditions (**EINTR**); system problems outside the program's control (**ENOMEM**, **EIO**); problems that might be correctable with user help (**ENOENT**); and programming errors (**EBADF**). Thus, it is useful to be able to group them, so that we may, for example, automatically retry temporary failures, ask the user for a new file name on permission or missing file problems, and abort the program on system errors and programmer errors. To do this, we organise our alerts in a hierarchy, similar to an inheritance hierarchy in OO programming (which is also used for exceptions – in Java, for example):

INHERITANCE

```
alert {Retry, AskUser, FatalSys, FatalBug};
alert {eNoEnt : AskUser};
```

The colon separates a sub-alert from its super-alert in a declaration. Multiple alerts can be assigned a common super-alert:

```
alert {eIO, eNoMem} : FatalBug;
```

Additionally, an alert may have more than one super-alert:

```
alert {StupidMistake};
alert {eBadF} : FatalBug : StupidMistake;
```

To avoid cycles in the inheritance graph, alerts must be declared before they are used as super-alerts. The built-in alert **Alert** is the super-alert of all other alerts.

Finally, we note that it is sometimes useful for the callee to pass some information back to the caller. To do this, the alert must be declared with one or more arguments. For example, an **Error** alert which allows a message to be passed to the caller:

FEEDBACK

```
alert {Error(char *msg)};
```

We will see below how the values are passed from callee to caller.

```

declaration ::= fun-dcltr (alert alert-rep | throws-clause)* [fun-body]
declaration ::= alertrepdef alert-rep alert-rep-name ;
declaration ::= funspace funspace (alert alert-rep)+
  alert-rep ::= [alert-name] pre [unless] (cond-expr)
  alert-rep ::= [alert-name] post [unless] (cond-expr)
  alert-rep ::= [alert-name] on throw exception-name
  alert-rep ::= alert-rep-name
  alert-rep ::= { alert-rep *, }

```

FIGURE 5.3: Grammar for specifying alert reporting. The *funspace* non-terminal is defined in Figure 5.5.

5.4.2 SPECIFYING THE ALERT REPORTS OF A FUNCTION

Alert reports are specified at the callee side with an **alert** clause in the function declaration, c.f. grammar in Figure 5.3. Possible reporting mechanisms include condition checks before (**pre** conditions, useful for guarding) and after (**post** conditions, for checking return values and global error flags) a call, and exceptions. For example, to declare that a hash table lookup function `lookup` returns `0` on failure, we write:

```
val lookup(tbl t, key k) alert post(value == 0);
```

To use a more specific alert than the default `Alert`, we simply add the name of the desired alert:

```
val lookup(tbl t, key k) alert NotFound post(value == 0);
```

The **post**-clause takes an expression, which is evaluated after the function call returns – if the expression is true, the function has failed. The special keyword `value` can be used to check the call's return value (the type of `value` is that of the return type of the callee). Similarly, the **pre**-clause takes a condition which is checked *before* the function is called.

The condition expressions may be arbitrarily complex, but should only use globally accessible names, arguments to the call, and `value`. Arguments are referred to by the name they are given in the function declaration, and have whatever values they have at the time of checking (before or after the call).

```

// alert if table t cannot be expanded
// due to memory constraints
int insert(tbl t, key k, val v)
  alert eNoMem post (value == -1 && errno == ENOMEM);

```

Specifying a condition with the **unless** keyword negates it, providing a more intuitive way of specifying invariants and pre/post conditions

(which are often specified in terms of what is normal, and not in terms of what is exceptional).

```
// separate success/failure flag if no return values
// can be used for alerting
val lookup(tbl t, key k, bool *success)
  alert NotFound post unless(*success);
```

Exceptions (if the language supports it) can be declared with a **throws** (Java) or **throw** (C++) clause, provided that the exception name has also been declared as an alert:

```
alert {AnException};
int f() throws AnException;
```

If a mapping between exceptions and alert names is desired, an **on throw** clause may be used:

```
int f() throws AnException
  alert Error on throw AnException;
```

In all cases, information may be passed to a handler using alert parameters:

```
val lookup(tbl t, key k)
  alert NotFound(k) post(value == null)
```

The callee must still provide some way sending this information, either through updating of arguments, return values, global variables or exception objects – alert parameters are merely a declaration of whatever mechanism is used. For exceptions, the exception object is available for use in alert parameters:

```
int f() throws AnException
  alert Error(e.msg) on throw AnException(e);
```

5.4.3 ALERT HANDLING

The *alert handler* will treat all alerts the same, whether they are reported by return value, condition check, or exception. The grammar for alert handlers is presented in Figure 5.4. There are two alert handling constructs: **on**, for specifying an alert handler at any scoping level, down to a single statement, and the *handler operator*, **<::**, which specifies a handler for a single expression.

The on Construct

The **on** declaration takes a *alert-pattern* and a statement. The declaration is lexically scoped and applies to all call sites it matches within its block. The statement form of **on** applies to a single statement. If more than

```

declaration ::= handler handler-name ( parameter-list ) statement
declaration ::= on alert-pattern statement
statement ::= retry [ ( argument-list ) ] [ max int-expr ] ;
statement ::= use expr ;
statement ::= handler-name ( argument-list ) ;
statement ::= statement-body on alert-pattern statement
alert-pattern ::= single-alert + , [ in funspace ]
alert-pattern ::= alert-pattern or alert-pattern
single-alert ::= alert-name [ ( parameter-list ) ]
expr ::= expr <: [ alert-pattern ] : handler
handler ::= expr | { statement }
handler ::= handler-name ( argument-list )

```

FIGURE 5.4: Grammar for alert handlers. The *funspace* non-terminal is defined in Figure 5.5.

one handler matches, the most specific one closest in scope applies, or a compile-time error is given if there is more than one equally suitable handler.

The *alert pattern* specifies for which combination of alerts and callees the handler applies. The handler itself is a single or compound (block) statement, which should provide a replacement value, retry the computation, refer to another handler, or terminate the caller. It is an error for a handler to complete without providing a replacement return value when one is needed – in this case, we terminate the program (though we could check statically whether this can occur, and other design choices are certainly possible).

REPLACEMENT
VALUES

Within a handler, the **use** statement may be used to provide a replacement value; **use** exits from the handler as if the callee had returned normally with the value provided. For example, the following defines a handler for **NotFound** in the **lookup** function which substitutes the value "Doe, Jane" for failed lookups (e.g., when mapping ID numbers to names):

```
on NotFound in lookup() use "Doe, Jane";
```

The following does the same for all alerts in **lookup**:

```
on Alert in lookup() use "Doe, Jane";
```

The next two declarations both do the same for **NotFound** in all functions (the % matches all functions):

```
on NotFound in % use "Doe, Jane";
on NotFound use "Doe, Jane";
```

The % was chosen to avoid confusion with pointers (*) in C/C++, and is used in a similar fashion in AspectC++ [139].

The following `NotFound` handler applies to just the preceding `print` statement:

```
print(lookup(tbl, key)) on NotFound use "Doe, John";
```

The `retry` statement tries the failed call again (possibly with a maximum retry count, specified with “*max number*” – the default is to retry indefinitely). The `retry` statement also takes an optional list of arguments, which will replace the arguments in the failed call. It will exit from the handler, continuing execution at the point of the call which reported the alert – except when the maximum retry count has been reached, in which case execution continues with the next statement after `retry`.

RETRYING

For example, the following specifies that on a `NotFoundError`, we should ask the user for a new name and try again (maximum 5 times). If our recovery attempts fail, we substitute an empty string.

```
on NotFoundError in readfile(char *name) {
    warn("trying again...");
    char* name = askUser();
    if(name != NULL) retry(name) max 5;
    warn("giving up...");
    use ""; }
```

The Handler Operator

The handler operator provides a convenient way of handling alerts at the expression level. The left operand is an expression to be evaluated, and the right operand is a handler to be used if an alert was reported in the expression. Within the operator itself, one may specify which alerts should be handled. Alerts are specified in the same way as for `on`, the handler can be either an expression giving a replacement value, a call to a previously defined handler, or a statement list (enclosed in braces). In the following example, a string is substituted if a lookup fails:

EXPRESSION-LEVEL
HANDLERS

```
print("result:", lookup(t,k) <:: "Unknown");
```

An alert pattern can be specified between the colons:

```
print("result:", lookup(t,k) <:NotFound: "Unknown");
```

Furthermore, handler code can be provided, as for `on`:

```
fd = open(name, flags) <:eNoEnt: {
    char *newname = askUser();
    if(newname != NULL) retry(newname, flags);
    else giveUp("couldn't open file"); };
```

This will try to open a file, and if the file is not found, the user will be asked for another name. If the user provides one, we try that instead, otherwise, we abort with a message.

5.4.4 ABSTRACTION

Our extension provides abstractions for alert handling and reporting. The **handler** construct declares handlers which may be used later on by the **on** declaration or the handler operator. For example,

```
handler log(msg, dflt) {  
    print("An error occurred: ", msg);  
    use dflt; }
```

which may be used as:

```
on NotFound in % log("Lookup failed", "");  
name = askUser()  
    <:eNoEnt: log("No response from user", "--");
```

Handler abstractions look deceptively like functions in both definition and use, but are not functions, since the **retry** and **use** statement would be tricky to implement in a separate function. Instead, the definitions are expanded inline wherever they are needed. Hence, (mutually) recursive handlers are not allowed.

The **alertreodef** declaration declares alert reporting mechanisms for use in a function declaration. It follows the same pattern as the C/C++ **typedef** construct. This is useful when several functions share the same alert behaviour. For example,

```
alertreodef alert Error post(value == 0) ErrorOnZero;
```

ErrorOnZero can then be used for functions raising errors with a zero return value:

```
int f() alert ErrorOnZero;
```

5.4.5 SENDING INFORMATION FROM CALLEE TO CALLER

Alerts can have associated values (alert parameters), allowing a callee to provide additional information to a caller. A similar idea is found in exception handling (e.g., in Java or C++), where exceptions are objects that may contain information relevant to the exceptions. As shown in Section 5.4.1, valued alerts should be declared with arguments:

```
alert {Error(char *msg)};
```

At the callee side, we provide a suitable value in the alert clause:

```
int read(int fd, void *buf, size_t count)  
    alert Error("read error") post(value == -1);
```

The value can be obtained at the handler side from the alert pattern:

```
on Error(msg) in read() { print(msg); exit(1); };
```

In this example, `msg` is declared as a string, and gets the value "read error" from the failed `read()`. If more than one alert is given in the alert pattern, all of them must have the exact same argument list. It is not necessary to mention the arguments if they are not needed by the handler.

If the return value of the callee is to be available to a handler, it must be passed as a parameter, as return values are not always available (e.g., for exceptions and `pre` conditions), and there is no way for the handler to distinguish between different alert reporting mechanisms.

Note that, unlike exceptions, we need not construct an alert object as an aggregate of values. Instead, code is generated in the handler which obtains the information directly (which is why only arguments, return values and global variables can be passed from the reporter). Thus, as for alert conditions, we are restricted to expressions which have the same meaning for both the callee and the caller (i.e., global names and operators, constants and arguments, either before or after the call).

In the case of functions which change their arguments (or modify global data structures), it is possible that the state of these variables is inconsistent when the handler is invoked. In this case, it is up to the handler to put things in a consistent state or terminate execution. Ideally, functions would ensure that the program state is rolled back to a safe point before an alert is reported, or at the very least, declare that this may not happen for some or all alerts. This problem is also found with the common exception handling mechanisms. We have not dealt with this problem yet.

DATA CONSISTENCY

5.4.6 GRANULARITY AND FUNSPACES

By granularity, we refer to the coarseness of a declaration in the hierarchy from expression through statement, function declaration, and optionally class, module and subsystem level, all the way to the system level. Our language extension provides additional granularity alternatives, among them groups of functions, which we call funspaces. Funspaces can be applied to both alert handling and reporting.

Granularity of Handlers

In most languages, exception handlers are specified at the statement level, and the exception declaration (e.g., `throws` in Java) occurs along with the function declaration.

In our system, the declaration of both alert handlers and alert reporters can occur at various levels of granularity. As we have seen, handlers are declared with `on` declarations. These can occur at any level in the scope hierarchy (from global, through namespace/package and

SCOPING

```
funspace ::= [return-type | %](function-name | %)  
          [( parameter-pattern-list )]  
funspace ::= funspace funspace-name  
funspace ::= { funspace *, }  
declaration ::= funspacedef funspace funspace-name ;
```

FIGURE 5.5: Grammar for declaration of function spaces.

class, down to blocks and single statements within a function) and apply to the scope in which they are declared. Additionally, handlers can be declared at the expression level using the handler operator (`<: :`). If multiple handlers are in conflict, the most specific handler takes precedence, i.e. the one with the most specific alert pattern at the finest level of granularity.

The concept of a scoping level can be refined using *funspaces*. A **funspace** declares a set of functions, i.e. a subspace of the namespace for function names. The grammar for funspaces and funspace declarations is presented in Figure 5.5.

A funspace is basically a list of function patterns. For example, (a non-exhaustive list of) the file operations of `posix` can be declared as follows:

```
funspacedef { open(), close(), creat() } posix_io;
```

Each entry in the funspace list conforms to a pattern. For C and languages without overloading, giving a function's name is sufficient. In languages with overloading, the full signature must be given;

```
funspacedef {  
    int open(const char *pathname, int flags),  
    int open(const char *pathname, int flags, mode_t m),  
    int close(int fd),  
    int creat(const char *pathname, mode_t m) }  
posix_io;
```

The pattern can also contain wildcards, with `%` matching any single item, and `..` matching any argument list. For example, The pattern `%(const char*, mode_t)` would match any function with any return value, that takes two parameters: a `const char*` followed by a `mode_t`, e.g. `creat`:

```
int creat(const char *pathname, mode_t mode);
```

Funspaces, being sets of functions, can be merged, allowing us to construct the `posix` funspace from smaller, task-specific funspaces.

```
funspacedef {
    funspace posix_io,
    funspace posix_memory,
    funspace posix_process }
posix;
```

This is not merely a syntactic convenience. Different subsets of a given API often use different sets of errors, each specific to that subset. Sometimes, the same numerical error value is reused with different meaning across different subsets. `ENOMEM` when returned from `mmap` has a different meaning than `ENOMEM` returned from `stat`. Using funspaces, these differences can be captured at the granularity of function groups, rather than having to be specified on per-function basis.

Granularity of Alert Reporting

In the previous sections we saw how alert reporting is declared on individual functions. Using funspaces, reporting mechanisms may conveniently be declared on groups of functions. The following declares that the `eNoMem` alert will be reported on any function in the `posix` funspace if it returns `-1` and the global variable `errno` is set to `ENOMEM`.

```
funspace posix alert eNoMem post(value == -1
    && errno == ENOMEM);
```

Both handlers and alerts can be declared at any scoping level and on funspaces, but the declarations are completely independent. For example, the alert `eNoMem` may be specified for all `posix` functions, as above, while a handler for this alert could be declared for only one expression inside a particular function in a given program, e.g.,

```
f() { open("foo", O_RDONLY) <:eNoMem:exit(EXIT_FAILURE); }
```

Or, it could be declared for all `posix` functions:

```
on eNoMem in funspace posix { exit(EXIT_FAILURE); }
```

Multiple, overlapping funspaces may be declared, and both alerts and handlers may be specified independently for each funspace.

5.4.7 INTERFACING WITH LEGACY CODE

Introducing new failure handling disciplines typically means that legacy code must be rewritten if it is to take advantage of it. This is the case with exceptions, for instance: if you want exceptions in an existing library which reports errors with return values, you will have to either rewrite the library or write a wrapper for it.

Funspaces, together with handling and reporting abstractions can be used to specify alert reporting mechanisms and handling for a large

```

alert {PrecondFailure, Whoops};
on PrecondFailure in * {fatal("Precond failed");}

int f(int x) alert PrecondFailure pre unless(x > 0)
    alert Whoops post(value > 10);
int ff(int a, int b)
{ on Whoops in f() {print("whoops!"); use 0;}
  return f(f(a));
}

```

```

int f(int x);

int ff(int a)
{ int r;
  if(a > 0)
  { r = f(a);
    if(r > 10) { print("whoops!"); r = 0; }
    if(r > 0)
    { r = f(r);
      if(r > 10) { print("whoops!"); r = 0; }
    } else fatal("Precond failed");
  } else fatal("Precond failed");
  return r;
}

```

FIGURE 5.6: Comparison of the Alert extension to C (top), and the corresponding normal C code (bottom).

CODE REUSE

number of existing functions in a few lines of code. This makes reuse of existing libraries simpler, which is especially important since many older libraries use exotic alert reporting mechanisms which may be inconsistent with newer code.

If the library comes with structured documentation, it may be possible to automatically extract alert specifications from the documentation. For example, the `posix` standard [119] comes with structured manual pages in HTML format, available online. A tool could be written to extract from the manual pages function names, return value style (“returns `-1` on error”) and which error codes are applicable to the function, and generate the necessary declarations. We are currently exploring this option.



FIGURE 5.7: The compiler pipeline for our extended C language.

5.5 Implementation

We have made a prototype implementation of the language extension for C that implements the compiler pipeline shown in Figure 5.7. The prototype is implemented using the Stratego/XT [25] program transformation language, and the C-Transformers [19] framework for C99 transformation.

The prototype consists of an extension to the C99 grammar [88] (written in the grammar formalism SDF2 [154]) and a set of transformations translating the extended C code to standard C. As seen in Figure 5.7, the parser will recognise the extended C language and produce an abstract syntax tree (AST). Minimal type analysis is then performed to check that the handlers are type consistent with the functions they will be applied to. Next, the alert extension is “peeled off” in a desugaring step before the the AST is pretty-printed into text and fed to a normal C compiler.

5.5.1 TRANSLATION SCHEME

The translation algorithm works roughly as follows (for C99, following traversals can be combined into one pass, since the language requires declaration before use):

First, traverse the AST and look at all function declarations and definitions. For each declaration or definition, extract the signature (i.e., name, return type, argument types) and the alert mechanism (i.e. pre/postconditions – exceptions are not available in C), and store this for later.

Next, traverse the AST and look for scopes, function calls, **on** handler declarations or handler operators. When seeing a *scope*, create a new, nested scope for subsequent **on** handler declarations. When later exiting this scope, drop all handlers registered in this scope. When seeing an *on handler declaration*, register its entire definition in the current scope. When seeing a *handler operator* ($<: :$, expand the pattern in Figure 5.8. When seeing a *function call*, check if the signature of the callee is matched by any of the registered handlers. Check the textually closest handlers first and proceed to parent scopes. If a matching handler is found, expand the pattern in Figure 5.8.

Keep in mind that a function call is only rewritten if at least one relevant handler is found for it. Let us call the closest (and thus active) handler *current-handler*. Given a function call $f(e_1, \dots)$ to function t_0 $f(t_1,$

```
t0 r;  
{ t1 v1 = e1; ...  
  if(<precond-f(v1, ...)>) {  
    <current-handler>;  
  } else {  
    r = f(v1, ...);  
    if(<postcond-f(r)>) {<current-handler>;}  
  } }
```

FIGURE 5.8: Template for desugaring function calls.

...) where t_0 is the return type, f the name, t_1, \dots the types of the formal arguments, and e_0, \dots the expressions for the actual arguments, the instantiation of the template in Figure 5.8 occurs as follows: First, a local variable r of type t_0 is declared, and will hold the eventual return value. Then, each of the actual arguments is evaluated and stored, each e_x into a local variable v_x . Then, the expression for the **pre** condition of f is evaluated on the variables v_x (the expression $\langle \text{precond}(v_0, \dots) \rangle$ means that the precondition code is expanded in-place – care is taken to avoid accidental variable capture), and if it succeeds, we must invoke the alert handler. The code for *current-handler* is expanded in place, and use statements in the body are translated into assignments to r . The process is similar for the **post** condition and the **post** condition handler. An instantiation of this template was given in a cleaned up, human-readable form in Figure 5.6.

5.5.2 IMPLEMENTATION ISSUES

In C, which lacks function overloading, matching a function call to the function's declaration can be done easily just by comparing names. In other languages, such as C++ and Java, overload analysis is needed to distinguish functions sharing a common name.

Although overload analysis is unnecessary for C, we still need to be able to determine the types of arbitrary expressions, in order to declare temporary variables and type-check substituted values. Support for this is lacking in Transformers, but was easily added.

A special problem occurs with function pointers, since it is in general impossible to determine statically which function will be called at run-time. In the case of dynamic loading, the function may not even be written yet. A similar problem occurs with exceptions in object oriented languages, where the preferred solution, in for example Java, is to require the exceptions of a function in a subclass to be an (improper) subset of the exceptions of the corresponding function in the superclass. This

technique translates to our alerts as well: We can add a declaration about alerting to the type declaration of the function pointer, i.e. the function pointer declaration now also declares the alert mechanism for the function it will eventually point to, and type checking on the function signature, with alert declarations, must be performed when function pointers are assigned to. Arguably, this will make function pointers even more difficult to read, but these syntactical issues can be remedied by judicious use of `typedefs`. Our current implementation does not yet support this.

5.5.3 COMPILING TO ASPECTS

The application of alert handlers to function call sites is a separate, cross-cutting concern, and can certainly be considered an aspect in the sense used by Kiczales et al [95]. If we targeted AspectJ rather than C, the template in Figure 5.8 could be realised as an around advice, where the invocation of `f` is replaced with a call to `proceed`. The `pre` and `post` condition expressions would be placed before and after the call to `proceed`, respectively, in the same fashion as now. `Use` statements in the handler body would translate into `returns`.

The full granularity of handler declarations from expression level to arbitrary function groups would be harder to capture faithfully, however. While funspaces can be captured by normal pointcuts, by listing all function names in the point-cut, we do not see any easy and robust way of encoding pointcuts that exactly match expression level handlers.

5.6 Related Work

Language support for exception handling has been introduced gradually since the 1960's. Research-wise, PL/I's condition system and later CLU [101; 103] and Ada's structured exception handling have perhaps been the most influential.

CLU

Hill [81] documents about ten different idioms for raising exceptions in languages such as Algol and Fortran, which at the time did not have any exception handling facility. He advocates disciplined control transfer over special return values.

Randell [124], introduces a failure recovery system inspired by "stand-by sparing", as found in hardware designs. Their technique assumes a nested, block-structured language, and provides transaction-like error recovery. Each block is a transaction, and may have one or several recovery blocks associated with it. A block has an acceptance test, which acts as a postcondition check.

STANDY-BY
SPARING

If a block fails, by failing its postcondition check, the program state is unrolled to the state before the block was entered, and the list of

associated recovery blocks is tried in order, each time preceded by an unroll, until one of the recovery blocks passes the acceptance test. If the list is exhausted before recovery occurs, the error is passed to the enclosing block.

This technique does not support raising of errors to handlers further up the call chain.

MacLaren [107] critiques the design for being too complex, and encouraging bad idioms, like a global `ON` handler for file exceptions which set global error flags that must be checked after by the caller of any file operation, thus effectively degenerating to global error values.

Borgida [20] discusses language features for exception handling with a focus on the interplay between exception handling and transactions found in database and information systems. He advocates the support for resumption, user-defined exception types, classification of exception types, and preventing the handler from modifying the context of the alerter. The language presented supports transactional unrolling in the case of unhandled exceptions and the capture of accountability in transactions using exceptions.

REFLECTION

In a sufficiently reflective language, such as Oberon, exception handling may be entirely implemented by the user without extending the language, as is shown in [85]. Oberon allows reflection over stack frames using “riders”. Exceptions are thrown by invoking a rider that locates the appropriate handler in an enclosing procedure on the stack. If the found handler returns, this is taken as termination, and the stack below the handler is cleaned. If the handler invokes `Resume`, execution resumes at the point of exception.

ADA

Romanovsky and Sandén [125] discuss good and bad practices in exception handling, dividing the problem into bad language design and misuse due to insufficient training. They argue that languages should support two kinds of exceptions with respect to their program units (modules or packages): internal and external. External exceptions must be declared and checked, i.e. a propagation discipline must be declared and the compiler must enforce it. They argue further, based on experience with Ada, that exceptions in OO-languages must be classes, and user-definable, so that information may be passed from the exception raiser to the handler with the exception, and so that the exceptions may be classified based on type. They also argue that exceptions can aid in, rather than complicate program validation and verification.

The critique of Romanovsky and Sandén about a propagation discipline was addressed by Luckham and Polak [106]. They describe a language extension to Ada for specifying the propagation of exceptions. This extension has not been included in later versions Ada, however.

Cui and Gannon [35] describe an alternative exception handling system for Ada than the school of Goodenough [57]. Instead of being

declared as part of the control structure, as markers on the stack, exception handlers are dynamically attached to objects. When an object raises an exception, its associated handler is invoked. The authors refer to this as data-oriented exception handling. Our alert system provides no easy way to support this form of exception handling.

An argument against exception handling for embedded systems – the difficulty of predicting timing constraints in the face of exceptions – is addressed by Chapman et al [30], where the authors presents a model for static timing analysis of exceptions in a subset of Ada83.

The Lisp condition system [121] is similar to PL/I's ON, but supports more of the features first described by Goodenough [57], such as resumption.

Other functional languages, such as SML and Haskell, also support exceptions. Wadler [156] shows how exceptions may be realised in purely functional programming languages, using monads.

Dony [41] describes an object-oriented exception handling for Smalltalk, where users can define new exceptions, where exception objects contain information passed from the alerter, and where different exceptions can be distinguished and organised in a hierarchy based on their type. Like Goodenough's approach, the possible action of the handler are resumption, termination, and retry, but the choice is determined by the type of the exception object, rather than the alerting primitive.

Smalltalk-80 allows the declaration of class-handlers: per-class exception handlers. These do not take handlers in the dynamic call context into account, as proposed by Goodenough; this is provided by Dony's extension.

Lippert and Lopes [100] describe how to untangle error handling code from algorithmic code using aspect-oriented programming. The solution is applied to a framework constructed using design by contract. Aspects are used to extract the contract checking code from the rest of the framework. A drawback of the proposed solution is that the contract is declared in documentation comments along with the framework (algorithmic) code, whereas the contract checking code is maintained elsewhere, thus opening up for deviations between the declared contract and the actual contract checking code.

FUNCTIONAL
LANGUAGES

SMALLTALK

ASPECT
ORIENTATION

5.7 Conclusion

The state of the art in alert handling provides reporting mechanisms that are both efficient and expressive. However, with the exception of the aspect-oriented approach to separating out failure behaviour [100], failure handling code is largely rigid. The alert reporting and handling are tangled, and the implementer must always choose a mechanism

when implementing a function, but in doing so, also makes an implicit choice about the handling policy.

We have presented a flexible alert language extension that supports decoupling of the reporting and handling mechanisms for exceptional behaviour. The extension user-defined alert handling and reporting at a wide range of granularities. It allows the caller to declare what is normal and what is exceptional, and to declare separately the desired handling policies. This improves reuse of existing libraries and components, as policies can now be specified retroactively by the library user.

We have sketched its implementation based on the Transformers program transformation framework. The extension functions as a compiler extension in the form of a pre-processing step to the C compiler.

The design space where the alerter and handler are decoupled is largely unexplored. This is unfortunate, since even the modest extensions we have shown to a simple, imperative language with exceptions could improve both the reuse of existing code bases and the clarity of failure handling code.

Some work on this topic has been done in the context of aspect-orientation, but we believe that our alert declaration language is more precise and concise than generic join points and advice. One could consider our language extension as a domain-specific aspect language for error handling.

While our proposed extension has only been realised for two rather simple languages, in the imperative style, we expect it to be transportable to other languages and paradigms. The syntax should be modified to fit the conventions of the language; in this paper, we have based the syntax on C-like languages; this would be out of place in Python, for instance.

A few obvious extensions may be necessary. In general, funspace patterns may need to be more like AspectJ [95] or AspectC++ [139] pointcuts, to deal with function overloading and namespaces. In our subject language, C, this was not needed, since we only have one global namespace and no overloading.

It is important that funspaces should continue to be function (or method) groups, so that funspaces can cross-cut namespaces. This will keep the flexibility of specifying alert mechanisms and handlers across namespaces.

More research is necessary to determine the best interaction between funspaces and method visibility, however.

ACKNOWLEDGEMENTS

This investigation has been carried out with the support of the Research Council of Norway. We thank the anonymous reviewers and May-Lill Sande for their helpful comments.

No chickens were harmed in the making of this article.




Saguaro Cactus.
Tucson, AZ (PLDI'o8)

The Magnolia Programming Language

6.1 Introduction

Although we initially used C++ to prototype the language constructs described in the previous chapters, this turned out to be a tedious and frustrating task [38]. The amount of infrastructure needed to support mutification, alerts and axiom-based transformation and testing in C++ is huge, essentially one needs a complete compiler frontend – and finding one that is both fairly complete and extensible enough for our purposes is next to impossible. C++ is a large language, meant for large tasks, and this makes it difficult to use as a basis for small-scale language experiments.

The original goal of Magnolia was to be a simplified, easier to process version of C++: subtracting some undesirable features (to ease processing, or to give cleaner semantics) and adding some new features. We would subtract pointers; some troublesome parts of the syntax, semantics and the template system; and some of the object-oriented features we weren't using – and add mutification, alerts, axioms and any other experimental construct we might desire.

A few desirable changes to C++ (in addition to the experimental features) quickly appeared on the feature list: a clearer syntactic distinction between different declaration types, a more sensible module system than just code inclusion, a few more overloadable operators (or, with all of Unicode to chose from, a  more) – and soon the language designer instinct kicks in, with the desire to create more than just a C++ variant.. Thus, Magnolia was born.

As outlined in Chapter 1, Magnolia is designed to support program

A SIMPLER C++

MAGNOLIA

development based on abstraction and specification. In this chapter, we'll take a closer look at some aspects of Magnolia: procedural and data abstraction (Sections 6.3 and 6.5), types (Section 6.4), and specification with concepts and axioms (Section 6.7). We'll also discuss some compiler implementation issues.

This chapter is not a complete description or reference manual for the language – any such document would be quickly outdated as the design work is still in progress with both minor and major changes being considered. Rather, this chapter gives an overview of the language, focusing on the features that set Magnolia apart from other languages.

6.2 Signatures, Concepts and Implementations

A *signature* is a set of declarations of abstractions (e.g., functions, procedures, types). Specifying the behaviour of the abstractions in a signature gives us a *concept*, and implementing this behaviour gives us an *implementation*. This relationship is illustrated in Figure 6.1.

A *concept* is a signature with *axioms* and *requirements* describing the behaviour of the abstractions. This corresponds to the idea of an algebraic specification. An *implementation* is a signature with associated implementation code, providing algorithms for the operations, and data structures for the types.

Magnolia programs consist of a set of modules, containing concepts and implementations. An implementation *models* a concept if it has the same signature, and its implementation satisfies the axioms in the concept. *Signature morphisms* can be used to map between signatures – for example, functionalisation can turn a procedure declarations in an implementation into function declarations matching a concept.

Most of this chapter describes the signature and implementation side of things, but concepts are discussed in Section 6.7.

6.3 Procedural Abstractions

Magnolia provides two kinds of procedural abstractions: *functions*, which abstract over expressions, and *procedures*, which abstract over statements. Collectively, they are known as *operations*. As explained in Chapter 2, functions are forbidden from changing their arguments or accessing global data. Procedures may update their arguments, but should also refrain from accessing global data or having side effects other than their effect on arguments (this restriction is loosened when the procedure is declared '*impure*').

A function takes zero or more arguments, and returns a value. A function declaration looks like this:

FUNCTIONS

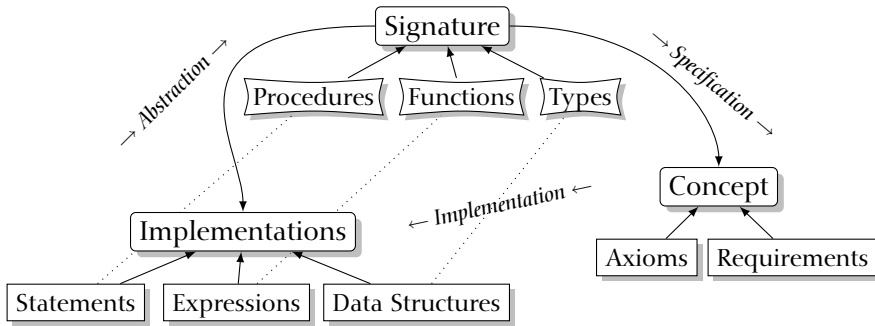


FIGURE 6.1: The relationship between signatures, concepts and implementations. Concepts add abstract definitions to a signature, implementations add concrete definitions.

```
function rettype funname(t1 p1, t2 p2, ...) = body;
```

where `t1` and `t2` are types and `p1` and `p2` are parameter names, and the optional body expression provides the implementation. A function application is an expression, with zero or more arguments that are also expressions. For example, the function above might be called like this: `funname(e1, e2)`, giving an expression of type `rettype`. For all functions, application with equivalent arguments must yield equivalent results. A function with zero arguments is known as a *constant*.

Procedures also take zero or more arguments, but have no return values. Procedure declarations look like this:

PROCEDURES

```
procedure procname(m1 t1 p1, m2 t2 p2, ...) {
    statement1;
    statement2;
    ...
}
```

where `t1` and `t2` are types, `p1` and `p2` are parameter names, and `m1` and `m2` are *parameter modes* – observe (default), update, output, delete or give. The optional procedure body (in braces) consists of a sequence of statements. Procedure calls are statements, and may not be nested. Arguments can be expressions (for `obs` parameters) or simple variables (for the other modes). An example call is:

PARAMETER
MODES

```
call procname(v1, e2);
```

Results are given through argument updates.

We have a clear separation between statement abstraction and expression abstraction – you can always be sure that nothing will happen to your variables in an expression.

Unlike in C and C++, it is not necessary to provide *forward declarations* – declaring an operation ahead of its use (and eventual implementation). The compiler takes care of finding all the operations declared in a program before attempting to resolve calls. This seems to be the trend in modern languages – except in some dynamic languages like Python, where declarations are run-time effects, and declarations must precede use at run-time (though not necessarily textually). Forward declarations simplified implementation on early computers, since the compiler could do the compilation in one pass and without having to keep the entire program in memory.

6.3.1 FUNCTIONALISATION AND MUTIFICATION

In Magnolia, a function can be declared either explicitly, using the function declaration syntax, or implicitly, by declaring a suitable procedure. Procedure declarations are *functionalised* by creating one or more function declarations in which the procedure's input parameters become function parameters, and the procedure's output parameters become return values. This process is detailed in Section 2.3.1. For example, consider the following procedure declaration:

```
procedure _++_(upd string a, string b);
```

FUNCTIONALISATION It declares an operation concatenating the string parameters *a* and *b*, storing the result in *a*. Based on this, the following function is declared:

```
function string _++_(string a, string b);
```

– where the **upd** parameter occurs both as a return value and a parameter. The resulting function may be used freely in expressions, without regard for the fact that the underlying implementation works by changing one of its arguments. Using operator syntax:

```
var s = "Hello, " ++ person ++ "!";
```

MUTIFICATION

To produce working code from this, the compiler applies *mutification* (explained in more detail in Section 2.3.2), where the function calls are translated into a sequence of procedure calls:

```
var s = "Hello, ";      // s = "Hello, "  
call _++_(s, person);  // s = "Hello, " ++ person  
call _++_(s, "!");     // s = "Hello, " ++ person ++ "!"
```

In general, mutification may require several temporary variables, particularly if the variable being assigned to isn't in the 'right' position in the argument list, or if an argument contains sub-expressions that also require mutification. Normal function call semantics also require temporaries to keep intermediate values – mutification will usually reduce the number of needed temporaries, and may also eliminate the need for them (as in the example above).

6.3.2 OVERLOADED OPERATION NAMES

Operation names may be overloaded in Magnolia – that is, distinct operations may share the same name, being distinguished only by their parameter types. The full combination of name and parameter types that make up a distinct operation is called the operation’s *signature*.

Internally in the compiler, all operations are identified by their signature. Connecting the use of a name – in a function application, for example – to a signature is called *overload resolution*, and is done together with type resolution early in the compilation process.

Overload resolution works as follows. First, a list of known candidate signatures is produced, based on the name used in the call. The list of candidates is filtered to remove clearly inappropriate candidates – function signatures for a procedure calls, for example. Then the list of formal parameters in each signature is compared to the list of actuals, using the type matching algorithm described in Section 6.4.1. Any non-matching candidates are discarded.

OVERLOAD
RESOLUTION

The resulting candidates are then ordered by preference, and if there is a clear best candidate, it is selected. If there are two or more equally good best candidates, an ambiguity error is reported to the programmer. The preference criteria are, in order:

1. The lowest number of implicit conversions needed for successful argument list match, or, if that is not sufficient,
2. the lowest number of type variables in the argument list match, or, if that is still not sufficient,
3. whether the candidate is locally defined or not – local names take precedence.

At some point it may be useful to give the programmer more control of the overload resolution – this could be done by allowing user-defined resolution functions that take a list of candidates and return the preferred one (or raise an error). The functions would be evaluated at compile time, or loaded into the compiler as an extension. The Common Lisp Object System (CLOS) has a feature similar to this for resolving dynamic dispatch at run-time [18].

6.3.3 OPERATORS

Operators are overloadable in Magnolia. The list of expression operators, presented in Table 6.1 is similar to that found in C and C++, and we are also considering letting the user define new operators by combining existing operator symbols, or by using Unicode characters. Operators don’t add anything to the semantics of the language, they are just syntactic sugar for function applications or procedure calls.

Operators	Associativity	Typical use
<code>_()</code> <code>_[]</code> <code>_.</code>	left to right	special
<code>!_</code> <code>~_</code> <code>+_</code> <code>-_</code>	right to left	unary ops
<code>_*</code> <code>_/</code> <code>_%</code> <code>_**</code>	left to right	binary ops
<code>+_</code> <code>-_</code> <code>_++</code>	left to right	binary ops
<code>_<<</code> <code>_>></code>	left to right	shifting
<code>_..</code>	left to right	ranges
<code>_<</code> <code>_<=</code> <code>_></code> <code>_>=</code> <code>_in</code> <code>_not in</code>	left to right	comparison
<code>_==</code> <code>_!=</code>	left to right	equality
<code>_&</code>	left to right	bitwise and
<code>_^</code>	left to right	bitwise xor
<code>_ </code>	left to right	bitwise or
<code>_&&</code>	left to right	logical and
<code>_ </code>	left to right	logical or

TABLE 6.1: Magnolia operators, in order of precedence (highest on top). Notable omissions compared to C/C++ are pre/post increment/decrement operators, pointer operators and assignment operators.

Each operator symbol has a corresponding name, which is made by putting an underscore in the position where the operator would take an operand. For example, binary plus has the symbol `+` and the operator name `_+_`. This makes it simple to distinguish unary and binary versions of the same operator. For example, unary plus is `+_`.

Most operators form expressions, and are implemented either directly or implicitly as functions. Assignment operators, on the other hand, form statements and must be implemented as procedures – this applies to regular assignment, `_=_`, and the set-element operation, `_[]=_`. For example, from the standard library:

forall type E

procedure `_[]=_`(**upd** array(E) A, int i, E elt);

Operators can be called either using operator notation:

`a[i] = x + y;`

or by using the function/procedure name:

call `_[]=_`(a, i, `_+(x, y)`);

The former is usually preferable for humans, the latter is what the compiler uses internally.

We experimented with having names for each operator – e.g., so the latter form above would be

call `setelem`(a, i, `plus`(x, y));

The trouble with this is that it is easy to forget which name corresponds to an operator. We followed the convention used by C++ STL (e.g., `divides` for `/`, `multiplies` for `*`), but still found we had to look it up too often. The underscore syntax is unambiguous, and should be easy to remember. Also, it has the advantage that it will also work with user-defined operators not foreseen by the language designers.

6.3.4 BUILT-IN / PRIMITIVE OPERATIONS

In a way, Magnolia doesn't really have any built-in operations – everything is declared in a library somewhere, including seemingly primitive types and operations for them. Their names aren't special keywords (like `int` in C), and normal overload and name resolution still applies. The compiler backend will recognise them and treat them specially (i.e., map them to primitive types in the target language). The usual primitive types (`int`, `string`, `bool`, ...) are defined in the `primitive` module, which is imported by default into all programs – though it is possible to disable this import, or override the declarations, so that `int` will refer to `myint.int` rather than `primitive.int` in your program. We may also defined multiple language flavours, with different sets of default declarations.

This leads to fewer special cases in the language design and in the compiler implementation, and also gives the conceptually clean feeling of user-defined operations being just as much (or as little) part of the language as any other operation. It also means that the programmer can get an overview of standard operations by reading the module declaring them (`primitive.mg`). This module doesn't provide definitions though, the definitions are provided by the compiler, or in an external library (implemented in C++).

A few operations can be considered pre-declared and built-in – default assignment, default construction of objects, and some tuple operations.

6.3.5 PARAMETER MODES

The parameter modes serve the purpose of directing parameter passing (by value or reference), restricting reads or writes to the parameter within the procedure, and declaring the data-flow behaviour of the procedure. The latter cannot be done in C++ – for example, you can't specify that an argument can be modified but will not be read (or, rather, will not contribute to the computation).

The parameter modes are detailed in Figure 6.2. The `observe` mode – guaranteeing that an argument will not be modified – is of particular importance when functionalising procedures and for ease of reasoning. All function parameters are by definition `obs.` This, together with disallowing procedure calls in expressions, means that the programmer (and

OBSERVE

	Formals		Actuals		Init	Notes
	Read	Write	Read	Write		
obs	yes	no	yes	–	yes	
nrm	yes	yes	yes	yes	yes	abstract value must not change
upd	yes	yes	yes	yes	yes	
out	no	yes	–	yes	yes	argument value is irrelevant
giv	no	yes	–	yes	no	is initialised upon return
del	yes	yes	yes	yes	yes	is uninitialised upon return

FIGURE 6.2: Magnolia's parameter modes.

the optimiser) can be sure that no variables are modified by Magnolia expressions.

UPDATE The reasoning behind the `update` mode is that it is usually cheaper, both time and memory-wise, to change an existing object than it is to build and return a new one. Through functionalisation and mutification, we can maintain the illusion of by-value semantics, while still getting the performance benefits of updating semantics.

OUTPUT The `output` mode is simply the same as `upd`, with the additional requirement that the procedure's computation is not affected by the value of the argument. This simplifies data-flow analysis of the program, allowing optimisations like eliminating computations when we know the result will be overwritten later on anyway.

Expressions are allowed as procedure arguments in `obs` positions. Modes that modify data require variables as arguments, and the variables must be of a mode that allows modification. Implicit conversions are also forbidden for non-`obs` arguments. This is not a problem for functions, where all parameters are treated as `obs` – the compiler will take care of making any necessary temporary variables for storing the result of expression evaluation or implicit conversion.

The current compiler only supports `obs`, `out` and `upd`. The other modes are planned, but aren't fully researched yet. The `nrm` (normalise) mode is between `obs` and `upd` in semantics – it guarantees that the *abstract value* of the argument doesn't change, but the data representation may change.

CONSTRUCTORS The value of an `out` parameter is expected to be initialised and valid. For procedures – or *constructors* – that initialise new data structures, we have the `give` mode. A `giv` argument must be uninitialised (either freshly allocated, or previously deleted), and the procedure should ensure that it is in a valid, initialised state upon return. The dual of `giv` is `delete`, which has the effect of marking the argument as uninitialised upon return.

Beyond the obvious applications in declaring constructors and destructors, `del` and `giv` may, for example, be used for operations that

translate between similar data structures. Consider an image object consisting of image data allocated on the heap and a few data items like size, bit depth, etc. Converting this data structure to the image structure expected by an external image processing library like `gd` might be done by a procedure <http://www.libgd.org/>

```
procedure convert(del myImage i, giv gdImage o);
```

which creates a new `gdImage` object reusing the heap allocated data. To avoid having two objects share the same data, the old image object is marked as deleted – even though the data isn’t actually destroyed, just moved to a different place.

Buffered I/O might be done in a similar way, with the output procedures ‘swallowing’ the data, and queueing it onto its list of things to be printed, without having to copy the data.

It is possible to adapt a procedure to behave like it is using other parameter modes automatically, by copying variables, introducing temporary variables, or extra constructor/destructor calls. This is part of what makes functionalisation possible – mutification will insert the necessary code to make a procedure behave like all its input parameters are `obs` parameters.

To make `upd` or `del` act like `obs`, we copy the argument to a temporary, which is discarded after the procedure call. To make `obs` work like `del`, we insert a call to a default destructor. Similarly, we can make `out` act like `giv` by using a default constructor call first. These conversions rely on certain operations being available – copying, default constructors and destructors (possibly no-op), and hence they may not work on all data types. For example, you can’t simply copy a network stream and expect to end up with two completely independent objects.

6.3.6 PARAMETER PASSING IN OTHER LANGUAGES

The parameter modes are distinct from *parameter passing* methods, which is what most other language designs specify. Magnolia doesn’t specify how parameters are passed from caller to callee, the compiler is free to choose the most efficient method for a given type or architecture. The idea is to be more concerned about what happens than how it happens. Knowing that some arguments are immutable (`obs`) is important when reasoning about code, particularly when relating procedures to functions – knowing whether this is implemented by copying values, or by passing a reference to a constant data structure is of little concern.

Among recent languages, C# [47] has fairly rich parameter passing methods, covering many of the same cases as Magnolia. C# types are either *value types* or *reference types*. Arguments are passed by value by default, though in the case of reference types, the reference is the value, giving an effect similar to reference passing. This is how Java works as

C#

well, except in Java only primitive types are value types (C# also allow user-defined value types `structs`). Explicit reference passing in C# is done using the `ref` keyword, which must be used both at the declaration site and the call site. Reference passing allows the value of a variable to be modified or replaced. C# also allows output parameters with the `out` keyword (which also must be given at both declaration and call site). Output parameters are like reference parameters, except they may be uninitialised, and are assumed to be initialised upon successful return of the call.

C#'s use of keywords at the call site has some appeal, and is something we will consider for Magnolia, particularly in the case of `del` and `giv`.

JAVA

Neither C# nor Java provide the equivalent of `obs` – guaranteeing that a method won't change a reference type parameter. In Java, you can declare a parameter as `final`, though this isn't part of the method signature, and can be changed when overriding the method. Protection against unwanted change is instead done through encapsulation, declaring member variables `private` so they can only be changed by methods belong to that class. The convention is that methods only change the current object (`this`), but the language doesn't enforce any protection. Determining statically whether a parameter can or can not be changed is difficult. Even having access to the method implementation may not help, since methods can be overridden through inheritance, and classes can be loaded dynamically.

ADA

Ada [146] provides `in`, `out` and `inout` modes – quite like Magnolia's `obs`, `out` and `upd`. The standard specifies how simple arguments of the different modes are passed (copy in, copy out), but it is up to the implementation to chose by-copy or by-reference for arrays and records. The choice of by-copy or by-reference makes a difference in the face of concurrency, or if an exception is raised after an `in out` argument has been modified – by-copy will show no change, by-reference will show a (perhaps undesirable) change. The Magnolia compiler currently uses by-reference to implement `upd` and `out`, and does not guarantee that arguments won't have changed if a procedure aborts due to an error – this is something we plan to deal with later.

6.4 Type System

The Magnolia type system is not fully researched, hence its features are based on necessity rather than careful design.

A Magnolia data type has a *name* (its identity) and a *representation* (its data structure). The representation is hidden, so type processing is done on names. A type name has term structure:

Simple types with just a simple name – a nullary term constructor.

Normally written without parenthesis. Example:

```

type int;           // declare simple type int
var int x;          // variable of simple type int

```

Parametrised types – a term constructor with one or more type arguments, given by a `forall` clause. Example:

```

forall type element, type index
  type array(type element, type index); // declaration
var array(string, int) a;              // use

```

Tuple types – parametrised types with an empty constructor name. The type of a tuple expression is a tuple of types. Example:

```

var (int, int, string) foo = (5, 2, "hello");

```

Tuple types are predeclared, so there is no need to declare them before use.

Type variables – formal parameters of a parametrised type, or types used in a generic operation. Type variables are scoped like normal variables. Type variables can bind to either a type or constructor name. For example, if `T`, `U` and `V` are type variables, `T(U,V)`, `array(U,V)` and `U` are legal type names. In the following two declarations, `T` is a type variable, local to each declaration:

```

forall type T
  type list(type T);
forall type T
  function T head(list(T) xs);

```

Type function – taking a number of arguments, either types or expressions, and yielding a type. Example:

```

define type first(type T, type U) = T;
define type intarray = array(int);

```

Type function applications are evaluated during type resolution, hence they do not create new types, just aliases for existing types.

Types with no variables in their name are *plain types*. Plain parametrised or tuple types are *instantiated types*, and can be thought of as a simple type with a fancy, structured name.

PLAIN &
INSTANTIATED
TYPES

6.4.1 TYPE MATCHING

Type resolution has two sub-steps; matching type names to their declarations (name resolution), and checking that the program is well-typed (type checking and overload resolution). Since operations are overloadable in Magnolia, type checking is integrated with overload resolution – you won't know the type of an expression until you know which

operation is used, and you won't know which operation is used until you know the types of the arguments. Just as type names are terms, operation names are also terms – constructed from the user-visible operation name and the parameter types.

Type checking and overload resolution is based on *type matching*, where an operation's actual arguments are matched against its formal parameters using pattern matching on the type name terms. A type τ' matches a type τ , if and only if a value of type τ' can be assigned to a variable of type τ , possibly using implicit conversions. Matching is like a unification operation, except that implicit conversions may make the matching asymmetric. A later extension of the type system to handle subtyping would also require asymmetric matching. Note that implicit conversion is illegal for arguments which must be variables (i.e., any non-obs procedure argument).

TYPE MATCHING

Type matching has the following inputs:

- A set of type variables X and an initial list of bindings σ_0
- A formal parameter type τ
- A actual parameter expression and type $e : \tau'$

For example, given the expression `f("foo", 5)` and function declaration

forall type T

function `int f(T a, int b);`

we have $X = \{T\}$, $\tau = (T, \text{int})$, $\sigma_0 = \{\}$, $e = (\text{"foo"}, 5)$ and $\tau' = (\text{string}, \text{int})$.

Matching will either succeed or fail. If it succeeds, it produces the following outputs:

- A set of type variable bindings σ
- A revised expression $e' : \tau\sigma$
- The number of implicit conversions applied

In the above example, $\sigma = \{T \mapsto \text{string}\}$, $e' = (\text{"foo"}, 5)$ and $\tau\sigma = (\text{string}, \text{int})$, with zero conversions.

The number of implicit conversions together with the number of type variable bindings is used in overload resolution to choose between multiple matching candidates.

PATTERN
MATCHING

Pattern matching is done according to the following algorithm:

1. If operating on argument lists, the lists are turned into tuples
2. We then proceed recursively:
 - A bound type variable is replaced by its binding before proceeding.
 - A plain type matches a plain type if the names match or if a single suitable implicit conversion exists.

- An unbound type variable matches any type expression and is bound to that expression.
- A type tuple matches a tuple of the same arity if their components matches.
- A parametrised type matches a parametrised type of the same name and arity if their arguments match.

After successful matching, bound type variables are checked against concept requirements.

For example, given $i : \text{int}$, $s : \text{string}$ and $x : \text{smallint}$, with an implicit conversion from `smallint` to `int`, the argument list of $f(i, s, x)$ will match the parameter list of

```
function int f(int a, string b, int c);
```

with no type variable bindings, and giving a revised expression $f(i, s, \text{int}(x))$. $f(i, i, x)$ will not match, nor will $g(i, s, x)$ match

```
function int g(smallint a, string b, int c)
```

since `int` does not match `smallint`, even though `smallint` matches `int` (due to implicit conversion). The function

```
forall type T
function T h(T a, T b);
```

has an argument list containing type variables, making the matching more interesting. The argument list (a, a) will match, with $t = \text{int}$, as will (a, x) , also with $t = \text{int}$, but yielding a revised argument list $(a, \text{int}(x))$.

6.5 Data Abstraction

Data types in Magnolia abstract over the data representation. For users of a data type, its representation doesn't matter – it is always hidden behind the operations of the type's interface. The users deal with *abstract values*, and the implementation relates those abstract values to a concrete *representation*.

ABSTRACT VALUES

Most parts of a program will operate on abstract types and abstract values. Only the implementations of core operations on a type need access to its representation. Operations where the inputs and outputs are abstract values are *abstract-level operations*, in contrast to *representation-level operations*. Note that a type's representation can – and typically will – be composed of abstract values of other types.

ABSTRACT &
REPRESENTATION
LEVELS

The implementation of a type is usually contained in a single module. One module may define multiple types – this is appropriate for closely related types. Defining a type is done by giving its representation, and defining its behaviour is done by defining operations on the type. These

abstract-level operations will then provide the interface for manipulating abstract values of the type.

6.5.1 DATA REPRESENTATIONS

STRUCT A **struct** is a compound data structure, with a representation made up of named fields of other types, similar to records or structs in other languages. Example:

```
type dog = struct {  
    var real tongueLength;  
    var string name;  
}
```

The fields of a structure are accessed using the conventional dot-notation:

```
myDog.name = "Spot";  
print myDog.tongueLength;
```

UNION A union or tagged union is data structure with named fields, only one of which can hold data at any time:

```
type pet = union {  
    var dog doggy;  
    var cat kitty;  
}
```

Unions are type-safe, and remember which field contains a value. Fields are accessed by case matching on the field name:

```
switch(myPet)  
    case doggy {  
        print doggy.tongueLength;  
    }  
    case kitty {  
        print kitty.livesLeft;  
    }
```

Unions are always *closed*, in that they can't be extended with new fields later on. Setting union field values is done using the dot notation. A data structure that dynamically keeps track of whether it has a defined value can be represented like this:

```
type maybe(type T) = union {  
    var T value;  
    var () undefined;  
}
```

OPAQUE DATA An *opaque data representation* is completely hidden, and can only be manipulated through operations defined outside Magnolia. The built-in types are declared this way; this would also be a way to interface with code from other languages. Examples:

```

type int;
type array(type E);

```

6.5.2 DATA TYPE VS. DATA REPRESENTATION

In order to implement operations on a type, we must provide a way to expose the representation to the implementation and to ensure the consistency of the representation and its relation to the abstract values.

Ensuring safe manipulation of the representation of an abstract type is done by bundling it with a *data invariant* and a *congruence relation*. The data invariant is a predicate on the representation that states which representation values constitute legal abstract values. The congruence relation states which legal representation values represent the same abstract value.

INVARIANT &
CONGRUENCE

We can have either strong or weak data invariants. A strong invariant completely describes which representation values are valid. A weak invariant is used when we are unable to write down the full invariant as a predicate. We will then have assumptions about the data representation in our implementation that we are unable to check algorithmically, and we must be extra careful when writing and modifying our implementation.

STRONG & WEAK
INVARIANTS

A strong invariant is implemented as a predicate `classInvariant`. For example, for a type `rational`, all representation values are legal except those with a zero denominator:

```

predicate classInvariant(rational a) = a.denom != 0;

```

For any operation on a data type we have that if the invariant holds for all inputs, it will also hold for all outputs. For a weak invariant, the predicate is named `dataInvariant`.

The congruence is needed because we may have multiple representations of the same abstract value. For instance, for rational numbers there are numerous representations for the abstract value “one half”: $1/2 = 2/4 = 50/100 = \dots$

CONGRUENCES

```

predicate congruence(rational a, rational b) =
  a.num*b.denom == b.num*a.denom;

```

Informally, if two values are congruent, they should be always treated the same, and we cannot distinguish between the abstract values.

The congruence predicate generates an equivalence operator `==`. Note that we may sensibly implement an equivalence that is weaker than congruence, by considering values of different types equivalent. For example, we may consider the two values `int(5)` and `real(5.0)` equivalent, but they are not congruent, since $5/2 = 2$ and $5.0/2 = 2.5$.

The data invariant and congruence predicates accept abstract values, but operate on the data representations.

Any abstract-level operation must always uphold the data invariant and the congruence. If an operation only uses other abstract-level operations, we can be assured that it is well-behaved, since it has no way of generating illegal representations or distinguish between different representations of the same abstract value. For representation-level operations, we must be more careful, and keep track of when an object is considered an abstract value, and when it is an instance of a concrete representation.

OPENING THE REPRESENTATION

To distinguish between an abstract type t and its representation, we name the representation $t\$$. If we have a variable of type t , we can obtain its representation by *opening* it:

```
var t x;  
open(x) {  
  // code manipulating x's representation  
}
```

Inside the `open` construct, we only have access to x 's representation, not the abstract value of x . We may manipulate x by manipulating its components, or by using operations on $t\$$. But we may not use any operations on t , since we do not know that the data representation is consistent.

Opening a variable implies that we swear to uphold the data invariant and congruence of the variable's type. For any procedure that uses the `open` construct, we automatically get axioms to that effect, which may be tested using axiom-based testing. For example, given a procedure

```
procedure _+(rational a, rational b) {  
  open(a, b) { ... }  
}
```

we get the axioms

```
axiom DI(rational a, rational b) {  
  if(classInvariant(a) && classInvariant(b))  
    assert classInvariant(a + b);  
}  
axiom DQ(rational a, rational b, rational a', rational b') {  
  if(congruence(a, a') && congruence(b, b'))  
    assert congruence(a+b, a'+b');  
}
```

We will note in the procedure declaration whether it uses `open`:

```
procedure _+(rational a, rational b) opens(a, b);
```

The compiler infers this automatically, but it is useful in the code for documentation. If the code contains no `open` construct for a variable mentioned in the `opens` clause, this is treated as a shorthand for opening

the variable in the entire procedure body. For example, the above `_+_` may be equivalently written:

```
procedure _+_ (rational a, rational b) opens(a, b) {
  ...
}
```

We only allow opening of parameters – not local variables – since we are only able to generate the necessary axioms at the operation level (though we could certainly check data invariants by inserting assertions before and after opening local variables).

Helper Operations

It is sometimes useful to break an operation up into several representation-level helper operations that operate on the data representation, without any guarantees about maintaining invariants and congruence. This is done using the representation name as the parameter type:

```
procedure normalise(upd rational$ a) {
  a.num = a.num/gcd(a.num, a.denom);
  a.denom = a.denom/gcd(a.num, a.denom);
}
```

We can also define operations that open or close a type. Presuming we have a `rational` representation in which the numerator and denominator should be as small as possible:

```
procedure box(upd rational$ a rational) {
  call normalise(a);
}
```

The procedure `box` updates its parameter `a`, and before the update it will have the representation type `rational$`, afterwards it will have the abstract type `rational`. It has the effect of closing the type, by ensuring that the data is in a normalised form, as expected by the data invariant.

Similarly, we could define `unbox` which opens the type:

```
procedure unbox(upd rational a rational$) {
}
```

The `open` construct will then have an effect similar to:

<pre>open(a, b) { ... }</pre>	≈	<pre>var a' = unbox(a); var b' = unbox(b); ... // with [a -> a', b -> b'] a = box(a'); b = box(b');</pre>
--	---	---

6.5.3 ACCESSING FROM OUTSIDE THE DEFINING MODULE

Normally we'll want to limit access to a types representation to operations defined within the same module. Even within the same module, the representation isn't exposed by default – we must explicitly open a variable to get access to its representation.

For types that are protected with a strong `classInvariant` we may not have to be so careful about guarding the implementation secrets. We can allow any operation in any module to open the data representation – as long as the operation swears to uphold the invariant and congruence. With invariant and congruence axioms, we can test whether the operation keeps its promise. If we are suspicious of out-of-module access to representations, we can have the compiler insert invariant checks after calls to such operations.

Incompatible changes to the representation will cause either compilation errors (when fields are no longer available) or errors that can be detected by testing. It is also easy for the compiler to figure out where out-of-module access to a representation occurs, so that the programmer can be aided in tracking down code that must be changed after a refactoring.

With a weak invariant, it is safer to keep the representation completely hidden within a single module, since any refinement of the representation may cause subtle incompatibilities which we won't be able to uncover by testing.

Note that even though out-of-module access may be *safe*, it is not necessarily a good idea from a software maintenance point of view.

6.5.4 CONSTRUCTING OBJECTS

Constructing a new object at the representation level is done with a constructor expression, giving the type name and a list of field values. All the fields of a `struct` must be given a value at construction time, and only one field must be given for a `union`. Example:

```
var myDog = dog${name := "Spot", tongueLength := 4.5};
```

It is also possible to define *constructor procedures* that create a new object in a `giv` parameter. This is preferred for non-trivial object construction. As with the constructor expressions, all the fields of the object must be initialised, by assignment or by calling appropriate constructors:

```
procedure dog(giv dog d, obs string n, obs string l) {  
  d.name = n;                               // by copy  
  call int(giv d.tongueLength, l); // by constructor call  
}
```

Any `giv` parameter is implicitly open at the start of the procedure, and implicitly closed at the end.

Constructor expressions can be interpreted as a call to a functionalised version of a default constructor:

```
procedure new_dog(giv dog d, obs string name,
                 obs int tongueLength);
```

Note that at the expression level (the algebraic language), there is no difference between constructing an object and returning an object – a constructor is just another function. For trivial types, a simple function may be sufficient to give an abstract-level interface to object construction:

CONSTRUCTION VS.
RETURN

```
function dog dog(string n, int l) = dog${name = n,
                                         tongueLength = l};
```

6.6 Expressions and Statements

Expressions in Magnolia can consist of variables, literals and function or operator applications. Expressions are always typed, with type resolution proceeding from the innermost sub-expression to the outermost (hence, overloading operations on the return type is not possible).

As explained in Section 6.3.1, expressions should always be side-effect free, with no data being modified, making Magnolia expressions more like expressions in functional languages than C or Java expressions. In particular the increment / decrement operators ++ / -- of C-like languages are unavailable in Magnolia.

In addition to procedure calls, Magnolia has control-flow statements such as **if** and **while**, similar to those found in C-like languages. Variables are introduced with the **var** declaration statement:

```
var foo = f(5);
var string bar;
```

The types of new variables can be omitted if the type can be inferred from the initialiser. With no initialiser, the variable type must be given explicitly.

6.7 Concepts and Axioms

Concepts serve two roles in Magnolia:

1. Allowing specifications in the form of axioms to be embedded in a program, and related to implementations using model declarations, and
2. constraining type parameters, so that only types supporting required operations are accepted.

Our research has mostly focused on the first aspect, and how axioms can be used for testing and optimisation. However, it is the second role

that was the motivation behind the proposal to add concepts to C++ [16], and also for similar features like Haskell type classes [71]. The design of concepts in Magnolia is far from complete, so we will only discuss them briefly here. Our first design was more or less a straight copy from C++, but given that our motivations are different and that we do not have to maintain compatibility with large amounts of existing code, it seems likely that the final design will be quite distinct from the C++ version.

As we saw at the beginning of this chapter, a concept consists of a signature and a set of axioms, and can include concept constraints on types in the signature. For example, this is an excerpt of a concept for integers:

```
concept Integer(type int) {  
  requires Equivalence(int);  
  function int _+(int a,int b);  
  function int _-(int a,int b);  
  function int _*(int a,int b);  
  function int _/(int a,int b) guard by b != int(0) ;  
  ...  
  axiom commutative_add(int a, int b) {  
    assert a + b == b + a;  
  }  
  axiom commutative_mult(int a, int b) {  
    assert a * b == b * a;  
  }  
  ...  
}
```

The `int` here is
entirely distinct from
the primitive type `int`.

The concept has one type, `int`, and lists some operations for it. There is one constraint given by the `requires` clause – that the type `int` models the *Equivalence* concept, basically stating that there should be an equivalence operation available on ints. Two axioms are shown, specifying commutativity for the addition and multiplication operators.

CONCEPT
REFINEMENT

Concepts can also be built on other concepts – for example, we could build *Integer* on top of *CommutativeRing*, which already contains the addition and multiplication operators, and the commutativity axioms for them. We can build many algebraic structures this way – Gottschling [60] provides a collection of fundamental algebraic concepts for concept-enabled C++.

MODELS

A `model` declaration states that some implementation types model a given concept – for example, that `myint` and primitive `ints` are integers:

```
model Integer(myint);  
model Integer(primitive.int);
```

Upon encountering the model declaration, the compiler will check that all the operations in the concept's signature are implemented for the

given types, and that the types also model any concepts mentioned in a **requires** clause. This corresponds to a concept map in C++. The axioms are not checked, as program verification is beyond the capabilities of the compiler. However, axiom based testing (Chapter 4) can be used to test the implementation.

The model declaration may optionally provide implementations for some operations, which is useful if the existing implementation does not fully model the concept, or if some operations in the implementation must be renamed to match the concept. For example, if our imaginary `myint` lacks equivalence, we could add one in the model declaration (assuming we can check if a number is zero):

```
model Integer(myint) {
  function bool _==(myint a, myint b) == isZero(a - b);
}
```

6.8 Genericity

Any operation, data structure or module with a type parameter is *generic*. Type parameters are given by the **forall** clause. Generic code can't be executed until we know what its arguments are – we'll typically need to know the size of data structures, and the signature of any operations called from the generic code. The former will vary, since Magnolia isn't purely object reference based, and the latter is hard to determine until concrete types are known, due to overloading.

Genericity can be handled either at compile-time or run-time. For the most part, Magnolia handles genericity by *instantiating* the generic code at compile-time, producing non-generic code. This has the advantage of simplifying the run-time system and allowing the instantiated code to be specialised for greater performance. The disadvantage is that program that rely a lot on generic code can grow quite large when all the instantiations are added.

INSTANTIATION

A generic data type has one or more parameters. Any use of the data type with non-generic arguments will trigger an instantiation. Only one instantiation is done for each distinct set of arguments. For example, the data type `stack` has an element type as a type parameter:

```
forall type elt
type stack(type elt) = {
  var array(elt) data;
  var int top;
}
```

Declaring a variable '`var stack(int) s`' will cause the type `stack(int)` to be instantiated, which in turn will trigger the instantiation of `array(int)`.

The compiler will think of these as plain structure types, except that the name is a bit funny. Instantiating the type does nothing to any operations defined on the type – this is done solely on the basis of calls to those operations.

Instantiation of generic operations is triggered whenever overload resolution returns with one or more type variable bindings. The algorithm is straight-forward (and is also used for generic types) – a non-generic signature is generated by substituting the type variables according to the bindings. If the resulting signature identifies an as-yet undefined operation, the definition of the operation is looked up and type variables in its body are substituted, producing an instantiated definition. To help further processing in the compiler, the instantiation is annotated with information of its generic source and the generic arguments. It is then added to the compiler's list of defined operations, just as if it had been defined directly in the source text.

There are two small complications that make the instantiation process not quite that simple: The first is that any generic type or operations used within an instantiation will also need to be instantiated. The second is that whenever a procedure is called through its functionalisation, we need to instantiate the procedure itself, and not just the functionalisation. To handle this, a second pass of semantic analysis can be applied to the instantiated code, recursively triggering instantiations of any generic users therein. Functionalised procedures are handled by making the function an internal compiler primitive representing “functionalisation of ...”, thus making it easy to find the corresponding procedure.

Note that type checking can be done without instantiation, since concept requirements will provide information about which operations are valid for various type parameters. In fact, instantiation can be seen as part of the compiler backend, and may not always be necessary as long as the generic code can be translated to C++ template code. However, doing instantiations early means that the instantiated code can be subjected to heavy optimisation.

6.9 Language Implementation

COMPILATION BY
TRANSFORMATION

Magnolia is implemented using *compilation by transformation*, using a sequence of transformation steps to transform code in the source language to a target language (object code, or another programming language). The Magnolia compiler is implemented using the Stratego/XT [26] transformation framework, where transformations are implemented using rewrite rules and strategies.

The overall design of the compiler follows the typical pipeline-like compiler structure, with various phases dealing with different aspects of

compilation. Language extension needs to take this into consideration, since a single extension will typically touch several phases of the compiler.

A ‘phase’ may not correspond to a single pass over the program, as multiple phases may be applied in sequence to a smaller part of the program, and a single phases may possible make multiple passes over the entire program.

6.9.1 FRONTEND

The parser is based on a grammar written in `SDF2` [154] and uses the `sglr` scanner-less GLR parser [151], which produces an abstract syntax tree in `ATerm` format [150]. New syntax can be added by extending the grammar. As `SDF2` is modular, this may be done without touching the existing grammar, simply by telling the compiler to use an alternate parse table, or to build a new parse table with an extra module included.

Although GLR grammars are theoretically composable, in that you don’t get the various conflicts that commonly occur with LALR parsers, adding a language syntax extension may result in an ambiguous grammar – one that produces more than one tree for a given input. There is no quick fix for this, checking that a context-free grammar is unambiguous is in general undecidable [28; 49]. Several approaches get around this by restricting the grammar formalism, for example, one may attempt to build an `LR(k)` parse table for the grammar, and check for conflicts [97; 130]. Schwerdfeger and Van Wyk [131] propose a system where language extensions can be checked for composability individually, which is important since it allows separately developed libraries with syntax extensions to be combined. Another approach may be testing by generating random strings in the language and see if any ambiguities occur.

SYNTAX EXTENSION

To provide syntax extensibility of the kind found in languages like Dylan [133], one could provide Magnolia syntax for syntax definition, then extract and compile the syntax definitions to `SDF2`. We’ll not bother too much with syntax extension though, but we will revisit language extensions in Chapter 7.

After parsing, the program code passes through desugaring, which normalises the abstract syntax tree. Some constructs that are syntactically dissimilar but semantically similar will be folded into the same representation here – for example, operator and function calls are basically the same thing, but with special prefix, postfix or infix syntax for operators. They are all folded into the same *function call* construct.

6.9.2 MAIN COMPILER PHASES

The main phases of the compiler may be invoked multiple times on smaller parts of a program, and one phase may call another phase as

needed. Overall, the phases proceed in the other listed:

Semantic Analysis: Expressions in the program are typed and the types of sub-expressions are used to resolve overloaded calls. Statements are also type-checked – for example, assignment requires that the variable being assigned to is writable, and of a type compatible with the value being assigned.

Concept Configuration: Generic code is typically written in terms of concepts, where multiple implementations can satisfy the same concept. Configuration replaces concepts with implementations and instantiates any axioms and default code provided in the concept.

Instantiation: Generic code and macro-like code is instantiated into concrete code. Instantiation is triggered by use (detected during semantic analysis), and may trigger further instantiation and application of semantic analysis to instantiated code.

Functionalisation: Translates statement-based code using pure procedure calls into expression-based code. This is sometimes desirable, because it makes code a bit easier to process, e.g. by expression rewrite rules. We may also do inlining of assignments to obtain as deeply nested expressions as possible – the effect can be reverse afterwards using common compiler optimisation techniques and mutification.

High-level Optimisation: The transformations done here are based on high-level semantic knowledge of the program – obtained from axioms, for example, or from user-provided transformations. Data-flow analysis is needed to track the propagation of properties. For example, knowledge of whether an array is sorted or not can be used to choose binary search over linear search. Sorting an array would set the property, inserting an arbitrary element would destroy it, and a rewrite rule to select binary search over linear search would check the property in its condition. This idea is explored further by Kalleberg [91].

Mutification: Translates expression-based code using pure function calls into statement-based code using update-based procedure calls. Needed if functionalisation (below) is used, and also on typical user code, since many functions are implemented as procedures. Mutification and functionalisation are described in more detail in Chapter 2.

Slicing: Removes dead code and specialises operations for particular outputs.

Various optimisations: Inlining, constant folding, code specialisation. These are called as needed from other phases and may also be applied in a separate optimisation phase.

6.9.3 BACKEND

The backend compiles away any remaining Magnolia-specific constructs and produces C++ source code, ready to be fed to a C++ compiler. The compiler normally operates as a whole-program compiler, producing one big C++ source file. Alternative backends for C and for special-purpose languages/libraries like Cuda and MPI are planned sometime in the future.

The code that is fed to the backend is actually close enough to C++ that only a few transformations are necessary. The C++ code is produced by a version of the Magnolia pretty-printer adapted to C++ syntax.

6.9.4 MODULES

A Magnolia program is represented internally as a tree (or term, depending on your point of view). A program typically consists of a main module, which imports several other modules (possibly from libraries), defining concepts and implementations. Some code may be implemented in another language (e.g., C++) with a module defining only the interface to the external code. All modules are loaded into the compiler during semantic analysis.

Imported modules are also subjected to semantic analysis. The compiler will store processed imports in the file system as trees to speed up later imports of the same module – modules are automatically reprocessed as needed if the source changes.

Although separate compilation would be possible, Magnolia uses whole-program compilation because we want to be able to aggressively inline and optimise code across module boundaries.

In the compiler, individual definitions (operations, types, etc.) are assigned an identification, *def-id*, and properties of a definition can be looked up based on the *def-id*. Uses of a definition are annotated with the *def-id* during semantic analysis, for example, a function call will be represented as a *def-id* together with a list of arguments – function body, return type and all other information can be found using the *def-id*.

6.10 Status of the Magnolia Implementation

Most basic aspects of the language are fairly complete. We have concrete and abstract data types, and operations on them in the form of functions and procedures. There is a module system, and we have generic types and operations, full overloading of type and operation names (including inference of generic arguments), the usual selection of statements, and a library of primitive types.

IMPLEMENTED
FEATURES

Functionalisation and mutification, as described in Chapter 2 is fully implemented, including slicing of multi-valued procedures for use as single-return functions. There are, however, more possibilities to explore here. Being able to convert a program to a form which uses only function

calls (as far as possible) will likely be useful for some optimisations. Slicing can be combined with other forms of specialisation to create tailored versions of performance critical operations. Finding the right balance between specialisation, inlining and code duplication would be important issues here.

Although it works for trivial examples like integers, the concept implementation is not mature enough for larger experiments with concept-based development. As this is one of the main areas we would like to apply Magnolia, finalising the design and implementation of concepts has high priority in the further development of Magnolia.

Supporting Language Extensions

As Magnolia is intended for experimentation in language design, it is important to be able to easily add language extensions and new language variants. This chapter describes an experimental meta-programming extension to Magnolia, designed to make it possible to do language extension without having to modify the Magnolia compiler.

The existing abstraction mechanisms in Magnolia already provide ways for the programmer to add to the vocabulary of the language. Language extension is simply another way to either add abstractions that behave similarly to those we already have, or to add new kinds of abstractions. Extended abstractions should, as far as possible, be integrated with features like mutification, functionalisation and axioms, so they behave seamlessly for the programmer, and can be subjected to high-level optimisation.

Although language extension was a very popular research topic in the sixties and early seventies [140], this line of research was mostly abandoned in favour of abstraction and object orientation. In recent years, however, interest in extensible languages and programming environments has surged [157], with, e.g., extensible compilers [48] and intentional programming [137].

We're not looking to make the ultimate extensible language, but rather to support those language experiments we want to perform – language extension itself being one of them.



"I See What You Mean" – The Big Blue Bear at Colorado Convention Center.
Denver, CO (GPCE / SLE '09)

Yet Another Language Extension Scheme

ANYA HELENE BAGGE

Department of Informatics, University of Bergen, Norway

ABSTRACT

Magnolia is an experimental programming language designed to try out novel language features. For a language to be a flexible basis for new constructs and language extensions, it will need a flexible compiler, one where new features can be prototyped with a minimum of effort. This paper proposes a scheme for compilation by transformation, in which the compilation process can be extended by the program being compiled. We achieve this by making a domain-specific transformation language for processing Magnolia programs, and embedding it into Magnolia itself.

7.1 Introduction

Implementing a compiler for a new programming language is a challenging but exciting task. As the language design evolves, the compiler must be updated to support the new design or to prototype the design of new features. Magnolia is both an experimental programming language, and a language for language experiments. We therefore need a compiler flexible enough to keep up with changes in the language design, and with features that make implementation of experimental features easy.

Use cases for a language extension facility include experimental features such as data-dependency based loop statements, embedding of domain-specific languages, restriction to sub-languages with stricter semantics and language implementation using a simple core language, and building the rest as extensions.

This is a preprint of: Bagge, A. H. Yet Another Language Extension Scheme. In Proceedings of the 2nd International Conference on Software Language Engineering (Denver, Colorado, USA, October 5 – 6, 2009). SLE '09. To appear in *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg. Final version may differ from the text presented here. Printed by permission.

In Magnolia, the programmer can express extra knowledge about abstractions as *axioms*. In the compiler, we would therefore like to preserve abstractions for as long as possible, in order to take advantage of axioms. Language extensions also provide abstractions, with knowledge we may also want to take advantage of. Desugaring extensions to lower-level language constructs at an early stage, as is done with syntax macros, discards any special meaning associated with the constructs, which could have been used for optimisation and extension-specific error checking.

The Magnolia compiler is implemented in Stratego/XT [26], using compilation by transformation, where a sequence of transformation steps transform code in the source language to a target language (object code, or another programming language). It is therefore natural to make use of transformation techniques for describing language extension. This paper presents an extension of the Magnolia language with transformation-based meta-programming features, so that extensions to the Magnolia language can be made in Magnolia itself, rather than by extending the Stratego code of the compiler. This gives more independence from the underlying compiler implementation.

The rest of this paper is organised as follows. First, we give a brief introduction to the Magnolia language, before we look at how to add language extension to it (Section 7.3). We have two extension facilities, macro-like *operation patterns* (Section 7.3.1) and low-level *transforms* (Section 7.3.2). We provide an example of two extensions, before discussing related work Section 7.3.6 and concluding (Section 7.4)

7.2 The Magnolia Language

We will start by briefly introducing the parts of Magnolia that are necessary to understand the rest of the paper. Magnolia is designed as a general-purpose language, with an emphasis on abstraction and specification. Abstractions are described by *concepts*, which consist of abstract types, operations on the types, and axioms specifying the behaviour of the operations algebraically. Multiple *implementations* may be provided for each concept, and *signature morphisms* may be used to map between differences in concept and implementation.

Operations can be either *procedures* or *functions*. Procedures are allowed to update their parameters, and have no return values. Pure procedures only interact with the world through their parameters (e.g., no I/O or global data). Functions may not change their parameters, and are always pure – the only effect a function has is its return value, and it will always produce the same return value for the same arguments. Function applications form expressions, while procedure calls are statements. In addition, Magnolia has regular control-flow statements like `if` and `while`.

A novel feature (detailed in a previous paper [8]) is the special relationship between pure procedures and functions. Procedures may be called as if they were functions – the process of *mutification* turns expressions with calls to functionalised procedures into procedure call statements. An expression-oriented coding style is encouraged. Procedures are often preferred for performance reasons, while expressions with pure functions are easier to reason about, and is also the preferred way of writing axioms.

7.3 Extending Magnolia

At least four types of useful extensions spring to mind:

1. Adding new operation-like constructs, that look like normal functions or procedures, but for some reason cannot or should not be implemented that way – for example, because we need to bypass normal argument evaluation, or because some of the computation should be done at compile time. This type of change has a local effect on the particular expressions or statements where the new constructs are used, and is similar to syntax macros in other systems.
2. Adding new syntax to the language, in order to make it more convenient to work with. We may also consider removing some of the default syntax. In Magnolia, this can be handled by extending the `SDR2` grammar of the language.
3. Disabling features or adding extra semantic checks to existing language constructs. This can be used to enforce a particular coding style, to disable general-purpose features when making a DSL embedding, or to ensure that certain assumptions for aggressive optimisation holds.
4. Making non-local changes to the language – features requiring global analysis, or touching a wide selection of code. Cross-cutting concerns in aspect orientation are an example of this. We can implement this by extending the compiler with new transformations and storing context information across transformations.

In a syntax macro system, new constructs are introduced by giving a syntax pattern and a replacement (or *expansion*). In languages like Lisp or Scheme, the full power of the language itself is available to construct the expansion. For Magnolia, things are a bit more complicated, since the extension may pass through several stages of the compiler before it is replaced by lower level constructs. We must therefore provide the various compiler stages with a description of how to deal with the language extension.

SYNTAX MACROS

To provide syntax extensibility of the kind found in languages like Dylan, one could provide Magnolia syntax for syntax definition, then extract and compile the syntax definitions to SDF2, as used in the compiler. We will not consider this here, however. A full treatment of compiler extension in Magnolia is also beyond the scope of this paper, we will therefore focus the macro-like *operation patterns* and briefly sketch the *transform* interface to compiler extension.

7.3.1 OPERATION PATTERNS

An *operation pattern* is a simple interface to language extension, similar to macros in Lisp or Scheme. Patterns are used in the same way as a normal procedure or function, but is implemented using instantiation with arbitrary code transformation. They are useful for things that need to process arguments differently from normal semantics.

The implementation of an operation pattern looks like a procedure or function definition, except that one or more of its parameters are meta-variables that take expression or statement terms, rather than values or variables. The argument terms and pattern body may be rewritten as desired by applying transforms to them (see examples below). When the operation pattern is instantiated, meta-variables in the body are substituted, and any transformations are applied. The resulting code is inlined at the call site.

Meta-variables are typed and are distinguished from normal variables through the type system, thus it is not necessary to use anti-quotation to indicate where meta-variables should be substituted. Operation patterns introduce a local scope, so local variables will not interfere with the call context.

The semantic properties (typing rules, data-flow rules, etc.) of an operation pattern are handled automatically by the compiler, and calls to operation patterns are treated the same as normal operation calls during type checking and overload resolution. This means that they can be overloaded alongside normal operations, and follow normal module scoping and visibility rules. Processing code with operation pattern calls requires some extra care, so that arguments that should be treated as code terms won't get rewritten or lifted out of the call.

Operation patterns can also conveniently serve as implementations of syntax extensions, by desugaring the syntax extension into a call to the pattern.

For example, the following operation pattern implements a simple way to substitute a default value when an expression yields some error value:

```
forall type T  
procedure default(T e, T f, expr T d, out T ret) {
```

```

ret = e;
if(ret == f)
  ret = d;
}

```

The `f` is the failure value (null, for example), `d` is the default replacement, and `e` is the expression to be tested.

Magnolia will automatically provide a function version of it:

```

forall type T
function T default(T e, T f, expr T d);

```

which we can use like:

```

name = default(lookup(db,key), "", "Lucy");

```

We can describe the behaviour of `default` by axioms, for example:

```

forall type T
axiom default1(T e, T f, T d) {
  if(e == f) assert default(e, f, d) <-> d;
  if(e != f) assert default(e, f, d) <-> e;
  if(f == d) assert default(e, f, d) <-> e;
  if(f != d) assert default(e, f, d) <!--> f;
}

```

7.3.2 TRANSFORMS

For further processing of language extension, we add a new meta-programming operation to Magnolia – the `transform` – corresponding to a rule or strategy in Stratego. Transforms work on the term representation of a program, taking at least one term plus possibly other values as arguments, and returning a replacement term. Provided semantic analysis has been done, term pattern matching in transforms are sensitive to typing, overloading and name scoping rules.

A transform may call other transforms and operations, and may also manipulate symbol tables and other compiler state. Several transforms can share the same name; when applied they are tried in arbitrary order until one succeeds. In addition to explicit calling, transforms can also be controlled through transform classes, which describe how and (possibly) when transforms should be applied. For example, a transform may have the classes *innermost* and *during(desugar)*, signifying that it should be applied using an innermost strategy during the desugaring phase of the compiler.

A sample transform is:

```

forall int i1, int i2, int i3
transform example(expr i1 * i2 + i3 * i2) [simplify,repeat]
  = (i1 + i3) * i2;

```

Traversals/modifiers		Compiler Phases		Uses
repeat	Can be used repeatedly	during(p)	apply during <i>p</i>	typecheck
once	In traversal: Apply only once	before(p)	apply before <i>p</i>	simplify
frontier	In traversal: Stop on success	after(p)	apply after <i>p</i>	mutify
topdown	Traversal type	requires(p)	run <i>p</i> first	ac
bottomup	Traversal type	triggers(p)	run <i>p</i> after	
innermost	Innermost reduction			
outermost	Outermost reduction			

TABLE 7.1: Transform classes: Topdown and bottomup traversals can be modified by repeat, once or frontier. The phase classes can be used to apply a transform before, during or after a particular compiler phase, or to trigger application of a compiler phase. Transforms can also be classified by use – for example, simplification transforms may be marked as such and used many places in the compiler. The **ac** class can be used to reorder expressions for associative-commutative matching.

This example has a pattern with three meta-variables, *i1*, *i2*, *i3*, all of which will match only integer expressions. The expression pattern in the argument list will be matched against the code the transform is applied to, and will only match the integer versions of *+* and ***. If the match is successful, the code is transformed to $(i1 + i3) * i2$. The transform classes *simplify* and *repeat* tell the compiler that this rule can be applied during program simplification, and that it will terminate if applied repeatedly. Table 7.1 shows a few different transform classes. Axioms, when used as rewrite rules, can also have classes assigned to them, making them usable as transforms [7].

Transforms can be applied directly in program code (most useful inside operation patterns). For example,

```
var x = example(a * b + c * b);
```

will apply the above transform (the expression to the left is implicitly passed as the first parameter) and rewrite the code to:

```
var x = (a + c) * b;
```

The double-bracket operator `[[...]]` can be used to apply inline rewrite rules, and to specify traversals – we’ll see examples of this later.

7.3.3 SEMANTIC RULES

Semantic analysis rules are described by the *typecheck* transform, which takes a statement, expression or declaration as argument, and returns a resolved version of its argument – and its type, in the case of an expression. Resolving means annotating each use of an abstraction with a unique identifier that leads back to its declaration – this is typically taken care of internally in the compiler. Type checking of a declaration will typically involve adding declarations to the symbol table; type checking other

constructs is typically a simple case of recursively type checking sub-constructs. A (simplified) typecheck rule for assignment statements is:

```
forall name x, expr e
transform typecheck(stat{x = e;}) = stat{x = e';}
where {
  var (e', t) = typecheck(e);
  if(!compatible(typeof(x), t))
    call fail("Incompatible types in assignment");
}
```

Axioms [7] can describe the abstract semantics of a construct. This is only applicable to expression-like constructs at the moment, we should also have a way of describing other constructs.

Implementation rules are used to compile constructs to lower-level code. *Instantiation rules* are triggered during semantic analysis, and receive the unique id of the abstraction and the use case, and produce an instantiated version. Other implementation rules are free-form and should be tied to a program traversal strategy and compiler phase. No effort is made on the part of the compiler to ensure that implementation rules don't leave behind uncompiled constructs, though we are looking at techniques that can handle this [3].

Other compiler phases may also need rules – for example, doing data-flow analysis and program slicing requires information about which variables are read and written in a statement – the *readset* and *writeset* transforms are used for this purpose. Transforms may also be provided for mapping between statement and expression forms.

By keeping track of semantic information, we can make more powerful extensions. For example, with the following extended version of **default** a failure value is no longer needed – it is obtained automatically from a function declaration attribute:

```
forall type T
function T default(expr T e, expr T d) =
  default(e, getAttr("fail_value", e), d);
```

This provides a primitive version of the *alerts* feature from Chapter 5.

7.3.4 MODULE-LEVEL AND GLOBAL EXTENSIONS

Language extension should normally be done at the module level, so that some modules in your program may use the extension, and others won't. For example, if your extension defines a restricted subset of Magnolia with some DSL features, you probably still want the compiler to process Magnolia libraries as if they were written in normal Magnolia. Therefore, Magnolia extensions have scope:

- The *names* of transforms and operation patterns are accessible in the module in which they are defined and in modules that import them, just as with other operations.

- Transforms are normally applied to the whole program. Semantically aware term pattern matching ensure that only relevant parts of the code are touched, not code that merely looks similar to what is described by the pattern.
- For syntax extensions and language-changing transforms that should only be applied to certain modules, there is a `language` declaration in the module header that can be used to import extension modules. Transforms imported via `language` are only applied to the local module.

7.3.5 EXAMPLE EXTENSIONS

We will give two example extensions, one which uses transforms to enforce a restriction on the language, and one which uses operation patterns to add a `map` construct.

PURE MAGNOLIA

Impure procedures are ones that violate the assumption that two calls with equivalent inputs give equivalent results. I/O is typically impure, a random generator that keeps track of the seed would also be impure. Since pure code is easier to reason about, we might want to have a sub-language of Magnolia where calls to impure code is forbidden. We implement this in a module `pure`, which is used by putting `language pure` in the module header of pure modules. Our language module contains the following transform:

```
transform purity(stat{call p(_*)}) [after(typecheck)]  
where {  
  if(getAttr("impure", p))  
    call error("In call to ", p, " -- impure calls forbidden");  
}
```

The transform `purity` will be applied to the code in all `language pure` modules after type checking is done (since the type checker might be used to infer impurity), and will match procedure calls. If the called procedure has the `impure` attribute, a compiler error is triggered.

MAP EXPRESSIONS

The *map* operation applies an operation element-wise to the elements of one or more indexable data structures (arrays, for example). Our `map` works on multiple indexables at the same time (like Lisp's `mapcar`), without the overhead of dealing with a list of indexables at runtime. For example,

```
A = map(@A * @B + @C); // map *,+ over elements of A, B, C  
A = map(@A * 5);       // multiply all elements of A by 5  
A = map(@A * V + @C);   // V is indexable, but used as-is
```

While `map` in Lisp and functional languages traditionally takes a function (or lambda expression) and one or more lists as arguments – we will

instead integrate everything as one argument, making it look more like a list comprehension. Indexables marked with an @-sign are those that should have element-wise. The @ is just a dummy operator, defined as:

```
forall type A, type I, type E where Indexable(A, I, E)
function E @_(A a);
```

This function is generic in E (element type), A (indexable/array type) and I (index type) – together, these must satisfy the *Indexable* concept. Applying the @-operator outside a map operation will lead to a compilation error – this should ideally be checked for and reported in a user-friendly manner.

A generic implementation of map is:

```
forall type A, type I, type E where Indexable(A, I, E)
procedure map(expr E e, out A a) {
  // define index space as minimum of input index spaces
  var idxSpace = min(e[[collect,frontier: @x:A -> indexes(x)]]);
  call create(a,idxSpace); // create output array
  for i in indexes(a) { // do computation
    a[i] = e[[topdown,frontier: @x:A -> x[i]]];
  } }
```

The implementation accepts an expression *e* (of the element type) and an output array *a*. The body of **map** is the pattern for doing maps, and this will be instantiated for each expression it is called with by substituting meta-variables and optionally performing transformations. Note that the statements in the pattern are not meta-level code, but templates to be instantiated. The `[[...]]` code are transformations which are applied to *e* – the result is integrated into the code, as if it had been written by hand. The first transformation uses a **collect** traversal, which collects a list of the indexables, rewriting them to expressions which compute their index spaces on the way. This is used in creating the output array. The computation itself is done by iterating over the index space, and computing the expressions while indexing the @-marked indexables of type *A*. The **frontier** traversal modifier prevents the traversal from recursing into an expression marked with @ – in case we have nested maps.

As an example of **map**, consider the following:

```
Z = map(@X * 5 + @Y);
```

where *X* and *Y* are of type `array(int)`. Here **map** is used as a function – the compiler will *mutify* the expression, obtaining:

```
call map(@X * 5 + @Y, Z);
```

At this point we can instantiate it and replace the **call**, giving

```
var idxSpace = min([indexes(X), indexes(Y)]);
```

```
call create(Z,idxSpace);  
for i in indexes(Z) {  
    Z[i] = X[i] * 5 + Y[i];  
}
```

which will be inlined directly at the call site.

Now that we have gone to the trouble of creating an abstraction for element-wise operations, we would expect there to be some benefit to it, over just writing for-loop code. Apart from the code simplification at the call site, and the fact that we can use `map` in expressions, we can also give the compiler more information about it. For example, the following axiom neatly sums up the behaviour of `map`:

```
forall type A, type I, type E where Indexable(A, I, E)  
axiom mapidx(expr E e, I i) {  
    map(e)[i] <-> e[[topdown,frontier: @x:A -> x[i]]];  
}
```

applying `map` and then indexing the result is the same as just indexing the indexables directly and computing the map expression. Furthermore, we can also easily do optimisations like map/map fusion and map/fold fusion, without the analysis needed to perform loop fusion.

7.3.6 RELATED WORK

There is a wealth of existing research in language extension [23; 140; 157] and extensible compilers [48; 116], and there is not enough space for a comprehensive discussion here.

Lisp dialects like Common Lisp [61] and Scheme [46] come with powerful macro facilities that are used effectively by programmers.

C++ templates are often used for meta-programming, where techniques such as expression templates [153] allow for features such as the map operation described in Section 7.3.5 (though the implementation is a lot more complicated).

Template Haskell [135] provides meta-programming for Haskell. Code can be turned into an abstract syntax tree using *quasi-quotation* and processed by Haskell code before being *spliced* back into the program and compiled normally. Template Haskell also supports querying the compiler's symbol tables.

MetaBorg [24] provides syntax extensions based on Stratego/XT. Syntax extension is done with the modular SDF2 system, and the extensions are desugared ("assimilated") into the base language using concrete syntax rules in Stratego.

Andersen and Brabrand [3] describe a safe and efficient way of implementing some types of language extensions using catamorphisms that map to simpler language constructs, and an algebra for composing

languages. We plan to integrate this work with ours as a safe way of doing simple extensions.

We aim to deal with semantic extension rather than just syntactic extension provided by macros. We do this by ensuring that transformations obey overloading and name resolution, by allowing extension of arbitrary compiler phases, and allowing the abstract semantics of new abstractions to be described by axioms. The language XL [108] provide a type macro-like facility with access to static semantic information – somewhat similar to operation patterns in Magnolia.

7.4 Conclusion

In this paper we have discussed how to describe language extensions and presented extension facilities for the Magnolia language extensions, with support for static semantic checking and scoping. The facilities include macro-like *operation patterns*, and *transforms* can perform arbitrary transformations of code. Transforms can be linked into the compiler at different stages in order to implement extensions by transforming extended code to lower-level code. Static semantics of extensions can be given by hooking transforms into the semantic analysis phase of the compiler.

A natural next step is to try and implement as much of Magnolia as possible as extensions to a simple core language. This will give a good feel for what abstractions are needed to implement full-featured extensions, and also entails building a mature implementation of the extension facility – currently we are more in the prototype stage. There are also many details to be worked out, such as a clearer separation between code patterns, variables and transformation code, name capture / hygiene issues, and so on.

CORE MAGNOLIA

The Magnolia compiler is available at <http://magnolia-lang.org/>.

Acknowledgements

Thanks to Magne Haveraaen and Valentin David for input on the Magnolia compiler. Thanks to Karl Trygve Kalleberg and Eelco Visser for inspiration and many discussions in the early phases of this research.



Reno, NV (Supercomputing '07)

Discussion

8.1 Related Work

In this section we will discuss related work particularly related to Magnolia, language design and our overall approach. For more detailed discussions of related work, see the individual chapters.

8.1.1 THE RISE OF DATA ABSTRACTION

The idea of data abstraction dates back to the early seventies. An *abstract data type* is a data structure with set operations to access the data structure – with all data access having to go through the operations. Thus the data representation itself is hidden or encapsulated, with access being controlled by a well-defined interface.

An important precursor to abstract data types was the class mechanism of Simula 67 [Dahl et al., 1968], which later evolved into the now almost-dominant object-oriented programming paradigm. Simula was designed for building simulation software, and provided *classes* as a way of implementing entities in the simulation. A class declares a self-contained program with data and procedures, serving as a pattern for making *objects*. Objects are used to model real-world objects in a simulation, with the procedures defining what sort of actions can be performed on them. Simula 67 lacked the encapsulation necessary to support abstract data types, but this was added later [Palme, 1973]. Simula also provided inheritance and virtual functions, which are important in object orientation, but which we have not considered for inclusion in Magnolia – at least not yet – as this would significantly complicate the language design, and limit the amount of processing that can be done statically.

Data abstraction was introduced as a language feature in the language

SIMULA

CLU

CLU [Liskov et al., 1981], with the *cluster* construct which defines a new abstract data type together with a data representation for objects and operations to manipulate the data. Access to the data representation is limited to operations defined within the cluster. The origins and motivations of CLU [Liskov, 1993] were similar to those of Magnolia – to support research on programming methodology, in particular, an abstraction-based approach to programming.

CLU has been a major influence on Magnolia, when it comes to data abstraction, exception handling, iterators, and separation between implementation and interface. Major differences from CLU include how Magnolia is centred around modules with multiple types and operations; how axioms are integrated into the languages and actively used by the compiler; and the mapping between imperative and algebraic coding styles. CLU also uses a heap-based reference oriented semantics (e.g., variables don't contain objects, they refer to them), similar to Lisp and Java, rather than the value-oriented semantics of Magnolia.

ALPHARD

Alphard [Shaw et al., 1977; Wulf et al., 1976] was another important early data abstraction language. The basic abstraction mechanism there is the *form*. Alphard provides *generators* to abstract over iteration and the structure of data collections – this is similar to the `map` operation discussed in Section 7.3.5.

SMALLTALK

Smalltalk (particularly Smalltalk-76/-80) [Kay, 1996] brought the ideas of object orientation further, basing the language on message passing between objects. Smalltalk dropped the distinction between primitive values and object (a distinction which is important in later OO languages like Java and C++), and in Smalltalk-80 even classes are objects, and can be manipulated through reflection. Similar features are found in modern languages like Python and Ruby.

ADA

Ada [Ichbiah et al., 1981] was built to support data abstraction and to simplify the creation of highly reliable software – design goals similar to Magnolia. As in Magnolia, Ada allows multiple types and operations to be specified together in a module. Ada features strong typing, modularity, generic modules, exceptions, run-time checking and also has support for multi-threaded programs – a feature not considered for Magnolia yet. Even the syntax is designed to avoid programmer mistakes. While previous versions had good support for data abstraction, Ada 95 added full support for object orientation, with inheritance and virtual functions, to the design.

SCALA

Scala [Odersky et al., 2006] is a modern data abstraction language with support for object orientation and (to some degree) functional programming. It is designed to tackle large scale abstraction (components) using the same concepts as are used at the small scale (i.e., class / module level). Every data type in Scala is organised in a type hierarchy, with `Any` at the top and `Nothing/Null` at the bottom, and even functions

are treated as objects (with the arrays being a subtype of functions). Scala makes extensive use of abstract classes (types), which are used in a manner similar to concepts in Magnolia, and *traits* which are a special form of abstract class used to do mixin composition of data abstractions.

8.1.2 SPECIFICATION, VERIFICATION AND RELIABILITY

Important early work on specification and verification of data abstractions was done by Hoare [1972]. Algebraic specification [Goguen et al., 1978; Guttag and Horning, 1978; Guttag et al., 1978; Liskov and Zilles, 1975] arose as a specification technique for abstract data types, focusing on specifying the behaviour of operations in relation to each other, rather than specifying the inputs and outputs of each operation. A simple algebraic specification consists of a signature, listing types and operations on them (giving the syntax), and a set of axioms (specifying the semantics). The signature idea is fairly common in programming languages – a well known flavour is the Java *interface*. Having both signatures and axioms – *concepts* in our terminology – is less common.

CLU provided a possibility to separate the abstract interface of a module (what we call a signature) from its implementation, and do type checking against interfaces. An interface could have multiple implementations, and the desired implementation could be selected at link time. CLU even allowed for procedure behaviour to be specified using *requires*, *ensures* and *modifies* clauses, although such specification was usually informal. This closely matches the idea of concepts, the main difference being that a CLU interface describes only one type. To deal with the problem of constraining polymorphism, CLU has a *where* clause which specified the operations that should be supported by a type parameter.

SPECIFICATION IN
CLU

Alphard *forms* – which are used to implement abstract data types – integrate formal specification through pre- and post-conditions, invariants, requirements, assumptions and proof rules. Unlike CLU and Euclid (below), the specification language is part of the language and not left to an external tool.

ALPHARD

The Euclid language [Popek et al., 1977] was designed as an improvement of Pascal [Wirth, 1971] with support for verification of programs, and also adding systems programming features missing from Pascal. The language design was guided by the goals of verifiability, reliability and understandability – goals that are also important in Magnolia, though we would consider ‘verifiability’ as ease of reasoning, and use it in support of optimisation. As aliasing is a major hindrance to reasoning, the language forbids aliasing in a similar, though slightly more elaborate way than Magnolia – Euclid allows for the use of pointers and passing array components and dereferenced pointers as arguments, by organising dynamic

EUCLID

variables and pointers in *collections*, and enforcing argument-passing rules that prevent access to both a dereferenced pointer / array component and its collection / array at the same time.

Like Magnolia, Euclid distinguishes functions (which are side-effect free) from procedures (which may have side-effects) – though Wortman and Cordy [1981] point out that functions may have limited usefulness in practise. Procedure parameters may be *var* (modifiable) or *readonly*, and all *var* arguments must be non-overlapping, to prevent aliasing. Access to module-level variables must be explicitly declared, and is read-only by default.

Specification and verification in Euclid is based on the axiomatic system developed by Hoare [1969]. Operations are specified by *pre*- and *post*-assertions, and *assert*-statements placed throughout the code. *Module invariants* – like data invariants in Magnolia – must hold on entry and exit from exported operations. The code can also be annotated by specifications to be read by external verifiers – the form of these annotations depend on the verifier, but any verifier is also expected to make use of the assertions in its reasoning. Furthermore, the compiler will insert *legality assertions* when the legality of some computation depends on run-time information – for example, checking that an array index is within bounds. As long as the legality assertions hold, a program is legal, with well-defined semantics (though it is not necessarily *correct* according to the program specification).

Exceptions are intentionally omitted from Euclid, as all programs are expected to be verified and be free for run-time errors. Later language designs have typically taken a more pragmatic approach, fortunately.

EIFFEL

The design of Eiffel [Meyer, 1992] provided several advances in both object orientation and reliability – and the combination of these – improving on earlier languages like CLU, Alphard, Euclid and Ada. Eiffel supports *design by contract* with pre- and post-conditions (*require* and *ensures*) on methods, and class invariants. As with Euclid module invariants and Magnolia data invariants, the class invariant must hold on entry and exit to any exported method. The pre- and post-conditions form a contract between the caller and callee – if the precondition is satisfied, the caller can rely on the postcondition to hold on return. Statement-level assertions and loop invariants are also provided by the language. The compiler can automatically insert assertion checks if desired.

As Eiffel is an object-oriented language, it also deals with contracts and assertions in the context of inheritance, polymorphism and dynamic binding – issues which are side-stepped in Magnolia, due to the lack of inheritance. A subclass must always obey the contract of the class or classes it inherits from, since objects of a subclass may be assigned to and used as variables declared as the superclass. Preconditions must therefore

not be stronger than those of the inherited classes, and postconditions may not be weaker. It is allowed, however, to have weaker preconditions and stronger postconditions in the subclass – this just means the subclass does a ‘better job’ at fulfilling the contract. Class invariants are always at least as strong in subclasses – additional clauses may be added to the invariant, but the invariants of inherited classes must still hold. In contrast with C++, which uses static binding of methods by default, Eiffel always uses dynamic binding (‘virtual functions’ in C++ terminology). This follows naturally from the use of class invariants; with static binding, the method of a superclass may be called to process an object of a subclass – but the superclass method has no knowledge of, and cannot be expected to uphold the invariant of the subclass. The design of contract inheritance ensures that methods always act according to their defined contract, even in the face of inheritance and dynamic binding, thus making it possible to reason about code that employs inheritance. This would be important for a possible future addition of inheritance to Magnolia.

Interface definitions are done using *deferred classes* (abstract classes in C++ / Java terminology). All classes that implement the interface inherit from the deferred class, implementing any missing functionality. The inheritance rules ensure that all such implementations uphold the same contract.

Eiffel also comes with an exception handling facility, which is further refined compared to Ada, PL/I and CLU. Exceptions are used in abnormal situations, and also when monitored assertions fail. Exception handlers should put objects back into a consistent state, and may then attempt to resume processing, or give up and propagate the exception through the call tree. The exception handler is not expected to fulfil the method’s postconditions, its only obligation is to ensure that objects are consistent and that the class invariant holds. The caller may then attempt to obtain its result some other way, or may propagate the exception again.

An important early work on mixing programming languages with algebraic specification is Extended ML [Sannella and Tarlecki, 1985, 1986], which introduced algebraic specification into the Standard ML language – rather than using axiomatic specifications with pre- and post-conditions as in Euclid. Extended ML separates signatures from implementations, and allows axioms to be given together with the signatures. It builds on the idea of *institutions* [Goguen and Burstall, 1984], in order to achieve independence from the underlying logic system. Sannella and Tarlecki [1999] report positive experiences with the use of Extended ML in teaching and for further research, but also note that dealing with specification in the context of the full Standard ML language in a fully formal way is too difficult to be practical; particularly dealing with things like exceptions / partiality, and higher-order functions. With Magnolia, we have full

EXTENDED ML

control over the base language, so we have a better hope of avoiding or controlling features that interfere with formal specification. For instance, we can deal with partiality using guarding, and avoid the use of higher-order functions.

CONCEPTS

Concepts in Magnolia and C++ match closely the ideas of signatures and axioms from algebraic specification, allowing us to use them for axiom-based testing and rewriting. Zalewski and Schupp [2007] discuss C++ concepts more closely from a specification and institution point-of-view. The concept approach, based on algebraic specification, is more similar to Extended ML than to the axiomatic / assertion approach of Euclid and Eiffel. Behaviour is specified by relating the operations of abstract data types to each other, rather than specifying requirements and effects of each operation. The algebraic approach has the added benefit of being directly usable for rewriting (see Chapter 3) and as a basis for testing (see Chapter 4). On the other hand, assertions are immediately useful as checks during the run-time of a program. In both cases, over-specification must be avoided if one is to be sure that the specification is applicable to a wide range of implementations – though the danger of this may be greater with assertions.

Magnolia still allows (or will allow) the use of pre- and postconditions, through the alert system (see Chapter 5) – though this is aimed at detecting and handling errors at run-time, rather than at program specification. *Guarding* [Haverdaen and Wagner, 2000] is a form of preconditions that can be used to hide partiality in an algebraic specification. Assertions will also play an important role in Magnolia, similar to their use in Euclid – providing hints for reasoning about programs. For example, asserting that an array is sorted, so that the compiler may take advantage of this when selecting a sorting algorithm – and of course also allowing for run-time checks of assertions, for debugging purposes.

8.1.3 CONCEPTS AND BOUNDED POLYMORPHISM

For a description of concepts in Magnolia, see Section 6.7.

Concepts in C++

Concepts were proposed for the C++0x standard [Gregor et al., 2008], primarily to make generic programming more user-friendly by allowing bounded polymorphism. In standard C++ it's not possible to specify requirements for template parameters. For example, a generic sorting library might include a template declaration like:

```
template<typename ArrayType, typename ElementType>  
void sort(ArrayType<ElementType> &a);
```

Concepts were unexpectedly dropped from the standard proposal late in the process (summer '09).

The `sort` function would rely on the `ArrayType` having an indexing operation `[]` and that the `ElementType` has a comparison operator (e.g., `<`). If a piece of user code tries to call `sort` with an inappropriate element or array type (for instance, a linked list having only head and tail operations), the user will get an error message pointing deep inside the library code. For complicated template libraries, such errors can give several pages of messages, making the actual fault difficult to trace.

With concepts, it is possible to specify requirements for template parameters. The above `sort` function may be specified instead as:

```
template<typename ArrayType,
        LessThanComparable ElementType>
requires Indexable<ArrayType, ElementType, int>
void sort(ArrayType &a);
```

– requiring that the array argument is `int`-indexable with `ElementType` elements, and that elements are *LessThanComparable*. The compiler is able to fully type check the `sort` implementation based on the concepts (something that previously had to be delayed until the template had been instantiated with concrete types), and will also give a clear error message if the user tries to instantiate the template with inappropriate parameters.

A *concept map* is used to declare that some types model a concept.

CONCEPT MAPS

```
concept_map Indexable<IntArray, int, int>;
```

The concept map can include some implementation code to replace or rename missing operations:

```
concept_map LessThanComparable<Orange> {
    bool operator<(const Orange&a, const Orange&b) {
        return a.weight < b.weight;
    }
}
```

This makes it much easier to fit code from different sources together, since differences in style or conventions can be translated away by the concept map. With plain templates (and with most other generic programming systems, for that matter), you're stuck with whatever operations and calling conventions are used in the template code. We examine this problem from another angle in Chapter 2.

Haskell Type Classes

Haskell type classes [Hall et al., 1996] provide a classification of types based on which operations are available on the types. A type class lists some operations, and types that support that operation forms the type class. Function parameters may then be typed by classes and be generic in all the types belonging to a type. Type classes provide for refinement

TYPE CLASSES

(building on existing classes), modelling, and constraints in much the same way as C++, as shown by a comparison by Bernardy et al. [2008].

As Haskell more or less uniformly uses curried functions, the proliferation of declaration forms that plague C++'s concepts is much less of an issue, lessening the need for a concept map feature.

Scala

ABSTRACT CLASSES Scala supports bounded polymorphism by specifying that a type parameter should be a subtype of some other type. Many of the ideas of concepts and Haskell type classes are realised in an object-oriented fashion through abstract classes, traits and views, but there is no support for axioms and specifications.

Signatures (interfaces) may be specified using abstract classes, for example:

Examples are adapted
from Odersky et al.
[2006].

```
abstract class Monoid[a] extends SemiGroup[a] {  
  def unit: a  
}
```

Implementations may be done using objects (classes with just a single instance) inheriting from the abstract class. Implementation objects are then sent as arguments to any method that makes use of the abstract class. For example, calling a `sum` method with a list of integers and an integer implementation of *Monoid*:

```
def sum[a](xs: List[a])(m: Monoid[a]): a = ...
```

```
sum(List(1, 2, 3))(intMonoid)
```

IMPLICIT
PARAMETERS

Explicitly passing the implementation as a parameter can be avoided through the use of `implicit` arguments or declarations, in which case the `sum` method can be called as:

```
sum(List(1, 2, 3))
```

and Scala will figure out from the context that `intMonoid` is the only applicable implementation. This gives an effect similar to how concepts are used in C++.

VIEWS

A *view* is used in a way similar to a C++ concept map or Magnolia model declaration, and is actually a way to define implicit type conversions. For example, one could define a view from `List[T]` to `Set[T]` by giving the definitions of the set operations, allowing lists to be used as arguments to methods that expect sets. *View bounded* type parameters accept types that have views into a given type – comparable to how a `requires` clause in C++ is used to constrain a type parameter to type that have a concept map for the given concept.

8.2 Evaluation

For Magnolia, current body of code is too small to properly evaluate the language design. An important next step will be to write larger programs, to see how the language features work in practise. Testing the language on more developers is also important – particularly developers of varying proficiency levels. A language that can only be used by its designers is almost useless – one that can only be used by grad students and above is also not likely to succeed.

USABILITY

Writing flexible, reusable code is generally considered difficult. This is often a problem of choosing the right abstractions – while the language can help make the implementation of those abstractions reusable in different situations, it takes a good deal of effort and experience on the part of the programmer to create reusable code.

Reliability, on the other hand, is something we would like to be within the grasp of junior programmers. After all, with all the experienced programmers busy designing good abstractions, someone has to write all the actual code. In order to verify that Magnolia meets the goal of enabling more robust and reliable code, we should do trials with both experienced and less experienced programmers, and see how reliability compares to other languages. From the small body of code we have written, we have noticed that code that passes the compiler checks usually works correctly the first time. Whether this will hold true for larger programs remains to be seen, but it is consistent with experiences with other languages with strict compiler checks, such as Euclid [Wortman and Cordy, 1981].

While we do not yet have enough data to evaluate the full language, we have enough experience to offer some insight on individual features. Our experience with functionalisation and mutification is that these features interact very well with the rest of the language design, particularly concepts and axioms. They are also quite convenient when writing Magnolia code. Previous experience with the Sophus numerical library (written in C++, in an algebraic style) shows that the ability to write math-intensive code in an algebraic style is of great benefit [Dinesh et al., 2000].

EXPERIENCE

While concepts are not completely designed and implemented in Magnolia, the Sophus library [Haveraaen and Friis, 2009; Haveraaen et al., 1999] has been designed around similar principles – although, implemented in an *ad hoc* manner, using C++ and the C++ preprocessor, and with a separate specification. Sophus is a medium-sized project (tens of thousands of code lines), with several applications available, including the Seismod seismic simulator. Sophus showed that numerical application could be built in a modular manner, with each module satisfying a particular specification. The well-specified and clean module interface

SOPHUS

allows for interchangeable implementations – such as replacing a sequential data structure with a parallel one. The planned reimplementations of Sophus in Magnolia will be a good exercise of concepts in Magnolia – and will also allow for a substantial refinement of the Sophus design, building on native support for concepts, axioms and mutification in the language.

JAxT

Axiom-based testing has not been implemented or tested in Magnolia yet, but we have gained some experience in this area with the JAxT [Haveraaen and Kalleberg, 2008] tool for Java. A team of experienced undergraduate students [Masood et al., 2009] successfully wrote axioms for some of the Java collection classes, formally specifying the behaviour dictated by the API documentation. Their work showed that axiom-based testing is certainly doable in an object-oriented context, but that there are problems associated with using axioms for code that relies heavily on side effects, as is common in object orientation. This is something we hope to avoid with functionalisation in Magnolia.

8.3 *On Language Design*

Our initial goal was not to design a new programming language, but rather to experiment with new language features. The design of Magnolia started as a means to the end of integrating and experimenting with new features, as our existing language platform, C++, turned out to be too difficult to work with [David, 2009]. In retrospect, the design and implementation work may have been easier if we had used another language than C++ as a starting point – CLU, Eiffel, Euclid, or even Pascal, perhaps – a language with many of the features we're interested in, and with a small, concise definition we could use as a starting point for our own language definition. We could then tweak the syntax to give it the necessary C++-like feel that our users desire, and perhaps still implement the compiler using compilation to C++.

The keynote was later published as a report and as the book chapter cited here.

In 1973, Hoare [1989] gave a keynote address on language design, discussing principles which are to a large degree still valid today. He recommends that features be designed separately from programming languages – perhaps as extensions to a well-known language – and the language designer (ideally a separate person) will then later on pick and choose the most suitable features and integrate them into a consistent design. This may be sound advice – though Meyer [1999] disagrees, as many Eiffel features have been successfully designed and integrated in a one-step process. We have gone with Meyer on this – Magnolia features and the language itself is designed by the same group. However, most Magnolia features are of course based on existing features in other languages, and here we have taken Hoare's cherry-picking approach.

Hoare point out five principles for good language design: simplicity, security, fast translation, efficient object code and readability.

Simplicity means the language should be as small as possible, so that programmers (and designers) can know and understand the full language. Language modularity, in Hoare's view, is not sufficient, as the programmer may then be left helpless on encountering features from an unfamiliar part of the language. Instead, simplicity should be a primary objective in language design. Meyer [1999] favours simplicity, but not at the cost of power to solve real problems. We agree – the focus should be on making the programmer's task simple, even if that means adding more language features. There is a danger, though, that some of Magnolia's features will be difficult to understand for programmers that lack training in algebraic specification, or is unfamiliar with some of the features, like alerts or mutification. This is something which will require very careful thought in the continuing design process.

SIMPLICITY

We have done some simplifications, though, in order to cut down on the language complexity. Almost any comparable modern language includes object orientation – this is dropped from Magnolia, as the simpler ideas of data abstraction are sufficient for us. We have not included concurrency constructs, or dynamic dispatch – features that would make programs more difficult to analyse statically. Other language research projects tackle these issues; object orientation and components in Scala [Odersky et al., 2006], parallelism and high-performance computing in X10 [Charles et al., 2005], Chapel [Callahan et al., 2004] and Fortress [Allen et al., 2007]; distributed computing in Creol [Johnsen et al., 2006], and so on – in addition to the wealth of other research going on in more established languages. If such features are needed later, we can build on the experience from other projects.

Security has been important in the design of Magnolia – with features like strict type checking, data invariants and alerts. Note that this is security in the sense of what we would now call safety or reliability, not security against attack from malicious software (though the latter to implies the former to a large degree). Security is also a major design concern in Eiffel. C++, on the other hand, forgoes security in some cases (allowing casts and pointer manipulation) in favour of compatibility with C and existing code.

SECURITY

Fast translation / compilation was important in the seventies, with slow hardware – and it is, surprisingly, still important. The edit-compile-debug cycle of programming will often require many recompilations of a project, and while processors have gotten a lot faster in the past thirty years, application size has also grown significantly. The same goes for object code efficiency – we have more processor power, but also even more difficult problems to solve. Based on our current tooling experience, keeping the compilation time low will likely be an issue as Magnolia

COMPILATION

programs grow larger. Many of the optimisations and specialisations we'd like to do require the entire program to be compiled at the same time, making time-saving separate compilation difficult. As long as we have language experimentation as a goal, however, compilation and executable speed is not a primary concern – though it may be a consideration when incorporating features into a production language. Stroustrup [2007] has been conscious of this in the design of C++ – avoiding features that add overhead, particularly features that have run-time costs even when not in use.

READABILITY

Readability is often undervalued, and many language designers feel that syntax is of little importance, since it is basically a skin over the underlying meaning. Still, syntax is one of the things that programmers easily get worked up about, and the issue of readability is important in maintenance and for understanding of code. Thus we have features like functionalisation / mutification, which makes the use of readable arithmetic expression notation accessible in more cases than usual. Some syntactic features of the C/C++/Java 'curly brace' style have a negative impact on readability, creating situations where an easy-to-miss extra semicolon can change the meaning of a program, or where the assignment and equals operators are easily confused. Meyer [1999] has taken this into consideration, and defined Eiffel syntax with a relaxed view of semicolons, and for the most part using an Algol/Pascal-like syntax. Magnolia has started out with the more popular C/C++/Java style, simply to remain as C++-like as possible; though it is easy to change the syntax later on if desired, or to provided multiple syntactic skins according to programmer preference.

Hoare [1989] also argues that procedures should give a clear indication of its effects on its parameters – and idea which is fully supported in Magnolia with the various parameter modes. He also suggests that variables should not just be typed, but also supplied with units – so that a radian value is distinct from a degree value, for instance. The Fortress language [Allen et al., 2007] supports this kind of type system, and keeps track of units through various operations – divide length by time to get speed, for example.

OVERLOADED
NAMES

Meyer [1999] argues against overloading – having more than one definition for the same name in the same class / module – as it interferes with reasoning, simplicity and readability. It is often difficult for the reader to determine which overloaded operation is called, as this is highly dependent on the context, and subtle differences in type can lead to the choice of an entirely different operation. The effect of subclassing in Eiffel is that you get different variants of the same operation, with the same basic semantics (specified by the contract), selected appropriately for the type – whereas with overloading you get multiple operations with the same name, but with possibly entirely different semantics. Additionally,

when combined with inheritance and dynamic binding, it becomes easy to confuse *overriding* an inherited operation with *overloading* it.

Magnolia does allow overloading – in fact, it is a pretty fundamental part of the language, since we do not use the object-oriented message passing model / dot-notation to decide which type an operation belongs to. The danger of confusing operations with different semantics remain, however. A possible way to handle this is to forbid overloading within concepts and within the same module. This way, every name in a concept has a unique and well-defined semantics – there may be many different implementations, but they must all follow the same specification, just as with subclassing in Eiffel. Confusion between overloaded names in different modules can be controlled through qualification of names and fine-grained imports.

To Stroustrup [2007] (and the C++ standards committee), compatibility is an important design criterion, both in the initial design (compatibility with C), and in further changes of the language. This is probably a very sensible approach for C++, given the many billions of lines of existing code. In contrast, Meyer [1999] considers language evolution as an important part of design, and is open to introducing incompatibilities with previous language versions – as long as there is agreement that the change will actually result in a better language, and a migration tool is made available for existing code. For Magnolia, being able to change and evolve the language is important – it is in fact the original purpose of the language. This is one of the language design dilemmas: on one hand, it is good to have few users and little existing code, so that one may easily evolve the language without concern for compatibility; on the other hand, having many users and a large code base makes it much more likely that one can make sound decisions about how to evolve the language to better serve its purpose. For us, compatibility issues are probably best handled with a migration tool.

COMPATIBILITY

Another way to approach compatibility is to ensure *interoperability* with existing languages and code, while still doing radical new design. This approach is taken in Scala [Odersky et al., 2006], which is fully interoperable with existing Java and C# classes and libraries, but still breaking new ground in the type system, and providing features like pattern matching of objects, algebraic data types and support for functional programming. Magnolia provides some degree of interoperability with C and C++, by allowing for calls to C/C++ operations and doing automatic translation of some simple data types (strings and primitive types) when calling C/C++ code. Data structure representations are not directly accessible, however, and must be accessed through functions. Similarly, C++ provides interoperability with C, with `extern "C"` declarations.

INTEROPERABILITY

8.4 *Future Work*

FUTURE WORK

We have five main areas of future research for Magnolia: optimisation, concepts and concept-based programming, language extension support, language description, and novel language features. Optimisation overlaps with concepts when it comes to axioms, and with language extension when it comes to transformation. Optimisation possibilities include axiom-based rewriting combined with property propagation, code specialisation, and other high-level optimisations. For low-level optimisations of the kind typically found in compiler books, we can rely on the backend compiler.

Concepts are still undergoing design work, and there should be ample opportunity for future research in this area as we begin to build larger programs based on concepts. Support for language extensions is still early in the design phase. While it is primarily intended for prototyping novel features, extending it to the point where it can be used to implement large parts of the compiler would be an interesting challenge.

As discussed above, determining how effective the language is in practical use is an important step that must be done. Furthermore, the current language design is closely tied to the implementation – to some degree, the implementation provides the specification of the language. This is clearly undesirable in the long term – we should swallow our own medicine and provide a separate specification of the language. Doing a prototype implementation before proper specification does have its value though, as it makes clear which features are easily implemented and easily compiled to efficient code – an important consideration in a language design project with extremely limited resources.



WALL-E

Conclusion

In the introduction, we discussed software quality, and techniques for developing quality software. Let us now look back on the preceding chapters, and see how this work extends the state of the art in software quality.

We pointed out *abstraction* as a fundamental approach to software quality, flexibility and for dealing with large problems. We would therefore like to encourage programmers to make extensive use of abstraction, and reduce any concerns programmers might have that the resulting code will be slow because of abstraction overhead. *Mutification / functionalisation* from Chapter 2 takes the existing idea of procedural abstraction and makes it more flexible, by decoupling implementation from use. Efficient code can be written in an imperative style, while the code can be used in an algebraic style, which is easier to reason about, and which is easily linked to program specification. Code built this way is aliasing-free, which enables aggressive optimisation by the compiler.

We also said that *separation of concerns* was important, both as a thought process, and when building software. Mutification and functionalisation provide a new kind of separation of concerns – between how you declare and implement something, and how you use it. We find the same separation in *alerts* – between how errors are dealt with at the implementation side, and how they can be dealt with at the use side.

Axiom-based rewriting, introduced in Chapter 3 can be used to apply high-level optimisations in the style of active libraries [Czarnecki et al., 2000]. Such transformations are easily applied to code written in the algebraic style enabled by mutification. Providing an optimisation benefit to using axioms will encourage programmers to make use of them. Axiom-based optimisation may also help further reduce any overhead

associated with more abstract code – and it also enables separation of optimisation from implementation, making it more likely that the programmer can keep the main implementation clean and simple. *Axiom-based testing* (Chapter 4) has a more direct quality impact – on reliability and maintainability – and provides even more benefit to the programmer who takes the time to program with concepts and axioms.

The *alert* feature (Chapter 5) abstracts over error behaviour, and cleanly separates between *reporting* errors and *handling* errors. The technique makes it easy to enforce proper parameter checking and checking of error conditions, contributing to increased robustness and reliability. Our work on alerts builds on the exception facility found in many programming languages, combines it with a formalisation of commonly used ad hoc techniques, and revisits some old ideas in error handling. Alerts are also an example of a *domain-specific aspect language* (DSAL), applying separation of concerns to the domain of error handling [Bagge and Kalleberg, 2006].

It is difficult to tell how effective our proposed techniques and features are at increasing software quality without performing trials on medium to large scale software projects. Such trials are beyond the scope of this dissertation, and will have to be left as future work.

Contributions

In this dissertation we have looked at language features supporting a development method for creating quality software, focusing on abstraction, specification, testing and separation of concerns. The specific contributions of this work are:

- A significant extension of previous work on mutification [Dinesh et al., 2000], providing a formalisation of functionalisation and mutification, and showing its usefulness in linking implementation and specification.
- A treatment of axiom-based testing in the context of concepts, and showing how concepts and axioms can be used for both rewriting and testing.
- A language construct, *alerts*, for dealing with errors, which unifies previous techniques and separates error reporting from error handling.
- A basic design and implementation for the new language Magnolia, providing a foundation for future work in this area.

Future Work

Main areas for future work are:

- Bringing the language design and compiler implementation to a level where it can be used for large projects, and doing trials to evaluate the effectiveness of the language, and our techniques and development method.
- Building a large base of fundamental concepts with accompanying axioms, as a basis for implementation work, and researching constructs for combining and constructing concepts.
- Integrate alerts with guards [Haveraaen and Wagner, 2000] and concepts.
- Do a proper design and specification of the type system. In particular, adding *dependent types* would allow the compiler to do even more checks at compile-time, such as array bounds checking, and dimension checks for matrix operations. Any checks that cannot be verified at compile time, can be pushed to run time, as with legality assertions in Euclid [Popek et al., 1977].
- Axiom-based testing should be extended with support for guards for controlling errors and undefinedness in axioms; a discussion of this and of the use of functionalisation to control side-effects in axioms would be a valuable research contribution.
- Explore further the use of axioms for optimisation, particularly in combination with optimisations that track values and properties throughout a program. Such tracking is particularly easy in Magnolia, because of the lack of aliasing, and since functions and procedures make data-flow properties easy to access.
- If possible, see if any of our experimental features or experiences can be contributed to other languages – such as in the further process of developing C++ concepts.



EVE

Summary

Developing robust, reliable, flexible and maintainable software is challenging, especially with large projects. In this dissertation we look at language constructs and features to support the development of high-quality software through *abstraction*, *specification* and *testing*. We are particularly interested in the high-performance / numerical software domain. The work culminates in the design of the Magnolia programming language, intended to serve as a foundation for further research in this area.

PLAYING WITH SIGNATURES: The same algorithm may often be realised in several different ways. For example, sorting can be seen as reordering a sequence, or as producing a new sorted sequence from an input sequence. The former maps naturally to an imperative procedure, while the latter maps naturally to how we would specify sorting in an algebraic specification. We introduce *mutification* and *functionalisation*, which allow us to map back and forth between imperative and algebraic code. In particular, algorithms may be implemented in an imperative style, which is often more efficient, especially in numerical software, but used – in both program specification and code – in the algebraic style. The algebraic style code can then be translated automatically to imperative style code.

WORKING WITH CONCEPTS: Concepts define the operation interface of a set of types, and specifies the behaviour of operations with axioms. We show how axioms can be used for optimisation purposes by annotating them with *axiom classes* that describe how they should be used. Such optimisation is particularly useful on code written in an algebraic style, as this is the preferred style for writing axioms. We also discuss how to generate automated tests from concepts and axioms.

AXIOM-BASED TESTING: We refine the idea of testing based on concepts and axioms, detailing how to generate tests in concept-enabled C++, and presenting a test generation tool for C++.

HANDLING FAILURE AND EXCEPTIONS: Proper handling of failures and unexpected situations that arise is important in creating reliable software. We introduce *alerts*, which provide a unified approach to reporting and handling failure. In particular, alerts can enforce checking of return codes, and provide local or global handlers for different error situations.

Alerts can also associate errors with particular argument values, thus providing precondition checks integrated with the alert handling system. Improper checking of arguments is a significant source of bugs and security problems, so this is an important feature.

THE MAGNOLIA PROGRAMMING LANGUAGE: We discuss the design of the Magnolia language, which is designed around the ideas and features mentioned above. Magnolia is intended to provide a flexible basis for further language experiments, while providing enough control over the execution model to be useful for developing high-performance software.

CONCLUSION: The idea of mutification and functionalisation provides a crucial link between implementation and specification, and serves as an enabling technology for the work on testing and optimisation with concepts and axioms. We provide for increased reliability and robustness with axiom-based testing and the alert facility, while encouraging use of abstraction and an easy-to-reason-with algebraic style by providing axiom-based optimisation and reducing overhead with mutification.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison–Wesley, 1986.
- [2] E. Allen, D. Chase, C. Flood, V. Luchangco, J.–W. Maessen, S. Ryu, and G. L. Steele, Jr. Project Fortress: A multicore language for multicore processors. *Linux Magazine*, pages 38–43, September 2007.
- [3] J. Andersen and C. Brabrand. Syntactic language extension via an algebra of languages and transformations. In T. Ekman and J. J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, Electronic Notes in Theoretical Computer Science, York, UK, March 2009. Elsevier.
- [4] S. Antoy. Systematic design of algebraic specifications. In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design*, pages 278–280, New York, NY, USA, 1989. ACM. ISBN 0-89791-305-1. doi: 10.1145/75199.75241.
- [5] S. Antoy and R. G. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Trans. Software Eng.*, 26(1):55–69, 2000.
- [6] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg–Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002. ISSN 0304–3975. doi: 10.1016/S0304-3975(01)00368-1.
- [7] A. H. Bagge and M. Haverdaen. Axiom-based transformations: Optimisation and testing. In J. J. Vinju and A. Johnstone, editors, *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238 of *Electronic Notes in Theoretical Computer Science*, pages 17–33, Budapest, Hungary, 2009. Elsevier. doi: 10.1016/j.entcs.2009.09.038.
- [8] A. H. Bagge and M. Haverdaen. Interfacing concepts: Why declaration style shouldn't matter. In T. Ekman and J. J. Vinju,

- editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, Electronic Notes in Theoretical Computer Science, York, UK, March 2009. Elsevier.
- [9] A. H. Bagge and K. T. Kalleberg. DSAL = library+notation: Program transformation for domain-specific aspect languages. In *Proceedings of the First Domain-Specific Aspect Languages Workshop (DSAL'06)*, Portland, Oregon, 2006.
- [10] A. H. Bagge, V. David, and M. Haverdaen. Axiom-based testing for C++. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 721–722, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-220-7. doi: 10.1145/1449814.1449829.
- [11] O. S. Bagge and M. Haverdaen. Domain-specific optimisation with user-defined rules in CodeBoost. In J.-L. Giavitto and P.-E. Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE'03)*, volume 86/2 of *Electronic Notes in Theoretical Computer Science*, Valencia, Spain, 2003. Elsevier.
- [12] J. Barnes. *High Integrity Ada. The SPARK Approach*. Addison-Wesley, 1997.
- [13] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002. ISBN 0321146530.
- [14] K. Beck. Extreme programming: A humanistic discipline of software development. In *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE'98)*, pages 1–6. Springer-Verlag, 1998.
- [15] K. Beck and E. Gamma. JUnit – Java Unit testing.
<http://www.junit.org> and
<http://sourceforge.net/projects/junit/> per 2009-01-24, 2009.
- [16] P. Becker. Working draft, standard for programming language C++. Technical Report N2857=09-0047, JTC1/SC22/WG21 – The C++ Standards Committee, March 2009.
- [17] J.-P. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz. A comparison of C++ concepts and Haskell type classes. In *WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming*, pages 37–48, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-060-9. doi: 10.1145/1411318.1411324.

-
- [18] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. *SIGPLAN Not.*, 23(SI):1–142, 1988. ISSN 0362–1340. doi: 10.1145/885631.885632.
- [19] A. Borghi, V. David, and A. Demaille. C-Transformers: A framework to write C program transformations. *ACM Crossroads*, 12(3), April 2006.
- [20] A. Borgida. Language features for flexible handling of exceptions in information systems. *ACM Trans. Database Syst.*, 10(4):565–603, 1985. ISSN 0362–5915. doi: 10.1145/4879.4995.
- [21] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. and H. Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, Pont-A-Mousson, France, September 1998. Elsevier Science.
- [22] J. M. Boyle, T. Harmer, and V. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, Boston, 1997.
- [23] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 31–40, New York, NY, USA, 2002. ACM. ISBN 1–58113–455–X. doi: 10.1145/503032.503035.
- [24] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM Press. ISBN 1–58113–831–9. doi: 10.1145/1028976.1029007.
- [25] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: Components for transformation systems. In F. Tip and J. Hatcliff, editors, *PEPM'06: Workshop on Partial Evaluation and Program Manipulation*. ACM Press, January 2006.
- [26] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52–70, June 2008. ISSN 0167–6423. doi: 10.1016/j.scico.2007.11.003. Special issue on experimental software and toolkits.

- [27] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52–60. IEEE Computer Society, April 2004.
- [28] D. G. Cantor. On the ambiguity problem of Backus systems. *J. ACM*, 9(4):477–479, 1962. ISSN 0004–5411. doi: 10.1145/321138.321145.
- [29] B.-Y. E. Chang, A. Diwan, and J. Siek. Gradual programming: bridging the semantic gap. In *Fun and Interesting Thoughts (FIT) at PLDI 2009*, Dublin, Ireland, 2009. URL <http://pldifit.blogspot.com/2009/06/gradual-programming-bridging-semantic.html>.
- [30] R. Chapman, A. Wellings, and A. Burns. Worst-case timing analysis of exception handling in Ada. In L. Collingbourne, editor, *Ada: Towards Maturity. Proceedings of the 1993 AdaUK conference*, pages 148–164. IOS Press: London, 1993.
- [31] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1–59593–031–0. doi: 10.1145/1094811.1094852.
- [32] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 7(3):250–295, 1998. ISSN 1049–331X. doi: 10.1145/287000.287004.
- [33] H. Y. Chen, T. H. Tse, and T. Y. Chen. Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Trans. Softw. Eng. Methodol.*, 10(1):56–109, 2001. ISSN 1049–331X. doi: 10.1145/366378.366380.
- [34] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1–58113–202–6. doi: 10.1145/351240.351266.
- [35] Q. Cui and J. D. Gannon. Data-oriented exception handling. *IEEE Trans. Softw. Eng.*, 18(5):393–401, 1992. ISSN 0098–5589. doi: 10.1109/32.135772.

-
- [36] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In M. Jazayeri, D. Musser, and R. Loos, editors, *Selected Papers from the International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39, London, UK, 2000. Springer-Verlag. ISBN 3-540-41090-2.
- [37] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Some features of the SIMULA 67 language. In *Proceedings of the second conference on Applications of simulations*, pages 29–31. Winter Simulation Conference, 1968.
- [38] V. David. *Language Constructs for C++-like Languages: Tools and Extensions*. PhD thesis, Research School in Information and Communication Technology (ICT), University of Bergen, Norway, 2009.
- [39] E. W. Dijkstra. *On the role of scientific thought (EWD 447)*, pages 60–66. Springer-Verlag New York, 1982. ISBN 0-387-90652-5. URL <http://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html>.
- [40] T. Dinesh, M. Haverlaen, and J. Heering. An algebraic programming style for numerical software and its optimization. *Scientific Programming*, 8(4):247–259, 2000.
- [41] C. Dony. Exception handling and object-oriented programming: Towards a synthesis. *SIGPLAN Not.*, 25(10):322–330, 1990. ISSN 0362-1340. doi: 10.1145/97946.97984.
- [42] R.-K. Doong. *An approach to testing object-oriented programs*. PhD thesis, Polytechnic University, Brooklyn, NY, USA, 1993.
- [43] R.-K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 165–177, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-449-X. doi: 10.1145/120807.120822.
- [44] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2): 101–130, 1994. ISSN 1049-331X. doi: 10.1145/192218.192221.
- [45] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of REFLECTION 2001: 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 170–186, Kyoto, Japan, 2001. Springer-Verlag Berlin Heidelberg. ISBN 978-3-540-42618-9. doi: 10.1007/3-540-45429-2_13.

- [46] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp Symb. Comput.*, 5(4):295–326, 1992. ISSN 0892–4635. doi: 10.1007/BF01806308.
- [47] ECMA–334. *ECMA-334: C# Language Specification*, 4th edition, June 2006. URL <http://www.ecma-international.org/publications/standards/Ecma-334.htm>.
- [48] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM. ISBN 978–1–59593–786–5. doi: 10.1145/1297027.1297029.
- [49] R. W. Floyd. On ambiguity in phrase structure languages. *Commun. ACM*, 5(10):526, 1962. ISSN 0001–0782. doi: 10.1145/368959.368993.
- [50] J. D. Gannon, P. R. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981. ISSN 0164–0925. doi: 10.1145/357139.357140.
- [51] M.–C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1995. ISBN 3–540–59293–8.
- [52] M.–C. Gaudel and P. L. Gall. Testing data types implementations from algebraic specifications. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 209–239. Springer, 2008. doi: 10.1007/978–3–540–78917–8.
- [53] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 28–38, New York, NY, USA, 1986. ACM Press. ISBN 0–89791–200–4. doi: 10.1145/319838.319848.
- [54] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978.
- [55] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, pages 221–256, London, UK, 1984. Springer-Verlag. ISBN 3–540–12896–4.

-
- [56] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991. ISSN 0360-0300. doi: 10.1145/103162.103163.
- [57] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975. ISSN 0001-0782. doi: 10.1145/361227.361230.
- [58] K. Gopinath and J. L. Hennessy. Copy elimination in functional languages. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 303–314, New York, NY, USA, 1989. ACM Press. ISBN 0-89791-294-2. doi: 10.1145/75277.75304.
- [59] J. Gosling, B. Joy, and G. L. Steele, Jr. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, USA, 1996. ISBN 0-201-63451-1.
- [60] P. Gottschling. Fundamental algebraic concepts in concept-enabled C++. Technical Report TR639, Department of Computer Science, Indiana University, 2006.
- [61] P. Graham. Common LISP macros. *AI Expert*, 3(3):42–53, 1987. ISSN 0888-3785.
- [62] D. Gregor and A. Lumsdaine. Core concepts for the C++0x standard library (revision 2). Technical Report N2621=08-0131, JTC1/SC22/WG21 – The C++ Standards Committee, May 2008.
- [63] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167499.
- [64] D. Gregor, B. Stroustrup, J. Siek, and J. Widman. Proposed wording for concepts (revision 7). Technical Report N2710=08-0220, JTC1/SC22/WG21 – The C++ Standards Committee, July 2008.
- [65] W. J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5): 661–674, 1999. ISSN 0098-5589. doi: 10.1109/32.815325.
- [66] J. V. Guttag. Notes on type abstraction (version 2). *IEEE Trans. Softw. Eng.*, 6(1):13–23, 1980. ISSN 0098-5589. doi: 10.1109/TSE.1980.230209.

- [67] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10:27–52, 1978.
- [68] J. V. Guttag and J. J. Horning. Report on the Larch shared language. *Science of Computer Programming*, 6:103–134, 1986.
- [69] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12):1048–1064, 1978. ISSN 0001-0782. doi: 10.1145/359657.359666.
- [70] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Softw.*, 2(5):24–36, 1985. ISSN 0740-7459. doi: 10.1109/MS.1985.231756.
- [71] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996. ISSN 0164-0925. doi: 10.1145/227699.227700.
- [72] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, 1990. ISSN 0098-5589. doi: 10.1109/32.62448.
- [73] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
- [74] M. Haverdaen. Institutions, property-aware programming and testing. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 21–30, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-086-9. doi: 10.1145/1512762.1512765.
- [75] M. Haverdaen and E. Brkic. Structured testing in Sophus. In E. Coward, editor, *Norsk informatikkonferanse NIK'2005*, pages 43–54. Tapir akademisk forlag, Trondheim, Norway, 2005. URL <http://www.nik.no/2005/>.
- [76] M. Haverdaen and H. A. Friis. Coordinate-free numerics: all your variation points for free? *Int. J. Comput. Sci. Eng.*, 5(x), 2009. ISSN 1742-7185.
- [77] M. Haverdaen and K. T. Kalleberg. JAXT and JDI: the simplicity of JUnit applied to axioms and data invariants. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 731–732, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-220-7. doi: 10.1145/1449814.1449834.

-
- [78] M. Haveraaen and E. G. Wagner. Guarded algebras: Disguising partiality so you won't know whether it's there. In *Recent Trends In Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 3–11. Springer Verlag, 2000. ISBN 978-3-540-67898-4.
- [79] M. Haveraaen, H. A. Friis, and T. A. Johansen. Formal software engineering for computational modelling. *Nordic Journal of Computing*, 6(5):241–270, 1999.
- [80] M. Haveraaen, H. A. Friis, and H. Munthe-Kaas. Computable scalar fields: a basis for PDE software. *Journal of Logic and Algebraic Programming*, 65(1):36–49, September–October 2005. doi: 10.1016/j.jlap.2004.12.001.
- [81] I. Hill. Faults in function, in Algol and Fortran. *Comp.J.*, 14(3): 315–316, 1970.
- [82] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. ISSN 0001-0782. doi: 10.1145/363235.363259.
- [83] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
- [84] C. A. R. Hoare. Hints on programming-language design. In C. B. Jones, editor, *Essays in computing science*, pages 193–216. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-284027-8.
- [85] M. Hof, H. Mössenböck, and P. Pirkelbauer. Zero-overhead exeption handling using metaprogramming. In *SOFSEM '97: Proceedings of the 24th Seminar on Current Trends in Theory and Practice of Informatics*, pages 423–431, London, UK, 1997. Springer-Verlag. ISBN 3-540-63774-5.
- [86] M. Hughes and D. Stotts. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 53–61, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-787-1. doi: 10.1145/229000.226301.
- [87] J. Ichbiah, J. Barnes, R. Firth, and M. Woodger. *Rationale for the design of the Ada programming language*. Cambridge University Press, New York, NY, USA, 1981. ISBN 0-521-39267-5.

- [88] ISO/IEC 9899. *ISO/IEC 9899: Programming languages – C*. ISO/IEC JTC1/SC22/WG14, 1999. URL <http://www.open-std.org/JTC1/SC22/WG14/www/standards>.
- [89] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: a type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1):23–66, 2006. ISSN 0304–3975. doi: 10.1016/j.tcs.2006.07.031.
- [90] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In R. Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, September 2001.
- [91] K. T. Kalleberg. User-configurable, high-level transformations with CodeBoost. Master’s thesis, University of Bergen, P.O.Box 7803, N-5020 Bergen, Norway, March 2003.
- [92] K. T. Kalleberg. *Abstractions for Language-Independent Program Transformations*. PhD thesis, University of Bergen, Norway, Postboks 7800, 5020 Bergen, Norway, June 2007. ISBN 978–82–308–0441–4.
- [93] K. T. Kalleberg and E. Visser. Fusing a transformation language with an open compiler. In T. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA’07)*, ENTCS, pages 18–31, Braga, Portugal, March 2007. Elsevier.
- [94] A. C. Kay. The early history of smalltalk. pages 511–598, 1996. doi: 10.1145/234286.1057828.
- [95] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001: Object-Oriented Programming: 15th European Conference*, volume 2072 of LNCS, pages 327–353. Springer-Verlag, June 2001. ISBN 3–540–42206–4.
- [96] P. Klint, A. T. Kooiker, and J. J. Vinju. Language parametric module management for IDEs. *Electr. Notes Theor. Comput. Sci.*, 203(2):3–19, 2008. doi: 10.1016/j.entcs.2008.03.041.
- [97] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
- [98] D. E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974. ISSN 0360–0300. doi: 10.1145/3556635.356640.

-
- [99] R. Lai. A survey of communication protocol testing. *J. Syst. Softw.*, 62(1):21–46, 2002. ISSN 0164-1212. doi: 10.1016/S0164-1212(01)00132-7.
- [100] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. *ICSE*, 00:418, 2000. doi: 10.1109/ICSE.2000.10148.
- [101] B. Liskov. A history of CLU. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-570-4. doi: 10.1145/154766.155367.
- [102] B. Liskov and S. Zilles. Specification techniques for data abstractions. In *Proceedings of the international conference on Reliable software*, pages 72–87, New York, NY, USA, 1975. ACM. doi: 10.1145/800027.808426.
- [103] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977. ISSN 0001-0782. doi: 10.1145/359763.359789.
- [104] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheiffler, and A. Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981. ISBN 3-540-10836-X.
- [105] P. Louridas. JUnit: Unit testing and coding in tandem. *IEEE Softw.*, 22(4):12–15, 2005. ISSN 0740-7459. doi: 10.1109/MS.2005.100.
- [106] D. C. Luckham and W. Polak. Ada exception handling: an axiomatic approach. *ACM Trans. Program. Lang. Syst.*, 2(2):225–233, 1980. ISSN 0164-0925. doi: 10.1145/357094.357100.
- [107] M. D. MacLaren. Exception handling in PL/I. In *Proceedings of an ACM conference on Language design for reliable software*, pages 101–104. ACM Press, 1977. doi: 10.1145/800022.808316.
- [108] W. Maddox. Semantically-sensitive macroprocessing. Technical Report UCB/CSD 89/545, Computer Science Division (EECS), University of California, Berkeley, CA, 1989.
- [109] M. Masood, E. Birkenes, K. T. Kalleberg, M. Haverlaen, and A. H. Bagge. Axiom-based testing of Java collections with JAXT. Technical Report 388, Department of Informatics, University of Bergen, P.O.Box 7805, N-5020 Bergen, Norway, August 2009. URL <http://www.ii.uib.no/publikasjoner/texrap/>.

- [110] B. Meyer. *Eiffel: The language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-247925-7.
- [111] B. Meyer. *Object-Oriented Software Construction, 2nd ed.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0136291554.
- [112] B. Meyer. Principles of language design and evolution. In J. Davies, B. Roscoe, and J. Woodcok, editors, *Millenial Perspectives in Computer Science*, Cornerstones of Computing, pages 229–246. Palgrave, 1999.
- [113] B. Meyer. Applying “Design by contract”. *Computer*, 25(10):40–51, 1992. ISSN 0018-9162. doi: 10.1109/2.161279.
- [114] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML – Revised*. MIT Press, 1997. ISBN 0262631814.
- [115] P. D. Mosses, editor. *CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004. ISBN 3-540-21301-5. doi: 10.1007/b96103.
- [116] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.
- [117] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001, EPFL, 1015 Lausanne, Switzerland, 2006. URL <http://www.scala-lang.org/docu/files/ScalaOverview.pdf>.
- [118] K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In R. Bodik, editor, *14th International Conference on Compiler Construction (CC’05)*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer-Verlag, April 2005. doi: 10.1007/b107108.
- [119] POSIX.1. *The Single UNIX® Specification, Version 3 / IEEE Std 1003.1-2001*. The Open Group, 2004. URL <http://www.unix.org/version3/online.html>.
- [120] J. Palme. Protected program modules in Simula 67. *Modern Datateknik*, 12:8, 1973.

-
- [121] K. M. Pitman. Condition handling in the Lisp language family. In *Advances in Exception Handling Techniques*, pages 39–59. Springer-Verlag, 2000.
- [122] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of Euclid. In *Proceedings of an ACM conference on Language design for reliable software*, pages 11–18, 1977. doi: 10.1145/800022.808307.
- [123] I. S. W. B. Prasetya, T. E. J. Vos, and A. Baars. Trace-based reflexive testing of OO programs. Technical Report UU-CS-2007-037, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2007. URL <http://www.cs.uu.nl/research/techreps/repo/CS-2007/2007-037.pdf>.
- [124] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM Press. doi: 10.1145/800027.808467.
- [125] A. Romanovsky and B. Sandén. Except for exception handling ... *Ada Lett.*, XXI(3):19–25, 2001. ISSN 1094-3641. doi: 10.1145/568671.568678.
- [126] D. Saff. Theory-infected: or how I learned to stop worrying and love universal quantification. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 846–847, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: 10.1145/1297846.1297919.
- [127] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 67–77, New York, NY, USA, 1985. ACM. ISBN 0-89791-147-4. doi: 10.1145/318593.318614.
- [128] D. Sannella and A. Tarlecki. Extended ML: An institution-independent framework for formal program development. In *Proceedings of the Tutorial and Workshop on Category Theory and Computer Programming*, pages 364–389, London, UK, 1986. Springer-Verlag. ISBN 3-540-17162-2.
- [129] D. Sannella and A. Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Comput. Surv.*, page 10, 1999. ISSN 0360-0300. doi: 10.1145/333580.333589.

- [130] S. Schmitz. An experimental ambiguity detection tool. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 69–84, Amsterdam, The Netherlands, The Netherlands, 2008. Elsevier Science Publishers B. V. doi: 10.1016/j.entcs.2008.03.045.
- [131] A. C. Schwerdfeger and E. R. Van Wyk. Verifiable composition of deterministic grammars. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 199–210, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542499.
- [132] K. Sen. Effective random testing of concurrent programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321679.
- [133] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996. ISBN 0-201-44211-6.
- [134] M. Shaw, W. A. Wulf, and R. L. London. Abstraction and verification in alphas: defining and specifying iteration and generators. *Commun. ACM*, 20(8):553–564, 1977. ISSN 0001-0782. doi: 10.1145/359763.359782.
- [135] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. doi: 10.1145/581690.581691.
- [136] J. G. Siek and A. Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 466–467, London, UK, 1998. Springer-Verlag. ISBN 3-540-65460-7.
- [137] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 451–464, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167511.
- [138] P. Sommerlad. *C++ Unit Testing Easier*, 2009. URL <http://r2.ifs.hsr.ch/cute>.

-
- [139] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002. Australian Computer Society, Inc.
- [140] T. A. Standish. Extensibility in programming language design. *SIGPLAN Not.*, 10(7):18–21, 1975. ISSN 0362-1340. doi: 10.1145/987305.987310.
- [141] F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167514.
- [142] P. D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In D. Wells and L. A. Williams, editors, *Proceedings of XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002*, volume 2418 of *Lecture Notes in Computer Science*, pages 131–143. Springer-Verlag, 2002. ISBN 3-540-44024-0.
- [143] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.
- [144] B. Stroustrup. Evolving a language in and for the real world: C++ 1991–2006. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 4–1–4–59, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X. doi: 10.1145/1238844.1238848.
- [145] B. Stroustrup and G. D. Reis. Supporting SELL for high-performance computing. In *Languages and Compilers for Parallel Computing (LCPC'05)*, volume 4339 of *Lecture Notes in Computer Science*, pages 458–465. Springer-Verlag, 2005. doi: 10.1007/978-3-540-69330-7_33.
- [146] S. T. Taft, R. A. Duff, R. Brukardt, and E. Plödereder, editors. *Consolidated Ada Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*, volume 2219 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. ISBN 3-540-43038-5.
- [147] X. Tang and J. Järvi. Concept-based optimization. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 97–108, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-086-9. doi: 10.1145/1512762.1512772.

- [148] Transformers Group. *Transformers*. The Transformers Group, LRDE, EPITA, 2006. URL <http://www.lrde.epita.fr/cgi-bin/twiki/view/Transformers/Transformers>.
- [149] H. Truong. Guaranteeing resource bounds for component software. In *EMOODS'05*, volume 3535 of *LNCS*, pages 179–194. Springer-Verlag, 2005. ISBN 3-540-26181-8. doi: 10.1007/11494881_12.
- [150] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y.
- [151] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer-Verlag. ISBN 3-540-41861-X.
- [152] G. van Rossum and J. F. L. Drake. *Python Language Reference Manual*. Network Theory Ltd, Bristol, UK, 2003. ISBN 0-9541617-8-5.
- [153] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. ISSN 1040-6042. Reprinted in *C++ Gems*, ed. Stanley Lippman.
- [154] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
- [155] E. Visser. Transformations for abstractions. In J. Krinke and G. Antoniol, editors, *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 3–12, Budapest, Hungary, October 2005. IEEE Computer Society Press.
- [156] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer-Verlag, 1995.
- [157] G. V. Wilson. Extensible programming for the 21st century. *Queue*, 2(9):48–57, 2005. ISSN 1542-7730. doi: 10.1145/1039511.1039534.
- [158] J. M. Wing and C. Gong. Experience with the Larch Prover. In *Conference proceedings on Formal methods in software development*, pages 140–143, New York, NY, USA, 1990. ACM. ISBN 0-89791-415-5. doi: 10.1145/99569.99835.

-
- [159] N. Wirth. The programming language pascal. *Acta Inf.*, 1:35–63, 1971.
- [160] D. B. Wortman and J. R. Cordy. Early experiences with euclid. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 27–32, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6.
- [161] W. A. Wulf, R. L. London, and M. Shaw. An introduction to the construction and verification of alghard programs. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 390, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
- [162] B. Yu, L. Kong, Y. Zhang, and H. Zhu. Testing java components based on algebraic specifications. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 190–199, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3127-4. doi: 10.1109/ICST.2008.39.
- [163] M. Zalewski and S. Schupp. C++ concepts as institutions: a specification view on concepts. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 76–87, New York, NY, USA, 2007. ACM. ISBN 978-1-60558-086-9. doi: 10.1145/1512762.1512770.
- [164] H. Zhu. A note on test oracles and semantics of algebraic specifications. In *QSIC '03: Proceedings of the Third International Conference on Quality Software*, pages 91–98, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2015-4.

Citation Index

- Aho et al. [1], 32
 Allen et al. [2007], 157, 158
 Andersen and Brabrand [3], 141, 144
 Antoy and Hamlet [2000], 6
 Antoy and Hamlet [5], 73
 Antoy [4], 73
 Astesiano et al. [2002], 5
 Bagge and Haveraaen [11], 38, 41, 44, 48, 49
 Bagge and Haveraaen [7], 22, 31, 33, 57, 65, 140, 141
 Bagge and Haveraaen [8], 71, 137
 Bagge and Kalleberg [2006], 164
 Bagge and Kalleberg [9], 77
 Bagge et al. [10], 74
 Barnes [12], 80
 Beck and Gamma [15], 74
 Beck and Gamma [2009], 6
 Becker [16], 56, 58, 126
 Beck [13], 55
 Beck [14], 56
 Beck [1998], 6
 Beck [2002], 6
 Bernardy et al. [2008], 5, 154
 Bobrow et al. [18], 111
 Borghi et al. [19], 74, 99
 Borgida [20], 102
 Borovanský et al. [21], 49
 Boyle et al. [22], 41, 43
 Brabrand and Schwartzbach [23], 144
 Bravenboer and Visser [24], 144
 Bravenboer et al. [2006], 7
 Bravenboer et al. [25], 43, 49, 74, 99
 Bravenboer et al. [26], 128, 136
 Callahan et al. [2004], 157
 Cantor [28], 129
 Chang et al. [2009], 8
 Chapman et al. [30], 103
 Charles et al. [2005], 157
 Chen et al. [32], 57, 72
 Chen et al. [33], 72
 Claessen and Hughes [2000], 6, 13
 Claessen and Hughes [34], 51, 67, 68
 Cui and Gannon [35], 102
 Czarnecki et al. [2000], 163
 Czarnecki et al. [36], 50
 Dahl et al. [1968], 147
 David [2009], 9, 156
 David [38], 107
 Dijkstra [1982], 3
 Dinesh et al. [2000], 15, 155, 164
 Dinesh et al. [40], 17, 32, 34
 Dony [41], 103
 Doong and Frankl [43], 72
 Doong and Frankl [44], 56, 57, 71
 Doong [42], 72
 Douence et al. [2001], 3
 Dybvig et al. [46], 144
 ECMA-334 [47], 115
 Ekman and Hedin [48], 133, 144
 Floyd [49], 129
 Gannon et al. [1981], 6, 12

- Gannon et al. [50], 46, 50, 56, 65, 74
Gaudel and Gall [52], 57, 74
Gaudel [51], 57
Gifford and Lucassen [53], 32
Goguen and Burstall [1984], 151
Goguen et al. [1978], 149
Goguen et al. [54], 72
Goldberg [56], 83
Goodenough [1975], 14
Goodenough [57], 81–83, 102, 103
Gopinath and Hennessy [58], 32
Gosling et al. [59], 83
Gottschling [60], 38, 61, 75, 126
Graham [61], 144
Gregor and Lumsdaine [62], 61
Gregor et al. [2008], 152
Gregor et al. [63], 48, 56, 58
Gregor et al. [64], 37, 39
Gutjahr [65], 68
Guttag and Horning [1978], 149
Guttag and Horning [67], 72
Guttag and Horning [68], 22
Guttag et al. [1978], 149
Guttag et al. [1985], 5
Guttag et al. [69], 72, 73
Guttag [66], 73
Hall et al. [1996], 5, 153
Hall et al. [71], 126
Hamlet and Taylor [72], 51, 68
Hamlet [1994], 7
Hamlet [73], 68
Haveraaen and Brkic [2005], 15
Haveraaen and Brkic [75], 38, 46, 48, 56, 57, 73
Haveraaen and Friis [2009], 155
Haveraaen and Kalleberg [2008], 13, 156
Haveraaen and Kalleberg [77], 57, 74
Haveraaen and Wagner [2000], 13, 152, 165
Haveraaen and Wagner [78], 77
Haveraaen et al. [1999], 9, 155
Haveraaen et al. [79], 32
Haveraaen et al. [80], 63, 73
Haveraaen [74], 38, 46
Hill [81], 82, 101
Hoare [1969], 150
Hoare [1972], 6, 149
Hoare [1989], 156, 158
Hof et al. [85], 102
Hughes and Stotts [1996], 6, 12
Hughes and Stotts [86], 56, 65, 74
Ichbiah et al. [1981], 148
Johnsen et al. [2006], 157
Jones et al. [90], 22, 49
Kalleberg and Visser [2007], 7
Kalleberg [91], 130
Kalleberg [92], 74
Kay [1996], 148
Kiczales et al. [2001], 4
Kiczales et al. [95], 101, 104
Klint et al. [2008], 7
Knuth [1974], 4
Knuth [97], 129
Lai [2002], 7
Lippert and Lopes [100], 87, 103
Liskov and Zilles [102], 72
Liskov and Zilles [1975], 149
Liskov et al. [103], 83, 101
Liskov et al. [1981], 148
Liskov [101], 101
Liskov [1993], 9, 148
Louridas [105], 46, 50, 74
Luckham and Polak [106], 102
MacLaren [107], 102
Maddox [108], 145
Masood et al. [109], 74
Masood et al. [2009], 156
Meyer [110], 85
Meyer [111], 85
Meyer [1992], 5, 150
Meyer [1999], 156–159
Milner et al. [114], 83
Mosses [115], 22, 40
Nystrom et al. [116], 144

Odersky et al. [2006], 148, 154, 157, 159
 Olmos and Visser [118], 29
 POSIX.1 [119], 81, 98
 Palme [1973], 147
 Pitman [121], 103
 Popek et al. [122], 32
 Popek et al. [1977], 149, 165
 Prasetya et al. [123], 68
 Randell [124], 101
 Romanovsky and Sandén [125], 80, 102
 Saff [126], 51, 57
 Sannella and Tarlecki [1985], 151
 Sannella and Tarlecki [1986], 151
 Sannella and Tarlecki [1999], 151
 Schmitz [130], 129
 Schwerdfeger and Van Wyk [131], 129
 Sen [132], 75
 Shalit [133], 129
 Shaw et al. [1977], 148
 Sheard and Jones [135], 144
 Siek and Lumsdaine [136], 73
 Simonyi et al. [137], 133
 Sommerlad [138], 74
 Spinczyk et al. [139], 92, 104
 Standish [140], 133, 144
 Steimann [2006], 4
 Stotts et al. [142], 57, 65, 74
 Stotts et al. [2002], 6, 13
 Stroustrup and Reis [2005], 7
 Stroustrup [143], 83
 Stroustrup [2007], 158, 159
 Taft et al. [146], 83, 116
 Tang and Järvi [147], 22, 31, 41, 48, 50, 57
 Transformers Group [148], 74
 Truong [149], 80
 Veldhuizen [153], 33, 144
 Visser [154], 99, 129
 Visser [155], 77
 Visser [1997], 7
 Wadler [156], 103
 Wilson [157], 133, 144
 Wing and Gong [1990], 5
 Wirth [1971], 149
 Wortman and Cordy [1981], 150, 155
 Wulf et al. [1976], 148
 Yu et al. [162], 56
 Zalewski and Schupp [2007], 152
 Zhu [164], 71
 ISO/IEC 9899 [88], 99
 van Rossum and Drake [152], 83
 van den Brand et al. [150], 129
 van den Brand et al. [151], 129
 van den Brand et al. [2001], 7

Index

- 1 (return code), 82
- abstract data type, *see*
 - abstraction, data
- abstract syntax tree, *see* AST
- abstraction, 2
 - abstract values, 119
 - and compilers, 136
 - by language extension, 136
 - by parametrisation, 2
 - by specification, 2
 - choice of, 155
 - control, 2
 - data, 3, 119–123, 147–148
 - procedural, 2, 108–116
- ac (axiom group), 41, 43
- access
 - outside module, 124
- active libraries, 50
- Ada, 148
 - exceptions, 83, 101, 102, 151
 - parameters, 116
- ADT, *see* abstraction, data
- advice, 4, 104
- agile methods, 6, 56
- alert (declaration), 88
- alert (clause), 90
- alerts, 77–104, *see also* errors
 - abstraction, 94
 - declaring kinds, 88–89
 - defaulting, 93
 - exceptions, 83
 - feedback, 94–95
 - granularity, 95–97
 - handler, 82
 - handlers, 91–93
 - implementation, 99–101
 - language extension, 87–98
 - legacy code, 97–98
 - mechanisms, 82–85
 - policies, 85
 - reporting, 82, 90–91
 - separation of concerns,
 - 86–87
 - syntax, 88, 90, 92, 96
- Algol
 - error handling, 101
- aliasing
 - prevention of, 24, 31
- Alphard, 148
 - specification in, 149
- AMD, 8
- analysis, 35
 - coverage, 56
 - data-flow, 28, 29, 114, 130,
 - 141
 - global, 137
 - semantic, 130
 - rules for, 140
 - types, 99
- Any
 - in Scala, 148
- API, 82–84, 97, 156
- applications
 - multi-threaded, 75
- arguments
 - immutable, 115
 - modification of, 113, 114

- ArrayEqual (axiom), 59
- Arrow Cross Militia, 36
- aspect orientation, 3, 4, 137
- AspectC++, 104
- AspectJ, 4, 101, 104
- aspects, 4, 86–87, 101
 - domain-specific, 77, 104
 - for error handling, 103
- assert**, 39, 47
 - in Euclid, 150
- assertions
 - in Euclid, 150
- assignment
 - in Magnolia, 112
- AST, 49, 99, 129, 131
- ASTOOT, 56, 57, 71, 72
- ATerm, 129
- Atlanta, 54
- axiom groups, 39–41, 51, 140
 - ac*, 41, 43
 - propagate*, 41, 45
 - simplify*, 41, 43
 - speedup*, 46
- axiom_group**, 40, 41
- axioms, 10, 35
 - ArrayEqual*, 59
 - Identity*, 39–41, 46, 48, 58
 - arbitrary code in, 47
 - for rewriting, 41–46
 - for testing, 12–13, 46–48
 - in C++, 39–41
 - in Magnolia, 108, 125–127, 140, 141
 - libraries, 70
 - selection of, 72–73
 - writing, 70
- axpy, 44
- Bank of America Plaza, 54
- binding
 - dynamic, 151
- BLAS, 44
- bottom-up, 43
- bounded polymorphism, 152
- BoundedStack (concept), 65
- brittle code, 1
- Budapest, 36
- by-copy, 116
- by-reference, 116
- by-value, 114
- C, 9, 99–101
 - expressions, 125
- C++, 9, 20, 35, 56, 100, 126, 148
 - concepts, 58–59, 152–153
 - difficulty of processing, 107
 - exceptions, 83
 - frontend, 48, 49
 - templates, 49
- C++0x, 20, 21, 152
- C#
 - parameters, 115–116
- call**, 125
- CASCAT, 56
- CASL, 5, 22
- Cell (processor), 8
- checking
 - of arguments, 77
 - preconditions, 77
- chickens
 - harming of, 104
- clarity
 - of code, 2, 34
 - of notation, 22, 29, 33
- class invariant
 - in Eiffel, 150
- classes, *see* types
- clonability
 - of objects, 31
- clos, 111
- CLU, 9, 147–148
 - constraining type
 - parameters, 149
 - exceptions, 83, 101, 151
 - interfaces, 149
 - specification in, 149
- cluster (in CLU), 148
- code

- reuse of, 33, 155
- three-address, 32
- CodeBoost, 41, 49
- collections
 - in Euclid, 150
- comma operator
 - in C++, 29, 35, 64
- Common Lisp, *see* Lisp
- Common Lisp Object System, 111
- CommutativeRing (concept), 22, 126
- commutativity, 57
- compilation
 - by transformation, 128
 - of functional languages, 32
- compiler
 - option, 43
 - pipeline, 99, 128
- compilers
 - backend, 160
 - design, 128
 - for Magnolia, 34, 128–133
 - high-performance, 8
 - optimising, 27, 37
 - phases, 129–131
- computing
 - high-performance, 8–9
- concept, 10
- concept
 - in C++, 37, 39–41
 - in Magnolia, 125–127
- concept map, 21, 58, 127, 153, 154
- concept-based programming, 160
- ConceptGCC, 41, 48, 50
- concepts, 5, 20, 149–154
 - BoundedStack*, 65
 - CommutativeRing*, 22, 126
 - DefaultIndexable*, 63
 - EqualityComparable*, 65, 71
 - Equivalence*, 126
 - HasDataSet*, 69
 - Indexable*, 23, 24, 37, 59, 60, 63, 143
 - Integer*, 126
 - LessThanComparable*, 37, 38, 153
 - Monoid*, 39–41, 46, 58, 62, 63, 154
 - SameShape*, 59
 - algebraic, 38
 - as specifications, 152
 - auto, 59
 - hierarchy of, 38
 - in C++, 35, 39–41, 58–59, 70, 72–75, 152–153
 - in Magnolia, 11–12, 23, 108, 125–127, 132
- concerns
 - cross-cutting, 3
 - separation of, 3–4, 86–87
- concurrency
 - and parameter passing, 116
- condition
 - in axioms, 41
- confluence, 42
- congruence
 - axioms, 75
 - relation, 70, 121
- const references
 - in C++, 64, 71
- constant, 109
- constraints
 - on type parameters, 58, 126, 149, 152
- constructors
 - default, 125
 - in Magnolia, 114, 124
- contract
 - design by, 5, 85, 103, 150–151
 - inherited, 150
- conversion
 - implicit, 111, 114, 118
- copy elimination, 32
- coverage, in testing, 7

- Cray, 8
- cross-cutting concerns, 137
- CUTE, 74
- Daistish, 56, 74
- DAISTS, 50, 56, 65, 74
- Danube, 36
- data dependency, 135
- data hiding, *see* encapsulation
- data invariant, 121
 - in Alaphard, 149
 - in Eiffel, 150, 151
 - in Euclid, 150
- data structures
 - in Magnolia, 120–125
 - opening of, 122
- data types, *see* types
- data-flow
 - analysis, 28, 29, 114, 130, 141
 - behaviour, 113
- David, Valentin, 15, 52
- declaration forms, 20
- default action, 86
- DefaultIndexable (concept), 63
- del, 114
- destructors
 - in Magnolia, 114
- desugaring, 136
- development
 - test-driven, 55
- Dijkstra, Edsger W., 3
- documentation
 - structured, 50, 98
- domain-specific language, *see* DSL
- dot notation, 120
- DSL, 141
 - embedded, 137
- Dylan, 129
- Eclipse, 64, 74
- Eiffel, 5, 56, 85
 - exceptions, 151
 - object orientation, 150
 - specification in, 150–151
- ELAN, 49
- encapsulation, 116, 119, 124
- end user, 20
- ensures
 - in CLU, 149
 - in Eiffel, 150
- equality
 - in axioms, 57
- EqualityComparable (concept), 65, 71
- equations
 - conditional, 42, 57, 58
- equivalence
 - behavioural, 57
 - in Magnolia, 121
 - in testing, 70–71
- Equivalence (concept), 126
- errno, 83
- errors, *see also* alerts
 - abstraction, 3
 - alerts, 77–104
 - aspects, 103
 - avoidance of, 80
 - condition system, 84
 - contracts, 85
 - event handlers, 84
 - exceptions, 83
 - global flag, 81–83
 - goto, 85
 - guarding, 84
 - handling, 81, 83–85
 - impossible, 56
 - in axioms, 70
 - in different languages, 101–103
 - long jumps, 83
 - posix codes, 89
 - reporting, 81–85
 - return codes, 81, 82
 - vs. normality, 87
- Euclid, 149
 - specification in, 149–150

evaluation
 axiom-based testing, 73–74
 of Magnolia, 155

evil
 root of, 4

exceptional situations, 80

exceptions, 81, 83, 148
 and alerts, 91
 and parameter passing, 116
 checked/unchecked, 84
 handlers, 84
 in testing, 47
 propagation, 86

expansion
 of macros, 137

experiences
 axiom-based testing, 73–74
 mutification, 155

expression templates, 33

expressions
 in Magnolia, 125

extension, *see* language extension

extern
 in C++, 159

failure, *see also* errors
 in transformations, 42
 of type matching, 118

fib, 21

fields, 120

final
 in Java, 116

flexibility, 1, 33, 155

fluent languages, 32

forall, 117, 127

form (in Alaphard), 148, 149

Fortran, 9
 error handling, 101

forward declarations, 110

FPGA, 8

frontend, 107
 C++, 48, 49
 Magnolia, 129

function
 in Magnolia, 108

function pointers, 100

functional languages
 compilation of, 32
 expressions in, 125

functionalisation, 10, 21, 110, 131
 compiler phase, 130
 evaluation of, 155
 of declarations, 25–26
 of procedure calls, 27–28

functions
 in Magnolia, 21, 23, 108, 131
 self-checking, 73
 single-return, 131
 type functions, 117
 virtual, 151

funspaces, 95–97

fusion, 144

gd (graphics library), 115

generators
 in Alaphard, 148

generic programming, 152

genericity, 148

generics
 in C++, 37, 152–153
 in Magnolia, 24, 127–128

giv, 114

Glasgow Haskell Compiler, 49

goto
 goto, 85

GPCE, 15, 55, 77

grammar
 ambiguity, 129
 modularity, 129

granularity
 of error handling, 87

group clause, 40

guarded algebras, 13

guarding, 77, 84, 90

handler (declaration), 94

- handlers, *see* alerts, errors
- handling, *see* alerts, errors
- HasDataSet (concept), 69
- hash table, 86
- Haskell, 9, 20
 - axiom-based testing in, 51
 - exceptions, 103
 - laws, 51
 - type classes, 126, 153
- Haveraaen, Magne, 15
- heap, 115
- Hoare, Tony, 156
- HPC, 8
- hygiene, 145
- I/O
 - and mutification, 31
 - and parameter modes, 115
- IBM
 - RoadRunner, 8
- Identity (axiom), 39–41, 46, 48, 58
- IDEs, 7–8, 64
- IEEE numbers, 83, 87
- if, 125
- implementation, 10, 35
 - in Magnolia, 108
 - of alerts, 99–101
 - of axiom-based rewriting, 48–49
 - of axiom-based testing, 74–75
 - of Magnolia, 128–131
- implementation signature, 21
- implementer, 20
- implicit
 - in Scala, 154
- impure
 - objects, 32
 - procedures, 108, 142
- in
 - in Ada, 116
- in out
 - in Ada, 116
- Indexable (concept), 23, 24, 37, 59, 60, 63, 143
- inequalities
 - conditional, 58
- informal formal methods, 75
- inheritance, 116
- inlining, 43, 130
- instantiation
 - compiler phase, 130
 - of C++ templates, 49
 - of generic code, 127
 - of operation patterns, 138
 - of operations, 128
 - of types, 117, 127
- institutions, 151
- instructions
 - reordering of, 28
- Integer (concept), 126
- integers
 - finite, 60
- interfacing
 - with other languages, 120
- invariant, *see* data invariant
- invariant
 - in Eiffel, 150
- iterators, 60
- Java, 56, 74, 100, 148, 149
 - collections, 156
 - exceptions, 83
 - expressions, 125
 - parameters, 115–116
 - reporting/handling errors, 86
- JAX, 57, 65, 73, 74
- JAXT, 57, 74
- Jews, 36
- join points, 4, 104
- JUnit, 6, 46, 50, 57
- Kalleberg, Karl Trygve, 15
- Knuth, Donald E., 4
- l-values, 24

- language, 142
- language extension, 77, 160
 - alerts, 87–99
 - global effects, 137, 141
 - in Magnolia, 133–145
 - local effects, 137
 - operation-like, 137
 - scoping, 141
- languages
 - domain-specific, 135, *see* DSL
 - fluent, 32
 - novel features, 160
 - sub-languages, 135
- Larch Shared Language, 22
- laws
 - algebraic, 27, 31
 - in Haskell, 51
- LDTA, 15, 17, 35
- LessThanComparable (concept), 37, 38, 153
- Liskov, Barbara, 9, 72
- Lisp, 137, 138
 - condition system, 103
 - map, 142
- list comprehension, 25
- LOBAS, 71, 72
- longjmp, 83
- lookup (function), 81
- macros, 136, 137
 - expansion, 137
 - hygiene, 145
- Magnolia, 9–14, 20, 23–24, 107–132
 - assignment operator, 112
 - axioms, 125–127
 - compiler, 128–132
 - concepts, 11–12, 108, 125–127, 132
 - congruence relation, 121–124
 - constructors, 114, 124
 - data abstraction, 119–123
 - data invariant, 121–124
 - data structures, 120–125
 - destructors, 114
 - equivalence, 121
 - evaluation, 155
 - expressions, 125
 - extension, 133–145
 - functions, 108, 131
 - generics, 127–128
 - implementation of, 128–131
 - implementations, 108
 - models, 108, 126
 - modules, 119, 131
 - object construction, 124
 - operators, 111–113
 - overloading, 111, 131
 - parameter modes, 113–115
 - parser, 129
 - partiality, 13–14
 - procedures, 109, 131
 - set-element operation, 112
 - signature morphisms, 108
 - signatures, 108
 - status, 131
 - tuples, 117
 - type matching, 117–119
 - types, 116–125
- maintainability, 1
- manual pages, 98
- map, 20, 142
- matching
 - asymmetric, 118
 - of patterns, 118
 - of types, 117–119
 - pattern, 41, 49, 118
 - semantic, 42
- message passing, 148
- meta programming, *see also* language extension
- meta variables, 42
- ML, 20
 - exceptions, 83, 103
- model, 10, 154
- model

- in Magnolia, 126
- modelling
 - of a concept, 108
- modifies**
 - in CLU, 149
- modularity, 148
- module invariant
 - in Euclid, 150
- modules
 - in Magnolia, 119, 131
- monads, 103
- monoid, 11
- Monoid (concept), 39–41, 46, 58, 62, 63, 154
- mouldable, 82
- MTL, 73, 75
- multi-threading, 75, 148
- multi-valued operations, 25, 131
- mutification, 10, 21, 71, 110, 130, 131
 - evaluation of, 155
 - of expressions, 26–27
 - of programs, 28–29
 - performance of, 32
- NaN, 83
- notation, *see also* style
 - algebraic, 33
 - dot, 120
 - proliferation of variants, 20
- Nothing**
 - in Scala, 148
- nrm**, 114
- Null**
 - in Scala, 148
- null**
 - in Java, 86
- numerics, 8–9
- Oberon
 - exception handling, 102
- object**
 - in Scala, 154
- object construction
 - in Magnolia, 124
- object orientation, 100, 148
 - testing, 57, 71–72
- obs, 21, 113
- OCaml, 9
- on (alert handler), 91
- ON (in PL/I), 84
- open**, 122, 123
- opens**, 122
- operations
 - abstract-level, 119, 122
 - extensions, 137
 - generic, 128
 - in Magnolia, 108
 - multi-valued, 25
 - operation patterns, 138–139, 141
 - primitive, 113
 - representation-level, 119, 122–123
 - specialisation of, 132
- operator
 - transform, 140
- operators
 - in Magnolia, 111–113
- Opteron, 8
- optimisation, 38, 160
 - axiom-based, 125
 - high-level, 27, 130
 - low-level, 160
 - premature, 4
- oracle problem, 57
- oracles, 59–60
- out, 21
 - in Ada, 116
 - in C#, 116
 - in Magnolia, 114
- over-specification, 72
- overloading
 - in C++, 49
 - in C, lack of, 100
 - in Magnolia, 111, 117, 131
 - of operation patterns, 138

of operators, 111
resolving, 100, 111

parameter modes, 109
delete, 114
give, 114
in Magnolia, 113–115
normalise, 114
observe, 115
output, 114
update, 114

parameter passing, 115–116

parsing
of C++, 74
of Magnolia, 129

partial operation, 13

partiality, 77
in Magnolia, 13–14

Pascal, 149

patterns, 42
matching, 41, 118
operation, 138–139, 141
replacement, 41
syntax, 137

PL/I, 84
condition system, 101, 151

PlayStation 3, 8

pointcuts, 4, 101, 104

polymorphism
bounded, 152

Portland, OR, 78

POSIX, 79, 81–84, 88, 89, 96–98

post (alert clause), 90

postconditions, 90
and inheritance, 150
in Alphard, 149
in Eiffel, 150
in Euclid, 150

pre (alert clause), 90

preconditions, 77, 90
and inheritance, 150
in Alphard, 149
in Eiffel, 150
in Euclid, 150

private
in Java, 116

proceduralisation
of function declarations, 26

procedure
in Magnolia, 109

procedure calls
in expressions, 115
statements, 125

procedures
impure, 108, 142
in Magnolia, 21, 23, 108, 131
multi-valued, 25, 131

programming
extreme, 6, 56
mouldable, 82
scientific, 8–9

propagate (axiom group), 41, 45

properties
propagation of, 45–46, 51, 130, 160
sorted, 45

prototyping
of language constructs, 107, 160

Python, 110, 148
exceptions, 83

quantification
universal, 51, 58, 69

QuickCheck, 51, 67, 69

random
test data, 68

rational numbers, 68, 121

readonly
in Euclid, 150

ref
in C#, 116

reference types, 115

reflection, 148

reliability, 1, 77, 148, 155

replacement pattern, 41

reporting, *see* alerts, errors

- representation mapping, 6, 73
- require**
 - in Eiffel, 150
- requirements
 - in concepts, 108
- requires**, 127
 - in C++, 153, 154
 - in CLU, 149
 - in Magnolia, 126
- resolution, 130
 - of overloading, 49, 111, 113, 117, 138
 - of types, 117
- retry** (alert handler), 93
- return codes, 77, 81, 82
- reuse, 33, 104, 155
 - and error handling, 79, 97
 - of tests, 61
- rewrite rules, *see* rewriting
 - user-defined, 49
- rewriting
 - axiom-based, 22, 31, 41–46, 50, 140, 160
 - confluence, 42
 - failure, 42
 - semantics-preserving, 38
 - strategies, 42–43
 - success, 42
 - termination, 42
- rings, 22
- robustness, 1, 155
- RtlUnwind**, 83
- Ruby, 148

- SameShape (concept), 59
- Scala, 148–149
- Scheme, 9, 137, 138
- scientific programming, 8–9
- SDF2, 39, 99, 129, 137, 138, 144
- security
 - and argument checking, 77
- Seismod, 32, 155
- semigroup, 12
- setjmp**, 83

- side-effects
 - in axioms, 64–65, 71
- signature, 10, 111
 - algebraic, 35
 - implementation, 21
 - in concepts, 126
 - in Magnolia, 108
 - use, 21
- signature morphism, 10
 - in Magnolia, 108
- signatures, 149
- simplify (axiom group), 41, 43
- Simula, 147
- SLE, 135
- slicing, 28, 130, 131, 141
- Smalltalk, 148
 - exceptions, 103
- SML, *see* ML
- software testing, 6
- Sophus, 9, 50, 56, 73, 155
- SPARK Ada, 80
- specialisation, 132
- specification, 4–5, 35, 125
 - algebraic, 29, 33, 49, 56, 72–73, 149
 - initial, 73
 - loose, 73
 - of abstract data types, 149
 - of object-oriented code, 71–72
 - over-/under-, 72
- speedup (axiom group), 46
- splicing
 - in Haskell, 144
- SSA form, 27, 29
- stack
 - bounded, 65
- static single assignment, 27, 29
- Steel Bridge, 78
- strategies, 42–43, 51
 - bottom-up, 43
 - choice, 43
 - repeat, 43
 - sequence, 43

Stratego, 43, 49, 74
 Stratego/XT, 99
struct
 in C#, 116
 in Magnolia, 120, 124
 style, *see also* notation
 algebraic, 22, 29, 35
 functional, 33
 imperative, 29
 object-oriented, 29
 of calls, 21
 of implementation, 21
 substitution, 44
 success
 in transformations, 42
 of type matching, 118
 supercomputers, 8–9
 symmetry
 in matrices, 46
 syntax
 definition, 129
 extension of, 77, 129, 137
 for alerts, 88, 90, 92, 96
 patterns, 137
 syntax macros, *see* macros
 Széchenyi Street, 36

 TAMPR, 41, 43
 tangling, 87
 TDD, 55, 56
 temporaries, 28, 32, 110, 115
 termination
 of programs, 80
 of rewriting, 42
 test coverage, 7
 test oracle, 46
 testing, 6–7
 axiom-based, 12–13, 38, 46–48, 53–76, 125
 data generation, 65–70
 experience, 73–74
 in Haskell, 51
 limitation of axioms, 47
 object-oriented code, 64–65, 71–72
 of exceptions, 47
 partition, 51
 reuse, 61
 suites, 64
 unit, 50
 TIL (language), 77
 tools, 7–8
 traits
 in Scala, 149, 154
transform, 139
 transform classes, 140
 transformation, *see also* rewriting
 high-level, 35, 130
 Transformers, 74, 99, 104
 transforms, 35, 139–141
 operator, 140
 typecheck, 140
 traversal
 bottom-up, 43, 140
 innermost, 139
 top-down, 140
 tuples
 in Magnolia, 117
 type checking, *see* resolution
 type classes, 5, 126, 153
 types
 abstract, 119–120, 122
 in Scala, 149, 154
 opening of, 122
 algebraic, 51
 analysis, 99
 constrained, 125
 implicit conversion, 118
 in Haskell, 51
 in Magnolia, 23, 116–125, 131
 instantiated, 117
 instantiation, 127
 matching of, 118
 parametrised, 117, 125, 127
 plain, 117
 representation, 120–125

- resolving, 117
 - resource-aware systems, 80
 - simple, 116
 - tuple types, 117
 - type variables, 117, 118
- typing
 - strong, 148
- undefinedness, 80
- unicode, 107
- unification, 118
- union, 124
 - in Magnolia, 120
- unit testing, 6
 - frameworks for, 50
- upd, 21, 114
- use signature, 21
- user, 20, 22
- using, 41
- value types, 115
- values
 - abstract, 119, 121
 - concrete, 121
- var, 125
 - in Euclid, 150
- variable
 - bindings, 118
 - temporary, 115
- variables
 - declaration of, 125
 - global, 31
 - in Magnolia, 125
 - in patterns, 42
 - initialisation of, 114, 125
 - modification in
 - expressions, 114
 - temporary, 28, 115
 - type variables, 117, 118
 - volatile, 32
- verification
 - of abstract data types, 149
 - of programs, 38
- views
 - in Scala, 154
- while, 125
- Willamette River, 78
- Win32, 82–84
- XT5, 8
- xUnit, 50

