A GENERATIVE APPROACH TO MESHING GEOMETRY

A GENERATIVE APPROACH TO MESHING GEOMETRY

By
MUSTAFA ELSHEIKH, B.SC.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the degree
Master of Applied Science in Software Engineering

McMaster University

MASTER OF APPLIED SCIENCE (2010)
(Software Engineering)

McMaster University
Hamilton, Ontario

TITLE: A Generative Approach to Meshing Geometry
AUTHOR: Mustafa Elsheikh, B.Sc. (Al-Azhar University)
SUPERVISOR: Dr. Jacques Carette, Dr. Spencer Smith
NUMBER OF PAGES: viii, 116

Abstract

This thesis presents the design and implementation of a generative geometric kernel suitable for supporting a family of mesh generation programs. The kernel is designed as a program generator which is generic, parametric, type-safe, and maintainable. The generator can generate specialized code that has minimal traces of the design abstractions. We achieve genericity, understandability, and maintainability in the generator by a layered design that adopts its concepts from the affine geometry domain. We achieve parametricity and type-safety by using MetaOCaml's module system and its support for higher order modules. The cost of adopting natural domain abstractions is reduced by combining MetaOCaml's support for multi-stage programming with the technique of abstract interpretation.

Acknowledgments

I would like to thank my supervisors for their support, and my parents for their love.

Contents

Chapter 1

Introduction

This chapter provides the background and motivations for this thesis. The contributions of this work are stated and the thesis organization is given.

This thesis has evolved out of the following works:

- The work by Smith et. al. [SMC07] on improving the quality of scientific computing applications using the program family approach.
- The work by Carette et. al. [Car06, CK05, CK08] on writing generic, efficient, and type-safe generators for a family of Gaussian Elimination algorithms using MetaOCaml's multi-staging facilities.

An explicit goal of this work was to build a program generator for a family of mesh generators that results in improving the quality of mesh generation software. As a first step towards this goal, the *generative geometric kernel* (GGK) was developed. GGK was designed and built using the generative meta-programming facilities of MetaOCaml. The resulting kernel can generate specialized geometric primitives and objects which can be used to support a family of mesh generators.

1.1 Motivation and Goals

The development of scientific computing programs faces several challenges. Some of the prominent challenges are:

- Managing the complexity of the software.
- Managing the anticipated changes in the software.
- Taking advantage of the similarity in the algorithms and data structures.

- Answering the trade-off between efficiency and accuracy.

Traditionally, the focus of scientific computing development is on answering the 'efficiency versus accuracy' trade-off. This leads to undermining other qualities of the software such as usability, reusability, and maintainability.

The use of proper software engineering practices can help in meeting the challenges of scientific computing programs without neglecting software quality. The most prominent software engineering approaches are *object-oriented design patterns*, *component-based development*, *aspect-oriented development*, *program families*, *generic programming* and *generative programming*. Most of those approaches advocate for *modularity*, *abstraction*, and *reuse*.

In particular, the ideas of program families have been recognized as a promising approach to develop quality scientific software. However, the program family approach as mean to analyze and design software does not address the following problem:

- How to achieve high parametricity, yet reduce the cost of abstractions? And how to reduce the cost of abstractions without undermining understandability and maintainability?

Many abstraction mechanisms (e.g., templates, polymorphism) can provide genericity and parametricity. However, they generally result in run-time costs. Some mechanism such as template meta-programming can help in reducing the cost of abstractions. However, the resulting design and code are difficult to understand and hence suffer from maintainability problems.

Generative programming can address the problem of implementing program families with high parametricity and low cost of abstraction. In particular, *multi-stage programming* is a viable approach to develop program generators where the overhead of abstraction can be reduced using program manipulation at different stages of execution. However, a typical problem incurred in generative programming is ensuring that the generated programs are well-formed and well-typed.

MetaOCaml, an extension of OCaml, has been recognized as a promising solution to those problems through its multi-staging facilities and its static type system. MetaOCaml ensures the well-formedness and well-typing of both the generator and the generated programs.

We extends the application of multi-stage programming to geometric computations related to the field of mesh generation, aiming at enhancing the quality of the mesh generation software.

1.2 The Mesh Generation Problem

Given a description of a geometric domain, the mesh generation problem [TW00, TSW99] tries to decompose the interior of that domain into a set of elements. Those elements take various shapes such as polygons in 2D, and polyhedrons in 3D. The set of elements is called a *mesh*. Meshes are widely used in the applications of computational science and engineering. There haver been some attempts for improving the quality of mesh generation software by using better software engineering practices [ESC04, SY09].

A Family of Mesh Generators

Mesh generation exhibits different variations, such as the shape and connectivity of the elements, the underlying geometry, the algorithms, and the data structures. There have been some attempts for improving the quality of mesh generation by using the program family approach [BHK06, Cao06, HLC+06]. A comprehensive study of the commonalities and the variabilties in a family of mesh generating systems can be found in [SC04].

1.3 The Scope of the Thesis

We propose the following layered design for implementing a family of mesh generators:

Meshing Algorithms
Meshing Data Structures
Meshing Topology
Meshing Geometry

Each layer defines a set of abstractions that represents the commonalities and hides the variabilities in the family.

We propose using the generative meta-programming methodology to build a program generator that can generate specialized members of the family. Generative meta-programming can reduce the cost of generic abstractions resulting from the family approach without sacrificing the understandability of the generator.

The Generative Geometric Kernel (GGK)

In this work, we limit our scope to the lower layer – the *meshing geometry*. We show the design and implementation of a geometric kernel, the *generative geometric kernel* (GGK). The requirements of the GGK were motivated by the needs of a

program family of mesh generators. GGK was designed and built using the generative meta-programming facilities of MetaOCaml. The resulting kernel is a generator that can generate specialized geometric primitives and objects. We show that using MetaOCaml's support for multi-stage programming, the cost of abstractions in GGK is reduced.

1.4 Contributions

The following list highlights the contributions of this work.

- Extending the techniques of [Car06] to the field of mesh generation. We combined multi-stage programming, abstract interpretation, and OCaml's module system to build a generator that it is well-typed, highly parametric, and has a low cost of abstraction.
- Designing a layered generative geometric kernel (GGK) that exhibits modularity, parametricity, domain abstractions, understandability, and maintainability.
- Developing a set of abstractions for coordinate number types that allows writing geometric computations independent of the number type implementations.
- Developing a set of geometric abstractions based on affine geometry that can provide a basis for a family of meshing algorithms.
- Providing a proof of concept implementation for the GGK kernel that successfully achieves the goals stated in Section 1.1.

1.5 Organization of this Thesis

Chapter 2 provides an overview of some of the common software engineering approaches to improving the quality of scientific software. The suitability of the program families approach to scientific computing is also discussed.

Chapter 3 provides a background on generative programming and multi-stage programming facilities in MetaOCaml. The technique of abstract interpretation is discussed. Abstract interpretation is used in this work to eliminate unnecessary computations in the generated code.

Chapter 4 gives an overview of the design. Chapter 5 describes the multi-staging layer of GGK. The multi-staging layer offers abstractions for building and manipulating immediate and code expressions. Chapter 6 describes the number types layer. Chapter 7 introduces various concepts of affine geometry that are implemented in

GGK. The layers: linear algebra, affine space, and geometric objects are described. Several examples are given showing the code generation in action.

Chapter 8 presents several modules in the implementation. Several examples for building code generators and the resulting code, are presented.

Chapter 9 concludes this thesis by providing a summary of the contributions, a review of the related work, and an outlook on the future work.

Appendix A is the reference guide for GGK. It lists the modules, and gives details on their interfaces.

Chapter 2

Software Engineering In Scientific Computing

Scientific computing (SC) software deals with numerical approximations of continuous quantities. The are three major challenges that face SC. First, SC software can be very complex. It can involve complex processing of large amounts of data. Second, SC has a wide-variety of trade-offs regarding its aspects of quality, such as usability, accuracy and efficiency. Last, SC has to be extensible. The underlying models of SC have an experimental nature. This means that the understanding of the model changes over the course of its development. Therefore, SC software should be built to accommodate change.

Section 2.1 introduces these major challenges. The answer to one concern would normally compromise some other concern. Good software engineering practices meet these challenges without compromising quality. Section 2.2 introduces software quality and section 2.3 provides an overview of several software engineering practices widely applied to SC. In section 2.4, we focus on the program family approach and its application to SC software.

2.1 Scientific Computing

Scientific computing programs [Hea02, KS08, OS06] deal with approximations of continuous quantities that involve extensive numerical processing. The information processing in SC is normally based on mathematical models that are continuous in nature. Most of the solutions to continuous models can be constructed in theoretically infinite iterative processes. Computers, however, have finite capabilities in both storage and processing times. Therefore, approximate solutions with finite number of steps are employed. In some cases, the exact solutions can be constructed in finite number

of steps (iterations). Convergence and accuracy are two distinguishing features of approximate methods. It is required that the approximate method converges rapidly and terminates with an accurate solution.

2.1.1 Complexity of SC

SC has applications in different fields such as physics, mathematics, chemistry, biology, and engineering. SC problems usually involve large amount of computations. The input can be large (such as in seismic imaging), the output can be large (such as when solving partial differential equations), or the computation can be complex (such as optimization problems).

A concern of SC is overcoming this complexity. The following list names a few examples of replacing complex system with equivalent, yet simpler, systems.

- Simplifying higher-order systems into low-order systems.
- Converting models with infinite dimensional spaces into models with finite-dimensional spaces.
- Converting non-linear problems into linear problems.

Another aspect is the readiness of the results. In some cases, SC problems are real-time problems, where the processing must be rapid. Processing large amounts of data within time constraints is a challenge to SC. The accuracy of results adds to the complexity of building these systems.

2.1.2 Real Numbers

Most SC problems deal with real numbers. A real number $x \in \mathbb{R}$ contains infinite amount of information. Digital computers, however, can only store and process finite number of digits. Hence, approximations are employed. Approximating real numbers using finite representations is a well-known issue in computing. The errors that result from processing such approximate representations take different forms. The following list names a few types of errors.

- Round-off errors: the difference between the exact (or, true) value of the quantity and the computed approximate value.
- Equality: deciding whether two numbers represent the same quantity or not.
- Overflow and underflow: when the computation results are too large or too small to be accurately represented.

Several schemes have been proposed for dealing with the issues of real numbers. Among the most popular schemes are floating-point arithmetics and interval arithmetic.

2.1.3 Change

Due to the nature of SC software, it often undergoes rapid change [Seg07]. Developing SC software involves successive transformations from the real world to mathematical models then to computational models then to software. During these transformations, several approximations are employed. This process has an iterative and experimental nature. It is desirable that the software construction process follows an engineering discipline that takes *change* into account.

2.2 Software Quality

Software engineering is concerned with technologies and activities for building and supporting software products. These technologies span over a range of concepts, methods, processes, and tools. Software engineering activities include planning, modeling, analysis, specification, design, implementation, testing, and maintenance.

Software products are built according to requirements. Requirements can be functional or non-functional. Functional requirements state how a feature should be implemented. Non-functional requirements state quality goals that the system is expected to achieve.

The notion of *software quality* relates how a software product posses a characteristic that fulfills its requirement [Kan02]. High quality is a desirable feature. However, there are different aspects of software quality and one aspect might have relative importance over another aspect. For example, it is acceptable for a mathematical package to have a non-friendly interface, if the user-base is a small group of mathematicians who are familiar with the notations implemented by the software. On the other hand, the same interface would be unacceptable if the user-base is a wider general audience.

Some aspects are related. For example, accuracy is usually traded-off with speed. It is up to the requirements to specify which aspects of quality are more important. The need to have different aspects of quality is reflected by different quality metrics. The precise definitions of software quality metrics depend on the context. The following definitions are adapted from [Kan05, MET02, Som04].

Correctness Correctness means the software conforms to its functional requirements and produces correct results.

Accuracy Accuracy measures how close the solution is to the true solution.

Reliability Reliability measures the system's ability to deliver services as specified under given usage conditions. Reliability can be quantified in terms of the probability of failure-free operation.

Performance Performance is a measure of the resources used to produce the results. There are typically two types of resources: processing time, and storage.

Maintainability Maintainability is a measure of the effort required to change the software.

Verifiability Verifiability is the effort required to verify the operation of the software.

Understandability Understandability is the extent to which the user can comprehend the software.

Portability Portability is a measure of the effort required to adapt the program for a new computing environment.

Usability Usability is a measure of the effort required to learn and use the software.

Reusability Reusability is a measure of how easy it is to use the software in a different project.

2.3 Approaches to Better Quality

Several approaches have been proposed for building better quality software. In this thesis, we are concerned with the approaches that addresses the design and implementation phases.

The rest of this section briefly discusses some of the approaches commonly applied to scientific computing.

Modularity, Abstraction, and Reuse

Modularity [Par72] is a central concept in modern software development. Modularity increases productivity, program flexibility, and results in better designs. Modularity is closely related to abstraction and information hiding. One of the challenges that faces modularity is dealing with concerns that spans several modules. Approaches such as aspect-oriented programming tries to deal with these concerns.

Abstraction transforms solutions from special cases into more general cases. It allows reusing the same solutions in various situations. However, abstract concepts

needs to be concertized at run-time. This usually introduces runtime overheads. Approaches such as generative programming aims at reducing this overhead. SC relies on mathematical models which have an abstract nature. For example, see [AHMK01] for a discussion of the role of mathematical abstractions in developing abstractions for scientific software.

The benefits of software reuse have been recognized for many years. In 1968, McIlroy [McI69] called for software components to be arranged in parametrized families according to certain qualities so they can be reused. Reuse reduces the cost of development and results in better software quality [LL97]. However, reuse promotes generic solutions which are not always optimal.

Program Families

It is common for software products to exist in different variations (or flavors) according to different requirements, environments and configurations. This situation results in producing and maintaining different flavors of the same software. This redundancy contributes to producing poor quality programs. For example, when a bug is discovered in one flavor of a program, the same bug has to be fixed in all other flavors. The cost of managing similarity is high and better software engineering practices can reduce that cost.

The program family (PF) approach [Par76] addresses the problem of managing similarity. This approach considers a set of programs that posses extensive common properties to be a *family* of programs. A single program is called a *member* of the family. PF promotes studying the common features of a family *before* studying the individual members. Designing a family requires studying the problem and solution domains. A particular process of interest is commonality analysis (CA) [CHW98, Wei98]. The process of CA defines families by studying the common aspects of a family before studying the special aspects of its individual members. The common aspects are called *commonalities*, and the special aspects are called *variabilities*. Section 2.4 will further discuss the application of program families in scientific computing.

Design Patterns

Gamma et. al. [GHJV95] introduced reuse on the level of the design concepts. A design pattern is a reusable solution for a recurring design problem. Design patterns offer generic abstractions for the interaction between modules. Design patterns are closely related with object-oriented programming. The latter is sometimes avoided by scientific programmers because of its performance overhead.

Blilie [Bli02] identifies two major benefits of using design patterns in scientific

computing that outweigh their cost. First, patterns introduce better code architecture through using proven structures. Second, patterns introduce a division of labor between the domain experts and programmers. Domain experts, i.e., scientists, are concerned with correctness. However, programmers can produce better design and code. Patterns distribute responsibilities, so that correctness is preserved while, at the same time, the quality of design and code are not compromised. Extending design patterns to scientific computing has been studied by [Gar04]. Cickovski et. al. [Cic05] presents the application of design patterns to create efficient, flexible and maintainable molecular modeling software.

Component-Based Development (CBD)

Components [Cro96, Mey03, Sam97, Szy02] are independent units of software defined by interfaces that describe their usage. The main difference between a component and a module is that the interface of a component is designed without considering other components that uses it. This decoupling facilitates the reuse of the component in different software projects. Components offer abstraction and modularity. Abstraction is gained by focusing on well-defined interfaces. Modularity is achieved by enforcing encapsulation of functionality. CBD addresses the problems of reuse and modularity. The use of components has been extended to scientific computing. For example, see [cca, Ber00].

Aspect-Oriented Development (AOD)

Components offer an expressive high-level view of design. However, this expressiveness is often hindered by the complex interactions between the concerns. AOD [KLM+97] addresses the problem of separating concerns between design and implementation. An aspect encapsulates a concern that cross-cuts several components. A typical aspect-oriented system has the following infrastructure:

- Component language: for writing abstract component programs.
- Aspect language: for programming the aspects.
- Aspect weaver: for combining components and aspects and generating concrete components.

AOD was applied with success to scientific computing [HG04, ILG+97, KG07]. AOD is a promising approach to implement a program family. However, the complexity of the underlying development infrastructure might increase the development and maintenance costs.

Generic Programming

The central concept of generic programming [MS89] is abstraction. Data, algorithms, and representations are abstracted from concrete implementations into more generic forms. Generic programming facilitates modularity, reuse, and flexibility by decomposing the software into generic components that make minimal assumptions about other components. Generic programming, however, relies mainly on language support for generic types and generic functions. Generic programming has wide success. Famous examples are: the C++ Standard Template Library (STL) [SL95], the Matrix Template Library (MTL) [SL98], and the Boost Library [Daw]. Generic programming has also been applied to scientific computing [GK03, LL02, MY01].

Generative Programming

Abstraction tackles the complexity of software design. However, abstractions introduce additional run-time performance penalties. Moreover, it is desirable to introduce automation to the software production process. Generative programming (GP) [CE00, CEG+00, Eis97] addresses those problems by treating the software as families and incorporating code generation in the production of the software. The GP approach involves all the phases of software production. The goals of GP resemble those of generic programming. However, GP focuses on incorporating more domain-specific abstractions without compromising performance. The goals of GP can be summarized as follows [CEG+00]:

- High intentionality: GP tries to reduce the gap between program code and domain concepts.
- High reusability and adaptability.
- Easy management of families of components.
- Increased efficiency.

A typical GP system has the following three elements:

- *Specification* of family members. Techniques such as domain engineering and feature modeling can be used to analyze and capture the common and variable features of a software family.
- *Components* which serve as a configurable base for code generation. Parametrization is employed to manage the differences and variations among families of components.

- *Configuration knowledge* which captures knowledge required to map requirement specification into specialized programs. Examples are: dependencies and interactions among components, domain specific optimizations, and illegal combinations of components.

Given a particular specification, a customized family member can be automatically generated (manufactured) from the components using configuration knowledge.

Generative programming has increased in popularity in scientific computing. For example, see [ABM09, Car06, McC07, RGZ+09, Vel98, WPD01].

Domain-Specific Languages (DSL)

DSLs [MHS05] aim at intentionality of program code. DSLs are high-level programming languages that provide appropriate notations and abstractions for a particular problem domain. In their domains, DSLs are more expressive, usable, and productive than general purpose languages. However, DSLs have limited applicability outside their domains. DSLs can be small and restricted and can also be embedded in general purpose languages.

DSLs allow expressing the solution at the domain level. Consequently, using a DSL enhances software qualities such as reliability and maintainability [KMB+96, vDK98] and allows for domain-specific optimizations to be implemented more naturally [Bru97, MP99].

DSLs can be used in a program family environment. For example, [WL99] proposes using a DSL to express the variations between family members. However, DSLs have potential cost associated with building the language system and the supporting compilers. A survey on the topic of DSLs can be found in [vDKV00].

Meta-programming

Meta-programs [She01] are programs that process themselves or other programs. Meta-programming can be used to construct programs. When used this way, meta-programs are known as *program generators*. Meta-programming requires no additional compiler technology and hence offers a low cost environment for implementing program generators.

The paradigm of generative meta-programming, that is using meta-programming for realizing generative programming systems, can result in efficient implementations in scientific computing. For example, see [EFP07, FSPL08, Vel96]. Chapter 3 discusses generative meta-programming techniques in further details.

2.4 Program Families in SC

The program families approach is suitable if the expected changes in that software are centered around the predictions about the needed family members. Weiss [Wei98] states three assumptions to be investigated before applying the family approach. We present the three hypotheses based on the investigation by [SMC07] on the suitability of program families in scientific computing.

- *The Redevelopment Hypothesis: Most software development is mostly redevelopment.* Scientific computing has produced large amount of software products, many of which are variations of the same program. For example, a study by Carette [Car06], showed that there are 35 different implementations of Gaussian Elimination in the industrial package Maple. A survey by Owen [Owe98], identifies 61 software packages which generate triangular meshes, 43 of which uses Delaunay triangulation algorithms. The FFTW website [FFT] lists 41 different libraries of FFT.

- *The Oracle Hypothesis: It is possible to predict the likely changes.* The underlying knowledge of scientific computing is scientific models. In most of the cases, the literature on those scientific models is stable. Hence, changes can be predicted by carefully analyzing the scope of the software within the literature and how the software can evolve.

- *The Organizational Hypothesis: It is possible to organize both software and the software developing organization to take advantage of predicted changes.* In scientific computing, it is possible to use proper abstractions such that predicted changes can be made independently of other types of changes. Mathematical abstractions provide a way to decouple the predicted changes from other changes in data structures or algorithms. See [AHMK01] for a discussion of the role of mathematical abstractions, such as computational domains and coordinate systems, in scientific computing.

Several examples of program families in scientific computing have been demonstrated. For example, see [BHK06, Cao06, SCM08].

2.5 Conclusion

This chapter presented a number of software engineering approaches and their applications to scientific computing. Considering software quality in the production of scientific software allows outstanding issues, such as efficiency and accuracy, to be

dealt with without compromising other quality aspects such as usability and portability.

In particular, program families and generative programming are promising approaches to develop quality scientific software. Chapter 3 discusses generative metaprogramming techniques in further detail.

Chapter 3

Generative Meta-programming

This chapter provides a background on generative meta-programming and abstract interpretation. Section 3.1 introduces the meta-programming. Section 3.2 introduces MetaOCaml and the staging annotations used to build and generate code. Section 3.3 introduces the technique of abstract interpretation.

3.1 Meta-programming

Meta-programs are programs that manipulate other programs [She01]. The program under manipulation is normally called an object-program. Meta-programming can be used to analyze the structure of *object-programs*. Such systems are called program analyzers. The other use of meta-programming is to construct object-programs. When used this way, meta-programs are known as *program generators*. Program generators fall in two categories: static generators and run-time generators. Static generators generate programs to be compiled by normal compilers. The generation is done in *one stage* before compilation or running. Examples of static generators are parser and lexer generators (e.g., Yacc [Joh79] and Lex [LS79].) Run-time generators generate and execute programs at run-time. If the generated program is in turn a run-time generator, i.e., the generation is done in multiple stages, then the approach is called *multi-stage programming*. Languages that support this paradigm are called multi-stage programming languages, such as MetaOCaml [moc] and MetaML [TS00].

When the meta-program and the object-programs are written in different systems, the meta-programming system is then called a *heterogeneous* system. If they are written in the same language, then the system is called *homogeneous*.

Homogeneous meta-programming systems offer program manipulation and require no additional compiler technology, which makes them useful for writing code generators.

3.2 MetaOCaml

MetaOCaml [moc] is a meta-programming extension for OCaml [Obj]. OCaml is a multi-paradigm language that supports functional, imperative and object-oriented styles of code. OCaml is statically-typed language, which improves the quality of the compiled programs through early detection of errors.

MetaOCaml is a multi-staged language. It provides mechanisms for constructing and combining code expressions that will be executed in future stages. For that purpose, MetaOCaml extends OCaml syntax by three constructs: *Bracket*, *Escape*, and *Run*. It also extends the type system by one additional type: `code`. The next sections introduce these extensions.

The statically-typed environment allow us to encode the variabilities of the generator as a static information. The type system can then ensure, at the generation time, that the configuration of the variabilities is consistent. In effect, the static information about future-stage computations can be exposed in an earlier stage; the generation stage.

MetaOCaml extends the OCaml tool set, making it fully compatible with OCaml. All OCaml programs are also MetaOCaml programs. Based on OCaml's support for functional programming, MetaOCaml provides good abstractions for meta-programming such as higher order functions, parametric polymorphism, algebraic datatypes, and parametric modules (functors). MetaOCaml provides homogeneous meta-programming embedded within OCaml. In effect, the overhead for writing program generators in MetaOCaml is reduced.

3.2.1 Staging Annotation in MetaOCaml

MetaOCaml provides three staging constructs: Brackets, Escape, and Run. These constructs allow the programmer to specify the order of evaluation of terms. This section introduces these constructs. For a detailed introduction on multi-stage programming in MetaOCaml, see [Tah04].

3.2.1.1 Bracket

The Bracket operator (`.<e>.`) delays the execution of the expression `e` and hence constructs a future-stage computation. Given a *valid* OCaml expression `e` of type `t`, the annotation `.<e>.`, of type `('a, t) code`[1], is the *lifted* code expression for `e`. The execution of this annotated expression is delayed to a future stage. However, both the syntax and the type of this expression are checked at the current stage. This

[1]The polymorphic type parameter `'a` is called the *environment classifier* [TN03]. The reasons and details behind environment classifiers are outside the scope of this thesis.

gives a static guarantee that the generated code will have the appropriate type when executed at a future stage. This allows writing code generators that can detect both syntax errors and type errors of the generated code. In effect, the resulting generated code is syntactically well-formed and well-typed.

Defining a code expression for a constant is given below[2].

```
# let e = 2.71828183 ;;
val e : float = 2.7182818
# let e = .< 2.71828183 >. ;;
val e : ('a, float) code = .<2.71828183>.
```

In the first user input, a normal OCaml constant of type `float` is defined. In the second input, a code expression for the same float constant is defined by putting brackets around the float literal. The expression passes the type checking and gets typed as `('a, float) code`, denoting a code expression of type `float`. The following fragment highlights the fact that computations inside the brackets are delayed to a future stage:

```
# .< 2.0 +. 3.0 >. ;;
- : ('a, float) code = .<(2.0 +. 3.0)>.
```

Normally, the evaluation of the expression `2.0+3.0` in OCaml would be `5.0`. However, due to the brackets, the computation is delayed to a future stage, and the resulting expression is the code for the computation. The same bracket construction can also be used to define code expressions for functions. The following example defines a code for a multiplication function.

```
# .< fun x y -> x * y >. ;;
- : ('a, float -> float -> float) code =
  .<fun x_1 -> fun y_2 -> (x_1 * y_2)>.
```

It is noticeable that the variables `x` and `y` are renamed in the resulting code. MetaOCaml renames all bound variables to avoid accidental variable name captures.

A noticeable feature of MetaOCaml is that the staging annotations allow construction of *only* valid OCaml expressions [Tah04]. Partial expressions such as `.< let z = 0 in >.` or `.< 2 * >.` cannot be constructed, and will result in syntax errors. Those limitations imposed by MetaOCaml contribute to the correctness of the generated code.

[2]This example and those that follow are verbatim copies of the MetaOCaml top-level interpreter. The user prompt is preceded by `#`, while the lines following are the interpreter's response.

3.2.1.2 Escape

Escape operator (`.~`) allows inlining code expressions inside larger code expressions. When applying `.~` to an expression `e`, MetaOCaml first evaluates `e` in the current stage, then splices the resulting expression inside a later stage expression. There are two conditions for splicing:

1. Escape is only allowed within code expressions. Splicing a code expression in a non-code context is meaningless and results in syntax errors.
2. Escape is only allowed on code expressions. Splicing a non-code expression results in a error.

The following example shows how to combine smaller code fragments into a larger expression.

```
# let numer = .< 3.0 >. and denom = .< 4.0 >. ;;
val numer : ('a, float) code = .<3.0>.
val denom : ('a, float) code = .<4.0>.
# .< .~numer /. .~denom >. ;;
-: ('a, float) code = .<(3.0 /. 4.0)>.
```

Splicing functions is no different than splicing expressions thanks to OCaml's support for functional programming, where functions are first class citizens. To highlight this, we present the following example. Let us assume we have a cost evaluation *generator* `eval`, that takes a cost function `cost` as a parameter, and inlines its code in a larger context. The function `eval` is defined as:

```
# let eval cost = .< fun x -> .~cost x >. ;;
val eval : ('a, 'b -> 'c) code ->
             ('a, 'b -> 'c) code = <fun>
```

Now, we define two example cost functions operating on different datatypes (`float` and `int`):

```
# let cost1 = .< fun x -> log x >. ;;
val cost1 : ('a, float -> float) code =
   .<fun x_1 -> (log x_1)>.
# let cost2 = .< fun x -> x / 2 >. ;;
val cost2 : ('a, int -> int) code =
   .<fun x_1 -> (x_1 / 2)>.
```

Applying `eval` to `cost1` and `cost2` results in:

```
# eval cost1 ;;
- : ('a, float -> float) code =
  .<fun x_1 -> ((fun x_1 -> (log x_1)) x_1)>.
# eval cost2 ;;
- : ('a, int -> int) code =
  .<fun x_1 -> ((fun x_1 -> (x_1 / 2)) x_1)>.
```

The resulting expressions have not only the inlined code, but also, the appropriate types (unary function on `floats` and `ints`, respectively). However, the resulting code, is not optimal and does not resemble a code fragment written by a human. For example, `eval cost2` would *naturally* be written as `.< fun x -> x / 2 >.` instead. This issue is addressed in the next section.

3.2.1.3 Meta-Programming (Or, On Splicing of Functions)

The power of the Escape operator lies in the fact that the inlined expression (`e` in `.~e`) is evaluated *before* splicing. This allows performing *general-purpose* computations at the inlining-time, i.e., at the *code-generation time*. This useful feature is a great aid to meta-programming. Consider the previous example, where ideally the generated code should look like the following *hand-written* code:

```
.< fun x -> log x >.
.< fun x -> x / 2 >.
```

This could be achieved by exploiting the aforementioned feature of the Escape operator: the generation-time computation before inlining. By carefully examining the function `eval`, we can notice that it inlines `cost` at the current stage but applies it to `x` at a future stage. The following modified `eval`, named `better_eval`, first applies cost to `x` at the current stage, then it inlines the result:

```
# let better_eval cost = .< fun x -> .~(cost .<x>.) >.  ;;
val better_eval : (('a, 'b) code -> ('a, 'c) code) ->
  ('a, 'b -> 'c) code = <fun>
```

This modification also requires changing the type of `cost` from a code of function, to a function that takes code as an input, and returns code as an output. Doing so, we get:

```
# let cost1' x = .< log .~x >. ;;
val cost1' : ('a, float) code ->
  ('a, float) code = <fun>
# let cost2' x = .< .~x / 2 >. ;;
val cost2' : ('a, int) code -> ('a, int) code = <fun>
# better_eval cost1' ;;
- : ('a, float -> float) code =
  .<fun x_1 -> (log x_1)>.
# better_eval cost2' ;;
- : ('a, int -> int) code =
  .<fun x_1 -> (x_1 / 2)>.
```

where the generated code is simplified and resembles the desired code with *no traces* of the generation-time helper functions `cost1'` and `cost2'`.

Using this technique, a code generator can be written to produce a clean code with minimal traces of the generator helper routines.

3.2.1.4 Run

The Run operator (`.!`) forces the execution of a code expression in the current stage. Let `e` be an expression of type `('a, t) code`. `.!e` is the expression of type `t` resulting from forcing the execution of `e` at the current stage. The following identity holds for Run: `.! .<e>. = e`.

The following code shows an example where the delayed expressions are evaluated as expected.

```
# .! .< 2.0 +. 3.0 >. ;;
- : float = 5.
# .< char_of_int 65 >. ;;
- : ('a, char) code = .<(char_of_int 65)>.
# .! .< char_of_int 65 >. ;;
- : char = 'A'
```

The same rule applies for functions:

```
# let incr_code = .< fun x -> x + 1 >. ;;
val incr_code : ('a, int -> int) code =
  .<fun x_1 -> (x_1 + 1)>.
# let incr_fun = .! incr_code ;;
```

```
val incr_fun : int -> int = <fun>
# incr_fun 4 ;;
- : int = 5
```

Using Bracket, Escape and Run makes it possible to write code generators. A notable feature of this approach is that the type system of MetaOCaml can ensure that the generated programs are syntactically well-formed and well-typed.

3.3 Abstract Interpretation

MetaOCaml has a purely generative approach. It treats the generated code as a black box. Abstract interpretation [CC77] offers an a priori technique to eliminate unnecessary computations in the generated code.

The technique of abstract interpretation approximates program semantics by replacing the concrete domain of computation by an abstract domain. The main application of abstract interpretation is static analysis. Abstract interpretation has been used in program generators (cf. [Car06]) to avoid the need for a posteriori optimizations. The following steps are typical in applying abstract interpretation for code generators [KST04]:

1. *Identify the concrete domain.* The concrete domain for a code generator is typically the `code` type, or, variations of it.

2. *Design an abstract domain that has more information about the code values.* The abstract domain explicitly encodes the implicit information about the concrete domain. For example, a concrete domain of statements has an identity statement of type `unit`. This information, i.e., the kind of the statement: unit or not, is useful to be encoded in the abstract domain. An operation such as statement sequencing can make use of this explicit information.

3. *Provide lifting and concretization functions.* Such functions can be used to convert between the abstract and concrete domains.

4. *Lift all the operators from the concrete type to the abstract type.* The abstract type provides more information and hence some work can be shifted to the first stage (typically, the generation-time). This results in less work to be done in the second stage (typically, the run-time).

5. *Express the program in terms of the staged operators.* The information provided by the abstract type and the staged operators allows some useful optimizations to be expressed in the abstract domain without the need to inspect the code.

We make use of abstract interpretation in writing code generators. Abstract interpretation allows manipulation of code expressions and a priori optimization. In chapter 5 we show that the generated code exhibit better characteristics such as: constant folding, constant propagation, and algebraic simplification.

Chapter 4

The Design of the Generative Geometric Kernel

This chapter describes the design of GGK. Section 4.3 discusses the use of OCaml module system to achieve a high degree of parametricity in the generator. Chapters 5, 6, and 7 discuss the details of the design. The reference guide can be found in Appendix A.

4.1 Design Overview

The generative kernel was designed and implemented as five loosely coupled layers. Figure 4.1 shows an overview of these layers. Each layer includes a set of concepts and defines a set of abstractions. These abstractions are designed to be generic and parametric.

Geometric Objects
Affine Space
Linear Algebra
Number Types
Multi-staging

Figure 4.1: Overview of the layers of GGK.

Geometric Objects Layer	Orient	Inside	Simplex
	Vertex	Hyperplane Operations	Hypersphere Operations
	Insphere	Hypersphere	Hyperplane
Affine Space Layer	Affine		Side
	Vector	Point	Orientation
Linear Algebra Layer	Tuple	Matrix	Determinant
Number Types Layer	Real	Order	Field
	Set	Ring	Sign
Multi-staging Layer	Code Constructs		
	Base Types		
	Staged Types		

Figure 4.2: Details of the layers of GGK.

4.2 Details of the Layered Design

The detailed design of GGK layers is shown in Figure 4.2. The purpose and functionality of each of the layers is described as follows.

- The *multi-staging layer* provides code generation facilities. It employs abstract interpretation and the multi-staging facilities of MetaOCaml to offer abstractions for representing and manipulating multi-staged types, values and functions. The concepts in this layer fall in three categories: staged types, base types, and code constructs. The staged types category provides the code generation facilities for types and functions. The base types category provides a collection of modules that facilitate code generation for the commonly used types such as boolean and string. The code constructs category offers a multi-staged version of the useful code constructs such as if-else. Chapter 5 discusses the design of this layer in detail.

- The *number types layer* provides abstractions for number types. This layer makes it possible for the upper layers to express computations independently of the number type implementations. The number types provided by this layer are staged number types. Chapter 6 discusses the design of this layer in details.

- The *linear algebra layer* provides linear algebra support for the geometric object and computations. This layer uses the multi-staging facilities provided by the lower layers and offers abstractions that have a reduced cost. The functionality and abstractions provided in this layer are motivated by the needs of the affine geometry, not by the general scope of linear algebra. Chapter 7 discusses the design of this layer in details.

- The *affine space layer* provides a basis for the geometric objects and computations. The concepts in this layer are built around affine geometry. This layer offers generic abstractions for points, vectors and operations on them. Chapter 7 discusses the design of this layer in detail.

- The *geometric objects layer* offers abstractions for geometric objects and computations. This layer provides facilities for writing geometric algorithms and data structures independent from the choices of coordinate types or dimensions. Chapter 7 discusses the design of this layer in details.

The following chapters discuss each of these layers in details.

4.3 OCaml Modules and Functors

One of the goals in building our code generator is achieving a high degree of parametricity. Parametrization is an aid to implementing families. Family variabilities become parameters. This is achieved by using OCaml's module system for expressing abstractions. OCaml's module system provides three constructs: *module types* (or, signatures), *modules*, and *functors*.

Module types (also called interfaces) are module-level type specifications. They provide a way to specify requirements on modules. Modules can be considered as implementations of module types. Modules can be checked against specification (i.e., module types) by the type checker. This ensure that the type checker will reject invalid implementations of the requirements.

The use of functors is two-fold. First, functors provide parametrization of modules. Parametric components can be implemented as functors. Second, functors, in the OCaml statically typed setting, ensure the composability of the parameters [CK05]. The inputs to functors are modules that match certain type signatures. Adding

constraints ensures that non-composable implementations will be rejected by the type checker.

Chapter 5

The Multi-Staging Layer

The multi-staging layer offers abstractions for building and manipulating staged types. This layer employs abstract interpretation and provides code generation facilities. Sections 5.1 and 5.2 provide details on building staged types and operators. An overview of the benefits resulting from staging and abstract interpretation is provided in Section 5.4. The details of the module types and interfaces can be found in Appendix A.

5.1 Building Staged Types

This section provides an overview of the types and functions that we built to provide facilities for staging types and functions.

Code Expressions

The type `code_expr`, defined as:

```
type ('a,'b) code_expr = { c : ('a,'b) code; a : bool }
```

represents a code expression that can be atomic or non-atomic. The field `c` captures a code expressions of polymorphic type `'b`. The field `a` is the atomicity flag. An expression `e` is atomic if it is:

1. an immediate value, or
2. a variable.

`e` is *not* atomic otherwise, i.e., a result of a computation. The atomicity of an expression is useful for inserting let-bindings which will be discussed later in this chapter.

```
type ('a,'b,'c) unary = {
  unow : 'b -> 'c ;
  ulater : ('a,'b) code_expr -> ('a, 'c) code_expr
}
```

with the field `unow` being f_n, and the field `ulater` being f_l. The following code shows the definition of `mk_unary`:

```
let mk_unary f = function
  | Now x -> Now (f.unow x)
  | Later x -> Later (f.ulater x)
```

5.2.2 Staging Binary Operators

The required staged binary function should satisfy the following equation:

$$f_s = \texttt{mk\_binary}(\langle f_n, f_l \rangle)$$

where

$$f_s(x,y) = \begin{cases} \texttt{Now}\, f_n(x_n, y_n) & x = \texttt{Now}\, x_n \text{ and } y = \texttt{Now}\, y_n \\ \texttt{Later}\, f_l(x_l, \mathcal{L}(y_n)) & x = \texttt{Later}\, x_l \text{ and } y = \texttt{Now}\, y_n \\ \texttt{Later}\, f_l(\mathcal{L}(x_n), y_l) & x = \texttt{Now}\, x_n \text{ and } y = \texttt{Later}\, y_l \\ \texttt{Later}\, f_l(x_l, y_l) & x = \texttt{Later}\, x_l \text{ and } y = \texttt{Later}\, y_l \end{cases},$$

and $\mathcal{L}$ is a lift operator from the a type `'b` to `('a,'b) code_expr` ($\mathcal{L}$ is realized by the function `lift_atom` in section 5.1).

In the equation above, cases 1 and 4 of $f_s(x,y)$ handle similar inputs of type Now or Later. Cases 2 and 3 handle mixed Now and Later inputs. Whenever f_s is applied to a mix of Now and Later values, the Now value is typecasted (or, lifted) into a Later value. The following code shows the type `binary` representing the *generalized binary function*, $\langle f_n, f_l \rangle$, and the function builder `mk_binary`.

```
type ('a,'b,'c,'d) binary = {
  bnow : 'b -> 'c -> 'd ;
  blater : ('a,'b) code_expr -> ('a, 'c) code_expr ->
           ('a,'d) code_expr
}
```

```
let mk_binary bop x y =
match x, y with
 | (Now x), (Now y) -> Now (bop.bnow x y)
 | (Now x), (Later y) -> Later (bop.blater (lift_atom .<x>.) y)
 | (Later x), (Now y) -> Later (bop.blater x (lift_atom .<y>.))
 | (Later x), (Later y) -> Later (bop.blater x y)
```

5.2.3 Staging Monoid Operators

It is common for binary operators to have special elements. For example, addition has an identity: zero. We can take advantage of abstract interpretation and build *better* staged operators that respect the identity laws of the base operators. That is, the resulting staged operator can generate code that does not contain unnecessary operations such as multiplication by zero, i.e., generating `0` as opposed to generating `0 * expression`.

In the concrete domain, a monoid operator $\star$ is a binary operator along with a special (unit) element u where:

$$x \star u = u \star x = x.$$

In the abstract domain, `staged`, u is an immediate value, i.e, a Now expression. x can be a Now expression or a Later expression. The staged version of $\star$, f_s can be described by the equation:

$$f_s(x, y) = \begin{cases} y & x = \texttt{Now}\ u \\ x & y = \texttt{Now}\ u \\ \text{Same cases as} & \\ \text{in } \texttt{mk\_binary} & \end{cases},$$

where the first two cases implement the monoid laws. The type `monoid` defined below represents a monoid operator as a generalized binary function along with an identity element.

```
type ('a,'b) monoid = {
 bop : ('a, 'b, 'b, 'b) binary;
 uelem : 'b
}
```

The monoid builder function, `mk_monoid`, defined as:

```
let mk_monoid mon x y =
match x, y with
| (Now x), y when x = (mon.uelem) -> y
| x, (Now y) when y = (mon.uelem) -> x
| x,y -> mk_binary mon.bop x y
```

is an extension of `mk_binary` which implements the aforementioned monoid laws.

5.2.4 Staging Ring Operators

Similar to monoid operators, the type `ring` is introduced to capture the quadruple:

$$(f_n, f_l, 0, 1),$$

where f_n is the Now version of the operator, f_l is the Later version, and 0 and 1 are the identity elements of the ring operator. The ring laws are:

$$x \star 0 = 0 \star x = 0$$

and

$$x \star 1 = 1 \star x = x.$$

The following code shows the type `ring` and `mk_ring`.

```
type ('a,'b) ring = {
  mon : ('a,'b) monoid;
  zelem : 'b
}

let mk_ring rng x y =
  match x, y with
  | (Now x), (Later y) when x = (rng.zelem) -> Now rng.zelem
  | (Later x), (Now y) when y = (rng.zelem) -> Now rng.zelem
  | x, y -> mk_monoid rng.mon x y
```

Both `ring` and `mk_ring` are defined as extensions of `monoid` and `mk_monoid`, respectively.

5.2.5 Staging Common Types

We lift some useful types which are commonly used during code generation. The modules `Int`, `String`, and `Bool` provide staged integers, strings, and boolean types, respectively. The reader is advised to view Appendix A for further details on the API of those modules.

5.2.6 Staging Code Constructs

We lift various code constructs useful in code generation. The function `seq`, defined below, is the lifted version of sequencing.

```
let seq a b = mk_binary
  { bnow = (fun x y -> (x; y));
    blater = fun x y -> lift_comp .< begin .~(to_code a) ;
                          .~(to_code b) end >. } a b
```

The function `ife`:

```
let ife c a b to_code = match c with
 | Now cc -> if cc then a else b
 | Later cc ->
    of_comp (.< if .~(to_code c) then .~(to_code a)
                else .~(to_code b) >.)
```

is the lifted if-else statement. There are two cases of the condition `c`:

1. `c` is a Now expression: `ife` reduces to an OCaml if-else statement evaluated at generation-time.

2. `c` is a Later expression, i.e., the condition is known at run-time: `ife` generates a code for the if-else statement which gets evaluated at run-time.

5.2.7 Generating Let Statements

Straightforward generation of code can easily result in duplication of code. Consider the following function `add` which computes the expression `x+(x+y)`. When applied to code expressions, add will just inline the expression for `x` twice.

```
(* Given that '+' is lifted *)
# let add x y = x + (x + y) ;;
# add (of_comp .<1. *. 2.>.) (of_atom .<3.>.) ;;
- : ('a, float) Staged.staged =
  Later {c = .<((1. *. 2.) +. ((1. *. 2.) +. 3.))>.;
         a = false}
```

Obviously, `x` is a common sub-expression that needs to be bound in a let-expression. We might try to improve `add` by introducing a let-binding in the function definition as:

```
let add1 x y = let ce = x in ce + (ce + y)
```

However, this is not a solution because the let binding is done at the meta-level and not inlined in the generated code. A solution is presented in [CK05] by using continuation passing style (CPS) and monads, which is outside the scope of the work of this thesis. We adopt a simpler approach and rewrite the expression `x+x+y` as `λx.(x+x+y)`, then we pass it to the let-generator, `let_`, where `let` is inserted and delayed for the future stage.

`let_` is defined as follows:

```
let let_ ce exp =
  match ce with
  | Now _ -> exp ce
  | Later c when c.a = true -> exp ce
  | Later c ->
      of_comp .< let _v = .~(c.c) in
        .~ (to_code (exp (of_atom .<_v>.))) >.
```

where `ce` is the common sub-expression, and `exp` is the body of the `let` expression. `exp` is a function in some parameter `v`. `v` replaces the duplications of `ce`. The common expression `ce` has three cases:

1. Now expression: `let_` reduces to function application.
2. An atomic code expression: `let_` reduces to function application as well. If the expression is atomic then there is no harm from duplicating it in the code.
3. A non-atomic code expression: `let_` generates code containing the binding of `ce` into a variable `v`.

To demonstrate `let_` in action, we apply `let_` to the same example as above:

```
# let add' z y = let_ z (fun x -> x + (x + y))
# add' (of_comp .<1. *. 2.>.) (of_atom .<3.>.)
- : ('a, float) Staged.staged =
  [rest omitted]
  .<let v_1 = (1. *. 2.) in
    (v_1 +. (v_1 +. 3.))>.
# add' (of_atom .<1.>.) (of_atom .<3.>.)
- : ('a, float) Staged.staged =
   [rest omitted]
  .<(1.  +. 1. +. 3.)>.
```

In the first application of `add'`, the expression `.<1. *. 2.>.` is labeled as non-atomic, and thus a let-binding is generated. In the second application, the expression `.<1.>.` is labeled as atomic (by the application of `of_atom`), and thus it was inlined in the resulting code.

5.3 Example: Building a Generator for the Power Function

We demonstrate the usage of this layer by an example. In this example, we take the power function and build a staged version of it. The staged version can use staging to unroll the recursion, and abstract interpretation to avoid generating unnecessary operations. We also build a code generator and demonstrate the use of module types and functors for expressing the variabilities of the code generator.

Consider the following classical example [JGS93, Tah04].

```
# let rec power n x =
   match n with
   | 0 -> 1
   | n -> x * (power (n-1) x);;

val power : int -> int -> int = <fun>
```

It is required to transform `power` into a generator that can unfold the definition of `power` and generate expressions such as `.<x*x*x>.` instead of `.<power 3 x>.`. We achieve this task through three steps:

(1) Staging `power`. First, we analyze the concrete domain of the variable `x`. Multiplication, 0, and 1 are the requirements of that domain. A ring operator over `int` type provides exactly those operations and constants. We define the multiplication operator as a generalized binary operator.

```
let mul_op =
  { bnow = (fun x y -> x * y);
    blater = (fun x y -> lift_comp .<.~(x.c) * .~(y.c)>.) }
```

Then we express the integer multiplication ring operator as a monid operator plus a zero element:

```
let mul_mon = { bop = mul_op; uelem = 1 }
let mul_ring = { mop = mul_op; mon = mul_mon; zelem = 0 }
```

The integer multiplication operator is then lifted by:

```
let ( * ) x y = mk_ring mul_ring x y
```

Now, the power function can be defined as:

```
let rec power n x =
  if n = 0 then Now 1
  else x * (power (n-1) x)

val power : int -> ('a, int) staged ->
   ('a, int) staged = <fun>
```

Applying power to several examples is shown below.

```
# power 4 (Now 3);;
- : ('a, int) staged = Now 81
# power 4 (of_atom .<3>.);;
- : ('a, int) Staged.staged =
  Later {c = .<(3 * (3 * (3 * 3)))>.;
         a = false}
```

(2) Building a generator. The function `power` only generates code if the input is a code expression. The required generator should build a code expression (i.e., perform abstraction), call `power`, extract the resulting code expression (i.e., perform concretization), and then splices the resulting expression in a code context for a function. The following generator performs those steps.

```
let gen n =
 .< fun x -> .~(to_code
      (power n (of_atom .<x>.))) >.

val gen : int -> ('a, int -> int) code
```

(3) Parameterizing `gen`. The function `power` is parametric in the exponent `n`. Therefore, the generator should have a variability for the exponent. The following module type `EXP` defines this simple variability.

```
module type EXP =
sig
  val n : int
end
```

The functor module `Power` below, provides the required parametrization over the exponent.

```
module Power (E : EXP) =
struct
 let gen () =
   .< fun x -> .~(to_code
       (power E.n (of_atom .<x>.))) >.
end
```

To demonstrate the generator in action, we generate the power for exponents 1, 2, and 3 using the following code. First we define three different values for the variabilities.

```
module Exp1 = struct let n = 1 end
module Exp2 = struct let n = 2 end
module Exp3 = struct let n = 3 end
```

Then we instantiate and call the generator for each.

```
module Power1 = Power(Exp1)
module Power2 = Power(Exp2)
module Power3 = Power(Exp3)

Power1.gen ();;
- : ('a, int -> int) code =
      .<fun x_1 -> x_1>.
Power2.gen ();;
- : ('a, int -> int) code =
      .<fun x_1 -> (x_1 * x_1)>.
Power3.gen ();;
- : ('a, int -> int) code =
      .<fun x_1 -> (x_1 * (x_1 * x_1))>.
```

It is noticeable that the base case of the recursion would normally result in a multiplication by 1. The resulting code however, has no extra multiplication by 1. This simplification is due to the ring laws implemented in `mk_ring`.

5.4 Conclusion

By combining abstract interpretation, and multi-staging facilities of MetaOCaml, we are able to represent, manipulate, and generate code expressions that exhibit the following features.

Constant folding If a certain computation involves only constants, then it is fully performed at generation-time, and the results are generated instead. For example, given that `add` is the staged version of addition, the expression `(to_code (add (Now 1) (Now 3)))` has the value `(to_code (Now 4)) = .<4>.`, instead of `.<1+3>.`. In other words, the constants are folded *first* at generation-time, then the code is generated.

Constant propagation If a variable `x` have an immediate value `v`, known at generation-time, then `v` is inlined for every occurrence of `x`. This is due to the fact that immediate values (constants) are labeled with the tag `Now`. With abstract interpretation, staged operators can extract the value of a `Now` expression and use it or inline it as needed.

Algebraic simplification Operations that involve special elements are simplified. Some examples are the identities for $+, -, \div$, and exponential with regard to

0 and 1. For example, given that `add` is the staged addition over integers, the expression `(to_code (add (Later .<x>.) (Now 0)))` evaluates to the code expression `.<x>.` instead of `.<x+0>.`.

Dead code elimination If a piece of code is not going to be executed at run-time, then it should not be generated in the first place. This is achieved by lifting the conditional statement. The following few examples show `ife` in action.

```
# ife (Now true) (Now 1) (Later .<3>.)
- : ('a, int) Staged.staged = Now 1
# ife (Now false) (Now 1) (Later .<3>.)
- : ('a, int) Staged.staged = Later .<3>.
# .< fun x -> .~(to_code
      (ife (Later .<x>.) (Later .<1>.) (Later .<3>.))) >.
- : ('a, bool -> int) code =
      .<fun x_1 -> if x_1 then 1 else 3>.
```

Common sub-expression elimination If a piece of code is used in several places in an expression, then it should be factored out, and bound to a variable name. An example was presented in Section 5.2.7.

Chapter 6

The Number Types Layer

The number types abstraction layer serves two purposes. First, it makes it possible to write algorithms independently of the number type choice and the number type implementations. In other words, it allows implementing a *family* of algorithms with the number type as a *variability*. Second, this layer offers staged number types. Staged typed, as described in previous chapters, form the base for code generation.

This chapter introduces the hierarchy of number type abstractions in this layer. The design of these abstractions is based on the traditional algebraic structures such as ring and fields. However, the design is motivated by the common data types available in OCaml. In other words, we do not provide a full hierarchy of algebraic structures such as semi-group or a magma. Instead, the number types are abstracted based on the behavior of their operators. Each interface (module type) specifies a minimal set of operations. To minimized duplications, we use OCaml's module inclusion for extension among types. The following types are ordered by the inclusion relation ($A \sqsubseteq B$ if module B includes module A):

$$\texttt{SET} \sqsubseteq \texttt{RING} \sqsubseteq \texttt{FIELD} \sqsubseteq \texttt{REAL}.$$

Sections 6.7, 6.8, and 6.9 provide the details of an implementation of this layer. Implementations for integer, rational and float types are provided. The details of exact and inexact floating point sign calculation are given. Section 6.10 demonstrates an example of using the modules of this layer. The details of the module types and interfaces can be found in Appendix A.

6.1 The SET Type

The `SET` type is an abstraction for a set of elements of some number type `n`. The type `n` is called the base number type. It represents the immediate values of that type. The staged version of `n` is the type `ns`.

The SET type is an interface requirement for the modules implementing the set concept. Such modules have to provide the following operations: an equality check, inequality check, and typecasting to strings. An additional version of equality with tolerance (eq_tol) is also required. This allows an opportunity for implementing inexact number types.

Two versions of the same function eq are required: the base version (eq_b), and the staged version (eq_s). An additional base version is included as a design choice that gives the users of the module the freedom to use either the staged version (eq_s), the Now version (eq_b.bnow), or the Later version (eq_b.blater). For the same reasons, to_string_b was added to the interface.

6.2 The SIGN Type

A notion of sign is required before introducing the signed number types. The module type SIGN defines the type of sign as follows.

```
type t = Pos | Zero | Neg | PosOrZero | NegOrZero
```

A given number is either positive, negative, or zero. The type t represents this information. To be able to conveniently express the $\geq$ and $\leq$ relationships, an additional two signs are added: PosOrZero and NegOrZero.

6.3 The ORDER Type

The module type ORDER represents a linear order relationship over a staged number type (referred to as the carrier set). The order is bounded from both sides. It has a top element and a bottom element. Example values for top (and bot) are positive (and negative) infinity. The interface for ORDER specifies the comparison functions: eq, neq, compare, lt, le, gt, ge, min, and max. As in SET, two versions are provided for each function, the base version (with suffix _b) and the staged version (with suffix _s).

6.4 The RING Type

The RING type is an abstraction of the following algebraic structure:

$$(S, +, *, -, 0, 1)$$

where S is a carrier set. RING is thus defined as an extension of the SET type by module inclusion. Elements of RING are signed numbers and have absolute values.

6.5 The FIELD Type

The FIELD type is an abstraction of the following algebraic structure:

$$(S, +, *, -, ^{-1}, 0, 1)$$

It is defined as an extension of RING by adding the multiplicative inverse operation: inv. Division is then possible.

6.6 The REAL Type

The type REAL is defined as an ordered FIELD with top and bottom elements. The interface requires the implementations to provide common functionality such as the square root operation.

6.7 A Model Implementation for Integer Number Types

The module Integer_Set provides an implementation of the interface SET. It is defined as:

```
module Integer_Set : SET =
struct
  type n = int
  type 'a ns = ('a, n) staged
  ....
```

The signature constraint Integer_Set : SET allows the OCaml type checker to ensure that the module Integer_Set implements the interface SET.

The module Integer_Ring, defined as:

```
module Integer_Ring : RING =
struct
  include Integer_Set
  let zero = 0
  let one = 1
  let negone = -1
  let two = 2
```

```
let add_op =
  { bnow = (fun x y -> x+y);
    blater = (fun x y -> lift_comp .<.~(x.c) + .~(y.c)>.); }
let add_b = { bop = add_op; uelem = 0 }
let add_s a b = mk_monoid add_b a b
...
```

is an extension of `Integer_Set`. It is an implementation of the interface `RING`. The code above shows the definition of the staged addition operator as a monoid.

6.8 A Model Implementation for Rational Number Types

The modules `Rational_Set` and `Rational_Ring` implements a rational number ring. They have the following definitions (partial code):

```
module Rational_Set : SET =
struct
  (* A rational number is represented as a fraction *)
  type n = int * int
  type 'a ns = ('a, int * int) staged
  ...
end

module Rational_Ring : RING =
struct
  include Rational_Set
  let zero = (0, 1)
  ...
  let mul_op = {
    bnow = (fun (a,b) (c,d) -> a*b,c*d);
    blater = (fun x y -> lift_comp .<
                        (fst .~(x.c))*(fst .~(y.c)) ,
                (snd .~(x.c))*(snd .~(y.c)) >.); }
  let mul_mon = { bop = mul_op; uelem = one }
  let mul_b = { mop = mul_op; mon = mul_mon;
                zelem = zero }
  let mul_s x y =
```

```
    if x = (Now negone) then neg_s y
    else if y = (Now negone) then neg_s x
    else mk_ring mul_b x y
end
```

The choice of representing rational numbers as an OCaml product type (pair) is hidden by the module abstraction. The interfaces in this layer make no assumption on the type n. This abstraction allows writing generators that are independent of the choice of representation and yet the generated code contains these choices. For example, generating code for the multiplication operation:

```
.< fun x y -> .~(to_code (Rational_Ring.mul_s
      (of_atom .<x>.) (of_atom .<y>.))) >.
```

results in generating the proper pair manipulation, and the proper type signature `int * int`:

```
('a, int * int -> int * int -> int * int) code =
.<fun x_1 -> fun y_2 ->
    (((fst x_1) * (fst y_2)),
    ((snd x_1) * (snd y_2)))>.
```

which is completely abstract at the generator level. Changing the implementation, results in a different generated code with proper type signature.

6.9 A Model Implementation for Float Types

The module `Float_Set` is an implementation of the interface `SET` for the floating point number type.

```
module Float_Set : SET =
struct
  type n = float
  type 'a ns = ('a, n) staged
  ...
end
```

The Inexact Sign Calculation

The interface `RING` specifies that the numbers have a sign. We provide two implementations for sign calculation: exact and inexact. The exact sign of a number x results from the comparison $x > 0$. The inexact calculation, however, depends on a tolerance in the calculation. The tolerance is a small number, referred to as ϵ. We use the following rule for determining the inexact sign of a floating point number:

$$\text{sgn}(x) = \begin{cases} \text{zero} & -\epsilon \leq x \leq \epsilon \\ \text{positive} & x > \epsilon, \\ \text{negative} & x < -\epsilon, \end{cases} .$$

By using functors, the sign calculation becomes parametric in the tolerance. The following module provides the exact sign calculation:

```
module Float_Sign_Exact =
struct
  let sgn x =
  ...
end
```

And the following functor provides the inexact sign calculation:

```
module Float_Sign_Inexact
  (E : sig val eps : float * ('a, float) code end) =
struct
  let eps = fst E.eps
  let eps_c = snd E.eps
  let sgn x =
  ...
end
```

The parameter `E` carries the tolerance information. The module `Float_Ring_Base` provides the ring operation except the sign calculation.

```
module Float_Ring_Base =
struct
  include Float_Set
  ...
end
```

`Float_Ring_Exact` extends `Float_Ring_Base` and adds the exact sign calculation.

```
module Float_Ring_Exact : RING =
struct
  include Float_Ring_Base
  include Float_Sign_Exact
end
```

Similarly, `Float_Ring_Inexact` extends `Float_Ring_Base` by the inexact sign calculation.

```
module Float_Ring_Inexact
 (E : sig val eps : float * ('a, float) code end) : RING =
struct
  include Float_Ring_Base
  include Float_Sign_Inexact (E)
end
```

`Float_Field_Exact`, and `Float_Field_Inexact` are implementations of `FIELD`. They are defined as:

```
module Float_Field_Exact : FIELD =
struct
  include Float_Field_Base
  include Float_Sign_Exact
end

module Float_Field_Inexact
 (E : sig val eps : float * ('a, float) code end) : FIELD =
struct
  include Float_Field_Base
  include Float_Sign_Inexact (E)
end
```

where `Float_Field_Base` extends `Float_Ring_Base` and implements the extra functionality of the division.

```
module Float_Field_Base =
struct
```

```
  include Float_Ring_Base
  ...
end
```

The same pattern is used for Float_Real_Base.

```
module Float_Real_Base =
struct
  include Float_Field_Base
  ...
end
```

Finally, the two modules `Foat_Real_Exact` and `Float_Real_Inexact` are defined as:

```
module Float_Real_Exact : REAL =
struct
  include Float_Real_Base
  include Float_Sign_Exact
end

module Float_Real_Inexact
 (E : sig val eps : float * ('a, float) code end) : REAL =
struct
  include Float_Real_Base
  include Float_Sign_Inexact (E)
end
```

6.10 Example: Staged Power

We demonstrate the use of this layer by an example. We reuse the same example from section 5.3. The difference here is that we choose two variabilities for the power function generator: the number type and the exponent. Recall, the power function is defined as:

```
# let rec power n x =
   match n with
   | 0 -> 1
```

```
    | n -> x * (power (n-1) x);;

val power : int -> int -> int = <fun>
```

We achieve the task of building a generator through four steps:

(1) Staging `power`. The `RING` type provides the multiplication operation and the constants 0 and 1. Let `R` be a module of type `RING`. `power` can then be rewritten into a generic form as follows:

```
let rec power n x =
  match n with
  | 0 -> Staged.of_immediate R.one
  | n -> R.mul_s x (power (n-1) x)
```

with the type:

```
val power : int -> 'a R.ns -> 'a R.ns = <fun>
```

(2) Parameterizing `power`. The function `power` is parametric in `R`. The functor `GenericPower`, below, provides the required parametrization over the number type.

```
module GenericPower (R : RING) =
struct
  let rec power n x =
    match n with
    | 0 -> Staged.of_immediate R.one
    | n -> R.mul_s x (power (n-1) x)
end
```

(3) Collecting variabilities. The generator has two variabilities: the number type, and the exponent. These two variabilities can be collected in the type:

```
module type POWER_VAR =
sig
  module R : RING
  val n : int
end
```

which becomes the input for the generator.

(4) Building the generator. The generator, `GenPower`, is defined as:

```
module GenPower (PV : POWER_VAR) =
struct
  let gen_power () =
    let module GP = GenericPower(PV.R) in
    .< fun x -> .~(Staged.to_code
                     (GP.power PV.n (Staged.of_atom .<x>.))) >.
end

module GenPower :
 functor (PV : POWER_VAR) ->
  sig
   val gen_power :
      unit -> ('a, PV.R.n -> PV.R.n) code
  end
```

`gen_power` is wrapped in a functor that takes the module `PV` as an input. `PV` contains the values for the variabilities. In other words, `PV` is the configuration for the generator. The function returned by `gen_point` has the code type of `PV.R.n -> PV.R.n`. That is, a unary function on the abstract number type `PV.R.n`.

Finally, we generate the power function for exponents 1, 2, and 3 using two different number types: `Float_Ring` and `Integer_Ring`. For the float type and exponent 2, the variability is:

```
module Float2 = struct
  module R = Float_Ring_Exact
  let n = 2
end
```

and the instantiation for the generator is: `module GenF2 = GenPower(Float2)`. The instantiated `gen_power` has the type:

```
gen_power : unit -> ('a, Float2.R.n -> Float2.R.n) code
```

where the abstract number type `PV.R.n` is bound to the concrete type `Float2.R.n`. Calling the generator results in:

```
# GenF2.gen_power () ;;

- : ('a, Float2.R.n -> Float2.R.n) code =
    .<fun x_1 -> (x_1 *. x_1)>.
```

The resultant code has no traces of `power`, and has the proper type `Float2.R.n -> Float2.R.n`. The following code snippet shows similar instantiations for `Float1`, `Float3`, `Integer1`, `Integer2`, and `Integer3`:

```
- : ('a, Float1.R.n -> Float1.R.n) code =
   .<fun x_1 -> x_1>.
- : ('a, Float3.R.n -> Float3.R.n) code =
   .<fun x_1 -> (x_1 *. (x_1 *. x_1))>.
- : ('a, Integer1.R.n -> Integer1.R.n) code =
  .<fun x_1 -> x_1>.
- : ('a, Integer2.R.n -> Integer2.R.n) code =
  .<fun x_1 -> (x_1 * x_1)>.
- : ('a, Integer3.R.n -> Integer3.R.n) code =
   .<fun x_1 -> (x_1 * (x_1 * x_1))>.
```

6.11 Conclusion

In this chapter we presented a hierarchy of staged number types. We presented a concrete implementation of integer, rational, and floating point numbers. Finally, through an example, we presented the usage of the abstractions in this layer.

Chapter 7

Affine Geometry

In this chapter we introduce various concepts that support geometric computations. We also augment the geometric concepts by introducing the modules and types for three layers:

- The linear algebra layer.
- The affine space layer.
- The geometric objects[1] layer.

The details of the module types and interfaces can be found in Appendix A. Several examples will be shown in chapter 8.

7.1 Introduction

Genericity in geometric computing can be achieved by abstraction, modularity, and layered designs. Proper abstraction includes identifying a number type, and a geometric algebra for geometric objects and operations on them. This approach resulted in generic higher-dimensional geometric kernels [MMN+97], and coordinate-free computations [GHW00]. A comparison of the different types of geometric algebras can be found in [FD03, Gol02]. Affine geometry treats n-dimensional spaces through the same set of abstract concepts. In effect, geometric primitives which are based on affine geometry scale to higher dimensions. However, the affine geometry does not allow a straightforward extension to different coordinate systems other than the orthogonal

[1] We use the term geometric *objects* to denote both objects such as hyperplanes, and computations such as orientation tests. The rationale is that the computations are *functional* objects, that is, objects with a functionality.

coordinate systems [DeR89]. A promising alternative geometric algebra can be found here [FD03].

The development of affine geometry presented here is adopted from [Aud03, dBvKOS97, Gal00, SE02]. The concepts are selected to satisfy two goals: scalability for n-dimensions, and high degree of abstraction. The higher abstraction allows a uniform (generic) representation for the same concept regardless of the dimension or the coordinate number type. The generative approach allows us to manipulate those generic forms and generate code that corresponds to the specialized forms of the concepts for a given dimension and number type.

7.2 Affine Spaces

Let $(\mathcal{D}, \mathcal{V})$ be a vector space over a division ring $\mathcal{D}$, and a set of free vectors $\mathcal{V}$. Free vectors are directions that can move freely in space. A vector v has a direction (denoted as $\hat{v}$) and a magnitude (denoted as $\|v\|$), but no fixed starting point. Let $\mathcal{P}$ be a set of points. Points are fixed positions in space. The triple $(\mathcal{D}, \mathcal{P}, \mathcal{V})$ is called an *affine space* if the following conditions hold:

1. For each pair of points p and q in $\mathcal{P}$ such that $p \neq q$, there exists a unique vector $v \in \mathcal{V}$ such that $v = p - q$.

2. For all points $p \in \mathcal{P}$ and vectors $v \in \mathcal{V}$, there exists a point $q \in \mathcal{P}$ such that $p + v = q$.

3. For any three points p, q, and r in $\mathcal{P}$, the following equation holds:

$$(q - p) + (r - q) = (r - p)\,.$$

Let $\{x_i\}_{i \in I}$ be a family of elements in space, having the same type (either points or vectors), and let $\{\lambda_i\}_{i \in I}$ be a family of scalars. The element $u = \sum_{i \in I} \lambda_i x_i$ is called the *linear combination* of the elements x_i. If the condition $\sum_{i \in I} \lambda_i = 1$ holds then u is called an *affine combination.*

A family $\{x_i\}_{i \in I}$ of points or vectors is said to be *linearly dependent* if there exists a family of non-zero scalars $\{\lambda_i\}_{i \in I}$ such that $\sum_{i \in I} \lambda_i x_i = \mathbf{0}$. The element $\mathbf{0}$ is the zero element (the zero vector or the zero point). The family is said to be *linearly independent* if there are no such scalars satisfying $\sum_{i \in I} \lambda_i x_i = \mathbf{0}$.

A family of points $\{p_i\}_{i \in I}$ is *affinely independent* if the family $\{p_k - p_i\}_{i \in I \setminus \{k\}}$ of vectors is linearly independent for some $k \in I$. In other words, if the point p_k is chosen as the origin.

7.2.1 Affine Subspaces

An affine subspace is a subset of an affine space closed under affine combinations. Formally, if $\mathcal{S} = (\mathcal{D}, \mathcal{P}, \mathcal{V})$ and $\mathcal{R} \subset \mathcal{P}$, then $\mathcal{R}$ is called a *subspace* of $\mathcal{S}$ if for all families of points $\{p_i\}_{i \in I}$ in $\mathcal{R}$, the affine combination $\sum_{i \in I} \lambda_i p_i$ is also in $\mathcal{R}$.

7.2.2 Bases

Let $\mathcal{S} = (\mathcal{D}, \mathcal{P}, \mathcal{V})$ be an affine space. Let $B = \{b_1, \ldots, b_k\}$ be a finite subset of $\mathcal{V}$. B is said to *span* $\mathcal{V}$ if for all v in $\mathcal{V}$ there exists an assignment of scalars $\lambda_1, \ldots, \lambda_k \in \mathcal{D}$ such that $v = \sum_{i=1}^{k} \lambda_i b_i$. In other words, all the vectors in $\mathcal{V}$ can be expressed (generated) as a linear combination of the vectors in B. B is called a basis of $\mathcal{S}$ if the following two conditions hold:

1. The vectors in B are linearly independent.
2. B spans $\mathcal{V}$.

7.2.3 Frames

Let $\mathcal{S} = (\mathcal{D}, \mathcal{P}, \mathcal{V})$ be an affine space, let $B = \{b_1, \ldots, b_k\} \subset \mathcal{V}$ be any basis for $\mathcal{V}$, and let O be any point in $\mathcal{P}$. The tuple $(O, b_1, \ldots, b_k)$ is called an *affine frame* of $\mathcal{S}$. The point O is called the *origin* of the frame. A frame is not unique. Given an affine frame, all the vectors and points in the space can be uniquely written as a linear combination of the frame elements. That is, every vector $v \in \mathcal{V}$ can be written as:

$$v = \sum_{i=1}^{k} v_i b_i = v_1 b_1 + \cdots + v_k b_k,$$

or as a matrix multiplication:

$$v = \begin{pmatrix} v_1 & \ldots & v_k \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_k \end{pmatrix}.$$

Similarly, every point $p \in \mathcal{P}$ can be written as:

$$p = O + \sum_{i=1}^{k} p_i b_i = O + \begin{pmatrix} p_1 & \dots & p_k \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_k \end{pmatrix}.$$

The scalar tuples $(v_1, \dots, v_k)$ and $(p_1, \dots, p_k)$ are called the *affine coordinates* for the vector v, and the point p, respectively. The coordinates depend on the choice of the frame.

7.2.4 Dimension and Codimension

The *dimension* of an affine space $\mathcal{S}$, denoted as $\dim(\mathcal{S})$, is equal to the cardinality of the set of basis B [Art91]. If $B = \{b_1, \dots, b_n\}$, then $\dim(\mathcal{S}) = n$ and the space is called an n-dimensional space.

The codimension is a relative concept between a space and it subspaces. If T is a subspace of S, then the *codimension of* T *in* S, denoted as $\text{codim}(T)$, is given by:

$$\text{codim}(T) = \dim(S) - \dim(T).$$

7.3 Affine Transforms

An affine transform is a mapping between two affine spaces which preserves the structure of the space [DeR89, SE02]. It maps points to points, vectors to vectors, and frames to frames. Other objects (e.g. lines and polygons) can be represented in terms of points and vectors, and thus can be mapped by affine transformations as well. Given two affine spaces $\mathcal{S}_1$ and $\mathcal{S}_2$, the function $f : \mathcal{S}_1 \rightarrow \mathcal{S}_2$ is said to be an *affine transform* (or affine transformation, or affine map) if for any family of points $\{p_i\}_{i \in I}$ in $\mathcal{S}_1$, and any family of scalars $\{\alpha_i\}_{i \in I}$ such that $\sum_{i \in I} \alpha_i = 1$, the equality:

$$f(\sum_{i \in I} \alpha_i p_i) = \sum_{i \in I} \alpha_i f(p_i)$$

holds.

Affine transformations can be composed to produce new transformations. Given two affine transforms T_1 and T_2, the transform $T_1 \circ T_2$ is the composition of T_1 and T_2 if and only if: for all elements x of the space, the following condition holds:

$$T_2(T_1(x)) = T_1 \circ T_2(x).$$

7.3.1 Matrix Representation of Affine Transforms

Affine transforms can be represented by the equation

$$T(x) = Ax + b$$

where A is an n by n matrix, and b and x are both n by 1 matrices. A is called the *linear transformation matrix*. x is the column matrix representing the affine coordinates of the object subject to transformation. b is the additive part of the transformation, normally called the *translation* part. To minimize the number of matrix operations involved, A and b can be augmented in a generalized matrix representation. Let $x = (x_1, \ldots, x_n)$, and $b = (b_1, \ldots, b_n)$, the transformation $T(x)$ can be rewritten as:

$$\begin{pmatrix} T(x) \\ 1 \end{pmatrix} = M \times \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix}$$

where M is called the *affine transformation matrix*. M is an augmented matrix which has the form:

$$M = \begin{pmatrix} & & b_1 \\ & A & \vdots \\ & & b_n \\ 0, \ldots, 0 & & 1 \end{pmatrix}.$$

The type of the transformation is determined by the values in the matrix M. We discuss two types of affine transforms: translation and scaling.

Translation. Translation corresponds to moving object in space. In a d-dimensional space, A is an identity matrix of size d. b is a column vector of d rows. b carries the translation information, that is, the displacement on each coordinate. Let the coordinates be numbered $1, 2$, up to d, the translation $b = (\Delta_1, \Delta_2, \ldots, \Delta_d)$ applied to an object with affine coordinates $(x_1, x_2, \ldots, x_d)$ results in a *translated object* $T(x)$ with coordinates $(x_1 + \Delta_1, x_2 + \Delta_2, \ldots, x_d + \Delta_d)$.

Scaling. In a d-dimensional space, A is a diagonal matrix of size d, and b is a zero column vector of d rows. The diagonal of A carries the scaling factor for each coordinate. Applying a scaling

$$A = \begin{pmatrix} s_1 & & 0 \\ & \ddots & \\ 0 & & s_d \end{pmatrix}$$

to an object with coordinates $(x_1, x_2, \ldots, x_d)$, results in a *scaled object* with coordinates:

$$(s_1 x_1, s_2 x_2, \ldots, s_d x_d).$$

7.4 Euclidean Geometry

Before introducing Euclidean space, we need to define the notion of a metric. Let $\mathcal{S} = (\mathcal{D}, \mathcal{P}, \mathcal{V})$ be an affine space, the *inner product* of two vector $u, v \in \mathcal{V}$, denoted as $\langle u, v \rangle$, is a mapping from $\mathcal{V} \times \mathcal{V}$ to $\mathcal{D}$. For an operator $\langle , \rangle$ to be an inner product it has to satisfy the following three conditions:

1. Symmetry: For all $u, v \in \mathcal{V}$, $\langle u, v \rangle = \langle v, u \rangle$.
2. Bi-linearity: For all $u, v, w \in \mathcal{V}$ and $a, b \in \mathcal{D}$:
   (a) $a \langle u, w \rangle + b \langle v, w \rangle = \langle au + bv, w \rangle$.
   (b) $a \langle u, v \rangle + b \langle u, w \rangle = \langle u, av + bw \rangle$.
3. Positive definiteness: Let $\mathbf{0}$ be a special vector in $\mathcal{V}$ called the zero vector with the property that $\langle \mathbf{0}, \mathbf{0} \rangle = 0$. $\langle , \rangle$ is positive definite if for all $v \in \mathcal{V} \setminus \{\mathbf{0}\}$, $\langle v, v \rangle > 0$.

The dot product [Aud03], denoted as $\langle \cdot \rangle$, is an inner product defined on vector spaces over real numbers. The dot product of two vectors $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$ is defined as:

$$u \cdot v = \sum_{i=1}^{n} u_i v_i.$$

Euclidean affine space, shortly Euclidean space, is an affine space equipped with the dot product $\langle \cdot \rangle$. The dot product defines the notions of length, distance and angle. The *length of a vector* v is defined as:

$$| v | = \sqrt{v \cdot v}.$$

A vector is called a *unit vector* if its length is 1. The *distance between two points* p and q is defined as:

$$\text{distance}(p, q) = | q - p | .$$

The angle between two vectors v and u is defined as:

$$\theta = \cos^{-1} \frac{v \cdot u}{| v | | u |}.$$

Orthonormal Basis. Two vectors v_1 and v_2 are *orthogonal* if $v_1 \cdot v_2 = 0$. A set of basis B is orthonormal if the following two conditions hold:

1. For all the vectors b in B: b is a unit vector.
2. For all b_i and b_j in B such that $i \neq j$: b_i and b_j are orthogonal.

7.5 The Linear Algebra Layer

An abstract implementation of affine geometry uses matrices. Matrices are generally inefficient in terms of time and space requirements. However, using the multi-staging layer, the cost of using matrices is alleviated and the generated code has no traces of the data structures of this layer. This is an example of how the cost of a convenient abstraction can be reduced by using multi-staging.

This section describes the linear algebra layer. This layer provides types and functionality for matrices and determinants over staged number types. The functionality and abstractions provided in this layer are motivated by the needs of the geometric computations in the other layers, not by the general scope of linear algebra.

7.5.1 The `TUPLE` Type

A tuple is a fixed-size ordered container of staged numbers. It supports projection, mapping, folding and conversion to and from lists. Tuples are used for representing the coordinates of vectors and points. Tuples can also be used to represent entries in some of the affine transforms.

For a tuple $t = \langle t_0, \ldots, t_i, \ldots, t_{d-1} \rangle$, the projection of the ith element is given by:

$$\texttt{proj}(t, i) = t_i.$$

The function `map` f t maps t into a new tuple given by:

$$\langle f(t_0), \ldots, f(t_{d-1}) \rangle .$$

The function `map2` fuses two tuples together. If $t = \langle t_0, \ldots, , t_{d-1} \rangle$ and $t' = \langle t'_0, \ldots, t'_{d-1} \rangle$, then:

$$\texttt{map2}\, f\, t\, t' = \langle f(t_0, t'_0), \ldots, f(t_{d-1}, t'_{d-1}) \rangle .$$

The fold operation is defined as:

$$\texttt{fold}\, f\, z\, t = f(t_0, f(t_1, \ldots f(z, t_{d-1}) \ldots)).$$

`mapfold` and `map2fold` are shorthands for the composition of `map` then `fold`, and `map2` then `fold`, respectively.

7.5.2 The `MATRIX` and `DETERMINANT` Types

The `MATRIX` type provides an abstraction of matrices of staged numbers. The interface specifies common matrix operations such as: matrix addition, subtraction, multiplication, augmentation and minor. Matrix augmentation and minor are important operations for the computations of the affine transforms.

The type `DETERMINANT` defined as:

```
module type DETERMINANT =
sig
  module N : FIELD
  module M : MATRIX with type 'a n_s = 'a N.ns
  val eval : 'a M.m -> 'a N.ns
end
```

is an abstraction of the determinant expansion computation. A module of type `DETERMINANT` is required to have two instances of `FIELD` and `MATRIX` types. The type constraint:

```
with type 'a n_s = 'a N.ns
```

ensures that the instantiation is appropriate. In other words, it allows the type checker to check whether the given matrix `M` is defined over the same number type of `N` or not. This encoding of the condition through type constraints, enhances the reliability of the generator by statically checking the composability of the variabilities.

The function `eval` takes a matrix `m` from the type `MATRIX` and returns the determinant expansion of it. The entries of `m` are staged numbers. It is up to the implementation to define `eval` in terms of the staged operators of `N` and `M`, or not. In the former case, `eval` can generate code for the determinant expansion. In the latter case, `eval` generates code to perform the expansion at run-time.

Two concrete implementations are given for `DETERMINANT` and `MATRIX`. The functor `Determinant` defined as:

```
module Determinant (N : FIELD)
 (M : MATRIX with type 'a n_s = 'a N.ns) : DETERMINANT =
struct
 ...
end
```

is a model of determinants. The functor `Matrix`

```
module Matrix (N : FIELD) : MATRIX =
struct
 ...
end
```

is a model for matrices.

7.6 The Affine Space Layer

The affine space layer is motivated by the definition of the affine spaces. The generic design is based on the triple $(\mathcal{D}, \mathcal{P}, \mathcal{V})$ defining a space. The number types layer provides abstractions for the division ring $\mathcal{D}$. The affine space layer provides abstractions for points, vectors and operations on them. The rest of this section introduces the the module types in this layer.

7.6.1 The VECTOR and POINT Types

The module type VECTOR, defined as:

```
module type VECTOR =
sig
  module N : REAL
  type vector
  type 'a vector_s = ('a, vector) staged
  ...
end
```

is the type of Euclidean n-dimensional free vectors. The interface defines two types vector and vector_s. vector is the type of vectors used at run-time, i.e., the base type. vector_s is the type of staged vectors used at generation time. The number type N is the type of coordinates. The choice of REAL is to accommodate the square root operation required for the Euclidean length metric.

The interface includes various vector operations, such as: vector addition and subtraction, dot product, cross product, scaling, and length. The functions have dimensionless types. This is due to the fact that all the functions are defined over the abstract type vector_s. For example, subtracting two vectors is defined as:

```
val sub : 'a vector_s -> 'a vector_s -> 'a vector_s
```

where the dimension is not dictated by the function signatures. However, the field val dim : int provides the dimension information. The module implementation is responsible for setting the value of the dimension.

Similar to VECTOR, the module type POINT, defined as:

```
module type POINT =
sig
```

```
  module N : REAL
  module V : VECTOR with module N = N
  type point
  type 'a point_s = ('a, point) staged
  val dim : int
  ...
end
```

is an abstraction of a point in an n-dimensional affine space. Points have coordinates from a `REAL` number type. Points rely on a `VECTOR` for point subtraction. That is why a module of type `POINT` has to know the type of vectors. The interface includes basic operations on points, such as extracting a coordinate, and subtracting two points to get a vector.

A common operation in computational geometry is ordering points. The module type `ORDERED_POINT` is an extension of the type point with the addition of an order relation. Iso-axis order is an interesting order on points. Iso-axis means in the direction of the axis. The ordering `ISO_AXIS_ORDERED_POINT` is a modification of `ORDERED_POINT`. It provides a coordinate-wise comparison. For example, the function `lt` in `ORDERED_POINT` has the type:

```
val lt : 'a point_s -> 'a point_s -> 'a Bool.b
```

whereas the same function in `ISO_AXIS_ORDERED_POINT` has the type:

```
val lt : int -> 'a point_s -> 'a point_s -> 'a Bool.b
```

The extra integer parameter specifies that the comparison is to be done on the ith coordinate.

7.6.2 The AFFINE Type

The type `AFFINE` provides an abstraction for the affine transformation concept. The interface `AFFINE`, defined as:

```
module type AFFINE =
sig
  module N : REAL
  module M : MATRIX with
               type 'a n_s = 'a N.ns
```

```
  module V : VECTOR with module N = N
  module P : POINT  with
             module N = N and module V = V
  module T : TUPLE
  type 'a t
  val apply_p : 'a t -> 'a P.point_s -> 'a P.point_s
  val apply_v : 'a t -> 'a V.vector_s -> 'a V.vector_s
  val compose : 'a t -> 'a t -> 'a t
  (* Soem affine transforms *)
  val id : int -> 'a t
  val translation : 'a V.vector_s -> 'a t
  val scaling : 'a T.t -> 'a t
end
```

provides the type of the affine transform. The interface includes functions to create, compose, and apply the transform to points and vectors. Three types of transforms are supported: identity, translation and scaling.

A concrete implementation, `AffineTransformations`, is given by the following functor:

```
module AffineTransformations
 (N : REAL) (V : VECTOR with module N = N)
 (P : POINT with module N = N and module V = V)
 (T : TUPLE) : AFFINE =
struct
  module N = N
  module M = Matrix (N)
  module V = V
  module P = P
  module T = T
 ...
end
```

The transforms are internally represented as matrices.

7.7 Hyperplanes

A hyperplane is an affine subspace of codimension 1. In an n-dimensional space, a hyperplane is uniquely defined by n points satisfying the equation:

$$\lambda_1 p_1 + \cdots + \lambda_n p_n = c$$

where $\lambda_1, \ldots, \lambda_n$ and c are scalars and at least one of the scalars $\lambda_1, \ldots, \lambda_n$ is not zero. A hyperplane can also be defined using a point p_0 and a normal vector n:

$$H = \{p : n \cdot (p - p_0) = 0\}.$$

Basis of Hyperplane. A hyperplane in d-dimensional space is a space of dimension $d-1$. This means that a hyperplane has a basis, an origin and a frame. Any point lying on the plane has local coordinates with respect to a given hyperplane frame. Finding the local coordinates of a point is useful for operations such as projecting points on a hyperplane.

Let H be a hyperplane defined by the points $p_1, \ldots, p_n$. Fix a point p_k to be the origin, where $1 \leq k \leq n$. A basis is given by:

$$B = \{b_1, \ldots, b_{n-1}\} = \{p_i - p_k : i \neq k\}.$$

The frame can then be defined as:

$$(p_k, b_1, \ldots, b_{n-1}).$$

The general problem of finding orthonormal basis of a hyperplane is not addressed here in this thesis.

Distance and Projection. The signed distance from a point p to a hyperplane H defined by a normal vector n and a point p_0 is given by [SE02]:

$$\delta(H, p) = n \cdot (p - p_0).$$

The projection of p onto H is given by:

$$p' = p - \delta(H, p)n.$$

Orientation. A hyperplane H partitions the space into two disjoint halfspaces: positive and negative, given by the inequalities:

$$\lambda_1 p_1 + \cdots + \lambda_n p_n < c$$

and

$$\lambda_1 p_1 + \cdots + \lambda_n p_n > c.$$

Any point x can lie either on the positive side (positive halfspace), the negative side (negative halfspace), or on the hyperplane. This relative position is usually called

the *orientation* of x relative to H. The orientation is determined by the sign of the following determinant:

$$\Delta_H(x) = \begin{vmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} & 1 \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,n} & 1 \\ x_1 & x_2 & \cdots & x_n & 1 \end{vmatrix}$$

where the hyperplane is defined by the points $p_1, \ldots, p_n$ ($p_{i,j}$ being the j^{th} coordinate of the point p_i). The orientation takes the following cases:

$$\text{orient}(H, x) = \begin{cases} \text{on} & \Delta_H(x) = 0 \\ \text{positive} & \Delta_H(x) > 0 \\ \text{negative} & \Delta_H(x) < 0 \end{cases} .$$

In the computational geometry jargon (cf. [dBvKOS97]), these sides are usually called *above* and *below* in the 3D case. In the 2D case, the orientations are called *left* and *right* turn. Counter-clockwise (ccw) and clockwise (cw) are other names for orientation. The following table summarizes the different namings:

positive	left	ccw	above
negative	right	cw	below
zero	collinear	collinear	coplanar

7.8 The Types `HYPER_PLANE` and `Hplane_Operations`

The interface `HYPER_PLANE` specifies a hyperplane constructable from a set of points. The interface allows the extraction of the hyperplane polynomial, the hyperplane bases, and local frame.

The functor `Hplane_Operations` takes a module of type `HYPER_PLANE` and builds a module that contains two hyperplane operations: `dist` and `project`. `dist` computes the distance between a hyperplane and a point. `project` computes the projection of a point on a hyperplane.

7.9 The Module `Orient`

The module `Orient` provides the orientation test. It is defined as a functor parametrized by the hyperplane type.

```
module Orient (H : HYPER_PLANE) =
struct
  ...
end
```

`Orient` defines a function `orient h p` which gives the sign of the orientation determinant for a point `p` and a hyperplane `h`. The functor also includes several conveniently-named functions for testing specific sides of the orientation. For example, `ccw h p` checks if the point `p` is counter-clockwise with the hyperplane `h`. `pos h p` has the same semantics as `ccw`. This redundancy was introduced to accommodate the different names of the orientation tests.

7.10 Hyperspheres

Hypersphere (also n-sphere) is a generalization of the circle concept in higher dimensions. A hypersphere S with a centre c and a radius r is defined as the set of all points that are r-equidistant from c. Formally:

$$S = \{x : \|x - c\| = r\}.$$

In an n-dimensional space, a hypersphere S is uniquely defined by $n+1$ points lying on its surface. If the points are $p_1, \ldots, p_{n+1}$, each described by a tuple of coordinates $(p_{i,1}, \ldots, p_{i,n})$, then the equation of S is:

$$\Delta_S(x) = 0$$

where $x = (x_1, \ldots, x_n)$ is any point lying on the sphere surface, and:

$$\Delta_S(x) = \begin{vmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} & (p_{1,1}^2 + \cdots + p_{1,n}^2) & 1 \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} & (p_{2,1}^2 + \cdots + p_{2,n}^2) & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{n+1,1} & p_{n+1,2} & \cdots & p_{n+1,n} & (p_{n+1,1}^2 + \cdots + p_{n+1,n}^2) & 1 \\ x_1 & x_2 & \cdots & x_n & (x_1^2 + \cdots + x_n^2) & 1 \end{vmatrix}.$$

The following table shows various examples of hyperspheres in different dimensions:

Dimension	Hypersphere	Points
1	Line segment	2
2	Circle	3
3	Sphere	4
n	n-sphere	$n+1$

Hypervolume. The n-dimensional volume (hypervolume) of a hypersphere with radius r is given by:

$$V_n = C_n r^n,$$

and the hyper-surface area is given by:

$$S_n = nC_n r^{n-1}$$

where

$$C_n = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}$$

The following table gives a few examples of evaluations of the volume and surface area at dimensions 1, 2, 3 and 4.

Dimension	C_n	S_n	V_n
1	2	2	$2r$
2	π	$2\pi r$	πr^2
3	$\frac{4\pi}{3}$	$4\pi r^2$	$\frac{4\pi}{3}r^3$
4	$\frac{\pi^2}{2}$	$2\pi^2 r^3$	$\frac{\pi^2}{2}r^4$

Sideness. The surface of the hypersphere divides the space into two disjoint sets: interior and exterior. Any point $x = (x_1, \ldots, x_n)$ can either lie on the surface of the hypersphere, outside, or inside. The predicate, inside(S, x), determines the position of a point x relative to a hypersphere S:

$$\text{inside}(S,x) = \begin{cases} \text{on} & \Delta_S(x) = 0 \\ \text{inside} & \Delta_S(x) > 0 \\ \text{outside} & \Delta_S(x) < 0 \end{cases}.$$

Circumcentre and Circumradius. The centre of a hypersphere can be computed using the matrix [Bou]:

$$M = \begin{pmatrix} p_{1,1}^2 + \cdots + p_{1,n}^2 & p_{1,1} & \cdots & p_{1,n} & 1 \\ p_{2,1}^2 + \cdots + p_{2,n}^2 & p_{2,1} & \cdots & p_{2,n} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{n+1,1}^2 + \cdots + p_{n+1,n}^2 & p_{n+1,1} & \cdots & p_{n+1,n} & 1 \end{pmatrix}.$$

Let M_k denote the minor of the matrix M resulting from deleting the column k. The centre $c = (c_1, \ldots, c_n)$ is given by:

$$c_i = (-1)^{i+1} \frac{M_{i+1}}{2M_1}$$

and the radius is given by:

$$r^2 = (-1)^{n+1}\left(\sum_{i=1}^{n} c_i^2 - \frac{M_{n+2}}{M_1}\right)$$

7.11 The Types HSPHERE and Sphere_Operations

The module type `HSPHERE` is an abstraction of the hypersphere concept. It provides an abstract type for spheres. In an n-dimensional space, a sphere can be constructed in two different ways: from $n+1$ points, or by specifying a centre point and a radius.

The functor `Sphere_Operations` takes a module of type `HSPHERE` as an input, and builds a module that contains the following functions:

- `centre`: computes the centre of a sphere.
- `radius`, `radius2`: computes the radius (resp. squared radius) of a sphere.
- `content`: computes the hyper-volume of a sphere.
- `surface`: computes the hyper-surface area of a sphere.

7.12 The Module Insphere

This module provides a generic test for a point inside a sphere. It is implemented as a functor with the input being a module `S` of type `HSPHERE`. `inside s p` tests if the point `p` is inside the sphere `s`. The interface is dimensionless. The dimension information is retrieved from the passed instance of `S`.

7.13 Simplex

An affine combination $\sum_{i\in I} \lambda_i p_i$ of a family of points $\{p_i\}_{i\in I}$, is called a *convex combination* if $\lambda_i \geq 0$ for all $i \in I$. A finite subset of points S is said to be *convex* if for all points x and y in S, the line segment xy lies completely in S. The *convex hull* of a set of points P, denoted as $\mathbb{CH}(P)$, is the smallest convex set containing P. $\mathbb{CH}(P)$ is defined as:

$$\mathbb{CH}(P) = \left\{\sum_{i\in I} \lambda_i p_i : p_i \in P \text{ and } \sum_{i\in I} \lambda_i = 1 \text{ and } \lambda_i \geq 0 \text{ for all } i\right\}$$

In other words, the convex combination of any set of points $p_1, \ldots, p_k$ in P is also in $\mathbb{CH}(P)$. The convex hull $\mathbb{CH}(P)$ defines a convex domain Ω with a boundary denoted as $\partial\Omega$.

d-Simplex

A d-simplex in a d-dimensional space is the convex hull of $d+1$ affinely independent points. 0-simplex is a vertex (point), a 1-simplex is a line segment, 2-simplex is a triangle, 3-simplex is a tetrahedron.

The points forming the simplex are called the *vertices* of the simplex. If V is the set of vertices of a simplex S, then any subset F of V is called a *face* of the simplex. The face opposite to vertex v_i is the face formed by the vertices $V \backslash \{v_i\}$.

Volume. Let S be a d-simplex defined by the points $p_1, \ldots, p_{d+1}$. The signed volume of S is given by the equation [Ste66]:

$$\frac{1}{d!}\begin{vmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} & 1 \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{d,1} & p_{d,2} & \cdots & p_{d,n} & 1 \\ p_{d+1,1} & p_{d+1,2} & \cdots & p_{d+1,n} & 1 \end{vmatrix}$$

Circumsphere. Given a simplex S with vertices $p_1, \ldots, p_{d+1}$, the *circumscribed sphere* (shortly, circumsphere), is the hypersphere passing through the vertices of S.

7.14 The Types `VERTEX` and `SIMPLEX`

A vertex is always attached to some simplex. This introduces a mutual dependency between the vertex and simplex types. To break the tie, the type `VERTEX` declares an abstract type `simplex_s` whereas the interface `SIMPELX` uses explicitly an instance module of the type type `VERTEX`.

The interface for `VERTEX` provides basic construction facilities. A vertex is created by specifying the point determining its location. The function `attach` attaches an incident simplex to a vertex.

The interface for `SIMPLEX` provides functionality for construction of simplices, retrieval of vertices and faces, and accessing neighborhood information. Two versions of face retrieval are included in the interface: non-oriented faces, and oriented faces. The vertices of an oriented face f opposite to a given vertex v are ordered such that: `orient h p = Ccw`, where `h` is the hyperplane passing through the vertices of f, and `p` is the point specifying the geometric location of v.

7.15 The Module `Inside`

This module provides a generic test for a point inside a simplex. The parametricity is realized using a functor. The inputs to the functor are two modules of type `HYPER_PLANE` and `HSPHERE`.

7.16 Conclusion

In this chapter we presented a development of the affine geometry concepts, and their corresponding modules and type in three layers: linear algebra, affine geometry, and geometric objects. Affine geometry was chosen as the underlying geometric algebra because it treats n-dimensional spaces through the same set of abstract concepts. In effect, geometric primitives scale to higher dimensions. The concepts of affine geometry that suites the mesh generation domain were implemented as abstract interfaces using OCaml's module types. The interfaces were chosen to be generic and dimensionless. The dimension information is statically fed at the generation-time by the modules implementing these interfaces. The resulting layers are abstract and generic.

Chapter 8

Implementation and Results

This chapter presents modules that implement the concepts in the following layers: linear algebra, affine geometry, and geometric objects. Other modules were presented in earlier chapters. Several examples for building code, generators and the resulting code, are presented.

It is worthnoting that the additional infrastructure required for building the code generator and capturing the variabilities using the OCaml module system, results in a simplified code. Several examples in this chaper show this cost trade-off in details.

8.1 The Tuple Models

The module `Tuple1D` defined below provides a model for a 1-tuple. A 1-tuple is a number.

```
module Tuple1D : TUPLE =
struct
  type 'a t = 'a
  let dim = 1
  ...
end
```

Two models of 2-tuples were implemented: `Record2D` and `Pair2D`. `Record2D` uses record types to store the two elements in the tuple, whereas `Pair2D` uses product types (pairs). `Record2D` is defined as:

```
type 'a rec2_type = { c0 : 'a ; c1 : 'a }
```

```
module Record2D : TUPLE =
struct
  type 'a t = 'a rec2_type
  ...
end
```

and Pair2D is defined as:

```
module Pair2D : TUPLE =
struct
  type 'a t = 'a * 'a
  ...
end
```

Record3D is an implementation of a 3-tuple using records.

```
type 'a rec3_type = { x : 'a ; y : 'a ; z : 'a }
module Record3D : TUPLE =
struct
  type 'a t = 'a rec3_type
  ...
end
```

8.2 The Module VectorStaged

The functor VectorStaged is a model for VECTOR. It implements an n-dimensional free vectors over a staged number type. VectorStaged is defined as:

```
module VectorStaged
 (N : REAL)
 (T : TUPLE) : VECTOR  =
...
```

It takes two parameters: the number type N, and the tuple type T. Internally, the vector type is represented as a staged tuple from T. The operations are written using staged constructs.

Eliminating Code Duplication

The operations of `VectorStaged` are implemented such that a let-insertion is used whenever duplication is expected. For example, consider the function `length` for computing the length of a vector. A naive implementation of `length` is:

```
(* N is a number type *)
let length v = N.sqrt_s (dot v v)
```

However, if `v` is an non-atomic expression, e.g. a computation, the resulting code will contain duplications. A let-binding should be used here. The variable `v`, repeated twice in `dot v v`, can be bound to a let expression by:

```
let length v = N.sqrt_s (Code.let_ v (fun v-> (dot v v)))
```

which avoids the code duplications.

8.3 Example: Generation of Dot Product

As an example, we write a generator for dot product of two vectors using `VectorStaged`. We fix the number type `Float`, and instantiate three different concrete implementations of `VectorStaged`:

1. `V1`: using `Tuple1D`, a 1D implementation of tuples.
2. `V2R`: using `Record2D`, a 2D implementation of tuples as records.
3. `V2P`: using `Pair2D`, a 2D implementation of tuples as pairs.
4. `V3R`: using `Record3D`, a 3D implementation of tuples as records.

The following generator can be used to generate the code for dot product:

```
module Gen (V : VECTOR) =
struct
  let dot _ =
    .< fun a b -> .~(Staged.to_code (
            V.dot (of_atom .<a>.) (of_atom .<b>.))) >.
end
```

Instantiating the generator for `V1`, `V2R`, `V2P`, and `V3R`, and calling `Gen.dot ()`, results in the following code (up to indentation):

```
(* 1D *)
module V1 = VectorStaged(Float_Real_Exact)(Tuple1D)
module GenV1 = Gen (V1)
GenV1.dot ()
('a, V1.vector -> V1.vector -> V1.N.n) code =
.<fun a_1 -> fun b_2 -> (a_1 *. b_2)>.

(* 2D Using Records *)
module V2R = VectorStaged(Float_Real_Exact)(Record2D)
module GenV2R = Gen (V2R)
GenV2R.dot ()
('a, V2R.vector -> V2R.vector -> V2R.N.n) code =
.<fun a_1 -> fun b_2 -> ((a_1.c0 *. b_2.c0)
    +. (a_1.c1 *. b_2.c1))>.

(* 2D Using Pairs *)
module V2P = VectorStaged(Float_Real_Exact)(Pair2D)
module GenV2P = Gen (V2P)
GenV2P.dot ()
('a, V2P.vector -> V2P.vector -> V2P.N.n) code =
.<fun a_1 ->
   fun b_2 -> (((fst a_1) *. (fst b_2))
      +. ((snd a_1) *. (snd b_2)))>.

(* 3D Using Records *)
module V3R = VectorStaged(Float_Real_Exact)(Record3D)
module GenV3R = Gen (V3R)
GenV3R.dot ()
('a, V3R.vector -> V3R.vector -> V3R.N.n) code =
.<fun a_1 ->
   fun b_2 -> ((a_1.z *. b_2.z) +.
     ((a_1.x *. b_2.x) +. (a_1.y *. b_2.y)))>.
```

The generator has the same interface: dot product of two atomic code expressions of vectors `a` and `b`. However, the generated code has high intentionality, and resembles exactly the choices of dimension (1D, 2D, or 3D), and the underlying data structure (pairs and records). Another notable features is: there are no traces of the internal

(generation-time) routines. For example, `dot` is internally implemented as fusion and map of two tuples:

```
(* map2fold on vectors using the representation of tuples *)
let map2fold m f z v v' =
  let bnow v v' = T.map2fold_n m f z v v'
  and blater v v' = (T.map2fold_c m f z v v') in
  mk_binary { bnow = bnow; blater = blater } v v'

(* The dot product *)
let dot v0 v1 =
  map2fold N.mul_s N.add_s N.zero v0 v1
```

However, none of the functions `T.map2fold_n`, `T.map2fold_c`, or other generation-time functions, appear in the generated code.

8.4 The Module `En_Point`

The functor `En_Point` provides a parametrized implementation of an n-dimensional point. The base type for points is a tuple. Hence the operations implemented in `En_Point` are dimensionless. The dimension is determined by `T`.

```
module En_Point (N : REAL)
  (V : VECTOR with module N = N)
  (T : TUPLE) : POINT =
struct
  module N = N
  module V = V
  type point = N.n T.t
  type 'a point_s = ('a, point) staged
  ...
end
```

The functor `Iso_Axis_Ordered_En_Point` defined as:

```
module Iso_Axis_Ordered_En_Point (N : REAL)
  (V : VECTOR with module N = N)
  (T : TUPLE) : ISO_AXIS_ORDERED_POINT =
struct
```

```
  include En_Point(N)(V)(T)
  ...
end
```

is a model for ordered points. It provides an axis-oriented ordering over the point type.

8.5 Example: Generation of 2D and 3D Translations

We use the following generator:

```
module Gen (AT : AFFINE) =
struct
  let translate () =
    .< fun p v -> .~(
         (* 1. lift to abstract *)
         let p' = of_atom .<p>.
         and v' = of_atom .<v>. in
         (* 2. call staged generator *)
         let t = AT.translation v' in
             let y = AT.apply_p t p' in
         (* 3. concretize *)
         Staged.to_code y) >.
end

module Gen :
 functor (AT : Affine.AFFINE) ->
  sig
   val translate :
     'a -> ('b,
       AT.P.point -> AT.V.vector -> AT.P.point) code
 end
```

Notice that the body of the generator has 3 steps: 1. Lift the function parameters into the abstract type (`staged`). 2. Call the staged generator routines. Those are the staged functions provided by the `AFFINE` interface. 3. Concretize the results back. Instantiating the generator with floating point numbers and vectors in 2D results in the following code (up to indentation):

```
module V2R = VectorStaged (Float_Real_Exact) (Record2D)
module P2R = En_Point (Float_Real_Exact) (V2R) (Record2D)
module AT2R = AffineTransformations
               (Float_Real_Exact) (V2R) (P2R) (Record2D)
module G2R = Gen (AT2R)
G2R.translate ()

('a, AT2R.P.point -> AT2R.V.vector -> AT2R.P.point) code =
 .<fun p_1 -> fun v_2 ->
     {c0 = (p_1.c0 +. v_2.c0); c1 = (p_1.c1 +. v_2.c1)}>.
```

and the following code in the 3D case:

```
module V3R = VectorStaged (Float_Real_Exact) (Record3D)
module P3R = En_Point (Float_Real_Exact) (V3R) (Record3D)
module AT3R = AffineTransformations
               (Float_Real_Exact) (V3R) (P3R) (Record3D)
module G3R = Gen (AT3R)
G3R.translate ()

('a, AT3R.P.point -> AT3R.V.vector -> AT3R.P.point) code =
.<fun p_1 -> fun v_2 ->
    {x = (p_1.x +. v_2.x);
     y = (p_1.y +. v_2.y);
     z = (p_1.z +. v_2.z)}>.
```

The generated code has no traces of the matrices used for the internal representation of the translation.

8.6 Example: Generation of Distances in 1D and 2D

We present two example generations of the distance between a point and a 0-hyperplane and a 1-hyperplane. A 0-hyperplane is a point in 1-dimensional space, and a 1-hyperplane is a line in 2D space. Recall that the distance between a point x and a hyperplane H with normal n and an origin p_0, is given by:

$$n \cdot (x - p_0).$$

The case of the 0-hyperplane is interesting because the normal vector $n = 1$, and hence the equation reduces to subtraction of two numbers. The instantiation below reflects exactly this derivation in the 1D case.

We use the following generator:

```
module Gen (H: HYPER_PLANE) =
struct
  module OP = Hplane_Operations (H)
  let dist () =
    .< fun ps p -> .~(
         (* Lift *)
         let p' = of_atom .<p>.
         and ps' = Array.to_list
           (Array.init H.V.dim (fun i -> of_atom .<ps.(i)>.)) in
         (* create a hyperplane *)
         let h = H.of_points ps' in
         let d = OP.dist h p' in
         to_code d) >.
end
```

where `dist` generates a code for a function `H.P.point array -> H.P.point -> H.V.N.n`. The point x is given by the second argument, and the points defining H are given by the first argument[1].

The code below instantiates the point and vector types in 1D.

```
module V1 = VectorStaged (Float_Real_Exact) (Tuple1D)
module P1 = En_Point (Float_Real_Exact) (V1) (Tuple1D)
module H0 = N_plane (Float_Real_Exact)(V1)(P1)
module GenDist1D = Gen (H0)
GenDist1D.dist ()
```

When the generator is invoked, the complex computations correctly reduces to subtraction of two numbers:

```
('a, H0.P.point array -> H0.P.point -> H0.V.N.n) code =
.<fun ps_1 -> fun p_2 -> (ps_1.(0) -. p_2)>.
```

[1]The usage of an array instead of one point is a specific choice in this example. This choice effectively allows the number of points defining a hyperplane to vary in different instantiations. This facilitates reusing the same generator with spaces of varying dimensions.

The instantiation and output for the 2D case is:

```
module V2R = VectorStaged (Float_Real_Exact) (Record2D)
module P2R = En_Point (Float_Real_Exact) (V2R) (Record2D)
module H1 = N_plane (Float_Real_Exact)(V2R)(P2R)
module GenDist2D = Gen (H1)
GenDist2D.dist ()

('a, H1.P.point array -> H1.P.point -> H1.V.N.n) code =
.<fun ps_1 ->
   fun p_2 ->
    let _v_10 =
     let _v_9 =
      let _v_7 =
       (* The hyperplane is represented by two points ps_1.(0)
          and ps_1.(1). The position vectors for which are _v_5
          and _v_6, respectively.
       let _v_5 = {c0 = (ps_1.(0)).c0; c1 = (ps_1.(0)).c1} in
       let _v_6 = {c0 = (ps_1.(1)).c0; c1 = (ps_1.(1)).c1} in
       (* Compute the normalized vector _v_9 = ps_1.(1) - ps_1.(0). *)
       {c0 = (_v_5.c0 -. _v_6.c0); c1 = (_v_5.c1 -. _v_6.c1)} in
      let _v_8 = (sqrt ((_v_7.c0 *. _v_7.c0) +.
                        (_v_7.c1 *. _v_7.c1))) in
      {c0 = (_v_7.c0 /. _v_8); c1 = (_v_7.c1 /. _v_8)} in
     (* The normal _v_10 = perp _v_9*)
     {c0 = (~-. _v_9.c1); c1 = _v_9.c0} in
    let _v_11 =
     (* Normal vector was computed. Evaluate the distance equation. *)
     let _v_3 = {c0 = (ps_1.(0)).c0; c1 = (ps_1.(0)).c1} in
     let _v_4 = {c0 = p_2.c0; c1 = p_2.c1} in
     {c0 = (_v_3.c0 -. _v_4.c0); c1 = (_v_3.c1 -. _v_4.c1)} in
    ((_v_10.c0 *. _v_11.c0) +. (_v_10.c1 *. _v_11.c1))>.
```

8.6.1 Example: Generating Insphere Test for 1D and 2D

We use the following generator to generate the insphere test. The module `Gen` is parametric in the number and tuple types. It instantiates the required modules, and generate an `in_` call.

```
module Gen (N : REAL) (T : TUPLE) =
struct
  module V = VectorStaged (N) (T)
  module P = En_Point (N) (V) (T)
  module S = Sphere (N) (P)
  module IS = Insphere (S)
  let insphere () =
    .< fun ps p -> .~(Staged.to_code (
         let p' = of_atom .<p>.
         and ps' = Array.init (V.dim+1)
                      (fun i -> of_atom .<ps.(i)>.) in
         let s = S.of_points (Array.to_list ps') in
         IS.in_ s p')) >.
end
```

In the 1D case, the test is equivalent to the following expansion:

$$\begin{vmatrix} \texttt{ps\_1.(0)} & \texttt{ps\_1.(0)*ps\_1.(0)} & 1 \\ \texttt{ps\_1.(1)} & \texttt{ps\_1.(1)*ps\_1.(1)} & 1 \\ \texttt{p\_2} & \texttt{p\_2*p\_2} & 1 \end{vmatrix} = 0.$$

The generated code (using `Float_Real_Exact` and `Tuple1D`) is:

```
- : ('a, GS1.S.P.point array -> GS1.S.P.point -> bool) code =
.<fun ps_1 ->
   fun p_2 ->
    (((ps_1.(0) *. ((ps_1.(1) *. ps_1.(1)) +. (~-. (p_2 *. p_2)))) +.
       ((~-. ((ps_1.(0) *. ps_1.(0)) *. (ps_1.(1) +. (~-. p_2)))) +.
         ((ps_1.(1) *. (p_2 *. p_2)) +.
           (~-. ((ps_1.(1) *. ps_1.(1)) *. p_2))))) > 0.)>.
```

In the 2D case, the test is equivalent to the following expansion (using `Float_Real_Exact` and `Record2D`):

$$\begin{vmatrix} \texttt{(ps\_1.(0)).c0} & \texttt{(ps\_1.(0)).c1} & \texttt{(ps\_1.(0)).c0*(ps\_1.(0)).c1} & 1 \\ \texttt{(ps\_1.(1)).c0} & \texttt{(ps\_1.(1)).c1} & \texttt{(ps\_1.(1)).c0*(ps\_1.(1)).c1} & 1 \\ \texttt{(ps\_1.(2)).c0} & \texttt{(ps\_1.(2)).c1} & \texttt{(ps\_1.(2)).c0*(ps\_1.(2)).c1} & 1 \\ \texttt{p\_2.c0} & \texttt{p\_2.c1} & \texttt{p\_2.c0*p\_2.c1} & 1 \end{vmatrix} = 0.$$

The generated code is long. We only show a part of it:

```
.<fun ps_1 ->
   fun p_2 ->
    ((((ps_1.(0)).c0 *.
        (((ps_1.(1)).c1 *.
           (let _v_7 = {c0 = (ps_1.(2)).c0; c1 = (ps_1.(2)).c1} in
            let _v_8 = {c0 = (ps_1.(2)).c0; c1 = (ps_1.(2)).c1} in
            ((_v_7.c0 *. _v_8.c0) +. (_v_7.c1 *. _v_8.c1)) +.
             (~-.
               let _v_9 = {c0 = p_2.c0; c1 = p_2.c1} in
               let _v_10 = {c0 = p_2.c0; c1 = p_2.c1} in
               ((_v_9.c0 *. _v_10.c0) +. (_v_9.c1 *. _v_10.c1))))) +.
          ((~-.
             (let _v_5 = {c0 = (ps_1.(1)).c0; c1 = (ps_1.(1)).c1} in
              let _v_6 = {c0 = (ps_1.(1)).c0; c1 = (ps_1.(1)).c1} in
              ((_v_5.c0 *. _v_6.c0) +. (_v_5.c1 *. _v_6.c1)) *.
               ((ps_1.(2)).c1 +. (~-. p_2.c1)))) +.
            (((ps_1.(2)).c1 *.
...
```

8.6.2 Example: Generating Orientation Test for 1D Using Exact and Inexact Zero

We demonstrate choosing different number types through this example. The following generator:

```
module Gen (N : REAL) (T : TUPLE) =
struct
  let col n =
    let module V = VectorStaged (N) (T) in
    let module P = En_Point (N) (V) (T) in
    let module H = N_plane (N) (V) (P) in
    let module HO = Orient (H) in
    .< fun ps p -> .~(Staged.to_code (
         let points = Array.init V.dim
                        (fun i -> of_atom .<ps.(i)>.) in
         let l = H.of_points (Array.to_list points) in
         HO.col l (of_atom .<p>.))) >.
end
```

can generate the collinearity test for a point and a hyperplane in any dimension. In the 1D case, the hyperplane is a point, and the test reduces to (where p is the hyperplane, and x is the test point):

$$\begin{vmatrix} p & 1 \\ x & 1 \end{vmatrix} = 0,$$

or

$$p - x = 0.$$

This equality test can be exact or inexact. The inexact number type `Float_Real_Inexact` uses a tolerance, ϵ, and this test takes the form:

$$-\epsilon \leq p - x \leq \epsilon,$$

or in expanded form:

$$(-\epsilon \leq p - x) \wedge (p - x \leq \epsilon).$$

We instantiate the generator with two different number type implementations. First, we instantiate with the inexact floating-point type:

```
module Float_E6 = Float_Real_Inexact(struct let
                                        eps = 1e-6, .<1e-6>. end)
module GH1' = Gen (Float_E6) (Tuple1D)
```

where the tolerance in the sign calculation set to 10^{-6}. Second, we use the exact floating-point type:

```
module GH1 = Gen (Float_Real_Exact) (Tuple1D)
```

Calling the generator results in the following code for the exact float:

```
    ('a,
     Float.Float_Real_Exact.n Tuple.Tuple1D.t array ->
     Float.Float_Real_Exact.n Tuple.Tuple1D.t -> bool)
    code
= .<fun ps_1 -> fun p_2 -> ((ps_1.(0) +. (~-. p_2)) = 0.)>.
```

which corresponds to testing $p - x = 0$. Whereas, the code for the inexact float is:

```
    ('a,
     Float_E6.n Tuple.Tuple1D.t array ->
     Float_E6.n Tuple.Tuple1D.t -> bool)
    code
=
.<fun ps_1 ->
   fun p_2 ->
    let x_3 = (ps_1.(0) +. (~-. p_2))
    and eps_4 = 1e-6 in
    ((x_3 >= (~-. eps_4)) && (x_3 <= eps_4))>.
```

which correspond to testing $(-\epsilon \leq p - x) \wedge (p - x \leq \epsilon)$.

8.6.3 Example: Generating Orientation Test for 2D

The following instantiation uses the same generator in the example above. The 2D generation follows from the choice of `Record2D`, an implementation of 2-tuples using record data structure.

```
module GH2 = Gen (Float_Real_Exact) (Record2D)
```

Calling the generator results in the following code (type signature omitted):

```
.<fun ps_1 ->
   fun p_2 ->
    ((((ps_1.(0)).c0 *. ((ps_1.(1)).c1 +. (~-. p_2.c1))) +.
       ((~-. ((ps_1.(0)).c1 *.
                ((ps_1.(1)).c0 +. (~-. p_2.c0)))) +.
         (((ps_1.(1)).c0 *. p_2.c1) +.
          (~-. ((ps_1.(1)).c1 *. p_2.c0))))) =
      0.)>.
```

which corresponds to expanding the determinant:

$$\begin{vmatrix} \texttt{(ps\_1.(0)).c0} & \texttt{(ps\_1.(0)).c1} & 1 \\ \texttt{(ps\_1.(1)).c0} & \texttt{(ps\_1.(1)).c1} & 1 \\ \texttt{p\_2.c0} & \texttt{p\_2.c1} & 1 \end{vmatrix} = 0.$$

Chapter 9

Conclusions

This chapter concludes this thesis by summarizing the contributions and discussing the related work. Finally, an outlook on the future work is given.

9.1 Summary of Contributions

The work presented in this thesis contributes to improving the quality of mesh generation software through building a program generator for geometric kernels suitable for mesh generation systems.

Three primary concerns of mesh generation software are managing the complexity of the software, managing the anticipated changes, and improving the quality of the software. Development as a program family is a promising approach to address those concerns. However, using good software engineering practices for implementing a family, such as modularity and abstraction, generally results in run-time overheads. To avoid these overheads, further techniques can be applied at the cost of the understandability and maintainability of the software. This thesis presents a step towards answering the question of the cost of the abstractions in the mesh generation domain without sacrificing understandability and maintainability.

The relative success of generative programming for implementing program families is due to its focus on abstraction, reusability, portability, maintainability, and efficiency. Generative meta-programming, and in particular multi-stage programming, offers an opportunity for building program generators without requiring additional compiler technology. Abstract interpretation – a technique usually applied in static analysis – can be combined with generative programming to eliminate unnecessary computations in the generated code. MetaOCaml offers multi-stage facilities and a static type system which ensures that the generated code is well-typed. Because MetaOCaml is based on OCaml, it enjoys all its features such as polymorphic types,

higher-order functions, and a powerful module system.

In this thesis, we have shown how abstract interpretation, layered design, and MetaOCaml's support for multi-staging can be combined to achieve our goal in writing a parametric, type-safe, abstract and maintainable generator for a geometric kernel. The generated code has minimal traces of the design abstractions and exhibits quality features such as: constant folding and propagation, algebraic simplification, and common sub-expression elimination. The generator is based on abstractions that are natural to the domains of mesh generation and computational geometry. The maintainability and understandability of the generator are not undermined, yet the cost of abstraction is reduced in the generated code.

It should be easy to see that these techniques generalize to other aspects of mesh generation systems.

9.2 Related Work

Previous works on the geometric core of mesh generation and computational geometry have mainly focused on genericity, flexibility, and performance. Qualities such as understandability and maintainability are usually sacrificed to achieve genericity and efficiency.

Simpson [Sim99] presents an attempt to decouple mesh generators from the underlying geometry by using object-oriented programming techniques to dynamically bind computations to different local coordinate representations. However, object-orient programming can introduce run-time overheads.

XYZ GeoBench [Sch91] offers a programming environment for implementing geometric algorithms by relying on object-oriented programming and virtual functions to implement genericity. By using object-oriented programming, geometric algorithms can be implemented in an arithmetic independent way. However, using dynamic binding for achieving flexibility, can result in performance penalties.

LEDA [MN99] is a comprehensive library of data types and algorithms. LEDA makes use of C++ templates to achieve genericity. The library has a layered design that decouples geometric algorithms from number types. The independence from coordinate systems is achieved by having two sets of geometric kernels: one for the Cartesian coordinates, and another for the homogeneous coordinates. This duplication is a challenge to the extensibility and maintainability.

The CGAL [FGK+00] library is written in C++ using generic programming to achieve flexibility, efficiency, and robustness. Adaptability and extensibility are achieved by parameterizing each geometric object by the geometric kernel type and the number type. CGAL relies on C++ template instantiation at compile-time to reduce the performance penalties of the parametrization. However, C++ template

meta-programming provides no guarantee on the correctness of the program generator. Programs written using this technique are difficult to understand, and hence their maintainability is a challenging task.

Outside the context of mesh generation and scientific computing, meta- and generative programming were applied to scientific computing.

Carette et. al. [CK05] provides a highly parametrized generator for a family of Gaussian Elimination algorithms. The use of monads and multi-stage programming allows eliminating all the abstraction overhead while preserving the type-safety of the generated code. Blitz++ [Vel98] uses C++ meta-programming to eliminate the cost of abstractions in vector mathematics.

9.3 Future Work

The results of this thesis work encourage future research in the development of program generators for a family of geometric kernels and mesh generation in general. Our time constraints have limited our scope to implementing a simple version of the generative geometric kernel. Therefore, more work should be done in the future to refine the kernel and expand its scope. The following list presents the prominent future investigations.

1. *Expand the staging layer.* Many refinements can be done in the staging layer. These refinements include:

   (a) Modifying the type `staged` to accommodate more complex types such as staged pairs and records.

   (b) Employing the monadic techniques from [CK05], to improve the generation of control structures, and let-bindings. Monads are promising in handling the generation of non-trivial data types and situations where nesting and side effects are involved.

   (c) Implementing more code optimizations such as loop unrolling and other code transformations from [CDG+06].

2. *Expand the number type layer for a comprehensive set of abstractions.* For example, the current abstractions do not differentiate between concepts such as division rings and commutative rings. Such a finer set of abstractions will allow expressing the geometric algorithms in terms as close as possible to the domain concepts and hence, reduce the gap between program code and domain concepts.

3. *Experiment with different geometric algebras.* Chapter 7 discussed the reasons behind choosing affine geometry. Other kinds of geometric algebras should be considered for experimentation. For example, the algebra in [FD03] is promising for handling different coordinate systems.

Bibliography

[ABM09] Ritu Arora, Purushotham Bangalore, and Marjan Mernik. Developing scientific applications using generative programming. In *SECSE '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 51–58, Washington, DC, USA, 2009. IEEE Computer Society.

[AHMK01] Krister Ahlander, Magne Haveraaen, and Hans Z. Munthe-Kaas. On the role of mathematical abstractions for scientific computing. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 145–158, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[Art91] Michael Artin. *Algebra.* Prentice-Hall, New Jersy, USA, 1991.

[Aud03] Michele Audin. *Geometry.* Springer-Verlag, Inc., New York, NY, USA, 2003.

[Ber00] Guntram Berti. Generic components for grid data structures and algorithms with C++. In *First Workshop on C++ Template Programming*, 2000.

[BHK06] María Cecilia Bastarrica and Nancy Hitschfeld-Kahler. Designing a product family of meshing tools. *Adv. Eng. Softw.*, 37(1):1–10, 2006.

[Bli02] Charles Blilie. Patterns in scientific software: An introduction. *Computing in Science and Engg.*, 4(3):48–53, 2002.

[Bou] Paul Bourke. Equation of a sphere from 4 points on the surface. http://local.wasp.uwa.edu.au/~pbourke/geometry/spherefrom4/.

[Bru97] D. Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In *Proceedings of the ACM SIGPLAN Workshop on Domain Specific Languages*, pages 17–35, Paris, France, January 1997.

[Cao06] Fang Cao. A program family approach to developing mesh generators. Master's thesis, McMaster University, Hamilton, Ontario, Canada, 2006.

[Car06] Jacques Carette. Gaussian elimination: a case study in efficient genericity with MetaOCaml. *Sci. Comput. Program.*, 62(1):3–24, 2006.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[cca] The Common Component Architecture Forum. http://www.cca-forum.org/.

[CDG+06] Albert Cohen, Sébastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kiselyov, and David Padua. In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.*, 62(1):25–46, 2006.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CEG+00] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Gluck, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, London, UK, 2000. Springer-Verlag.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Softw.*, 15(6):37–45, 1998.

[Cic05] Trevor Cickovski. Design patterns for generic object-oriented scientific software. In *ICSE05, Twenty-Seventh International Conference on Software Engineering*, 2005.

[CK05] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In Robert Glck and Michael Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 256–274. Springer Berlin / Heidelberg, 2005.

[CK08] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*, In Press, Corrected Proof, 2008.

[Cro96] Thomas W. Crockett. Beyond the renderer: Software architecture for parallel graphics and visualization. Technical report, 1996.

[Daw] B. Dawes. Home page of C++ Boost. http://www.boost.org/.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1st edition, 1997.

[DeR89] T. D. DeRose. A coordinate-free approach to geometric programming. *Theory and practice of geometric modeling*, pages 291–305, 1989.

[EFP07] Martin Erwig, Zhe Fu, and Ben Pflaum. Parametric FORTRAN: program generation in scientific computing. *J. Softw. Maint. Evol.*, 19:155–182, May 2007.

[Eis97] U. W. Eisenecker. Generative programming (GP) with C++. *Lecture Notes in Computer Science*, 1204:351–365, 1997.

[ESC04] A. H. ElSheikh, S. Smith, and S. E. Chidiac. Semi-formal design of reliable mesh generation systems. *Adv. Eng. Softw.*, 35(12):827–841, 2004.

[FD03] Daniel Fontijne and Leo Dorst. Modeling 3D Euclidean geometry. *IEEE Comput. Graph. Appl.*, 23(2):68–78, 2003.

[FFT] FFTW Home Page. http://www.fftw.org/benchfft/ffts.html.

[FGK+00] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. Pract. Exper.*, 30(11):1167–1202, 2000.

[FSPL08] Joel Falcou, Jocelyn Sérot, Lucien Pech, and Jean-Thierry Lapresté. Meta-programming applied to automatic SMP parallelization of linear algebra code. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 729–738, Berlin, Heidelberg, 2008. Springer-Verlag.

[Gal00] Jean Gallier. *Geometric methods and applications: for computer science and engineering.* Springer-Verlag, London, UK, 2000.

[Gar04] Henry Gardner. Design patterns in scientific software. In *Computational Science and Its Applications - ICCSA 2004*, pages 776–785. 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GHW00] Philip W. Grant, Magne Haveraaen, and Michael F. Webster. Coordinate free programming of computational fluid dynamics problems. *Sci. Program.*, 8(4):211–230, 2000.

[GK03] J. Gerlach and J. Kneis. Generic programming for scientific computing in C++, Java, and C#. *Lecture Notes in Computer Science*, 2834:301–310, 2003.

[Gol02] Ron Goldman. On the algebraic and geometric foundations of computer graphics. *ACM Trans. Graph.*, 21(1):52–86, 2002.

[Hea02] Michael T. Heath. *Scientific Computing: an Introductory Survey.* McGraw-Hill, New York, NY, USA, 2002.

[HG04] Bruno Harbulot and John R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 122–131, New York, NY, USA, 2004. ACM.

[HLC+06] N. Hitschfeld, C. Lillo, A. Caceres, M. Bastarrica, and M. Rivara. Building a 3D meshing framework using good software engineering practices. In Sergio Ochoa and Gruia-Catalin Roman, editors, *Advanced Software Engineering: Expanding the Frontiers of Software Technology*, volume 219 of *IFIP International Federation for Information Processing*, pages 162–170. Springer Boston, 2006.

[ILG+97] John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, pages 249–256, London, UK, 1997. Springer-Verlag.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report July 31, 1978.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Kan05] Ronald Kirk Kandt. *Software Engineering Quality Practices (Applied Software Engineering).* Auerbach Publications, Boston, MA, USA, 2005.

[KG07] Chanwit Kaewkasi and John R. Gurd. A distributed dynamic aspect machine for scientific software development. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 3, New York, NY, USA, 2007. ACM.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP) , Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1997.

[KMB+96] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *ICSE '96: Proceedings of the 18th international conference on Software Engineering*, pages 542–552, Washington, DC, USA, 1996. IEEE Computer Society.

[KS08] Dianne Kelly and Rebecca Sanders. Assessing the quality of scientific software. 2008.

[KST04] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 249–258, New York, NY, USA, 2004. ACM.

[LL97] Nam-Yong Lee and Charles R. Litecky. An empirical study of software reuse with special attention to Ada. *IEEE Trans. Softw. Eng.*, 23(9):537–549, 1997.

[LL02] Lie-Quan Lee and Andrew Lumsdaine. Generic programming for high performance scientific applications. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 112–121, New York, NY, USA, 2002. ACM.

[LS79] Michael E. Lesk and Eric Schmidt. Lex A Lexical Analyzer Generator. In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report in 1975.

[McC07] J. McCutchan. A generative approach to a virtual material testing laboratory. Master's thesis, McMaster University, Hamilton, Ontario, Canada, 2007.

[McI69] Doug McIlroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Proceedings of Software Engineering Concepts and Techniques*, pages 138–155. NATO Science Committee, January 1969.

[MET02] Maurizio Morisio, Michel Ezran, and Colin Tully. Success and failure factors in software reuse. *IEEE Trans. Software Eng.*, 28(4):340–357, 2002.

[Mey03] Bertrand Meyer. The grand challenge of trusted components. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 660–667, Washington, DC, USA, 2003. IEEE Computer Society.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[MMN+97] Kurt Mehlhorn, Michael Müller, Stefan Näher, Stefan Schirra, Michael Seel, Christian Uhrig, and Joachim Ziegler. A computational basis for higher-dimensional computational geometry and applications. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 254–263, New York, NY, USA, 1997. ACM.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: a platform for combinatorial and geometric computing.* Cambridge University Press, New York, NY, USA, 1999.

[moc] MetaOCaml Home Page. http://www.metaocaml.org/.

[MP99] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *DSL'99: Proceedings of the 2nd conference on Conference on Domain-Specific Languages*, pages 5–15, Berkeley, CA, USA, 1999. USENIX Association.

[MS89] D. R. Musser and A. A. Stepanov. Generic programming. In P. (Patrizia) Gianni, editor, *Symbolic and algebraic computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, pub-SV:adr, 1989. Springer Verlag.

[MY01] Peter McGavin and Roger Young. A generic list implementation. *SIGPLAN Fortran Forum*, 20(1):16–20, 2001.

[Obj] Objective Caml. http://caml.inria.fr/ocaml/.

[OS06] Suely Oliveira and David E. Stewart. *Writing Scientific Software: A Guide to Good Style.* Cambridge University Press, New York, NY, USA, 2006.

[Owe98] Steven J. Owen. A survey of unstructured mesh generation technology. In *Proceeding of the 7th International Meshing Roundtable*, pages 239–267, 1998.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

[RGZ+09] Suman Roychoudhury, Jeff Gray, Jing Zhang, Purushotham Bangalore, and Anthony Skjellum. Modularizing scientific libraries with aspect-oriented and generative programming techniques. *Acta Electrotechnica et Informatica*, 9(3):16–23, 2009.

[Sam97] Johannes Sametinger. *Software engineering with reusable components.* Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[SC04] S. Smith and C. H. Chen. Commonality analysis for mesh generating systems. Technical Report CAS-04-10-SS, McMaster University, 2004.

[Sch91] Peter Schorn. Implementing the XYZ GeoBench: A programming environment for geometric algorithms. In *CG '91: Proceedings of the International Workshop on Computational Geometry =- Methods, Algorithms and Applications*, pages 187–202, London, UK, 1991. Springer-Verlag.

[SCM08] Spencer Smith, Jacques Carette, and John McCutchan. Commonality analysis of families of physical models for use in scientific computing. In *Proceedings of SECSE08 conference*, 2008.

[SE02] Philip J. Schneider and David Eberly. *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[Seg07] Judith Segal. Some problems of professional end user developers. In *VLHCC '07: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 111–118, Washington, DC, USA, 2007. IEEE Computer Society.

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[Sim99] R. Bruce Simpson. Isolating geometry in mesh programming. In *Proc. of the 8th Int'l Meshing Roundtable*, pages 45–54, South Lake Tahoe, California, October 1999.

[SL95] Alexander Stepanov and Meng Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1995.

[SL98] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 59–70, London, UK, 1998. Springer-Verlag.

[SMC07] S. Smith, J. McCutchan, and F. Cao. Program families in scientific computing. In J. Sprinkle, J. Gray, M. Rossi, and J.-P. Tolvanen, editors,

7th OOPSLA Workshop on Domain Specific Modelling, pages 39–47, Montreal, Quebec, 2007.

[Som04] Ian Sommerville. *Software Engineering*. Addison-Wesley, 7th edition, May 2004.

[Ste66] P. Stein. A note on the volume of a simplex. *The American Mathematical Monthly*, 73(3):299–301, 1966.

[SY09] S. Smith and W. Yu. A document driven methodology for developing a high quality parallel mesh generation toolbox. *Adv. Eng. Softw.*, 40(11):1155–1167, 2009.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Tah04] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, pages 30–50. Springer-Verlag, 2004.

[TN03] Walid Taha and Michael Florentin Nielsen. Environment classifiers. *SIGPLAN Not.*, 38(1):26–37, 2003.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[TSW99] Joe F. Thompson, Bharat K. Soni, and Nigel P. Weatherill, editors. *Handbook of grid generation*. CRC Press, Boca Raton, FL, 1999.

[TW00] Shang-Hua Teng and Chi Wai Wong. Unstructured mesh generation: Theory, practice, and perspectives. *Int. J. Computational Geometry and Applications*, 10(3):227–266, Jun 2000.

[vDK98] Arie van Deursen and Paul Klint. Little languages: little maintenance. *Journal of Software Maintenance*, 10(2):75–92, 1998.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[Vel96] Todd Veldhuizen. *Expression templates*, pages 475–487. SIGS Publications, Inc., New York, NY, USA, 1996.

[Vel98] Todd L. Veldhuizen. Arrays in Blitz++. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230, London, UK, 1998. Springer-Verlag.

[Wei98] David M. Weiss. Commonality analysis: A systematic process for defining families. In *Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, pages 214–222, London, UK, 1998. Springer-Verlag.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[WPD01] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

Appendix A

Reference Guide

A.1 The Multi-staging Layer

A.1.1 Staged Types and Operators

Types

`code_expr` A code expression `c` with its atomicity flag `a`.

`staged` Staged expression.

`Now` Type constructor for Now staged expression.

`Later` Type constructor for Later staged expressions.

`unary` Generalized unary function.

`binary` Generalized binary function.

`monoid` A binary function and a unit element.

`ring` A monoid and a zero element.

Functions

`of_immediate` Lifts an immediate expression into a Now staged expression.

`of_atom` Lifts a code expression into a Later staged expression with atomic flag = true.

`of_comp` Lifts a code expression into a Later staged expression with atomic flag = false.

`to_later` Typecasts a staged expression into a Later expression.

`to_code` Typecasts a staged expression into a MetaOCaml's `code` value.

`mk_unary` Builds a unary staged operator from a generalized unary function.

`mk_binary` Builds a binary staged operator from a generalized binary function.

`mk_monoid` Builds a monoid staged operator from a monoid.

`mk_ring` Builds a ring staged operator from a ring.

A.1.2 The Module `Int`

Types

`t` Staged integer type.

Constants

`zero` The constant zero.

Functions

`random_n` Generates an immediate random integer. Input is an immediate expression for the seed.

`random_c` Generates a code expression for generating a random integer. Input is a code expression for the seed.

`random` Takes a staged integer seed and returns a random staged integer.

`succ` The successor function over staged integers.

A.1.3 The Module `String`

Types

`t` Staged string.

Functions

`concat_b` Concatenation monoid. Empty string "" is the unit element.

`concat_s` Concatenates two staged strings.

A.1.4 The Module `Bool`

Types

`b` Staged boolean.

Constants

`false_` False.

`true_` True.

Functions

`not_b` Generalized unary function for logical negation.

`not_s` Negation of staged boolean.

`and_b` Conjunction ring. False is ring's zero and true is ring's one.

`and_s` Conjunction of two staged boolean expressions.

`or_b` Disjunction ring. True is ring's zero and false is ring's one.

`or_s` Disjunction of two staged boolean expressions.

`eq_s` Equality test for two staged expressions.

A.1.5 Staging Code Constructs

`ife c a b` Abstraction of the if-else construct: `if c then a else b`.

`let_ ce exp` Abstraction of the let construct: `let t = ce in (expt t)`.

A.2 Number Types

A.2.1 The Module Type `SET`

Types

`n` Base number type.

`ns` Staged number type.

Functions

`eq_b` Equality test for two numbers of type `n`.

`eq_s` Equality test for two staged numbers.

`eq_tol` Equality test with tolerance for two staged numbers.

`neq_b` Inequality test for two numbers of type `n`.

`neq_s` Inequality test for two staged numbers.

`to_string_b` Returns the string representing a number of type `n`.

`to_string_s` Returns the staged string representing a staged number.

A.2.2 The Module `Sign`

Types

`t` Type of sign.

Functions

`pos s` True if `s` is a positive sign.

`neg s` True if `s` is a negative sign.

`zero s` True if `s` is a sign marker for zero.

`bind` `bind a b c d e` builds a function that takes a sign `s` and returns `a` (resp. `b`, `c`, `d`, `e`) if the sign is positive (resp. zero, negative, positive or zero, negative or zero).

A.2.3 The Module type `ORDER`

Types

`t` Base type of elements of the carrier set.

`t_s` Staged type of elements of the carrier set.

Constants

`bot` Bottom element of the order.

`top` Top element of the order.

Functions

`eq` Equality test for two staged elements.

`neq` Inequality test for two staged elements.

`compare_b` Generalized binary function. `compare_b x y` returns 0 if `x = y`, 1 if `x > y` and `-1` if `x < y`.

`lt_b` Generalized binary function. `lt_b x y` returns `true` if `x` $<$ `y` according to the order relation.

`le_b` Generalized binary function. `le_b x y` returns `true` if `x` $\leq$ `y` according to the order relation.

`gt_b` Generalized binary function. `gt_b x y` returns `true` if `x` $>$ `y` according to the order relation.

`ge_b` Generalized binary function. `ge_b x y` returns `true` if `x` $\geq$ `y` according to the order relation.

`min_b` Generalized binary function. `min_b x y` return the minimum of `x` and `y` according to the order relation.

`max_b` Generalized binary function. `max_b x y` return the maximum of `x` and `y` according to the order relation.

The functions `compare_s`, `lt_s`, `le_s`, `gt_s`, `ge_s`, `min_s`, and `max_s` are the staged versions of their `_b` counterparts.

A.2.4 The Module Type `RING`

Constants

`zero` Zero of the ring.

`one` One of the ring.

`negone` The constant 'negative one'.

`two` The constant 'two'.

Functions

`add_b`	Monoid operator for addition.
`sub_b`	Generalized binary operator for subtraction.
`mul_b`	Ring operator for multiplication.
`neg_b`	Generalized unary operator for negation.
`sgn`	`sgn n s` returns true if the staged number `n` has the sign `s`.
`abs_b`	Generalized unary operator for absolute value.
`pow`	`pow x y` returns x^y where both `x` and `y` are staged numbers.
`int_pow`	`int_pow n x` returns x^n where `x` is a staged number and `n` is an integer.

The functions `add_s`, `sub_s`, `mul_s`, `neg_s`, and `abs_s` are the staged versions of their `_b` counterparts.

A.2.5 The Module Type `FIELD`

Constants

`pi`	The constant π.

Functions

`inv_b`	Generalized unary function. `inv_b x` returns $\frac{1}{x}$.
`div_b`	Generalized binary function. `div_b x y` returns $\frac{x}{y}$.

The functions `inv_s`, and `div_s` are the staged versions of their `_b` counterparts.

A.2.6 The Module Type `REAL`

Constants

`bot`	Bottom element of the type.
`top`	Top element of the type.

Functions

`sqrt_s`	Square root of a staged numbers.
`eq`	Equality test for two staged numbers.
`neq`	Inequality test for two staged numbers.
`compare_b`	Generalized binary function. `compare_b x y` returns `0` if `x = y`, `1` if `x > y` and `-1` if `x < y`.
`lt_b`	Generalized binary function. `lt_b x y` returns `true` if $x < y$.
`le_b`	Generalized binary function. `le_b x y` returns `true` if $x \leq y$.
`gt_b`	Generalized binary function. `gt_b x y` returns `true` if $x > y$.
`ge_b`	Generalized binary function. `ge_b x y` returns `true` if $x \geq y$.
`min_b`	Generalized binary function. `min_b x y` return the minimum of `x` and `y`.
`max_b`	Generalized binary function. `max_b x y` return the maximum of `x` and `y`.

The functions `compare_s`, `lt_s`, `le_s`, `gt_s`, `ge_s`, `min_s`, and `max_s` are the staged versions of their `_b` counterparts.

A.3 Linear Algebra

A.3.1 The Module Type TUPLE

Types

`t`	Tuple type.

Constants

`dim`	Number of elements in the tuple.

Functions

`init` `init n f` creates a tuple ⟨`f 0, f 1, f 2,... , f (n-1)`⟩.

`proj_n` `proj_n x i` returns the i^{th} element of `x`. `x` is of type `t`.

`proj_c` `proj_c x i` returns the i^{th} element of `x`. `x` is of type `t code`.

`of_list_n` Creates a tuple from a list of values.

`of_list_c` Creates a code expression for a tuple from a list of code values.

`to_list_n` Converts a tuple into a list of values.

`to_list_c` Converts a code expression of a tuple into a code expression of list of values.

`map_n` `map_n f t` maps `t` element-wise by `f`. `f` and `t` are immediate values.

`map_c` `map_n f t` maps `t` element-wise by `f`. `f` and `t` are code values.

`map2_n` `map_n f t t'` fuses `t` and `t'` element-wise by `f`. `f`, `t`, and `t'` are immediate values.

`map2_c` `map_n f t t'` fuses `t` and `t'` element-wise by `f`. `f`, `t`, and `t'` are code values.

`mapi_n` `mapi_n f t` maps `t` element-wise to ⟨`f 0 (proj_n t 0), f (proj_n t 1), ...`⟩. `f` and `t` are both immediate values.

`mapi_c` `mapi_n f t` maps `t` element-wise to ⟨`f 0 (proj_c t 0), f (proj_c t 1), ...`⟩. `f` and `t` are both code values.

`fold_n` `fold_n f z t` folds the tuple elements by `f`. `z` is an initial value.

`fold_c` The code version of `fold_n`.

`mapfold_n` `mapfold_n m f z t` is equivalent to `fold_n f z (map_n m t)`.

`mapfold_c` `mapfold_c m f z t` is equivalent to `fold_c f z (map_c m t)`.

`map2fold_n` `map2fold_n m f z t t'` is equivalent to `fold_n f z (map2_n m t t')`.

`map2fold_c` `map2fold_c m f z t t'` is equivalent to `fold_c f z (map2_c m t t')`.

A.3.2 The Module Type MATRIX

Types

`n_s` Types of the matrix entries.

`m` The matrix type.

Functions

`nrows` `nrows m` returns the number of rows of `m`.

`ncols` `ncols m` returns the number of columns of `m`.

`dim` `dim m` returns (`nrows m, ncols m`).

`create` `create r c f` creates a matrix `r` by `c`, where the elements of `m` are given by: `m(i,j) = f i j`.

`create_row` `create_row n f` creates a matrix `n` by `1`, where the elements of `m` are given by: `m(i,j) = f i j`.

`create_col` `create_col n f` creates a matrix `1` by `n`, where the elements of `m` are given by: `m(i,j) = f i j`.

`get` `get m i j` returns the element indexed (`i`, `j`).

`zero` `zero r c` creates a zero matrix `r` by `c`.

`diag` `diag n f` creates a diagonal matrix where diagonal elements are given by `(f 0)`, `(f 1)`, etc.

`id` `id n` creates an identity matrix of size `n`.

`transpose` Transpose of a matrix.

`haugment` Horizontal augment of two matrices.

`vaugment` Vertical augment of two matrices.

`map` `map f m` maps the matrix `m` element-wise by `f`.

`map2` `map2 f m m'` fuses `m` and `m'` element-wise by `f`.

`add` Matrix addition.

`sub` Matrix subtraction.

`mul` Matrix multiplication.

`sadd` `sadd m s` adds the scalar `s` to every element in `m`.

`ssub` `ssub m s` subtracts the scalar `s` from every element in `m`.

`smul` `smul m s` multiplies the scalar `s` by every element in `m`.

`sdiv` `sdiv m s` divides every element in `m` by the scalar `s`.

`minor` `minor m i j` return the minor of `m` resulting from removing row `i` and column `j`.

A.3.3 The Module Type DETERMINANT

Constants

`N` Module of type `FIELD`.

`M` Module of type `MATRIX`.

Functions

`eval` `eval m` evaluates the determinant of `m`. `m` is a matrix of type `M.m`, with entries of type `N.ns`. The result of `eval` is a staged expression.

A.4 Affine Space

A.4.1 The Module Type VECTOR

Types

`vector` Base type of vectors. Used at run-time.

`vector_s` Staged vectors. Used at generation-time.

Constants

`N` Type of the coordinates. `N` is a module of type `REAL`.

`dim` Dimension of a vector.

Functions

`zero`	Returns a staged zero vector.
`of_coords`	Creates a vector from a list of coordinates of type `N.ns`.
`coord`	Returns the nth coordinate of a vector.
`eq`	Tests equality of two staged vectors.
`neq`	Tests inequality of two staged vectors.
`mirror`	Mirrors a staged vector.
`add`	Adds two staged vectors.
`sub`	Subtracts two staged vectors.
`dot`	Returns the dot product of two staged vectors.
`bcross`	Returns the cross product of two staged vectors.
`gcross`	Returns the generalized cross product of a list of staged vectors.
`scale`	Scales a staged vector by a scalar of type `N.ns`.
`shrink`	Shrinks a staged vector by a scalar of type `N.ns`.
`length`	Returns the length of a staged vector.
`length2`	Returns the squared length of a staged vector.

`direction` `direction v` returns the unit vector in the direction of `v`.

A.4.2 The Module Type `POINT`

Types

`point`	Base type of points. Used at run-time.
`point_s`	Staged points. Used at generation-time.

Constants

`N`	Type of the coordinates. `N` is a module of type `REAL`.
`V`	Type of the position vector. `V` is a module of type `VECTOR`.
`dim`	Dimension of a point.

Functions

`to_code`	Converts a staged point into a code expression of type `('a, point) code`.
`of_code`	Converts a point code expression into a staged point.
`of_list`	Creates a staged point from a list of coordinates of type `N.ns`.
`to_list`	Returns a list containing the coordinates of a staged point.
`eq`	Tests the equality of two staged points.
`neq`	Tests the inequality of two staged points.
`eq_tol`	Tests the equality, with a tolerance, of two staged points.
`orig`	Returns a zero point (the origin).
`coord`	Returns the ith coordinate of a point.
`pos_vec`	Returns the position vector a point. The vector is of type `V.vector_s`.
`add`	`add p v` returns the point `(p+v)`.
`sub`	`sub a b` returns the vector `(b-a)`.
`to_string`	Returns the string representing a point.

A.4.3 The Module Type `ORDERED_POINT`

Constants

`bot`	Bottom point.
`top`	Top point.

Functions

`lt`	Less than.
`gt`	Greater than.
`le`	Less than or equal.
`ge`	Greater than or equal.
`min`	Minimum.
`max`	Maximum.

A.4.4 The Module type `ISO_AXIS_ORDERED_POINT`

Constants

`bot`	Bottom point.
`top`	Top point.

Functions

`lt`	`lt i a b` is the $<$ relation on the `i`th coordinate of `a` and `b`.
`gt`	`gt i a b` is the $>$ relation on the `i`th coordinate of `a` and `b`.
`le`	`le i a b` is the $\leq$ relation on the `i`th coordinate of `a` and `b`.
`ge`	`ge i a b` is the $\geq$ relation on the `i`th coordinate of `a` and `b`.
`min`	`min i a b` returns the minimum of `a` and `b` with respect to the `i`th coordinate.
`max`	`max i a b` returns the maximum of `a` and `b` with respect to the `i`th coordinate.

A.4.5 The Module Type `AFFINE`

Types

`t`	Type of the affine transform.

Constants

`N` Number module.

`P` Point module.

`V` Vector module.

`M` Matrix module for transformation matrix.

Functions

`apply_p` Applies a transformation to a point.

`apply_v` Applies a transformation to a vector.

`compose` Composes two transformations.

`id` Identity transformation.

`translation` Creates a translation transform from a given vector.

`scaling` Creates a scaling transform. Scaling parameters are given by an input tuple.

A.4.6 The Module `Orientation`

Types

`t` Type of orientations.

Functions

`ccw` `ccw x` returns true if the orientation `x` is a counter-clockwise orientation.

`cw` `cw x` returns true if the orientation `x` is a clockwise orientation.

`col` `col x` returns true if the orientation `x` is a collinear orientation.

`ccw_or_col` `ccw x` returns true if the orientation `x` is a counter-clockwise or a collinear orientation.

`cw_or_col` `cw x` returns true if the orientation `x` is a clockwise or a collinear orientation.

`of_sign` Returns the orientation equivalent to a given sign.

`of_sign`	Returns the sign equivalent to a given orientation.
`bind`	`bind a b c d e` builds a function that takes a orientation `x` and returns `a` (resp. `b`, `c`, `d`, `e`) if `x` is `ccw` (resp. `col`, `cw`, `ccw_or_col`, `cw_or_col`).

A.4.7 The Module `Side`

Types

`t`	Type of sideness.

Functions

`in_`	`in_ x` returns true if the sideness `x` is 'in'.
`on`	`on x` returns true if the sideness `x` is 'on'.
`out`	`out x` returns true if the sideness `x` is 'out'.
`in_or_on`	`in_or_on x` is equivalent to `in_ x` $\vee$ `on x`.
`out_or_on`	`out_or_on x` is equivalent to `out x` $\vee$ `on x`..
`of_sign`	Returns the sideness equivalent to a given sign.
`of_sign`	Returns the sign equivalent to a given sideness.
`bind`	`bind a b c d e` builds a function that takes a sideness `x` and returns `a` (resp. `b`, `c`, `d`, `e`) if `x` is `in_` (resp. `on`, `out`, `in_or_on`, `out_or_on`).

A.5 Geometric Objects

A.5.1 The Module Type `HYPER_PLANE`

Types

`t_s`	Type of hyperplanes.

Constants

`N`	Number module.
`V`	Vector module.
`P`	Point module.

Functions

`dim`	Dimension of a hyperplane.
`of_points`	Constructs a hyperplane from a list of points.
`point`	Returns the ith point of a hyperplane.
`points`	Returns the list of points used to construct the hyperplane.
`poly`	Returns the polynomial of a hyperplane.
`normal`	Returns the normal of a hyperplane.
`orig`	Returns the origin of the local frame of a hyperplane.
`basis`	Returns the ith basis of the local frame of a hyperplane.
`bases`	Returns the list of bases of the local frame of a hyperplane..
`frame`	Returns the local frame of a hyperplane..
`coord`	Returns the local coordinates of a given point on a hyperplane.
`pos_vec`	Returns the position vector of a given point on a hyperplane.

A.5.2 The Module `Hplane_Operations`

Inputs

`H`	A module of type `HYPER_PLANE`.

Functions

`dist`	`dist h p` returns the distance between a point `p` and a hyperplane `h`.
`project`	`project h p` returns the projection of a point `p` on the hyperplane `h`.

A.5.3 The Module Type `HSPHERE`

Types

`sphere_s`	Type of spheres.

Constants

`N` Number module.

`P` Point module.

Functions

`dim` Dimension of a sphere.

`of_points` Constructs a sphere from a list of points.

`point` Returns the ith point of a sphere.

`points` Returns the list of points used to construct the sphere.

`of_centre_radius` Constructs a sphere from a centre point and a radius.

A.5.4 The Module `Sphere_Operations`

Inputs

`N` A number type module.

`H` A sphere module.

Functions

`centre` Computes the centre of a sphere.

`radius` Computes the radius of a sphere.

`radius2` Squared radius.

`content` Returns the hyper-volume of a sphere.

`surface` Returns the hyper-surface area of a sphere.

A.5.5 The Module Type `VERTEX`

Types

`vertex_s` Type of vertices.

`simplex_s` Type of incident simplices.

Constants

`P`	Point type module.

Functions

`dim`	Dimension of a vertex.
`eq`	Tests equality of two vertices.
`iso`	Tests isomorphism between two vertices.
`create`	Creates a vertex from a geometric point.
`point`	Returns the geometric location of a vertex.
`incident`	Returns the simplex incident to a vertex.
`attach`	Sets a simplex to be incident to a vertex.

A.5.6 The Module Type `SIMPLEX`

Types

`simplex_s`	Type of simplices.

Constants

`V`	Vertex type module.

Functions

`dim`	Dimension of a simplex.
`of_vertices`	Creates a simplex from a list of vertices.
`eq`	Tests equality of two simplices.
`vertex`	Returns the ith vertex of a simplex.
`vertices`	Returns the list of vertices in a simplex.
`is_vertex`	`is_vertex s v` returns true if `v` is a vertex of `s`.
`v_index`	`v_index s v` returns the index of vertex `v` in `s`.

`face` Returns the ith face of a simplex.

`faces` Returns the list of faces in a simplex.

`oface` Returns the ith oriented face of a simplex.

`ofaces` Returns the list of oriented faces of a simplex.

`is_face` `is_face s f` returns true if `v` is a face of `s`.

`f_index` `f_index s f` returns the index of face `f` in `s`.

`is_neighbor` `is_neighbor s n` returns true if `n` is a neighbor to `s`.

`n_index` `n_index s n` returns the index of neighbor `n` in `s`.

A.5.7 The Module `Orient`

Inputs

`H` Type of hyperplanes.

Functions

`orient` `orient h p` returns the orientation of point `p` with regard to hyperplane `h`.

`col` Tests if a point is collinear with a hyperplane.

`pos,ccw,below` Tests if a point is below a hyperplane.

`neg,cw,above` Tests if a point is above a hyperplane.

`pos_or_col,ccw_or_col,below_or_on` Tests if a point is below or collinear with a hyperplane.

`neg_or_col,cw_or_col,above_or_on` Tests if a point is above or collinear with a hyperplane.

A.5.8 The Module `Insphere`

Inputs

`S` Type of hypersphere.

Functions

`inside`	`inside s p` checks if the point `p` is inside the sphere `s`.
`in_`	`in_ s p` returns true if the point `p` is inside the sphere `s`.
`on`	`on s p` returns true if the point `p` is on the sphere `s`.
`out`	`out s p` returns true if the point `p` is outside the sphere `s`.
`in_or_on`	`in_or_on s p` returns true if the point `p` is inside or on the sphere `s`.
`out_or_on`	`out_or_on s p` returns true if the point `p` is outside or on the sphere `s`.

A.5.9 The Module `Inside`

Inputs

`H`	Type of hyperplanes.
`S`	Type of simplices.

Types

`point_inside_simplex` Type of the sideness test results.

Functions

`in_`	`in_ s v` returns true if the vertex `v` is inside the simplex `s`.
`on`	`on s v` returns true if the vertex `v` is on the simplex `s`.
`out`	`out s v` returns true if the vertex `v` is outside the simplex `s`.
`in_or_on`	`in_or_on s v` returns true if the vertex `v` is inside or on the simplex `s`.