

User-configurable, high-level transformations with CodeBoost

Karl Trygve Kalleberg

Cand. Scient. Thesis
Department of Informatics,
University of Bergen

2003

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Background	2
1.3	Contents	2
1.3.1	Organization	3
1.4	Remarks	4
2	Toolkits and libraries	5
2.1	Sophus	5
2.1.1	Higher-level language constructs	6
2.1.2	Coordinate-free numerics	7
2.2	Object-oriented numerics	8
2.2.1	A typical Sophus application	10
2.3	Hierarchical data structures	11
2.4	Stratego	11
2.4.1	Terms, rules and strategies	12
2.4.2	Modularisation and pipeline	15
2.5	Auxiliary tools	15
2.5.1	Syntax-aware editor	15
2.5.2	Version control	16
3	CodeBoost	17
3.1	Background	17
3.2	Transformation pipeline	18
3.2.1	Parsing	18
3.2.2	Semantic analysis	19
3.2.3	Transformation	19
3.2.4	Pretty-printing	20
3.3	Some transformation examples	21
3.3.1	AST dump	21
3.3.2	Simple mutification	22
3.4	Framework	24
3.4.1	Configuration files	25

3.5	Totems and totem propagation	27
3.6	Shortcomings	29
3.7	Other transformation systems	29
3.7.1	OpenC++	31
3.7.2	TAMPR	31
3.7.3	ANTLR	32
3.7.4	XTC	32
3.7.5	ASF+SDF	32
3.7.6	ELAN	33
3.7.7	Blitz++	34
3.7.8	Simplicissimus	34
3.8	Summary	36
4	Transformations	37
4.1	Augmented libraries	37
4.1.1	Source-to-source transformations	38
4.1.2	Program traits	39
4.1.3	Domain-specific optimisations	39
4.2	Inspirational paradigms	40
4.2.1	Partial evaluation and program specialisation	40
4.2.2	Generative programming	41
4.3	Algebraic simplification	42
4.4	Shortcomings	44
4.5	Summary	45
5	TigerBoost	47
5.1	Motivation	47
5.2	Transformations	47
5.2.1	Constant propagation and folding	48
5.2.2	Elementary totem propagation	48
5.2.3	Loop unrolling	49
5.2.4	Optimising matrix multiplication	50
5.3	The relation to CodeBoost	53
5.4	Planned obsolescence	54
5.5	Summary	55
6	Symbolic differentiation	57
6.1	Motivation	57
6.2	Overview	57
6.3	Implementation	58
6.3.1	Function constraints	59
6.3.2	Collect diff operator applications	60
6.3.3	Collect function bodies	61
6.3.4	Evaluate and inline result	61

6.3.5	Differentiate expression	61
6.3.6	Repeat cycle	63
6.3.7	Emit code	63
6.4	Shortcomings	64
6.5	Application	64
6.6	Summary	65
7	Mold-L, a matrix description language	67
7.1	Motivation	67
7.2	Syntax	69
7.3	Semantics	69
7.3.1	Operations	70
7.4	Writing layouts	71
7.5	Shortcomings	71
7.6	Summary	72
8	Matrix transformations	73
8.1	Motivation	73
8.2	Overview	74
8.3	Implementation	74
8.3.1	Matrix layouts	75
8.4	Compile-time evaluation	77
8.4.1	Integration	81
8.5	Shortcomings	81
8.6	Application	82
8.7	Summary	83
9	Further work	85
9.1	Extending Mold-L	85
9.2	Further augmenting the Sophus library	86
9.3	Robust error handling	87
9.4	Improving CodeBoost running times	88
9.5	Documenting and promoting CodeBoost	89
9.5.1	Release early, release often	89
9.5.2	Central hub for development	90
9.6	Summary	90
10	Conclusion	93
A	Differentiation rules	97

B	Installing CodeBoost	99
B.1	Downloading	99
B.2	Compiling	99
B.3	Testing	100
B.4	Summary	101

Preface

As with most scientific work, not all the initially enticing ideas pan out. In retrospect, I probably spent too much time on:

The CodeBoost configuration language; the CodeBoost configuration language, described in Section 3.4.1 has been through a few incarnations, and each of them had its own Emacs mode and parsing utilities, which naturally took a bit of time to craft.

The totem mechanism; apart from the totem experiments done in TigerBoost (see Chapter 5), I tried out a few different syntaxes for the C++ language before I finally settled on the very simple explained in this thesis.

Porting CodeBoost; the research group wanted CodeBoost to run on the supercomputers where we run the Sophus programs. This turned out to be a very tricky undertaking. Stratego generates GNU C-specific code, so we were locked to the GNU C compiler. On the MIPS 64bit architecture, where we run all our Sophus programs, the GNU C backend is not entirely stable. After a few bug reports to the GNU C compiler team, we finally managed to compile the code without the C compiler segfaulting. The matter has still not been resolved entirely, so all boosting is still done on our workstations.

A few notes on style: When we say “conceptually”, e.g. “conceptually, CodeBoost proceeds as follows”, we mean that in theory, this is how it should be done, but in practice, nasty details prevent us from doing it exactly that way. Where this difference is non-trivial, we explain how and why.

At times, our explanations and examples may appear very detailed and basic to people familiar with the paradigm of rewriting strategies and Stratego. The level of detail is deliberate. For CodeBoost to have a future, we need additional hands to help out. The entry barrier for becoming productive with CodeBoost is heightened by the lack of proper documentation. We hope that by providing detailed explanations in this thesis, the entry barrier will be lowered somewhat.

Finally, I would like to hand out my thanks to the people around me throughout this work.

A heapload of thanks goes to Otto Skrove Bagge for letting me lift his Stratego formatting tools, L^AT_EX style files and for being an overall nice guy. The weekly

Sophus meetings and its participants have also been a steady source of stimulation.

It is only fitting that Eelco Visser should be honoured for writing Stratego, a really neat transformation tool without which the current CodeBoost would not exist.

Moral support has been provided by my colleagues and friends. Of special mention are: Knut A. Erstad, Stig E. Sandø, Espen Riskedal. Most missing grammatical mistakes are due to the delightful assistance of Tilde Broch Østborg. I am eternally grateful to Sigurd Thune, who loaned me his computer during the final stages of this thesis, when my own computer completely broke down.

And last, but not least, I would like to direct a huge thank you to my supervisor Magne Haveraaen for getting me into this in the first place.

This investigation has been carried out with the support of the Research council of Norway (NFR), and by a grant of computing resources from NFR's Supercomputer Committee.

Chapter 1

Introduction

In this chapter, we outline our work and put it in a general perspective.

We start by detailing our motivation, continue by putting it in a historical context and conclude by outlining how the remainder of the thesis is organized.

1.1 Motivation

When writing large or complex applications for a specific domain, one often wishes for a domain-specific language, custom-fitted for the task at hand. A *domain specific language* (DSL) is a programming language intended to solve problems within a limited application domain.

If one primarily implements mathematical models, a language with the full plethora of mathematical operators and abstractions may be desirable.

Such domain-specific languages, be they built from the ground up, adapted from existing languages or written in a programmable programming language such as Lisp, always come at a certain cost.

Usually, the performance of applications written in DSLs is lower than if the application had been developed in a general programming language.

Additionally, building an entire language is usually very tricky and requires a significant investment of time. It can seldom be argued beyond doubt that developing a language to solve a given problem will pay off in the long run, as very few projects have sufficiently long estimated lifetimes.

One alternative to developing a full-fledged language, is to employ a *domain specific embedded language* (DSEL). A domain specific embedded language is a DSL implemented inside some other language.

An example of this is the Flex (and Lex) lexer generators. The Flex language deals with regular expressions, and is embedded into a C language container.

Yet another alternative, related approach, is to extend an existing language,

henceforth termed the *subject language*, with a semantically aware, transforming preprocessor and an accompanying library.

This is the approach we have taken. In this thesis, we show that such a preprocessor can give many of the advantages of a domain-specific language.

1.2 Background

The original idea for a preprocessor for the Sophus numerical library was proposed by Magne Haveraaen in 1995, as part of the SAGA project. From [Hav03]:

The aim of the SAGA project is to demonstrate and quantify the benefits of using algebraic programming techniques as a breakthrough technology in the area of computational modeling.

The preprocessor is intended to help fill the gap between the abstract model of a mathematical problem and the concrete implementation that solves it; we want to write well-structured programs without sacrificing performance.

This implies that the preprocessor is not merely intended to be a high-level optimizer, but also a tool for adapting and extending the subject language beyond what a simple textual preprocessor can offer.

When the SAGA project was financed, Jan Heering at CWI used additional funds to allow T.B. Dinesh to implement the first generation of such a preprocessor. It was named CodeBoost (CB1).

This implementation was done using a system called Syntax Definition Formalism + Algebraic Specification Formalism, SDF+ASF (see Section 3.7.5 and [BvdH⁺01]). Dinesh applied CodeBoost to selected parts of the Sophus library and programs before it was decided that SDF+ASF was not sufficiently powerful to solve the problems at hand.

Later, Haveraaen was introduced to Eelco Visser's Stratego rewriting system, also by way of Heering. Stratego had a more powerful approach to tree traversals. Otto Skrove Bagge was sent to the University of Utrecht in the winter of 2000 to learn Stratego and construct the second generation of CodeBoost (CB2).

The work in this thesis is based on the second implementation. Bagge's master thesis [Bag03] is the best resource for understanding the innards of CB2, henceforth just named CodeBoost.

1.3 Contents

The crux of our work is a handful of optimizations written using the CodeBoost transformation system (see Chapter 3), mostly tailored for the Sophus numerical library (see Section 2.1).

These transformations were chosen as we wanted:

- To show the flexibility and power available in CodeBoost.
- To investigate the practical aspects of having the user control the application and aggressiveness of the transformations, instead of just naively applying them to the entire body of code.
- To marry optimization and programming techniques from the fields of partial evaluation, program derivation and specialization, aspect-oriented programming with the high-level clarity of coordinate-free numerics.
- To investigate the feasibility of programming general abstractions in an algebraic programming style without penalties to performance.

In the course of implementing our transformations, initial prototyping was done in the “toy language” Tiger [App98]. Here, we experimented with control and parameterization mechanisms for the transformations, and gained familiarity with the Stratego language on which CodeBoost itself is based.

We then proceeded to implement some simple but vital transformations in CodeBoost, to gain familiarity with both the transformation system itself, and to gain practical experience with transformations on the complex C++ AST. These transformations included various drafts for the in-source syntax for controlling optimization (analogous to `#pragma`), as well a rudimentary algebraic simplifier.

Next, we implemented a transformation for doing symbolic differentiation of simple algebraic expressions, showing that CodeBoost can be utilized for regular generative programming (in particular program derivation).

Finally, we implemented a very specific optimization for algebraic operations on sparse matrices. If the matrix layout is known at compile-time, we show that we can generate hugely improved functions for various algebraic operations on matrices, and that changing the matrix layout amounts to no more than a recompilation of the program. We are essentially specializing by performing very restricted partial evaluation.

We also show that transformations need not be written in Stratego, and that we can tie in external programs as helpers in the transformation pipeline, and argue that this makes CodeBoost very flexible with regards to the nature of transformations it may facilitate.

1.3.1 Organization

The remainder of this thesis is organized into three parts.

The first part, consisting of Chapters 2, 3 and 4, discusses the general theoretical foundations and background material. A tiny primer on rewriting strategies

and the Stratego language is given in Chapter 2. Also given in that chapter is an overview of the Sophus numerical library, and some of its motivating factors. A backgrounder on the transformation framework itself follows in Chapter 3. The part is concluded by Chapter 4 which puts our work in relation to the existing corpus of transformation work.

The second part, comprised of Chapters 5, 6, 7 and 8, starts by introducing TigerBoost and outlines our initial experiences with writing algebraically related transformations. Thereafter, the complex transformations we have developed is covered.

The thesis is concluded in part three, where Chapter 9 discusses future extensions and improvements, while Chapter 10 summarizes our findings.

Additionally, Appendix B is included as a short guide to installing and testing the CodeBoost framework.

1.4 Remarks

Familiarity with the C++ language is a prerequisite for understanding some of the subtle points addressed in the discussion of the transformation implementations.

In general, the C++ code listings have been modified somewhat in the name of clarity. For instance, in Fig. 8.1 we introduce the global functions `loadMatrixFromFile`. The Sophus library has a slightly different, more complex and flexible way of loading data from files.

The same goes for the listings in Chapter 8. Here, we reference the matrices using the standard C++ table notation. The matrix classes in Sophus do not have the `[]` operator defined on them. The Sophus indexing system is a lot more complex and flexible. The notation in this thesis has been adopted in order to illustrate the relevant concepts of the transformation more easily.

In this thesis, we assume there are only two different matrix abstractions in Sophus, the `Tensor` and the `Mesh` classes. The actual situation is more complicated. For instance, the tensor abstractions are implemented using an abstraction called `MeshScalarField`, which in turn is derived from the `MeshCont` abstraction.

These intricacies are not relevant for the discussions in this thesis, and will not be mentioned again, but when perusing the transformation code, some names will have changed from the descriptions included herein.

Chapter 2

Toolkits and libraries

In this chapter, we describe the toolkits and libraries used during this project.

We start by describing the Sophus numerical library and the algebraic style on which the library is built. Next, we present the Stratego language and the concept of rewriting strategies for program transformation.

We conclude with a short presentation of auxiliary tools we found indispensable in the course of development. Some of these have been developed by members of the research group, the others are well-known tools.

2.1 Sophus

The Sophus authors describe the Sophus library thus:

Sophus is a software library carefully designed to mimic the abstract structure of the mathematics of partial differential equations [PDEs], as used in the description of many natural and industrial phenomena. Its apparatus supports coordinate free numerics in the formulation and development of solvers to problems. By requiring strict adherence to specified interfaces, we have been able to achieve that different implementations of the same mathematical concepts are basically interchangeable. The Sophus library components can be presented through different layers of abstractions.

Application layer: solvers for PDEs such as a seismic simulator, Seis-Mod, or a coating problem.

Tensor layer: handles coordinate systems, matrices and vectors and general differentiation operators.

Scalar field layer: numerical discretisations such as finite differences and finite elements with partial derivatives.

Mesh layer: implements grids for sequential and parallel HPC machines.

The interchangeability of modules within a layer allows software developers to experiment with different solution strategies and HPC architectures without the need for extensive reprogramming. For example, the change from a sequential to a parallel version of a seismic simulator does not involve any reprogramming of the rest of the solver application. Currently, the Sophus library is implemented using C++.

From a software engineering standpoint, Sophus is considerably more than a regular C++ library. It brings with it the concept of coordinate-free numerics (explained below) and a strict style of programming, dubbed the Sophus style.

This style narrows down the immense flexibility of the C++ language to a well-defined subset by disallowing the usage of some language constructs (pointers), minimising the usage of some (inheritance) and promoting others (the functional style).

The main motivation for the style is to follow regular mathematics very closely, so that implementing a mathematical model using Sophus becomes easier.

As will become apparent throughout the rest of this work, the Sophus style vastly improves the feasibility of advanced optimisations that in an unrestricted C++ programming styles would be practically impossible.

2.1.1 Higher-level language constructs

In most languages, C++ included, the chief method of extending the language is to add new data types and functions, and chunk them together in a library.

Conceptually this is very simple, and can easily remain so in practice as well, unless the problem domain is particularly complex. The interrelations between the various entities (in particular methods and data types) in the library may be well documented and well understood by the library user, but that is not to say that the compiler itself has any grasp of their semantics.

For languages targeted at a particular domain, it is common to include considerable knowledge about the standard libraries into the compiler: OpenMP support in the SparcWorks and MIPSPro compilers (as introduced in [CDK⁺00]) is one example of this.

Fortran compilers include what is normally considered to be library functions (such as the intrinsic trigonometric functions) into the compiler proper, in order to be able to optimise their usage.

In our case, we want the compiler to be aware of properties of the abstractions provided by Sophus, primarily in order to perform compile-time optimisations.

However, hard-coding this knowledge into a generic compiler's pipeline is hardly appropriate. Many domain-specific optimisations have a very restricted audience, so its higher cost of maintenance is seldom justified.

The domain-specificity leads us to think that the optimiser should rather go with the domain-specific library instead of the generic compiler. Thus, one would ideally like to have library-specific high-level optimisers that can plug into all of the existing compilers quite easily.

By plug-in, we mean using the compiler's front-end to do the parsing and semantic evaluation, then perform library-specific optimisation in the internal AST.

Few compilers provide such functionality, IBM's Montana compiler is one [Kar98]. Nevertheless, there exists no standardised extension API for modern compilers. Therefore, the only common interface mechanism between the high-level optimiser and the compiler proper is by through rewritten source code.

Even then, we run into the very real nuisance that all compilers are not created equal. Especially for C++ language, the compilers have a varying degree of language compliance, forcing us to either restrict our compatibility list, or spend time on writing compiler-specific workarounds.

Once we have established how much of the C++ language we can reasonably use, we need to tackle the problem of what our abstractions should look like, and how they should interface with each other to gain maximal flexibility. This is the the domain of coordinate-free numerics.

2.1.2 Coordinate-free numerics

In mathematics, there is the distinction between *abstract* and *concrete* definitions and formulations. The concrete definition tells us something about the representation of a given mathematical object, whereas the abstract definition tells us something about its properties, regardless of representation.

This is analogous to the what/how dichotomy found in software engineering. Such a dichotomy is also one the bases for object-oriented programming.

The concrete representation of a natural number may be in the binary or the decimal system. The chosen representation does not change the fundamental properties of natural numbers, such as each having a unique successor (Peano).

Uncovering and exploiting the abstract properties of mathematical objects forms the basis of category theory, which goes so far as to claim that the only interesting properties of an object are its interactions with other objects.

In software engineering, we have as a basic tenet the separation between *specification* (what) and *implementation* (how).

Solving differential equations numerically is a very common problem in numerics. Traditionally, selecting a coordinate system in which to perform the calculations has been done early on in the implementation process, and permeates

the entire program. With this approach, changing the representation by going to a different coordinate system is a sizable chore.

The name “coordinate-free” does not imply that the coordinates are dispensed with altogether. They are contained at a much lower implementation level, and the different coordinate systems (such as Cartesian, radial, polar, axis-symmetrical, etc) may be readily interchanged.

One can think of PDEs as being coordinate-free in the same way as lists are pointer-free; the complexity of the coordinates is encapsulated inside a well-defined interface.

If we select the representation early on, solving the the simple linear equation $Ax = b$ for dense and sparse representations of A will then need two entirely different implementations.

The alternative, advocated by the coordinate-free approach and common software engineering practice, would be to implement the solver not based directly on the sparse or dense representation of the objects involved, but as an algorithm on their common interface properties.

In the case of the linear equation, both dense and sparse matrices have the operations *vector addition* and *inner products* defined on them, which may be used to formulate a general solver.

When have formulated a solver for the problem, targeted towards the the *abstract* definition of the objects, switching between the sparse and dense representations is trivial.

To attain maximum efficiency, we may still want to specialise the solver to make use of the underlying properties of hidden in the concrete definition. If the matrix A above had been lower-triangular, the solver algorithm could be simplified.

Uncovering and exploiting such properties is what we wish to use CodeBoost for. For that to be possible, we need clearly defined semantics of all data types, and relevant operations on those. This is partly achieved through a strict coding style, described next, and partly by tagging properties onto objects, described in Section 3.5.

2.2 Object-oriented numerics

C++ is a huge language. There is virtually always more than one way to solve a given problem, and the solution taken often depends on the programmer’s preference.

To make the optimisation problem feasible, we must enforce some restrictions on the coding style, in our case the algebraic style. On the macro level, the goals of such a restriction are as follows:

- The code should be as close to regular mathematics as practical.
- A mathematical model should map easily over to code so that;
- Changes in the model may easily be transported over to the code.
- Mathematical familiarity and cursory computer programming experience should be enough to read the code.

Coordinate-free numerics, as described in the previous section, maps nicely onto the concept of objects and classes in the C++ language. The abstract definition of a mathematical object becomes a class with a given set of operators. The concrete definition becomes the actual implementation of that class.

In a micro level, we force particular conventions in order to maximise the opportunity for beneficial transformations. One such convention is how we denote and implement parameter passing between Sophus abstractions.

An argument to a member function of a Sophus class is always in one of the following modes:

- *given*; the argument is initialised to a given value. Used as first argument to constructors.
- *updated*; the argument is modified.
- *observed*; the argument is not modified.
- *deleted*; the argument is deleted. Only used for destructors.

By knowing that parameter x is not modified, the transformer can later infer that all the data invariants of x that were true before the call are still true afterwards.

If we compare the *observed* parameter passing mode to the `const` keyword, we see that the former is a much stricter convention. In C++, member function `void Foo::f() const` is, under given conditions, allowed to modify members of a `const Foo` object, thus we cannot really be certain that the data invariants hold after invoking `f`.

It must be noted that the conventions in the Sophus style are manually enforced. We do not have any tools that verify adherence to the style. An interesting future project may be writing a transformation that flags style breaches and common mistakes.

2.2.1 A typical Sophus application

A typical application written using the Sophus library proceeds through the following steps:

1. Initialisation and data input
2. Solver loop
3. Cleanup and file dump

In (1), the problem set is loaded from disk, and possibly massaged into a more optimal internal representation, as well as allocated the necessary memory slots and processors used in the following step.

Step (2) is usually a loop that applies the solving algorithm in a step-wise fashion. It is common that the solver slowly converges towards the correct solution, so that one can weigh the benefit of an accurate solution against the computing costs and optionally stop the simulation early when the solution is deemed “good enough”. This is exemplified with the solver for the coating problem, described in [GHW00].

A variant of step (2), the one used in Seismod, is to run the simulation over time. Each iteration of the loop calculates a step in time. In this case, there will be no convergence. The simulation stops when the desired time span has been evaluated.

Finally, step (3) will deallocate the memory and processors used, then dump the internal solution to a file for later processing by other programs.

An example of a Sophus program is the Seismod simulator for elastic waves. Fig. 2.1 shows the solver loop for the simulator.

```
while(t < stoptime) {  
  e = Lie(un, g);  
  sigma = apply(Lambda, e);  
  a = div(sigma) / rho + fsource;  
  unp1 = 2 * un - unm1 + a * deltat * deltat;  
  unm1 = un;  
  un = unp1;  
}
```

Figure 2.1: The Seismod solver loop

A fuller explanation of the variables and meaning behind the solver is given in [FJH⁺01]. We are primarily interested in the properties of the data and functions involved in this solver loop, and the `apply` function, which is a general matrix multiplication, also referred to as `mmult`.

2.3 Hierarchical data structures

The data structures processed by the code in Fig. 2.1 are hierarchical. For instance, in the Λ and e tensors, the individual elements are huge meshes.

Fig. 2.2 depicts the relation between the tensors and meshes.

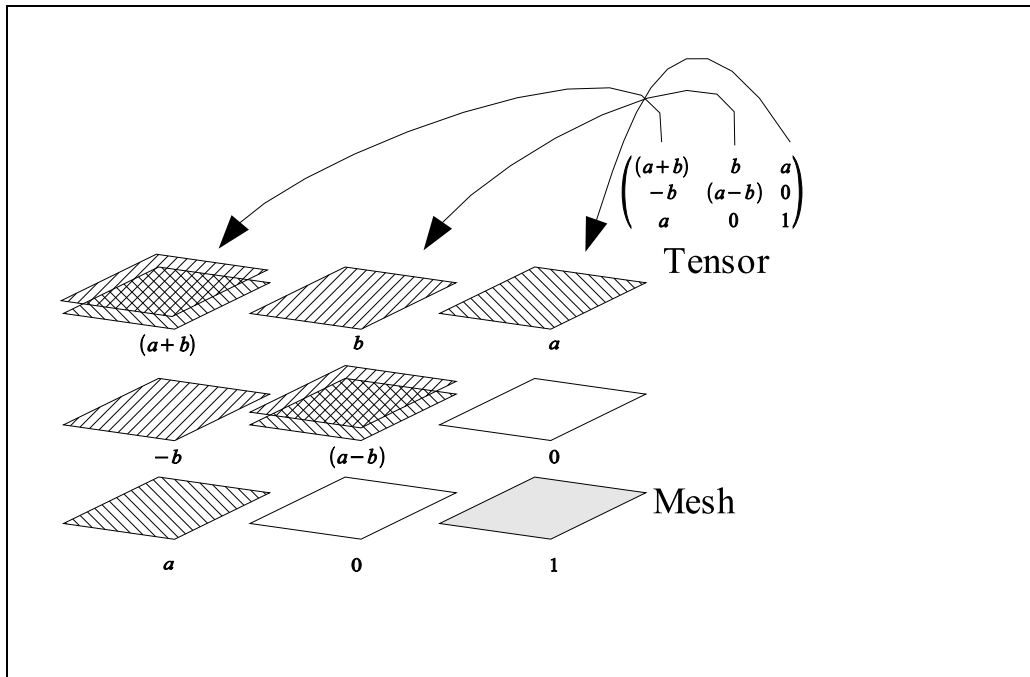


Figure 2.2: The Tensor/Mesh hierarchy

Chapter 7 treats the properties of meshes and tensors in more detail.

2.4 Stratego

As explained in the introduction chapter, the current version of CodeBoost was implemented in Eelco Visser's Stratego language ([Vis01a] and [LVV02]).

In a nutshell, Stratego is "a modular language for the specification of fully automatic program transformation systems based on the paradigm of rewriting strategies."

From [EV03]:

In Stratego, basic transformation rules are expressed by means of labeled conditional rewrite rules. Exhaustively applying all rewrite rules in a collection of valid rules is often not desirable; a system of

rules can be non-terminating, or, more frequently, non-confluent. The latter means that different outcomes of the normalization process are possible depending on the position of application and the selection of rules. Therefore, it is necessary to have more control over the application of rules. In standard systems based on rewriting, normalization is controlled by a fixed default `RewritingStrategy`. In such systems more control is achieved by encoding the desired strategy with additional rewrite rules that spell out a traversal over the abstract syntax tree and apply the transformations in the desired order.

To control the application of transformation rules, Stratego provides a language for defining rewriting strategies based on primitives for sequential programming and abstract syntax tree traversal. A rewriting strategy selects a number of rules from the available rules and defines in what order these rules are applied to a program fragment. Thus, the separation of concerns between logic (transformation rules) and control is maintained.

Transformation strategies can be defined generically, i.e., parameterized with a rule selection or other strategy to apply. The language comes with a large library of generic strategies for syntax tree traversal and built-in data type manipulation. In particular, the library contains generic, language independent strategies for language processing such as substitution, unification, and bound variable renaming that are parameterized with the shape of the relevant language constructs. Strategies cannot only be used to combine rules into transformations, but can also be used inside rules to test applicability conditions and perform local transformations.

In the following sections, we will give a brief primer on the Stratego language, so that the reader may enjoy the code excerpts provided in subsequent chapters.

2.4.1 Terms, rules and strategies

In Stratego, we perform rewrites (alterations of tree nodes and subtrees) on a tree structure consisting of *terms* (nodes) by applying rewriting *rules*. The application of these rewriting rules are controlled by *strategies*.

Given a program expressed as an abstract tree structure, transforming the program is a question of applying the transformation rules in the correct order, before converting it back to a concrete syntax (usually called pretty-printing).

In the next sections, we will explain the meaning of terms, rules and strategies.

Terms

The basic data abstraction we work on in Stratego is the term. Terms make up a tree structure, which in turn is also a term. The word *term* is used to refer to both leaf nodes, internal nodes and subtrees of this tree structure.

While the human-readable version of terms is on the form `Parent(Child1,Child2)`, they are stored, modified and passed around throughout the pipeline in a more efficient binary format.

Figure Fig. 2.3 is provided as an illustration on how actual terms in CodeBoost appear.

```
Expr(
  Infix(Op("=",
    Sig(FunDecl(RootName(OpName("=")),
      ArgList([
        VarDecl(NoName,Type([],IdName("__any_t"),DNil),[],NoExpr),
        VarDecl(NoName,Type([],IdName("__any_t"),DNil),[],NoExpr)]),
      Type([],IdName("__any_t"),DNil),
      [],NoQual,NoCInit,NoDecl),(Builtin,Public),[])),
    Var(IdName("c"),Type([],IdName("int"),DNil),Local),
    Infix(Op("*",
      Sig(FunDecl(RootName(OpName("*")),
        ArgList([
          VarDecl(NoName,Type([],IdName("int"),DNil),[],NoExpr),
          VarDecl(NoName,Type([],IdName("int"),DNil),[],NoExpr)]),
          Type([],IdName("int"),DNil),
          [],NoQual,NoCInit,NoDecl),(Builtin,Public),[])),
        Var(IdName("a"),Type([],IdName("int"),DNil),Local),
        Var(IdName("b"),Type([],IdName("int"),DNil),Local))))))
```

Figure 2.3: AST for $c = a * b$

This term is an *abstract syntax tree* (AST) for the expression $c = a * b$.

As we can see, the AST representation is a good deal more complex than the corresponding concrete syntax. This should not be surprising, given the additional detail provided in the AST.

We work on this term by applying transformation rules, described next.

Rules

Rules are snippets of Stratego code that explain how to rewrite (modify) one term to another. It generally has a name ¹, consists of a left-hand side which must match a given term, a right-hand side which is the resulting term, and a *where* clause, which may specify restrictions and expressions to be evaluated in order for the rule to be applied.

Take the rule in Fig. 2.4 as an example.

```
simple-rule:  
Foo(s) → Bar(s')  
where  
<to-string> s ⇒ s'
```

Figure 2.4: A simple rule

Mental evaluation of this rule proceeds as follows:

First, the left-hand side of \rightarrow will match a term $\text{Foo}(s)$, where s is a free variable that is bound at matching time, i.e. it can be any term.

The *where* clause will apply the strategy `to-string` to the s term, and assign the result to the variable s' . Typically, the *where* clauses get rather long for non-trivial rules.

s' is then used to construct the resulting tree, $\text{Bar}(s')$.

This rule will succeed iff the term on which it is applied has `Foo` as its root node and the strategy `to-string` succeeds. So, in order to ensure success of this rule, we must be certain it is applied where we can expect to find a `Foo` term. Deciding when and where to apply rules is the job of strategies, described next.

Strategies

A strategy is a recipe for how to apply the named rules to a given term. Stratego comes with a number of traversal strategies that allows you to walk the tree in various ways, applying rules to each node, to some nodes, to least one node or other combinations thereof.

For example, if we wanted to apply our `simple-rule` from the previous section at least once inside a given term, we could do `<onced(simple-rule)> term`. `onced` will visit each subterm in term, and attempt to apply `simple-rule` at each visit. Once `simple-rule` succeeds, it will stop the traversal and signal a success. If `simple-rule` fails at all nodes, `onced` will signal a failure.

¹All strategies and rules are named in lower-case-separated-by-hyphens. In the code listings, they are marked green.

In Stratego, it is the programmer's responsibility to deal with such failures. Sometimes, we just want to the failure to propagate all the way up, and terminate the running program. Other times, we want to try alternative strategies when one fails.

There is a rich set of combinators for this. The `;` combinator acts as a logical and. Given `strategy1 ; strategy2`, the expression will only succeed if both `strategy1` and `strategy2` succeed, in that order. The choice combinator `+`, can be would succeed if either of `strategy1` or `strategy2` succeeded. The order of evaluation is undefined. There is also a left-choice `<+` operator to force the order of evaluation.

Armed with the built-in strategies and a rich set of combinators, one can construct new strategies that apply rules (or other strategies) to the terms in an arbitrarily complex manner.

The strategy code excerpts in this thesis are all fairly simple. For a more complete tutorial, refer to [Vis00].

2.4.2 Modularisation and pipeline

The strategies and rules as described in the previous sections are grouped into modules. A module is a conceptual unit concerned with handling one particular transformation. It is analogous to modules in modern OOP languages (e.g., packages in Java). An example of a module is the `simplify2` transformation described in Section 4.3.

In CodeBoost, we use modules in two ways: they are either separate transformation steps in a pipeline (details in Section 3.2), or they are library modules, included into a transformation module.

The `dump` example in Section 3.3.1 shows a very tiny, but complete module.

2.5 Auxiliary tools

Our basic development toolkit includes a syntax-aware editor, a version control system and only basic Unix command line utilities.

It is our goal that CodeBoost should be usable in any Unix-like environment.

2.5.1 Syntax-aware editor

We selected Emacs [Sta97] as our editor. Our motivating reason for using Emacs, apart from prior familiarity, is its ability to be made syntax-aware of our subject languages.

The syntax-awareness allows practical conveniences such as automatic indenting, syntax highlighting and rudimentary semantic searching (i.e., list all methods in file).

We have added two new syntax modes for Emacs: `stratego-mode`, which is used for editing Stratego source code, and `codeboost-mode`, which is used for editing CodeBoost configuration files. Both are available in the `codeboost-utils` CVS module².

2.5.2 Version control

The CodeBoost source code is managed in a CVS [BF02] repository.

As CodeBoost is research code, disruptive changes happen a lot. At the same time, smaller bugs are reported, and often we want to issue a fix immediately.

To accommodate this, we follow the common development practice of introducing radical changes by way of branches.

A branch is a separate line of development; it is conceptually a separate copy of the source code directory tree. One branch, termed `HEAD`, always contains the latest stable source code.

When we decide to rewrite a central part of CodeBoost (which happens frequently), we create a new, experimental, named branch of the tree, introduce the disruptive changes into that branch, perform adequate testing, then merge the experimental branch back with the `HEAD` branch.

The idea is that the ongoing disruptive work, which may and often does last for a week or more, is performed on a separate tree. This way, both the experimental branch and the stable branch (`HEAD`) is available to interested parties at all times.

The merging process is usually quick, and if extra care is taken, can in practice be finished in much less than an hour.

From time to time, we take a snapshot of `HEAD` and release it as a versioned tarball. This typically happens when we have successfully merged in a series of significant improvements, or when we need a new baseline for presenting ongoing work.

²See [OSB03] for instructions on how to obtain them.

Chapter 3

CodeBoost

This chapter introduces the CodeBoost transformation framework.

We start by putting CodeBoost in a detailed context, then discuss the transformation pipeline. A small example of a transformation is given.

We continue by introducing parameterization of transformations and the CodeBoost configuration mechanism. Finally, perceived and experienced shortcomings of the current implementation are discussed.

3.1 Background

The CodeBoost (CB1) concept was first described in [DHH98]. The underlying idea is to bridge the gap between the compiler's high-level optimizations and hand-coded, application-specific optimizations for the domain of numerical software written in the algebraic style.

Optimizations performed by the compiler are usually

- *fine-grained*; work at the expression level. With advanced dependency analysis, some code-motion is allowed, but is usually fairly restricted.
- *general*; very little information about user-defined data types can be exploited. For the compiler to bring out its workhorse optimization tricks, the user-defined data types have to be broken down into primitive types.
- *restricted*; the resulting program must be correct according to the semantic rules of the language.

CodeBoost is an attempt at remedying these shortcomings in regular compilers by making explicit domain-specific relations among the data types at hand.

The version of CodeBoost, described in [DHH98], has been deprecated and will not be discussed further.

The remainder of this thesis concerns itself with the second implementation, detailed in [Bag03]

3.2 Transformation pipeline

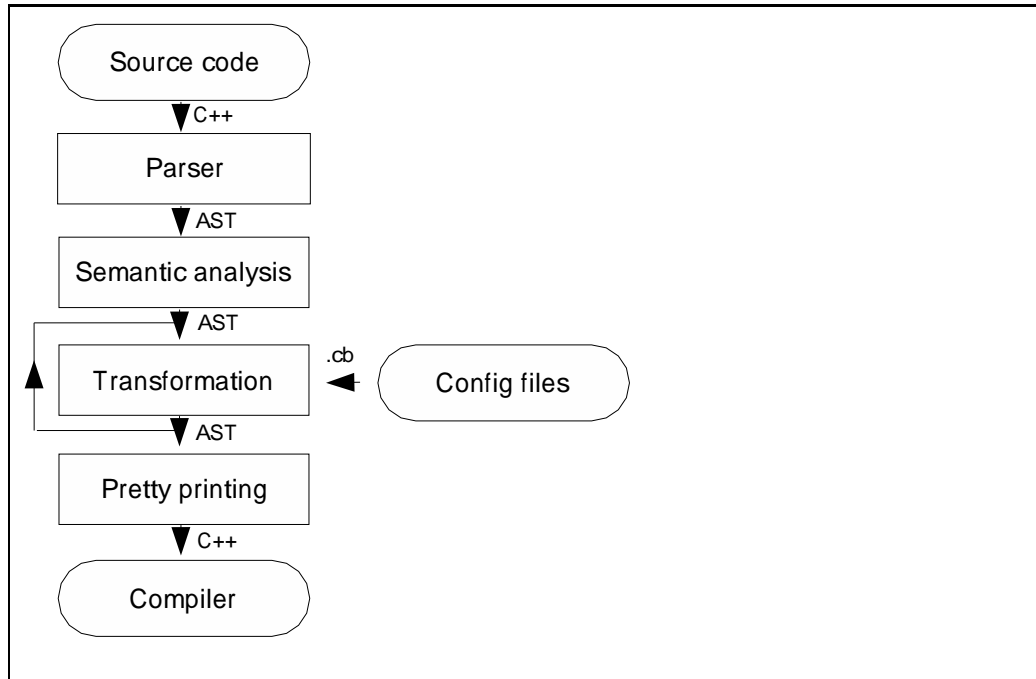


Figure 3.1: The CodeBoost pipeline

3.2.1 Parsing

The incoming C++ code is preprocessed using the regular C++ preprocessor and a CodeBoost-specific Perl script. This allows us to easily control which parts of the code should be seen and modified by CodeBoost in the later steps.

For example, it is not desirable for CodeBoost to see the header files and template declarations for the standard C++ library, as we are not interested in performing transformations on them.

The preprocessed code is then parsed using the OpenC++ [Chi96] front-end, which has been modified to output its internal concrete syntax tree (CST) in the ATerm format [vdBdJKO00].

From here on out, the incoming code is passed between the pipeline steps as ATerms. This allows us to insert pipeline steps written in any language we desire,

provided we can parse and output valid ATerms. The ATerm library has bindings for the C and Java languages.

3.2.2 Semantic analysis

The semantic analyzer is divided into three parts:

- *local variable annotation*; the local variables are annotated with their appropriate types, using a simple environment-passing scheme.
- *symbol table generation*; non-local declarations are collected into a symbol table, and variable names are properly resolved. Function and operator applications are annotated with a list of possible candidates.
- *overload resolution*; the function and operator applications are resolved by calculating the type of all parameter expressions and picking the best-match candidate from the candidate list.

For proper overloading resolution, CodeBoost has been made aware of the default language declarations for the built-in operators and some of the standard library functions. These will be used if no other suitable candidates are found first.

Undeclared variables and functions will not cause CodeBoost to stop, but can cause trouble at the transformation stage, as complete type information will not be available. The undeclared variables will be marked as having a type of `_any_t`.

At the end of the semantic analysis step, all variables have been tagged with their appropriate types, and all function/operator applications have been annotated with the function declaration of their resolved candidate.

3.2.3 Transformation

The primary extension point in CodeBoost is at the transformation stage. A typical transformation is a Stratego module which receives the AST of the program. The transformation rewrites the AST and passes the result on to the next step in the pipeline.

As mentioned, the pipeline steps can be written in any language. However, CodeBoost provides a convenient library of rules and strategies available to transformations written in Stratego for tasks such as symbol table lookups, type operations (comparison, matching, conversion), various traversal strategies, totem annotation, pretty-printing for error messages.

In addition, transformations written in the Stratego language will naturally be able to benefit from Stratego's extensive collection of traversal strategies and generic support for variable renaming.

We have already implemented a handful of transformations for CodeBoost in the Stratego language. Even though most are primarily targeted at the Sophus library, some of these transformations are more generally applicable. Adapting the other transformations to another library is probably not worthwhile, as they are extremely specific.

As we separate the low-level transformation rules from the high-level traversal strategies, parts of the specific transformations may be reused. This rule/strategy separation has made it considerably easier to adapt complete transformation modules to changing conventions within the evolving Sophus library.

The order in which the transformations are performed, are specified explicitly by the user at boosting-time. This gives the user complete control, but also presents a few drawbacks:

1. *uncovering optimization possibilities*; sometimes, applying transformation *A* may open up for further optimization by transformation *B*. Unless the user is aware of this, he may miss out on transformation opportunities.. Example: Running the expression simplifier is generally desirable after running the symbolic differentiator, as the output from the differentiator can usually be simplified considerably.
2. *order of transformation*; for transformation *B* to work, it must be preceded by transformation *A*. Example: The mutation transformation (described below) must be applied before in-lining.

In time, we expect to be able to specify these dependencies, and have them resolved automatically by CodeBoost.

3.2.4 Pretty-printing

After the transformations have been performed, the resulting AST is verified to be syntactically correct, then fed to the pretty-printer.

The AST is transformed to the device and language-independent, pretty-printing Box language. The result is emitted as properly indented, human-readable C++ code.

The pretty-printing step can also highlight parts of the code, and produce a nice-looking PostScript file using GNU `a2ps`. We acknowledge that visualizing the code is very useful to inspect and verify that the transformed code does what it is intended to.

As we gain experience, it would be desirable to improve this aspect of CodeBoost, to allow easy annotation of profiling results, coloured diffs against the original code and other conveniences.

3.3 Some transformation examples

As instructive examples, we have included two transformations in this chapter. The first is a debugging aid, the second is a tiny mutation transformation.

3.3.1 AST dump

When debugging both the transformation pipeline and the individual transformations, it is practical to look at dumps of the internal AST at the various stages.

```

module dump
imports lib cb
strategies
main = cbio(dump-options, doit)
module-name = !"dump"
module-version = !" $Revision: 1.1 $"
module-id = !" $Id: codeboost-dump-1.r,v 1.1 2003/03/08
20:44:27 karltk Exp $"
dump-options = ![]
doit = debug

```

Figure 3.2: AST dumper

The code in Fig. 3.2 is a complete transformation module. Its sole purpose is to dump the internal AST to `stderr`, and leave the tree intact. It can be inserted at any stage in the pipeline.

Line one names the module (`module dump`). Line two, (`imports lib cb types`), imports the strategies available in the `lib`, `cb` and `types` modules.

In the `strategies` section, the strategies `module-name`, `module-version` and `module-id` are meta information sometimes used for debugging of Code-Boost itself.

`main` is the starting point of all transformations. The strategy `cbio`, imported from the `cb` module, loads the AST and auxiliary information from the input pipe (i.e. from the previous pipeline step), performs parameter parsing based on its first parameter, `dump-options`, then hands off control to its second parameter, the `doit` strategy.

`dump-options` is a strategy that constructs an empty list: the `dump` strategy recognizes no command line parameters. Alternatively, the `dump-options` parameter to `cbio` may be skipped altogether.

`doit` simply invokes the Stratego library function `debug`, imported from the `lib` module, which will dump the entire tree to `stderr`.

After `doit` completes, control is returned to `cbio`, which takes care of internal cleanup before the transformation exits.

3.3.2 Simple mutification

This is a cut-down, but complete and working version of the `mutify` transformation described in [Bag03].

Fig. 3.3 shows a very simple algebraic expression.

```
Tensor<float> A,B; A = A + B;
```

Figure 3.3: A simple algebraic expression

We can expect that the the C++ compiler will internally rewrite this code into the code given in Fig. 3.4.

```
Tensor<float> A,B,t0; t0 = A + B; A = t0;
```

Figure 3.4: Code from Fig. 3.3 after compiler rewrite

If `A` and `B` are large structures, the unneeded copying from `t0` to `A` can easily result in an annoying performance hit.

If the code in 3.3 adheres to the Sophus style (see Section 2.2, we know that there is a relation between the operators `+` and `+=` on the `Tensor` class.

This will allow us to rewrite 3.3 to the the expression in Fig. 3.5.

```
A += B;
```

Figure 3.5: After mutifi cation

In the Sophus `Tensor` class, the `+=` operator is implemented using no unnecessary copying.

The regular C++ compiler could not have performed this optimization, as the C++ language does not specify that there need be any such relation between `+` and `+=` on user-defi ned types.

The Codeboost `mutify` transformation solves this problem for us by rewriting expressions on the form $x_0 = x_0 + x_1$ to $x_0 += x_1$.

A cut-down version of the `mutify` transformation is given in Fig. 3.6.

The structure of this module is very simple. It has two main sections; `strategies` and `rules`. The former contains the code for traversals and I/O

```

module mutify
imports lib cb types
strategies
main = cbio(mutify-options, doit)
module-version = !" $Revision: "
module-name = !"mutify"
module-id = !" "
mutify-options = ![]
doit = Program(id,doit',id)
doit' = topdown(try(mutify-plus))
rules
mutify-plus:
  Expr(Infix(Op("=",sig),Var(n,t,s),Infix(Op("+",sig'),Var(n,t,s),e))) →
  Expr(Infix(Op("+=",sig''),Var(n,t,s),e))
  where
    ⟨type-of-expr ; is-tensor⟩ e ;
    ⟨is-tensor⟩ t ;
    ⟨oncetd(\OpName("=",) → OpName("+=",))\⟩ sig ⇒ sig''
is-tensor =
  ?Type(.,IdName("Tensor"),.-)

```

Figure 3.6: Cut-down mutification transformation

with the CodeBoost pipeline. The latter contains the low-level transformations rules we want to apply.

The program starts at `main`, which calls off into `cbio`. The strategy `cbio` abstracts away the untidy details of loading and storing the AST as well as parameter parsing.

This transformation module has no parameters, so `mutify-options` is a strategy constructing an empty list.

`cbio` will eventually call `doit`, which starts by loading the symbol table. `Program(id,doit',id)` signifies that we emit a new top-level `Program` term, with the first and third parameters (version info and auxiliary tables [such as symbol table], respectively) are left untouched. The second parameter (program AST) is handed to the `doit'` strategy, and the result is output.

`doit` is a very simple strategy that traverses all nodes in the AST in a top-down manner. At each term (node), it tries to apply the `mutify-plus` rule. Where the application succeeds, the original term is replaced with the transformed term.

The `mutify-plus` rule will only allow itself to be applied to expressions on the form $A = A + B$, where both `A` and `B` are `Tensors`. It first checks that the type of the term `e` is `Tensor`, then it checks that the type of the variable is also a `Tensor`. The `<type-of-expr ; is-tensor>` syntax signifies strategy composition; first `type-of-expr` is applied, then `is-tensor` is applied. Both must succeed for the transformation to continue.

Additional simple transformations are described in Chapter 4. More advanced transformations are described in [Bag03] and in Chapters 6 and 8.

3.4 Framework

As described in the previous sections, CodeBoost takes care of parsing C++, maintaining a symbol table, providing name and type lookup, pretty printing the AST, parameter passing between transformations.

Additionally, the CodeBoost framework contains convenience scripts for pretty-printing `ATerms`, Emacs modes for `Stratego` and the CodeBoost configuration languages, diffing and folding `ATerms`. We expect the collection of auxiliary programs to increase as CodeBoost is applied to real-world tasks.

An extension we made to this framework, was adding the ability to tag language constructs with auxiliary information to aid transformations. For example, we can use a tag to tell CodeBoost that a given variable `A` should be considered a null matrix for the remainder of the scope.

The tags we put on the language constructs range from simple flags (null matrix), to an arbitrarily complex description (a complete description of a matrix

layout). The complex descriptions are stored in their own configuration files, and linked at boosting time.

We will first describe these configuration files, then the process of tagging their contents onto the language constructs.

3.4.1 Configuration files

In the course of our work, we found it necessary to extend CodeBoost with configuration files. The configuration file extension was mainly prompted by the need to specify matrix layouts for the matrix optimizations described in Chapter 8.

In the interest of reducing development time, we wanted to go for the simplest configuration language possible. We started out by a simple `attribute = value` format, as shown in Fig. 3.7.

```
max-unrolled-instances = 100
max-nesting = 10
min-chunk-size = 0
max-chunk-size = 100
add-comment = "CodeBoost was here."
start-unroll-at = 0
end-unroll-at = max-unrolled-instances
enabled = true
```

Figure 3.7: Initial CodeBoost configuration format

It quickly became apparent that if we wanted to group parameters to several transformations into one file, we would have problems with namespace collisions for the attribute names (in particular, the experimental `enabled` setting).

Consequently, it was decided to have all attribute names on the form `<context>.<name>`. A typical `attribute = value` pair then becomes `unroll-loops.max-nesting = 10`.

As we started specifying matrices, we experienced the need to change and extend the syntax considerably. With that came the realization that we would probably be forced to extend the configuration syntax slightly for each new, non-trivial transformation that dealt with mathematical structures, e.g. a set transformation may want to specify some properties of its set in a set-syntax.

We wanted to avoid having to rewrite and extend the configuration file parser for each new transformation added to CodeBoost. We looked at existing formalisms which address the problem of using one common syntax across many and varied semantics. The immediate candidates were XML [TBSM00], SGML [Gol90] and symbolic expressions.

Earlier experience with SGML and XML told us that SGML is a more complex language than XML, and that the added power of SGML was not needed for this task. The choice between XML and symbolic expressions fell in the favour of symbolic expressions, as we did not want to bring in a full-blown XML parser at the current stage of development.

Fig. 3.7 rewritten for the new symbolic expression syntax is given in Fig. 3.8.

```
(context
  unroll-loops
    (max-unrolled-instances 100)
    (max-nesting 10)
    (min-chunk-size 0)
    (max-chunk-size 100)
    (add-comment "CodeBoost was here.")
    (start-unroll-at 0)
    (end-unroll-at max-unrolled-instances)
    (enabled true)
)
```

Figure 3.8: The current CodeBoost configuration format

Notice that we have introduced a block-like `context` construct instead of the `<context> . <name>` syntax shown earlier. The `context` construct is always on the form `(context <name> ...)`, where the body `...` can be any list of symbolic expressions.

This body is handed off as a concrete syntax tree (CST) to each transformation. The task of evaluating this tree is then left to the transformation itself.

With the reformulation into symbolic expression, we were able to specify matrix layout specifications quite easily. The matrix

$$\begin{pmatrix} a & a-b & 0 \\ b-a & b & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

would be written as shown in Fig. 3.9.

The `(layout ...)` subexpression will be fed directly to the matrix transformation as a CST for further processing.

For details on how the specification in Fig 3.9 is handled, refer to Chapter 8.

The open-endedness of the Codeboost configuration format allows almost any kind of entity to be described. Once an entity is specified in a configuration file, it must be related to the program source code. This is the task of the totem mechanism, described next.

```

(context
  tensor-optimiser
  (layout
    tensor-layout-a
    (explicit
      (( a (- a b) 0)
        ((- b a) b 0)
        (0 0 1)))
    )
  )
)

```

Figure 3.9: A complete matrix layout specification

3.5 Totems and totem propagation

We have implemented a mechanism in CodeBoost for tagging language constructs with auxiliary information, and propagating these tags throughout the program.

The tags can be specified by the programmer in the source code itself, in external configuration files or in some cases, they can be inferred automatically by CodeBoost.

In our nomenclature, such a tag is called a *totem*, as the tag is usually an emblem signifying membership in a special class of entities.

The function call `CB_TAG(A, ``algebraic-simplification'', ``unit-matrix'')` tags the variable `A` with membership in the exclusive `unit-matrix-club`, in the `algebraic-simplification` context.

Each transformation typically has its own context, but as the context name is decoupled from the transformation name, several transformations may share a given context, or one transformation may operate on several contexts.

When a transformation, say an algebraic simplifier, is applied to the source tree containing the expression $A \circ B$ (or `mmult(A, B)`), where `A` has been marked by the `unit-matrix` tag, it knows the expression can be simplified to `B` by the properties of matrix multiplication and unit matrices.

The `CB_TAG` construct is useful only for very simple totems that can easily be written using the C++ syntax. For more advanced totems, we use the CodeBoost configuration files, described in Section 3.4.1.

A configuration file may be loaded into the source code using the `CB_IMPORT` construct. Once loaded, all definitions available in the given configuration file may be tagged onto the desired language constructs.

Fig. 3.10 shows how this is done.

Here, the configuration file `matrix-settings.cb` is loaded, and all its definitions are available for tagging. From the context `matrix-layout`, we

```
f() {
  Mesh<float> A;
  CB_IMPORT("matrix-settings");
  CB_TAG(A, "matrix-layout", "layout-a");
}
```

Figure 3.10: Importing and tagging totems

retrieve definition `layout-a` and tag it onto the variable `A`.

In many cases, we can propagate totems throughout the program using a *general totem tracker* that implements a handful of simple, transformation-independent rules.

For variables, totems are kept throughout their lifetime, until either of the following occurs:

- *assignment*; if `A` is assigned to, it loses all its acquired totems.
- *passed as an updated parameter*; if `A` is passed to a function as an updated parameter, `A`'s internal state may change, so all totems are dropped.

Totems may propagate from one variable to another over the assignment operator (`B = A`). This clearly resembles data flow analysis, but we do not use the dependencies to reorder any expressions.

Currently, totems on classes and functions stick throughout the program.

Some challenges for the totem system remain unsolved. Consider the code in Fig. 3.11.

```
template <class T>
Mesh<T> f(Mesh<T> x, Mesh<T> y)
{ return x * y; }
g() {
  Mesh<T> a,b,c;
  CB_TAG(a, "matrix-layout", "unit-matrix");
  c = f(a,b);
}
```

Figure 3.11: Tracking totems through calls

Unless we do function inlining of `f`, we cannot currently optimize the multiplication expression properly. An alternative to inlining would be to track totems through the function call and generate a new, specialized version of `f`.

In general, tracking totems is not always straight-forward. As we show in Chapter 8, sometimes the rules for the propagation of a totem are closely tied to a given transformation and cannot be satisfied using the generic totem tracker.

3.6 Shortcomings

The primary shortcoming of CodeBoost at the present time is the lack of proper documentation for end-users and external contributors. While some rudimentary documentation does exist alongside the source code, we do not have a reference manual.

Another, more minor goal has been to maintain a certain degree of platform independence and portability. As our experimentation domain is parallelized numerical software for medium to large computations, we have restricted ourselves to Unix systems.

Our work has been conducted on Linux, IRIX and Solaris 9, using the gcc 3.0.x, 3.1.x and 3.2.x series of compilers, as well as the MIPSPro compiler on the IRIX platform and the SparcWorks compiler on Solaris.

However, it turns out that Stratego is not portable between all of our target platforms. Currently, we can only host CodeBoost on Linux and Solaris.

When developing CodeBoost, we experienced that the Stratego compiling step is rather time-consuming. This is ameliorated by continual improvements in workstation processor speeds and improvements made to the Stratego compiler itself. Consequently, we do not foresee this to be a long-term affliction.

In the process of developing transformation modules, we often have the need to inspect the intermediate stages of the AST. As Fig. 3.12 shows, this can sometimes be a daunting proposition. It would be nice to have an Emacs mode for inspecting ATerms, or a command-line ATerm beautifier.

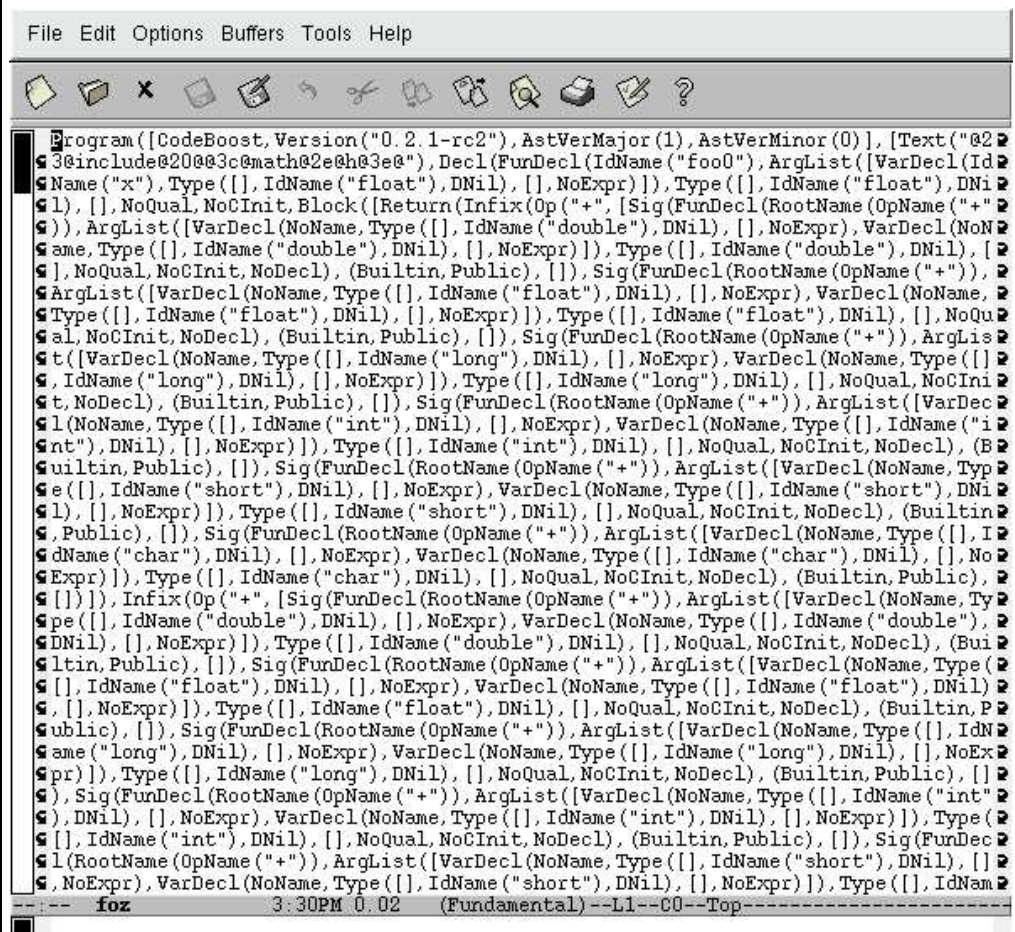
3.7 Other transformation systems

CodeBoost and Stratego are by no means the only transformation systems available for transforming algebraic and numerical software.

The system that appears to share most of CodeBoost's goals is the TAMPR rewriting system.

Generic alternatives to Stratego include OpenC++, SORCERER, ASF+SDF, ELAN, OBJ[GM97] and Maude [CELM96], among others.

The following sections will introduce the most relevant of these systems.



```
Program([CodeBoost, Version("0.2.1-rc2"), AstVerMajor(1), AstVerMinor(0)], [Text("@2  
3@include@20@3c@math@2e@h@3e@"), Decl(FunDecl(IdName("foo0"), ArgList([VarDecl(Id  
Name("x"), Type([], IdName("float"), DNil), [], NoExpr)]), Type([], IdName("float"), DNi  
l), [], NoQual, NoCInit, Block([Return(Infix(Op("+", [Sig(FunDecl(RootName(OpName("+"  
)), ArgList([VarDecl(NoName, Type([], IdName("double"), DNil), [], NoExpr), VarDecl(NoN  
ame, Type([], IdName("double"), DNil), [], NoExpr)]), Type([], IdName("double"), DNil), [N  
oQual, NoCInit, NoDecl), (Builtin, Public), [], Sig(FunDecl(RootName(OpName("+")),  
ArgList([VarDecl(NoName, Type([], IdName("float"), DNil), [], NoExpr), VarDecl(NoName,  
Type([], IdName("float"), DNil), [], NoExpr)]), Type([], IdName("float"), DNil), [], NoQu  
al, NoCInit, NoDecl), (Builtin, Public), [], Sig(FunDecl(RootName(OpName("+")), ArgLis  
t([VarDecl(NoName, Type([], IdName("long"), DNil), [], NoExpr), VarDecl(NoName, Type([  
IdName("long"), DNil), [], NoExpr)]), Type([], IdName("long"), DNil), [], NoQual, NoCIni  
t, NoDecl), (Builtin, Public), [], Sig(FunDecl(RootName(OpName("+")), ArgList([VarDec  
l(NoName, Type([], IdName("int"), DNil), [], NoExpr), VarDecl(NoName, Type([], IdName("i  
nt"), DNil), [], NoExpr)]), Type([], IdName("int"), DNil), [], NoQual, NoCInit, NoDecl), (B  
uiltin, Public), [], Sig(FunDecl(RootName(OpName("+")), ArgList([VarDecl(NoName, Typ  
e([], IdName("short"), DNil), [], NoExpr), VarDecl(NoName, Type([], IdName("short"), DNi  
l), [], NoExpr)]), Type([], IdName("short"), DNil), [], NoQual, NoCInit, NoDecl), (Bui  
ltn, Public), [], Sig(FunDecl(RootName(OpName("+")), ArgList([VarDecl(NoName, Type([], I  
dName("char"), DNil), [], NoExpr), VarDecl(NoName, Type([], IdName("char"), DNil), [], No  
Expr)]), Type([], IdName("char"), DNil), [], NoQual, NoCInit, NoDecl), (Builtin, Public),  
[])])], Infix(Op("+", [Sig(FunDecl(RootName(OpName("+")), ArgList([VarDecl(NoName, Ty  
pe([], IdName("double"), DNil), [], NoExpr), VarDecl(NoName, Type([], IdName("double"),  
DNil), [], NoExpr)]), Type([], IdName("double"), DNil), [], NoQual, NoCInit, NoDecl), (Bui  
ltn, Public), [], Sig(FunDecl(RootName(OpName("+")), ArgList([VarDecl(NoName, Type(  
[], IdName("float"), DNil), [], NoExpr), VarDecl(NoName, Type([], IdName("float"), DNil),  
[], NoExpr)]), Type([], IdName("float"), DNil), [], NoQual, NoCInit, NoDecl), (Builtin, P  
ublic), [], Sig(FunDecl(RootName(OpName("+")), ArgList([VarDecl(NoName, Type([], IdN  
ame("long"), DNil), [], NoExpr), VarDecl(NoName, Type([], IdName("long"), DNil), [], NoEx  
pr)]), Type([], IdName("long"), DNil), [], NoQual, NoCInit, NoDecl), (Builtin, Public), [  
]), Sig(FunDecl(RootName(OpName("+")), ArgList([VarDecl(NoName, Type([], IdName("int")  
), DNil), [], NoExpr), VarDecl(NoName, Type([], IdName("int"), DNil), [], NoExpr)]), Type(  
[], IdName("int"), DNil), [], NoQual, NoCInit, NoDecl), (Builtin, Public), [], Sig(FunDec  
l(RootName(OpName("+")), ArgList([VarDecl(NoName, Type([], IdName("short"), DNil), [  
], NoExpr), VarDecl(NoName, Type([], IdName("short"), DNil), [], NoExpr)]), Type([], IdNam
```

-- foz 3:30PM 0.02 (Fundamental) --L1--C0--Top

Figure 3.12: A typical debugging session

3.7.1 OpenC++

OpenC++ is a version of C++ with a Metaobject Protocol (MOP); it is a tool for source-code translation of C++ code. It can be used to define new syntax, new annotations and new object behavior.

However, in practice it turned out not to be sufficient for our needs. It only provides us with a CST to work on, and it does not offer any kind of tree-rewriting system. Also, the output after a transformation is pure textual code, not even a new CST.

Thus, we needed to extend OpenC++ with a rewriting system.

3.7.2 TAMPR

TAMPR[BHW97] is a fully automatic, rewrite-rule based program transformation system. It generates correct and efficient programs from specifications.

The authors highlight the following features of TAMPR as being novel:

- A declarative approach to specifying transformations.
- A restricted repertoire of constructs for analyzing and transforming programs.
- Application of transformations to exhaustion.
- An emphasis on sequences of canonical forms for carrying out large-scale refinements.
- Completely automatic operation.
- The ability to effortlessly "replay" the application of transformations when either the program being transformed or the transformations themselves change.

TAMPR has been used to derive both sequential and parallel programs from the same high-order functional style specification. Sets of transformation rules which together define a canonical form are automatically applied to the input specification until the canonical form is reached.

The transformations can usually be proven to be semantically preserving. The underpinnings of the TAMPR system were first described in [Boy70].

The user controls where the transformations should be applied by inserting directives into the source code. This is somewhat related to the concept of totems.

3.7.3 ANTLR

According to the authors, “ANTLR (ANother Tool for Language Recognition) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C++, or C# actions.”

Furthermore, ANTLR has the ability to do tree transformations on the internal AST. The transformations are specified in terms of a tree grammar, where the rule bodies are specified as actions associated with the grammar productions. The grammar is compiled into a tree parser by a tool named SORCERER [Par93].

The resulting parser will, when applied to an input tree, build an output tree according to the production rules. It is thus analogous to a regular token parser, except the input stream is a tree, not a token stream.

This system only allows for a one-pass traversal. There is no separation between strategies and rules, as exists in Stratego. Nor is there any form of generic tree traversal. We consider it too limited for our needs.

3.7.4 XTC

XTC (Transformation Tool Composition) implements the XT component model and provides support for creating compositions of XT components. An XT component is typically written in Stratego.

The XTC API supports easy calling of external Stratego components and mixing them with internally defined transformations.

The XTC system is akin to the CodeBoost pipeline, only a lot more generic; it is not tied to a particular subject language.

3.7.5 ASF+SDF

The first implementation of CodeBoost was done using the ASF+SDF framework, described in [BvdH⁺01]. This framework is divided into the SDF2 syntax formalism and the ASF rewriting language.

SDF2 has the following features:

- Modular syntax definition (parametrized modules, symbol renaming).
- Integrated lexical and context-free syntax.
- Declarative disambiguation constructs (priorities, associativity, and more).
- Regular expression shorthands.
- All non-circular context-free grammars allowed!

SDF2 is implemented in two parts: The parse table generator and a scanner-less generalized LR parser (SGLR). It accepts arbitrary context-free grammars as input.

The ASF is term rewriting language features:

- Conditional rewrite rules.
- Arbitrary user-defined syntax (the complete language is called ASF+SDF).
- Default rewrite rules (fire only when all other rules fail).
- Traversal functions (for automatically traversing trees).
- Rewriting while preserving layout and source code comments.
- Supported by the MetaEnvironment.

ASF can be used for the definition of semantics of programming languages, defining many-sorted algebras and defining source code transformations.

The implementation consists of a compiler and an interpreter. The compiler translates ASF to highly efficient C code. This code is compiled to a standalone tool by gcc. The interpreter has a slower execution time, but a quicker startup time.

The ASF+SDF framework was eventually rejected as an implementation toolkit for CodeBoost. It was replaced with Stratego, as Stratego has a considerably more powerful rewriting language that allows for generic traversals and detailed control over the rule applications.

3.7.6 ELAN

According to the authors, “The ELAN system provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies.” As such, it strongly resembles Stratego.

Compared to Stratego, ELAN is more focused towards formal proofs and constraint solving:

“It offers a natural and simple logical framework for the combination of the computation and deduction paradigms as it is backed up by the concepts of rewriting calculus and rewriting logic.”

“It permits to support the design of theorem provers, logic programming languages, constraint solvers and decision procedures and to offer a modular framework for studying their combination.”

To this end, it offers two kinds of rewrite rules: *unlabeled rules*, which are applied using a fixed innermost strategy, and *labeled rules*, which are applied according to user-defined strategies.

As ELAN does not support generic traversals, it does not stand up to our requirements.

3.7.7 Blitz++

The goal of the Blitz++ project is “to develop techniques which will enable C++ to rival – and in some cases even exceed – the speed of Fortran for numerical computing, while preserving an object-oriented interface.”

To that end, the Blitz++ numerical library has been developed. Blitz++ uses templates to do loop fusion, unrolling, tiling, and algorithm specialization can be performed automatically at compile time, it is an example of an augmented (or active) library.

Blitz++ thus compares more directly with the Sophus+CodeBoost combination than with CodeBoost directly.

The core optimization mechanisms used in Blitz++ are expression templates [Vel95b], template meta-programming and generative programming.

A side-effect of the template mechanism in C++ is that it can be used to perform partial evaluation at compile-time [Vel95a]. Basic control structures such as if/else, loops and case switches are available. These structures can be used to do selective code generation, as the compiler “interprets” these structures at compile-time. This technique is called *template meta-programming*.

The template mechanism can also be used to inline expressions into function bodies, removing the overhead of callback functions. This can be applied to calculating vector and matrix expressions in a single pass, without temporaries.

Most abstractions in Blitz++ are generic; they are parameterized objects and functions designed to be customized at use-time. They are programmed using the techniques mentioned above, to take advantage of the C++ template mechanism, so as to remove the performance penalty of abstraction.

Compared to Sophus and CodeBoost, Blitz++ is limited to the power available in the C++ template system. The syntax cannot be extended, nor can arbitrary transformations be specified. The optimizations and abstractions are tightly interwoven. The Blitz++ library maintainer is only required to have a good knowledge of the C++ template mechanism, whereas for Sophus+CodeBoost, good knowledge of Stratego is also required to extend and maintain the optimizations.

3.7.8 Simplicissimus

The authors introduces the Simplicissimus project thus:

For abstract data types (ADTs) there are many potential optimizations of code that current compilers are unable to perform. These op-

timizations either depend on the functional specification of the computational task performed through an ADT or on the semantics of the objects defined. In either case the abstract properties on which optimizations would have to be based cannot be automatically inferred by the compiler. In the *Simplicissimus* project our aim is to address this level-of-abstraction barrier by showing how a compiler can be organized so that it can make use of semantic information about an ADT at its natural abstract level.

In [SGML01], they present an expression simplifier for user-defined data types. They use the Matrix Template Library (MTL) [SL998], written in C++, as their subject body of code.

Their approach is concept-based, with concept defined as follows.

Definition 3.1

Concept: A set A of abstractions together with a set R of requirements, such that an abstraction is included in A iff it satisfies all the requirements in R.

Their simplifier is equipped with a given set of rules, that are decoupled from the types they apply to. Each rule has a set of requirements that must be met for it to be applicable.

Users later provide formal descriptions of their data types, using an in-syntax specification language, i.e. the descriptions are written as C++ code.

Given the rule requirements and data type descriptions, *Simplicissimus* will determine when its various rules are applicable.

In summary, the the compiler is viewed as a compilation generator, and users are allow to extend the semantic knowledge available to the compiler. Their primary target group is library designers, who are usually able to justify the extra the cost of writing the data descriptions for their library data types.

The implementation makes use of advanced C++ techniques, such as expression templates, template meta-programming, and traits (interface templates), and separate pipeline step for the GNU C++ compiler.

Simplicissimus is not a complete transformation framework. It is rather an add-on to the GNU C++ compiler that works on its internal TREE representation [Cod00]. Similarly, we could replace our OpenC++ front-end and plug the transformation part of CodeBoost into the GNU C++ pipeline. We have already rejected this approach, partly because the GNU C++ compiler emits suboptimal binary code on the platforms we use, and partly because its template handling is not fully ANSI compliant.

The in-syntax descriptions that allow for expression rewriting are in some ways similar to the concept of user-defined rewrite rules, available in CodeBoost. [Bag03] demonstrates a transformation where the user can write simplification

rules (user-defined rules) inside the C++ syntax. This lowers the entry barrier for writing simple transformations considerably, as the user no longer needs any knowledge of Stratego or the CodeBoost framework.

3.8 Summary

The CodeBoost transformation framework is a general source-to-source transformation framework for the C++ language. To show its real-world applicability, we have decided to implement a set of domain-specific transformations for programs written in the algebraic style.

The internal workings of CodeBoost mirror that of a regular compiler, having a pipeline with preprocessing, parsing, transformation (usually called optimization step for regular compilers) and an output steps. The main difference between CodeBoost and a regular compiler is that CodeBoost's transformations are higher-level and domain-specific, and its output is in the same language as the input.

When the transformation modules are written with attention towards separating the transformations from the traversal strategies, extracting and reusing parts of them becomes considerably easier. Often-used strategies and rules are good candidates for inclusion into the CodeBoost standard library.

A more detailed overview of CodeBoost can be found in [BHV01]. The most complete description of its implementation, background and inner workings to date is given in [Bag03].

Chapter 4

Transformations

In this chapter, we present the theoretical groundwork for the transformations we have implemented.

We start by discussing the decision to group the transformations with the library instead of the compiler, and its implications.

We then proceed to present the most important paradigms we have been inspired by.

Next, we discuss a simple transformation that performs algebraic simplification and round off with the experienced shortcomings.

4.1 Augmented libraries

We have earlier argued that a library as collection of classes does not alone fulfill the stated goals of the Saga project.

In particular, the classes alone cannot fulfill the requirement for improved performance. Also, the Sophus style dictates that some operators may be specified implicitly, for notational convenience. If the operator $+=$ is specified, the operator $+$ must be automatically generated. Previous experience told us that a mere class library could properly handle these cases.

In order to accommodate this, we have augmented our library with abstraction-specific transformations. These are transformations are then applied to programs written for the Sophus library through a process called *boosting*. [CEG⁺] calls this an “active library”.

Definition 4.1

The application of (potentially) performance-improving transformations to the source code of a program, resulting in a semantically equivalent program, also on source code form, is termed boosting.

A boosted program may exhibit significantly improved running times compared to their unboosted versions, but the boosting process may also be done for notation convenience or to generate parts of the program from specifications.

The Broadway compiler [GL99] has provided inspiration for our work. The Broadway approach consists of an annotation language and a modified compiler that together can tailor a library to a specific application.

The annotation language describes properties of the functions and data types provided by the library. The annotations are used to rank functions by the specificity. A function multiplying a diagonal matrix by a general matrix is considered more specific than a function multiplying two general matrices.

A program written for the annotated library is run through the Broadway compiler, along with the library annotations. The compiler then rewrites occurrences of general function applications to more specific function applications. This generally results in improved running times, as the specialised functions are less computationally expensive than their general counterparts.

4.1.1 Source-to-source transformations

In the realm of high performance computing (HPC), each computer vendor usually supplies his own computer suite. We have mentioned earlier that there is no standardised API for plugging new optimisations and transformations into these compilers.

As one of our requirements for CodeBoost was that it would work on different HPC computers, the only option left was to implement it as a source-to-source transformer on the subject language (C++).

After developing the transformations outlined in this thesis, we have had very good experience with the source-to-source approach.

Its advantages can be summarised as:

- The resulting code is human-readable.
- The resulting code can be fed directly into most C++ compilers.
- We can decide to only transform parts of a program, then use the C++ pre-processor to mix transformed and non-transformed code.
- We did not need to invent a new intermediate language.

We do not currently anticipate any obstacles that will require a different representation than C++ source code for the transformed result.

4.1.2 Program traits

As we concern ourselves primarily with the functional and object-oriented styles of programming, we are limited to a fairly well known set of language abstractions. The most interesting of these are *classes*, *templates*, *functions* and *expressions*.

Each of these abstractions can have various traits. For instance, a method within a class can be declared private. It has a given return type. It has a given set of formal parameters. In the C++ language, each abstraction's possible traits are clearly defined beforehand.

We also consider invariants as program traits. We define an invariant:

Definition 4.2

*Invariant: A rule, such as the ordering of an ordered list or heap, that applies throughout the life of a data structure or procedure. Each change to the data structure must maintain the correctness of the invariant.*¹

For programs written for the Sophus library, we want our transformer to be informed of these invariants, so we need to extend the subject language with a mechanism for specifying these traits.

Please observe that extending the language with new semantics like this does not imply extending the syntax as well. We will show later that it is possible to specify additional traits for the various abstractions within the regular C++ syntax. We will also comment on where it would be preferable to have the luxury of changing the language syntax.

Testing that invariants hold is a cherished software engineering technique for introducing some stability scaffolding into the program that has recently come into vogue again, in the form of extreme programming. In particular, see [BF01] and [JAH01].

If an invariant is known to hold at a particular point in the program, we can sometimes use that information to optimise the code around that point.

In some ways, this relates to aspect-oriented programming [KLM⁺97]. We are concerned with separating the abstract mathematical properties of our objects from the optimisation opportunities that arise when combining them. We want programmers to write formally correct programs, then apply our cross-cutting transformations to obtain a program with improved running time.

4.1.3 Domain-specific optimisations

With our approach, the key to good performance is domain-specific optimisations. We set a premise that the compiler that ultimately produces the binary code does

¹From Free On-Line Dictionary of Computing

an adequate job when provided with well-massaged code.

One of our goals is to show that augmenting high-level libraries with suitable domain-specific optimisations is feasible given access to proper transformation tools.

Another goal is to show that it is possible to write clean, high-level code without sacrificing runtime performance.

4.2 Inspirational paradigms

4.2.1 Partial evaluation and program specialisation

Given detailed knowledge about our program's mathematical abstractions we can automatically specialise both the code and the data structures to fit the problem at hand better.

Given the expression $C = A \circ B$, where A, B and C are matrices, and we know that A is a diagonal matrix, we can safely transform the code into $C = \text{diagonalMultiply}(A, B)$ without loss in program meaning (the `diagonalMultiply` function is assumed to be hand-written in advance).

Thus, if we know various characteristics of the data our program works on, we can specialise our program to work more optimally on those specific data.

The paradigm of partial evaluation concerns itself with problems of this kind. By performing partial evaluation, we evaluate and simplify as much of the program as possible at compile time (we can seldom evaluate everything at compile-time, hence "partial"), so that the resulting program has a (often much) shorter running time.

In Fig. 4.1, we have a *subject program* with two inputs; *static* and *dynamic*. The static input we know at compile-time, so we are able to specialise our subject program to that part of the data. The result of this specialisation is the *specialised program*, which will still accept the dynamic input, to produce *output*

The program specialiser is not allowed to change the semantics of the program. Given the program p , static input in_1 and dynamic input in_2 , output out is given by Equation 4.1.

$$out = [[p]] [in_1, in_2] \tag{4.1}$$

That is, p applied to inputs in_1 and in_2 gives output out .

Similarly, the partial evaluation program itself, named `mix` can be applied to inputs p and in_1 to obtain the specialised program p_{in_1} , see Equation 4.2.

$$p_{in_1} = [[mix]] [p, in_1] \tag{4.2}$$

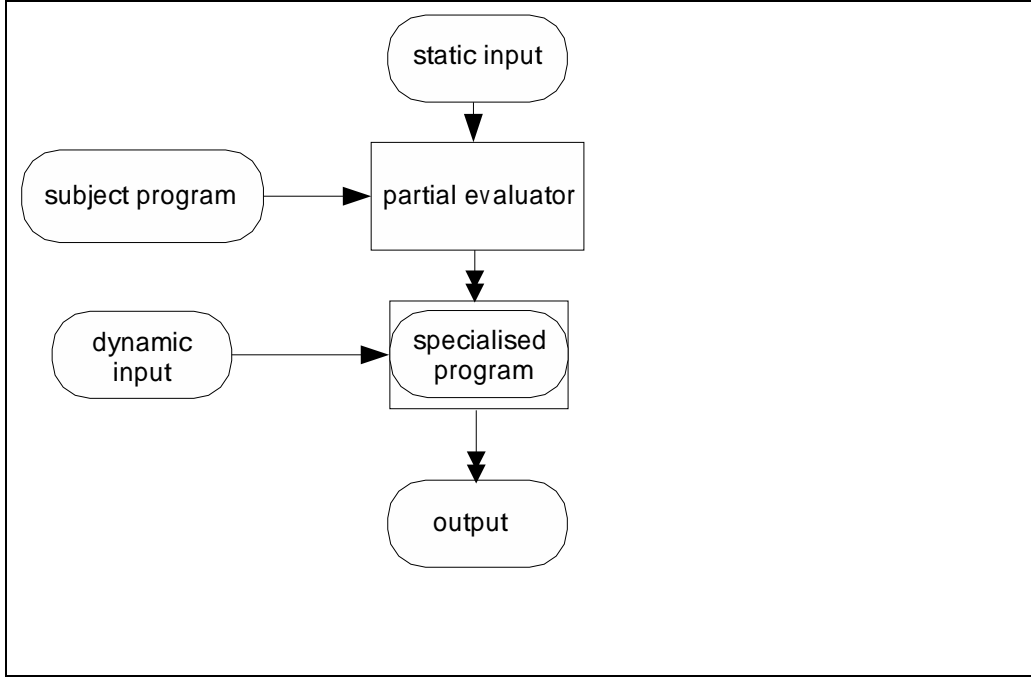


Figure 4.1: A partial evaluator

The specialised program will, when applied to the dynamic input, yield the *out* we had earlier, as shown in Equation 4.3.

$$out = \llbracket p_{in_1} \rrbracket [in_2] \quad (4.3)$$

Thus, we have the equality in Equation 4.4.

$$\llbracket p \rrbracket [in_1, in_2] = \llbracket \llbracket mix \rrbracket [p, in_1] \rrbracket [in_2] \quad (4.4)$$

A more thorough introduction to partial evaluation can be found in [JGS93].

The *mix* program above is typically an interpreter for the language in which the subject program was written. In our case, we have only implemented a partial evaluator for a restricted set of operators on a restricted set of data types.

Chapter 8 details our implementation of a transformation that specialises the matrix multiplication (\circ) and point-wise operations (point-wise $+$, $-$, $*$, $/$) on matrices.

4.2.2 Generative programming

According to [KC00], generative programming is “a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate

or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge”.

We have made a transformation for automatic symbolic differentiation of simple algebraic expressions. It is far from a “software system”, but it was inspired by the concept of automatically generating a specific family of functions given a well-defined rule-set.

Currently, the building blocks of our “end-product” are the primitive mathematical abstractions of real numbers, but with some additional work, we should be able to differentiate over the mathematical abstractions available in Sophus as well (primarily tensors and meshes).

Given the function `float f(float x) { return x*x; }` the user may apply the differentiation operator to this function: `main() { float k = 10; D<f,1>(k); }` and have CodeBoost automatically generate the differentiated function `template <float f(float), int> D(float x) { return 2*x; }`

The differentiation operator `D` takes as arguments the function to be differentiated (in our case `f`) and then a list of parameters to differentiate on (here, only the first).

Naturally, only the derivations requested will actually be generated.

Chapter 6 gives a detailed description of this transformation.

It must be mentioned that there is a branch of computer science named program derivation, first described in [BD77]. It concerns itself with generating an efficient program from an initial specification using semantically preserving transformations.

Our goal for the differentiation operator is not efficiency but removing mechanical coding work. We therefore argue that the transformation is closer to the paradigm of generative programming.

4.3 Algebraic simplification

Both the function differentiation transformation (see Chapter 6) and the matrix transformations (see Chapter 8) generate algebraic expressions that are not optimal.

For instance, the differentiator may emit code as shown in Fig. 4.2.

```
(((0 * x) + (42.0 * 1)) * (x + 10)) + ((42.0 * x) * (1 + 0))
```

Figure 4.2: A generated expression from `diff`

If `x` is one of the primitive types, the compiler is perfectly able to simplify the expression on its own. However, if `x` is a user-defined type, the normal sim-

simplification may no longer apply, so the C++ compiler cannot perform any further optimisation.

The `simplify2` transformation module implements a few very simple algebraic simplification rules, which holds for all mathematical objects that are rings.

The rules are given in Fig. 4.3. Additionally, we evaluate constants, i.e. the expression `10 * 10` is rewritten to `100`.

$1 * x \rightarrow x$	$0 + x \rightarrow x$	$x * 1 \rightarrow x$	$x + 0 \rightarrow x$	$0 * x \rightarrow 0$
$0 + x \rightarrow x$	$x * 0 \rightarrow 0$	$0 - x \rightarrow -x$	$x - 0 \rightarrow x$	$x / 1 \rightarrow x$

Figure 4.3: The simplification rules

The transformation starts by loading the configuration file for this transformation, then continue by applying the simplification rules in a bottom up manner to all terms of the program.

The configuration file is simply a list of which data types are to be considered for simplification. By default, only the built-in types `float`, `int`, `long` and `double` are considered.

The simplification rules themselves are very short. Fig. 4.4 lists the rules for simplification over the multiplication operator.

<code>simplify-times: Infix(Op(" * ", _), e, n) → e</code> <code>where <is-one> n ; <type-of-expr ; is-ring> e</code>
<code>simplify-times: Infix(Op(" * ", _), n, e) → e</code> <code>where <is-one> n ; <type-of-expr ; is-ring> e</code>
<code>simplify-times: Infix(Op(" * ", _), e, n) → n</code> <code>where <is-zero> n ; <type-of-expr ; is-ring> e</code>
<code>simplify-times: Infix(Op(" * ", _), n, e) → n</code> <code>where <is-zero> n ; <type-of-expr ; is-ring> e</code>

Figure 4.4: `simplify-times`

The strategies `is-one` and `is-zero` check if the given expression evaluates to one or zero, respectively. For variables of user-defined types, this gets a bit tricky, as the only way to know this is by tagging the variables. The next section explains the details of our tagging mechanism.

`is-ring` ensures that the expression is of a type we know can be simplified.

If one spends time inspecting the `simplify2.r` code, it becomes apparent that performing simple algebraic transformations like the one presented here is

nearly trivial using the CodeBoost framework. No significant helper strategies are introduced.

For many of the trivial cases here, this transformation could easily have been implemented as a collection of user-defined rules (see [Bag03]), but then we would need a separate constant folding step afterwards. Furthermore, we suspect that the simplification transformation described here will be extended significantly with various identities on Sophus data types, which may prove difficult to specify accurately using user-defined rules.

4.4 Shortcomings

After finishing the simplification transformation, we consider the the following to be immediate candidate areas for improvement:

- *handle complex data types*; The algebraic simplification transformation is currently very simple. It would be desirable to extend it to simplify all operators for the Sophus-specific types, meshes in particular.
- *simpler totem interface* ; The totem subsystem is currently completely decoupled from the CodeBoost symbol table. The reason being that totems do not in general follow the scoping rules of variables and functions. This makes it necessary to intermix the totem collector with the traversals, which is a nuisance.

When trying to make the totem mechanism completely transparent for everyday use, one bumps into the problem that some transformations require their own, special totem tracking rules. Thus, we would need some kind of hookable traversal mechanism where the transformation is allowed to hook in its own totem collection and propagation rules.

Another aspect not thoroughly evaluated yet, is the necessity of the totem mechanism itself. Essentially, totems are annotations on the AST, that change during the course of transformation. It may turn out that parsing the configuration files and replacing the in-syntax occurrences of `CB_IMPORT`, `CB_TAG` with suitable tree-annotations will be sufficient.

Separating the totem propagation from the transforming traversals would then be easier, as the totem state would not need to be reconstructed on each traversal; a totem propagation pass may be inserted before the transforming traversal, and its result will be stored at the relevant terms.

4.5 Summary

We have presented two simple transformations for the CodeBoost transformation framework. The first one, an algebraic simplifier, is a stand-alone transformation intended to be plugged in after the differentiation and matrix optimisation transformation described later in this thesis.

The second one, a totem tracking mechanism, is intended to be used by other transformations, mainly as a very simple aid to tracking data-invariants throughout the source code.

Writing the totem system and the algebraic simplification transformation proved pretty simple, given a modicum of Stratego knowledge. The code is available in the files `src/trans/simplify2.r` and `src/trans/totem.r`.

It is not entirely clear whether the customizability offered by the CodeBoost configuration format is ultimately practical in the real world, at least for small transformations as the ones presented in this chapter. Given the fact that the CodeBoost source code is freely available, it may be easier to provide clearly commented modification spots in the transformation source code itself.

Chapter 5

TigerBoost

In this chapter, we present the TigerBoost transformation tool for programs written in the Tiger language.

We start by outlining the motivation for TigerBoost. We then present the transformations we implemented, and our experiences from doing so. We conclude with a comparison with CodeBoost and discuss the continued need for TigerBoost.

5.1 Motivation

TigerBoost was created out of a desire to learn Stratego while at the same time gain worthwhile knowledge on transformations ultimately relevant to the Sophus library.

TigerBoost can be thought of as a very primitive transformation system for Tiger [App98] code. It is thus not immediately useful for Sophus, which is written in C++.

However, compared to CodeBoost, the simpler transformation tool laid out in TigerBoost allowed for rapid prototyping and experimentation, especially with respect to matrix optimisations and totems.

In reality, TigerBoost is nothing more than very slight adaptation of Eelco Visser's Tiger front-end as it appeared in summer/autumn 2001. Its inception took place at a meeting between M. Haverlaen, O. Bagge, E. Visser and myself in June 2001, where E. Visser pointed out that toying with his Tiger front-end would probably be a productive route towards gaining better familiarity with Stratego.

5.2 Transformations

We implemented four fairly easy transformations in TigerBoost, each outlined briefly below.

5.2.1 Constant propagation and folding

The concept of constant propagation is fairly trivial and a well-known optimisation technique employed in virtually all modern compilers. In TigerBoost, we only exploit constant propagation to evaluate simple algebraic expression at compile-time and to unroll loops.

Given the code in Fig. 5.1 TigerBoost would perform both constant propagator

```
let
  var a := 10
  var b := 20
in
  c := a + b * b
  d := c
end
```

Figure 5.1: Tiger code fragment

and algebraic simplification to yield the code in Fig. 5.2.

```
let
  var a := 10
  var b := 20
in
  c := 410
  d := 410
end
```

Figure 5.2: After constant-propagation and folding

In and of itself, the constant propagation transformation is not very interesting, but as we shall see, it is vital for some of the transformations covered later in this chapter.

5.2.2 Elementary totem propagation

The full concept of totems is explained in Section 3.5. The first implementations were done using TigerBoost, and immediately told us that we could easily give high-yield hints to the optimiser without extending the syntax of the source language.

Totems are tags we put on variables and functions to explain to TigerBoost properties of the code that are otherwise difficult or impossible to infer. Given

these totems, TigerBoost can do more invasive optimisations than would otherwise be possible.

As the following code shows, totems in TigerBoost follow the normal language scoping rules. Given the code snippet in Fig. 5.3,

```

let
in
  let
  in
    setTotem(A, diagonalMatrix())
    setTotem(B, diagonalMatrix())
    C := matrixMul(A,B) ;
    D := matrixMul(A,C)
  end ;
  matrixMul(A,B)
end

```

Figure 5.3: Tiger code fragment

TigerBoost will transform it into the optimised snippet in Fig. 5.4.

```

let
in
  let
  in
    C := matrixMulDiag(A,B) ;
    D := matrixMul(A,C)
  end ;
  matrixMul(A,B)
end

```

Figure 5.4: After totem propagation

Only the first `matrixMul` can be optimised, as we do not know the layout for `C`, and the totems are no longer valid once we leave the inner `let`-block.

Note that the rewrite from `matrixMul(A, B)` to `matrixMulDiag(A, B)` if both `A` and `B` are known to be diagonal matrices is an explicit rule in TigerBoost.

5.2.3 Loop unrolling

Unrolling loops is another very well-known optimisation technique, especially when dealing with high-performance compilers intended for numerical applications.

Our loop-unroller is very straight-forward. Given the code in Fig. 5.5, the

```
let
in
  for x := 0 to 5 do
    j := x
  end
```

Figure 5.5: A simple loop

resulting code after run through TigerBoost will be as given in Fig. 5.6.

```
let
in
  j := 0 ;
  j := 1 ;
  j := 2 ;
  j := 3 ;
  j := 4
end
```

Figure 5.6: After loop unrolling propagation

Again, this is not a very exciting optimisation in and of itself. Its power becomes apparent when we combine it with the other transformations presented earlier.

5.2.4 Optimising matrix multiplication

Putting the previous optimisations together, we can now perform a potentially high-yield optimisation on a the very common matrix multiplication operator.

We have already showed one way to optimise the multiplication operator in Section 5.2.2; Given the tag `diagonalMatrix` on matrices `A` and `B`, the expression `matrixMul(A, B)` was rewritten to `matrixMulDiag(A, B)`.

This was further extended upon in TigerBoost to account for properties on symmetrical, unit, lower and upper matrices. The pattern was simple; given property `x` on `A` and `y` on `B`, we can rewrite `matrixMul` as shown in Fig. 5.7

The table tells us that `matrixMul(A, B)`, where `A` is diagonal and `B` is symmetrical, can be rewritten to `matrixMulSymm(A, B)`¹.

¹The full table of combinations was never actually implemented in TigerBoost, just sufficient amounts to know that it was doable.

x/y	unit	diagonal	symmetrical	upper	lower
unit	unit()	B	B	B	B
diagonal	A	diag(A,B)	symm(A,B)	upper(A,B)	lower(A,B)
symmetrical	A	symm(A,B)	symm(A,B)	upper(A,B)	lower(A,B)
upper	A	upper(A,B)	upper(A,B)	upper(A,B)	mul(A,B)
lower	A	lower(A,B)	lower(A,B)	mul(A,B)	lower(A,B)

Figure 5.7: Simple rewrite rules for matrix multiplication

The drawback of this method is that as we add combinations of x and y to the table, we need to hand-code a specialised version of `matrixMul`. It would be preferable if, given a detailed layout of A and B , we could automatically generate an optimal version of `matrixMul`.

Armed with the transformations described in the previous sections, this proved fairly easy.

The way we approached the problem was by inlining the `matrixMul` function when we encounter it, then applying constant propagation and folding, loop-unrolling and then finally some specific rewriting rules for accessing the matrix elements.

This clearly resembles a partial evaluation of the loop-nest. Its performance gain is governed largely by the sparseness of the matrices involved.

Let's step through the details for the code given in Fig. 5.8.

The `a_2` element in `matrixLayout` signifies that the element is non-zero, but that its exact value is unknown.

Thus, we see that B is a diagonal matrix, whereas A is symmetrical.

TigerBoost will conceptually proceed as follows:

1. Tag A , B and C with the totems described.
2. Inline functions
3. Unroll loops
4. Simplify matrix accesses
5. Propagate any constants
6. Algebraic simplification

The workings of transformations 1,2,3 and 5 should be apparent.

²We would ideally have liked to use `_`, but the Tiger grammar does not allow `_` by itself to be an identifier.

```

let
function matrixMul(A, B) =
  let var T : matrix := C
  in
    for i := 0 to T.x do
      for j := 0 to T.y do
        for k := 0 to T.x do
          C.d[k].d[i] := C.d[k].d[i] + A.d[k].d[j] * B.d[i].d[k];
        end
      end
    end
  in
    let
      var A := createMatrix(4,4)
      var B := createMatrix(4,4)
    in
      setTotem(A, matrixDim(4,4));
      addTotem(A, matrixLayout(a_, a_, a_, a_,
                               a_, a_, 0, 0,
                               a_, 0, a_, 0,
                               a_, 0, 0, a_));
      setTotem(B, matrixDim(4,4));
      addTotem(B, matrixLayout(a_, 0, 0, 0,
                               0, a_, 0, 0,
                               0, 0, a_, 0,
                               0, 0, 0, a_));
      setTotem(C, matrixDim(4,4));
      C := matrixMul(A,B);
    end
  end
end

```

Figure 5.8: A more complete matrix example

After inlining the function in step (2), the loops can be unrolled completely, as the value of $T.x$ and $T.y$ are known from the dimensions of C specified by `setTotem(C, matrixDim(4,4))`.

After this loop unrolling, we have a sequence of expressions such as `C.d[0].d[1] := C.d[0].d[1] + A.d[0].d[2] * B.d[1].d[0];` These expressions are subjected to a matrix access simplification transformation.

The transformation rule `MatrixAccessSimplify` considers an expression of the form `M.d[i].d[j]`. The matrix M 's element layout, given by a previous `matrixLayout` tag, is looked up. If the value at $M[i, j]$ is a constant, the expression `M.d[i].d[j]` is replaced with this constant.

For `MatrixAccessSimplify`, M must have a `matrixDim` tag, a `matrixLayout` tag and the indexes must be within their valid ranges (although our prototype transformation does not verify this).

Thus, `C.d[0].d[1] := C.d[0].d[1] + A.d[0].d[2] * B.d[1].d[0];`, given the layouts above, would be translated into `C.d[0].d[1] := C.d[0].d[1] + A.d[0].d[2] * 0;`³, which would in step (5) be rewritten to `C.d[0].d[1] := C.d[0].d[1];`, and in step (6) be removed altogether.

The net result is then as given in Fig. 5.9.

```

C.d[0].d[0] := C.d[0].d[0] + A.d[0].d[0] * B.d[0].d[0] ;
C.d[0].d[0] := C.d[0].d[0] + A.d[0].d[1] * B.d[0].d[0] ;
C.d[0].d[0] := C.d[0].d[0] + A.d[0].d[2] * B.d[0].d[0] ;
C.d[0].d[0] := C.d[0].d[0] + A.d[0].d[3] * B.d[0].d[0] ;
C.d[1].d[1] := C.d[1].d[1] + A.d[1].d[0] * B.d[1].d[1] ;
C.d[1].d[1] := C.d[1].d[1] + A.d[1].d[1] * B.d[1].d[1] ;
C.d[2].d[2] := C.d[2].d[2] + A.d[2].d[0] * B.d[2].d[2] ;
C.d[2].d[2] := C.d[2].d[2] + A.d[2].d[2] * B.d[2].d[2] ;
C.d[3].d[3] := C.d[3].d[3] + A.d[3].d[0] * B.d[3].d[3] ;
C.d[3].d[3] := C.d[3].d[3] + A.d[3].d[3] * B.d[3].d[3]
```

Figure 5.9: Completely optimised code

5.3 The relation to CodeBoost

Calling TigerBoost a transformation framework is would be a gross boast. There is no concept of a tunable transformation pipeline, no symbol table exists, there

³In Fig. 5.8, we see that `B.d[1].d[0]` is 0

are no provisions for printing sensible error messages, etc.

We only have the option of running a pre-defined set of transformations once the tool is compiled. The order cannot be changed without a recompilation. In the prototyping phase, this did not present a problem, but it would clearly be inadequate for everyday use.

As Tiger is a very simple language, the transformations can be simple; they do not need to deal with the complex type system of C++, as we do in CodeBoost. This has at least two aspects to it:

lower implementation complexity; if we can formulate our example programs using simple types (structs) and functions, TigerBoost transformations are usually very small and neat. We experienced that refactoring the code to accommodate changes in the conceptual idea for a given transformation was vastly easier for the Tiger language, compared to equivalent refactorings with CodeBoost.

missing language constructs; there are numerous aspects we cannot fully investigate due to Tiger's small size. Of notable interest is genericity (templates), inheritance and operator overloading. The mutation transformation would make no sense for Tiger.

Our net experience shows that if the problem can be satisfactorily formulated as a Tiger program, and that the desired transformation is non-trivial, doing an initial prototype with TigerBoost is well worth the effort.

However, we must emphasise that transporting the transformations from TigerBoost to CodeBoost is generally not possible. While Stratego brings with it the possibility of generic, language-independent strategies, this cannot always be utilised; the underlying languages are too dissimilar.

It has proven more practical to consider the transformations written in Tiger as throw-away prototypes⁴, and reimplement the CodeBoost formulation from scratch.

5.4 Planned obsolescence

As mentioned, TigerBoost was a fork off an early implementation of E. Visser's Tiger compiler [Vis01b]. It has not been updated to the more recent versions of this compiler, nor has it been updated to compile cleanly with the recent releases of the Stratego language.

The only version of XT [MdJ03] it is known to work with is 0.8.

The lack of updates has been deliberate. It has not been deemed worthwhile to keep TigerBoost current throughout the year since its creation. Focus has been on

⁴As Fred Brooks said: "plan to throw one away, you will anyway."

CodeBoost. Reviving TigerBoost should not be particularly difficult, if a genuine need for it arises.

5.5 Summary

Our experience with TigerBoost, and later with CodeBoost, tells us that implementing transformations for the Tiger language is (not surprisingly) significantly easier.

It was therefore in our opinion a huge win to experiment with various approaches to totem tagging and propagation in Tiger, as it allowed us to cover more ground easier.

Furthermore, the loop-optimisations show a nice theoretical gain. Since we know that the algebraic operations on matrices are mainly implemented as loop-nests, any improvements on loop structures are potentially valuable.

We managed to show that generating both fully unrolled code for very sparse matrices and generating specialised loops for the multiplication of matrices was feasible using the Stratego language. The simplest approach where we only rewrote `matrixMul` to a hand-written specialised version, such as `matrixMulDiag` is nearly trivial to implement, given a working totem mechanism.

Fully evaluating the matrix multiplication and generating unrolled, optimised code was also fairly straight-forward. However, we suspected at the time that for larger matrices of intermediate sparseness, re-rolling the code back into loops would probably be a worthwhile attempt.

Transporting these optimisations over to CodeBoost proved worthless. Given the simplicity of the Tiger language and also the simplicity of the transformations outlined here, it was decided to rewrite the transformations from scratch within the CodeBoost framework.

Experience with both TigerBoost and CodeBoost shows that changing all or significant parts of a transformation written for TigerBoost is considerably easier than doing similar changes on transformations written for CodeBoost.

Given this fact, we would recommend TigerBoost only when rapid development of “mock-up” transformations is desirable. If a fairly clear understanding of the transformation is available, implementing directly for CodeBoost is recommended.

Chapter 6

Symbolic differentiation

In this chapter, we present a transformation for doing automatic program differentiation.

We start by giving our motivations for implementing the transformation, continue with an explanation of the implementation, then conclude with a discussion on the current shortcomings of its design and implementation.

6.1 Motivation

It has been shown elsewhere [Bag03] that CodeBoost is suited for implementing optimising transformations.

In this chapter we seek to show that CodeBoost also can be applied to the field of generative programming, as introduced in Section 4.2.2.

In particular, we consider the problem of automatic program differentiation.

Definition 6.1

By automatic program differentiation, we mean the process of applying the established mathematical rules for differentiation to algebraic expressions in the source code.

Our goal is to demonstrate that CodeBoost can be employed to reduce the amount of trivial programming and maintenance required by the developer.

6.2 Overview

We introduce to the C++ language the differentiation operator D , disguised as a template function. Application of the operator amounts to a function call in the subject language.

The prototype for the operator is on the form $D\langle f, n_0, n_1, \dots, n_N \rangle (a_0, a_1, \dots, a_M)$ where

- f is a function reference
- n_0, \dots, n_N are indexes into a_0, \dots, a_M
- the function $f(a_0, a_1, \dots, a_M)$ must already be defined

Thus, conceptually D can be applied to the function f , to yield an expression for f' . The indexes n_0, \dots, n_N are used to control which variables to differentiate with respect to. Given the application $D\langle f, 1, 3 \rangle (u, v, w, z)$, the operator yields the expression for $\frac{d}{dw} \frac{d}{du} f(u, v, w, z)$.

```
double f(double x, double y, double z) { return x*y*z; }
usage(double u, double v, double w) { D<f,1>(u,v,w); }
```

Figure 6.1: An application of D

As we can see, this definition only allows for differentiating named expression, i.e. functions. We currently have no analogue for $[expr]'$. Implementation-wise, we know of no known obstacle for doing this, although the syntax for the operator itself would have to change slightly.

The currently implemented form is more applicable to Sophus. If the expression is non-trivial enough to warrant automatic program differentiation, it should be put into a separate function body anyway.

As $D\langle f \rangle$, which in reality is a convoluted function call to f , is replaced with a strongly modified version of f , the transformation cannot be seen as semantically preserving from within the C++ language semantics.

Ideally, we would like to compile programs containing the D operator independently of CodeBoost. If we defined D as a template function that performed numerical differentiation when not boosting, the program would not be reliant on CodeBoost to run.

In the following sections we shall present the implementation in detail, show some examples of its application and discuss its current shortcomings.

6.3 Implementation

The code for the differentiation transformation resides in `src/trans/diff.r`. It will accept any valid C++, and try to perform the following steps on its input:

1. Collect all applications of the differentiation operator.

2. Collect all function bodies on which the diff operator is applied.
3. Evaluate and inline result of diff operator applications.
4. Repeat cycle.

The following definition will come in handy:

Definition 6.2

A function that can not be automatically differentiated, nor is allowed to be differentiated numerically, is classified as non-differentiable.

If step 1 does not uncover any applications of the diff operator, no (further) transformation is performed.

If step 2 uncovers the application of the diff operator to a non-differentiable function, an error will be emitted and the transformation fails.

Step 3 may result in the generation of an application of the diff operator, therefore we repeat the entire cycle until we have exhaustively evaluated all diff operator applications.

Before detailing the steps above, we shall discuss the criteria for differentiability.

6.3.1 Function constraints

The current version of the function differentiator is fairly strict with regards to which kinds of functions it is willing to differentiate.

One must manually insert a numerical approximation on the functions it fails to differentiate automatically. In that case, a call to the template function `Dn`, shown in figure Fig. 6.2, may be attempted.

The function parameter `double f(double, double)` in the template function `Dn` must be replaced with the prototype of the function we want to differentiate.

Both precision- and performance-wise, this replacement may not be satisfactory. One may end up hand-coding the differentiated function in the cases where the transformation fails.

As previously stated, only fairly simple functions are candidates. They all must be on the form `T0 f(T1 a0, T2 a1, ..., Tn an) { return <expression>; }`

Here, `T0` through `Tn` must be types which have the appropriate operators used in `<expression>` defined on them.

That is, if `a0` is `complex`, and the `<expression>` is `sqrt(a0)`, the differentiated expression is `1 / (2 * sqrt(a0) * 1)`. It is then assumed that type-promotion operators from `int` to `complex` are defined, and that `T0` is of an appropriate type.

```

template <double fn(double,double), int d1>
double Dn(double u, double v) {
    if(d1==1) {
        double du = 0.001;
        double y1 = fn(u+du,v);
        double y0 = fn(u,v);
        return (y1 - y0) / du;
    } else if(d1==2) {
        double dv = 0.001;
        return (fn(u,v+dv) - fn(u,v)) / dv;
    } else {
        std::cerr << "Derivation parameter out of bounds" << endl;
        exit(-1);
    }
}

```

Figure 6.2: Numerical differentiation of the first order, using Euler differentiation

Currently, this checking is pretty lax, so for cases where these assumptions do not hold, we leave it to the compiler to emit appropriate errors. This is not ideal, as:

- The compiler will emit error messages with line numbers referring to the generated code, which the user is never supposed to look at.
- The error message can be arbitrarily cryptic, as the error pertains to the generated code, not the original user code.

It would be preferable if most of the common mistakes could be caught at transformation time, and flagged as errors there, when enough knowledge is around to emit sensible error messages. This is material for further work.

Next, we will detail the steps in the transformation.

6.3.2 Collect diff operator applications

Step (1) traverses the entire program's AST in a top-down manner, looking for function applications on the form $D\langle f, i, j, \dots \rangle(x, y, \dots)$.

On each encountered application, the required signature of f is calculated, and put into a collection.

At the end of the traversal, we have a list of all functions in the program that needs to be symbolically differentiated.

6.3.3 Collect function bodies

Step (2) does another top-down traversal, this time collecting all function bodies matching the function signatures calculated in step (1).

6.3.4 Evaluate and inline result

Step (3) does the final top-down traversal. Again, we search for all applications of the D operator, just as in step (1). This time around, we have collected sufficient information to replace the operator application with the resulting differentiated expression.

This in itself is a staged process:

- Inline function body for f
- Differentiate expression, on each index in turn.

Inline function body

This stage is straightforward. We inline the function body of f , and do appropriate variable substitution.

Consider the function in Fig. 6.3.

```
double f0(double x, double y, double z) { return x*y*z; }
```

Figure 6.3: A sample formula

When we apply the operator D to this function, $D\langle f0, 1 \rangle(u, u, v)$, its body will be inlined and elementary variable substitution takes place. The result is the expression $u*u*v$.

6.3.5 Differentiate expression

This stage is the trickiest in the process. In the implementation, it is divided into multiple, nested strategies.

The outermost strategy concerns itself with differentiating on each index parameter, in reverse order. Given $D\langle f0, 1, 2 \rangle(u, u, v)$, the strategy will first differentiate the body of function f on its second parameter, in this case v .

The result will then be subjected to further differentiation, this time on parameter 1, which is u .

The actual symbolic differentiation is left to the `diff` strategy, shown in Fig. 6.4.

```

diff(var) =
  ( diff-trivial(var) +
    diff-mul(var) + diff-div(var) + diff-plusminus(var) +
    diff-literal +
    diff-tan (var) + diff-sin(var) + diff-cos(var) +
    diff-arctan(var) <+ diff-arcsin(var) <+ diff-arccos(var) +
    diff-sinh(var) <+ diff-cosh(var) <+ diff-tanh(var) +
    diff-sqrt(var) + diff-exp(var) + diff-pow(var) + diff-log(var) )
  <+ diff-funcall(var)

```

Figure 6.4: The `diff` strategy

The `diff` strategy is applied to the root of the expression tree. Conceptually, it will traverse this tree in a top-down manner, selecting from the `diff-*`¹ rules at each node in the tree.

Together, the `diff-*` rules cover all the differentiation rules, given in Appendix A.

Each of `diff-*` rules will call back to `diff`. This way, we recursively differentiate the expression tree.

Notice that in Fig. 6.4, the `diff-funcall` is at the bottom, because we want to handle known mathematical functions like `pow`, `cos` and `log` as our primitives.

We could alternatively have employed primitives on the form $D\langle \text{pow}, 1 \rangle(x, y)$, but this proved to be an implementation hassle, due to both the syntactic and semantic complexity of template applications.

The `diff-mul` should illustrate how `diff` is called recursively, see Fig. 6.5.

```

diff-mul(var):
  Infix(Op(" * ", s), f, g) → res
  where
    <logn(DEBUG)> "Differentiating * " ;
    <make-op> (" * ", <diff(var)> f, g) ⇒ l ;
    <make-op> (" * ", f, <diff(var)> g) ⇒ r ;
    <make-op> (" + ", l, r) ⇒ res ;
    <logn(DEBUG)> "Differentiated * "

```

Figure 6.5: `diff-mul`

The rule matches a term on the form `Infix(Op(' * ', s), f, g)`, where `s`, `f` and `g` are free variables bound at matching time.

¹Shorthand for the `diff-mul`, `diff-trivial`, ... rules in figure Fig. 6.4

f is the left-hand side of the $*$ -operator, while g is the right-hand side. s is just the signature of the $*$ -operator, and is in this case of no interest.

We see that both arguments to the $*$ -operator, named f and g , are recursively differentiated.

The `make-op` strategy generates the term for an application of a given operator on two arguments. In this case, we generate for the operators $*$ and $+$. As the strategy succeeds, `res` contains a term, representing the application of Equation A.1.

6.3.6 Repeat cycle

After stage (2) of step (3) has finished, we have made a complete first pass over the input program.

The application of the rule `diff-funcall` in Fig. 6.4, may result in newly generated applications of the D operator.

These will need to be resolved, so we repeat the three first stages again, and keep repeating until step (1) does not uncover any calls to the D operator anymore.

6.3.7 Emit code

When all applications of the differentiation operator have “bottomed out”, and step (1) fails, the modified program is emitted and the transformation ends.

In general, it is not always the case that the `diff` operator will “bottom out”. Consider the function in Fig. 6.6.

```
float f(float x) { return D<f,1>(x); }
```

Figure 6.6: A differential equation.

This case, where f is defined in terms of its own derivative. It is thus a differential equation.

Our transformation will cannot solve differential equations, so in this case, the entire transformation step will fail.

The same applies for cases where there is a mutual interdependence between the functions f and g , see Fig. 6.7:

```
float f(float x) { return D<g,1>(x); }
float g(float x) { return D<f,1>(x); }
```

Figure 6.7: Mutual interdependence

A mutual interdependence of this kind may or may not bottom out, depending on whether the user has specified $D\langle f, 1, \dots, 1 \rangle(x)$ or $D\langle g, 1, \dots, 1 \rangle$ elsewhere in his program. This is also a differential equation, so the transformation will fail when trying to differentiate it.

6.4 Shortcomings

Our current implementation has several known shortcomings, summarised below:

- *differential equations*; We do not solve differential equations. This is considered intractable. However, the current implementation also suffers from not reporting this problem to the end-user in a sensible manner.

We should extend it with proper error messages that explain why the transformation fails, and perhaps also even suggest remedies.

- *not fully Sophus-aware*; We do not differentiate operations on tensors and meshes. This is a complex issue. Given the symbolic representation of a tensor, we would be able to differentiate it symbolically. Using Mold-L (see Chapter 7), we do have the symbolic representation of the tensor, but we currently do not have code to make it readily available to the Stratego transformation code.
- *general error handling*; The error handling is insufficient for general use. As we do not track line numbers throughout the transformations, it is difficult to emit sensible warnings and error messages that pertain to the original source code. The issue of line numbers is also discussed in Chapter 9.
- *reliant on CodeBoost*; The strong type-checking in the C++ language prohibits defining one global prototype for the D and Dn operators; we need one prototype for each f we apply the diff operator to.

Consequently, we cannot generally replace applications of the symbolic D operator with applications of the Dn operator, without a semantically aware preprocessor.²

6.5 Application

The symbolic differentiator may be utilised in the method of manufactured solution (MMS), as explained in [PK02].

²I.e. naïvely replacing the string $D\langle$ with $Dn\langle$ would not work

MMS is a novel and recently developed technique that verifies the observed order-of-accuracy of the implementation of a numerical algorithm.

The central idea is to modify the governing equations and the boundary conditions by adding forcing functions or source terms in order to drive the discrete solution to a prescribed or “manufactured” solution chosen a priori.

Briefly, the method works as follows: We make a PDE solver, such as Sophus. We then choose a known function for which we want to test the solver. With the aid of symbolic differentiation, we construct a dataset to guide the PDE solver towards the correct solution.

By applying either solution accuracy analysis or residual error analysis, one may determine the accuracy of the solver.

In order to construct the test dataset, a symbolic differentiator is required.

6.6 Summary

The first draft of this transformation was fairly simple to implement. In fact, as long as we only consider types that act analogously to reals (arguably a sensible assumption, given the nature of differentiation), the transformation is simple.

To fully accommodate the differentiation operator syntax $D\langle f, n \rangle$, we had to extend CodeBoost with the ability to handle templates with function arguments. This was also fairly simple, and indicates that the CodeBoost framework itself is readily extensible.

Given the complexity of C++, generating terms for function calls can sometimes be a bit tricky. In particular, the function signatures are long, and getting them right by hand can prove tedious. It would probably be a good idea to extend CodeBoost with helper-functions to alleviate this.

Gaining experience with every-day use of the transformation on production code would prove invaluable to further improve its practical application. This would especially be useful for finding which human errors prove to be common, and how to best deal with these in terms of outputting sensible error messages.

Chapter 7

Mold-L, a matrix description language

In this chapter, we introduce the Mold-L description language for matrix layouts.

After explaining our motivations, we describe the syntax and semantics of the language. We give a short note about writing the layout descriptions before concluding with the currently observed shortcomings of the design and implementation.

7.1 Motivation

The purpose of Mold-L is to as precisely as possible describe the layout of N-indexed matrices to all relevant CodeBoost matrix transformations.

As we currently focus CodeBoost towards the Sophus numerical library, our focus has been on specifying layouts for two different families of matrices available in Sophus: *tensors* and *meshes*.

Both are implemented as N-dimensional array structures, and follow a similar (but not identical) abstract interface. Fig. 7.1 lists the commonalities and differences in the two abstractions.

Given the disparate characteristics, we thought it prudent to support two different ways of specifying matrix layouts: *implicit* and *explicit*.

Definition 7.1

A matrix layout specification where the actual elements are not individually specified is termed an implicit layout.

Definition 7.2

A matrix layout specification where all the actual elements are individually specified is termed an explicit layout.

<p>Common</p> <p>The operators +, - and * are defined.</p> <p>The elements 0 and 1 are defined.</p> <p>They are indexed by multiple indexes.</p>	
<p>Tensors</p> <p>Usually less than 200 elements.</p> <p>Usually indexed by more than two indexes.</p> <p>Its elements are typically meshes.</p> <p>The function <code>apply</code> is defined.</p>	<p>Meshes</p> <p>Typically millions of elements.</p> <p>Usually only two indexes.</p> <p>Its elements are typically rational values.</p> <p>Data ordered in a block-like layout.</p>

Figure 7.1: Properties of meshes and tensors

Fig. 7.2 shows a sample implicit layout. A sample explicit layout can be found in Fig. 7.3.

```
(layout
 mesh-a
 (implicit
 (dim
 (x 200)
 (y 200))
 (any -)
 ((> y 100) 1)
 ((> x 50) (>= y 30) 0)))
```

Figure 7.2: A sample implicit layout specification

```
(layout
 tensor-layout-b
 (explicit (
 (1 0 0)
 (0 1 0)
 (1 0 0))))
```

Figure 7.3: A sample explicit layout specification

These layouts are intended to be written by the Sophus library end-user. The layout and source-code is then fed through CodeBoost, which emits a program specialised to the given layout.

It is the responsibility of the end-user to tell CodeBoost about the program's matrix layout

A precise Mold-L description will be maintained and propagated through the program alongside the matrix data, using the totem system developer for CodeBoost.

At any point in the program where matrices are operated on, the operation may be sped up considerably if the operation is known to the matrix transformation collection and precise Mold-L specifications exist for the operation's parameters.

7.2 Syntax

The Mold-L language is embedded into the CodeBoost configuration files, and its syntax is therefore based on the well-known symbolic expressions.

```

layout ::= ( layout name mold )
mold ::= ( explicit ( {row}+ ) )
row ::= ( {arith-expr}+ )
arith-expr ::= ( arith-op {arith-expr}+ ) | name | value
mold ::= ( implicit
           ( dim {axis-spec}+ )
           ( {bounds-spec}+ ) )
axis-spec ::= ( name integer-number integer-number )
bounds-spec ::= ( {axis-bound}+ value )
axis-bound ::= ( rel-op name integer-number )
rel-op ::= < | > | <= | >=
value ::= integer-number | -

```

Figure 7.4: Complete syntax specification for matrix layouts

A complete syntax for layout specifications is given in Fig. 7.4. *name* is a regular symbol matching the regular expression `[a-zA-Z-]+` and *integer-number* is a regular integer number.

7.3 Semantics

Conceptually, the specifications are parsed, then fully expanded N-dimensional arrays where each entry contains an algebraic expression are constructed in-memory. This matrix of symbolic expressions can then be queried at boosting time.

For the optimisations we have implemented, we only do queries about individual elements, so lookups always amount to array references.

In reality, we only do full in-memory expansion for the explicit case. The implicit specification is kept in memory as a collection of blocks describing the matrix element values.

At the lowest level, there are 1-dimensional spans, that describe a strip along one axis in the N-dimensional matrix (usually, but not necessarily along a row). Each span specifies the number of elements in the span (its length) and the element value of common to those elements (either of *unknown*, *one* or *zero*).

At the next level, the spans are in turn collected in spans. If n adjacent spans have the same length and the same element value, they are collected into a span with length n .

For regular, two-dimensional matrices, the compactification process stops here. For N-dimensional matrices, all N levels will be considered.

Consider the matrix

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The compactification process will start by considering the first row. It will generate a span of length 3, with element value 1, denoted $\langle 3, 1 \rangle$. An identical span will be constructed for the next row. These two identical spans are then collected into a higher-level span, denoted $\langle 2, \langle 3, 1 \rangle \rangle$. The algorithm proceeds to generate the spans for the two lower rows, $\langle 2, \langle 3, 0 \rangle \rangle$.

The list of top-level spans ($\langle 2, \langle 3, 1 \rangle \rangle$ and $\langle 2, \langle 3, 0 \rangle \rangle$) is then used as the internal representation of the matrix layout for this matrix.

If we did not take steps to compactify the internal representation, boosting a program working on huge datasets would require equally huge amounts of memory, preventing us from doing the boosting on regular workstation machines.

Even with our naïve implementation of the span collection, the speed-loss experienced compared with the direct array lookups is next to negligible.

Needless to say, the implementation details of both the implicit and explicit case is hidden beneath a predefined interface. Thus, we can change the internal representation at whim.

7.3.1 Operations

The matrix layouts are meant to aid in specialising matrix operations throughout the program code. Given the expression $A = B \oplus C$ and layouts for A and B, we can specialise the code for \oplus to perform the minimum amount of calculations necessary for that particular expression.

We can also calculate the resulting layout A must have, and propagate the newly calculated layout as a totem. Thus, if A later should show up in a new

expression, say $D = A \oplus B$, we know A 's layout and specialise this application of \oplus as well.

Definition 7.3

Merging layout: Calculating the layout for A in the expression $A = B \oplus C$ is termed merging the layouts of B and C over the operator \oplus .

Currently, we support the point-wise versions of multiplication, addition and subtractions for both tensors and meshes. Additionally, we merge and specialise for the tensor's `apply` function.

7.4 Writing layouts

As an aid to writing the layouts, we have developed a special Emacs mode for editing CodeBoost configuration files — called `codeboost-mode.el` — that takes care of automatic indentation and syntax highlighting. It has been tested with Emacs 21.

Additionally, we developed a small tool that can automatically generate the implicit layout specifications from meshes stored in the Norsar format, named `norsar2layout.py`. It will currently only uncover any block structures present in the datasets. Extending it to also recognising other patterns should be fairly trivial, but will most likely require extending the Mold-L language.

Both of these utilities are available in the `codeboost-utils` CVS module from the CodeBoost home page [OSB03].

7.5 Shortcomings

While Mold-L was designed to be tiny from the start, we have already encountered some areas where it would be advantageous to make extensions:

- *diagonal and symmetric layouts*; The current Mold-L format is highly geared towards the Sophus library. It does not cater for specifying all regular matrix patterns such as diagonal and symmetrical matrices. Both of these are desirable primitives in the language, as many numerical problems are formulated with matrices of this kind.
- *Voigt notation*; When specifying symmetries, one well-known technique is the Voigt notation, described in [Aul90]. We have been unable to find any documented adaptations of the Voigt notation for the computer, nor are we familiar with its everyday use. It will therefore be necessary to consult expertise familiar with the notation in order to implement a good adaptation.

We have erred on the side of simplicity for fear of over-engineering a solution that would never be applied in real-life. Incrementally extending the language based on experience and user-feedback is our preferred route.

7.6 Summary

The Mold-L language can describe most matrix properties mentioned in [HMMK92], though it must be mentioned that manually specifying non-trivial symmetries is tedious.

Both the syntax and the semantics are extensible. The internal semantics of a Mold-L specification is extremely simple: for the explicit form, it is merely an N-dimensional array structure where each entry is an algebraic expression; for the implicit form, it is a collection of potentially overlapping N-dimensional blocks.

Chapter 8

Matrix transformations

In this chapter, we present an optimising transformation that performs partial evaluation of some algebraic operators on matrices based on matrix layouts specified at boosting time.

First, we put the transformation in perspective. Next, we describe how the layouts are specified and tracked throughout the program. We continue by explaining the partial evaluation of an operator, show some benchmark results and round off by discussing further improvements.

8.1 Motivation

Matrices are arguably the workhorse of mathematical abstractions for numerical software. This is evidenced by the number of articles written on the subject, the matrix-related abstractions available in common and uncommon numerical languages and the concept's central place in the curriculum for students of numerical software.

In many numerical programs, the majority of the execution time is spent doing calculations on matrices. Improving the speed of matrix calculations should therefore have a good chance of improving the overall running time of the program.

Volumes have been filled treating this topic before ([ABB⁺00],[GL96]), but few of them treat matrix operations within the coordinate-free paradigm.

With the classical approach, usually implemented in languages such as C and Fortran, the programmer is largely responsible for breaking up the abstract mathematical notion of the matrix into arrays and loops, from where the compiler takes over and produces the fastest code possible. This breakdown is performed based on the knowledge the programmer has about the properties of and operations on the underlying matrices.

The coordinate-free approach makes matrices a full-fledged abstraction of its

own. This requires a language where the programmer can create his own abstractions, in the form of classes. In order to reach high efficiency, the compiler must be smart enough to make use of the properties inherent in these new abstractions, and break the operations down into optimal loops in its own.

We started experimenting with exploiting the well-defined properties of unit, null, diagonal and (anti-)symmetrical matrices. Refer to Chapter 5 for details.

Work done by F. Gustavsson shows that a block-structured approach to exploiting the sparseness can result in significant speedups [Gus01]. As the exact block structures are very problem-specific, we decided to investigate the feasibility of partially evaluate the matrix operators with respect to the a priori known layout properties.

8.2 Overview

The input to this transformation is regular C++ source code where the layouts of the matrix variables have been specified in CodeBoost configuration files, introduced in Chapter 3.

A top-down traversal of the input program is performed. Totems specifying matrix layouts are tracked during the traversal. When an operator application on two matrices with known layouts is detected — on the form $A \otimes B$ — the evaluator is called, and a specialised function for that exact expression is generated. The expression $A \otimes B$ is replaced by a function call to the generated function.

Only point-wise addition, subtraction, division and multiplication on matrices, as well as the general matrix multiplication `mmult` (known as `apply` on the tensor abstraction) are considered.

These elementary algebraic operations were selected as they are conceptually and algorithmically very simple, they occur often, and they are, especially in the case of general matrix multiplication, fairly expensive operations.

8.3 Implementation

The implementation consists of two parts: traversal and evaluation.

The traversal part is written in Stratego, and traverses the subject program, tracks the totems, invokes the evaluator and incorporates the evaluated result back into the program tree.

The evaluator is written in Python. It takes as input the matrix layouts and the desired operator, then generates the specialised code. This code is then stitched back into the program AST by the traverser.

All data exchange between the parts is done through Unix pipes, where AST subtrees are passed in the form of textual ATerms.

The traverser also takes care of parsing the matrix layout files, and passing them to the evaluator.

8.3.1 Matrix layouts

The previous chapter described the Mold-L specification language and how we specify the layout of a given matrix. The layouts come in two forms, *explicit* and *implicit*. Explicit layouts are used for meshes, while implicit layouts are used for tensors.

We use the totem mechanism described in Chapter 3 to tag the layouts onto the matrix variables in the program. The totems are tracked throughout the program code. Whenever operations are performed on a totemized variable, we attempt to calculate the new layout, if possible.

Specifying layouts

The first hurdle to overcome is how to tell CodeBoost what the layout of the matrix variables are. The code in Fig. 8.1 shows how this is done.

```
fn() {
  Tensor<float> A,B,C;
  A = loadMatrixFromFile("A");
  B = loadMatrixFromFile("B");
  CB_IMPORT("layout-a");
  CB_IMPORT("layout-b");
  CB_TAG(A, "matrix-layout", "layout-a");
  CB_TAG(B, "matrix-layout", "layout-b");
  C = mmult(A,B);
}
```

Figure 8.1: Totemizing matrices.

Assume that the `loadMatrixFromFile` function loads the matrix data from a given file.

`CB_IMPORT` loads the configuration file named `layout-a`, with all its contexts. From the context `matrix-layout`, we pick the layout named `layout-a` and tag it onto the variable `A`.

The same goes for variable `B`.

When we get to the statement `C = mmult(A, B);`, CodeBoost knows the layout of both `A` and `B`, it knows that they are both of type `Tensor`, so the evaluator is invoked.

In addition to the specialised operator code, the evaluator also generates a specification layout. This layout is propagated upwards in the expression tree, and in our case is tagged onto `C`.

Tracking the layouts

The matrix layout is stored as a totem, and is tracked throughout the program using a totem tracker which implements to a few simple rules.

A matrix is only modified when it is either assigned to, or is sent as a mutable parameter to a function. In the Sophus style, most C++ operators do not modify their arguments ¹.

The propagation through an algebraic expression is demonstrated by walking through the transformation for the first line in Fig. 8.2.

```
D = A * B + C;
nomodify_0u(C);
modify_1u(C);
```

Figure 8.2: Tracking totems.

First, `A * B` will be replaced with a call to a generated function; `f000(A, B)`. The layout of `A * B` will be calculated and tagged onto `f000`. When `f000(A, B) * C` is replaced with `f001(f000(A, B), C)`, the layout from `f000` is combined with the layout of `C` to calculate the new layout, which is tagged onto `f001`. The `f001` layout is then propagated over the assignment operator to `D`.

The Sophus style tells us which parameters are mutable for a given function. When a matrix is passed as a mutable parameter to an unknown function, we must assume that it completely loses its known structure.

On line two, the `_0u` suffix tells CodeBoost that `C` is not modified, so all totems remain intact.

On line three, the `_1u` tells us that the first parameter, `C`, is modified. As Codeboost does not know the semantics of `modify`, we must assume that `C` has completely changed; all its totems are dropped.

The user is free to specify the matrix layout after a call to an unknown function, so as to help the transformation along. It should be noted that though the layout

¹An exception is the `<<` operator, which in Sophus is used to read data into a variable.

specification should be as precise as possible, even a coarse-grained specification will be helpful for optimising purposes.

8.4 Compile-time evaluation

The basic dense matrix multiplication algorithm we want to evaluate is given in Fig. 8.3.

```

for(int i=0;i<A.rows();i++)
  for(int j=0;j<B.cols();j++)
    for(int k=0;k<B.rows();k++)
      C[i][j] = A[i][k] * B[k][j];

```

Figure 8.3: The basic dense multiplication algorithm

A naïve partial evaluation of this algorithm in Tiger is shown in Chapter 5. In that chapter, we expand the three-level dense multiplication loop code above, then remove all statements where at least one of the operands is zero.

With a less naïve partial evaluator, it is possible to extract common subexpressions, re-roll unrolled statements into new loops to save code (and also code cache) size.

Our current optimizer does both of these things, but only when the layouts of A and B are given on the implicit form.

Implicit matrix multiplication

For very large matrices, we specify the layout on implicit form. As explained in Chapter 7, the implicit layout specifies the block structure of the matrix.

When at least some of the blocks are filled with zeros, we can rewrite the regular $O(n^3)$ algorithm to account for these zeros, and avoid performing scalar multiplication operations on the zero elements.

The transformation works by conceptually unrolling the i,j and k -loops, then considering the operation $C[i, j] += A[i, k] * B[k, j]^2$ for all appropriate values of i,j and k .

In the layout of A , an element $A[i, k]$ can be either of *unknown* (denoted as $_$), *zero* (denoted as 0) and *unit* (denoted as 1). The same holds for $B[k, j]$.

When multiplying $A[i, k]$ by $B[k, j]$ in the k -loop, we can simplify the multiplication according to the rules in Fig. 8.4.

²For notational convenience, we use $A[i, k]$ as a short-hand for $A[i][k]$.

$A[i,k] = 0$ or $B[k,j] = 0$	$C[i,j] += 0$
$A[i,k] = 1$ and $B[k,j] = 1$	$C[i,j] += 1$
$A[i,k] = 1$ and not $B[k,j] = 1$	$C[i,j] += B[k,j]$
$B[i,k] = 1$ and not $A[k,j] = 1$	$C[i,j] += A[k,j]$
$A[i,k] = -$ and $B[k,j] = -$	$C[i,j] += A[i,k] * B[k,j]$

Figure 8.4: Scalar multiplication rewriting rules

After unrolling and expression simplification, we have a long series of similar statements. In fact, we often observe that several consecutive expressions are identical, save for the indexes, as is exemplified in Fig. 8.5.

```
C[i][j] += 0; /* A[i][0] * B[0][j] = 0 */
...
C[i][j] += 9; /* A[i][9] * B[9][j] = 0 */
C[i][j] += A[i][10] * B[10][j]
...
C[i][j] += A[i][99] * B[99][j]
```

Figure 8.5: Unrolled code

We can easily re-roll the code in Fig. 8.5 back into loops shown in Fig. 8.6.

```
for(int k=0;k<10;k++) {
  C[i][j] += 0; // A[i][k] * B[k][j];
}
for(int k=10;k<100;k++) {
  C[i][j] += A[i][k] * B[k][j];
}
```

Figure 8.6: Rerolled code

This re-looping has a clear resemblance to the simple run-length encoding (RLE) algorithm known from compression theory (refer to [NIG95] for thorough introduction).

Interestingly, we can apply the same re-rolling at the i - and j -loop levels as well. The code in Fig. 8.8 shows a j -level loop rerolled from the code in Fig. 8.7, using the same algorithm.

Implementation-wise, the conceptual unrolling never occurs. It would prohibitively expensive to completely unroll the i,j,k -loop for the typical ranges of i,j,k we operate in (typically 1000 and up).

```
for(int k=0;k<10;k++) {
    C[i][0] += 0; // A[i][k] * B[k][0];
}
for(int k=10;k<100;k++) {
    C[i][0] += A[i][k] * B[k][0];
}
// j += 1
for(int k=0;k<10;k++) {
    C[i][1] += 0; // A[i][k] * B[k][1];
}
for(int k=10;k<100;k++) {
    C[i][1] += A[i][k] * B[k][1];
}
// j += 1
for(int k=0;k<10;k++) {
    C[i][9] += 0; // A[i][k] * B[k][9];
}
for(int k=10;k<100;k++) {
    C[i][9] += A[i][k] * B[k][9];
}
```

Figure 8.7: Unrolled code

```
for(int j=0;j<10;j++) {
    for(int k=0;k<10;k++) {
        C[i][j] += 0; // A[i][k] * B[k][j];
    }
    for(int k=10;k<100;k++) {
        C[i][j] += A[i][k] * B[k][j];
    }
}
```

Figure 8.8: Rerolled code

We therefore intermix the compression with the evaluator itself. Whenever an expression (k -level arithmetic expression or i/j -level loop) is produced, we compare it to the previous expression emitted, and collate them if they are alike. They are considered alike if their index differ by one on the level (i, j, k) under consideration.

After the loops have been rerolled, the transformation emits the code for the loop nests, then calculates the layout for the resulting matrix C .

We know that $C_{i,j} = \sum_{i,j,k} A_{i,k} * B_{k,j}$. If all of $A_{i,k}$ or all of $B_{k,j}$ are zero, $C_{i,j}$ is also zero. If not, we mark it as unknown ($_$). The resulting table is a new explicit layout. As with the generated code, the new layout table is subjected to RLE compression on one axis at a time, to reduce memory requirements. The compressed result is passed on as a totem.

At the end of evaluation, both the resulting code and the layouts are passed back from the evaluator to the traverser as $ATerms$.

Explicit matrix multiplication

For matrices specified on the explicit form, we always produce fully unrolled loops, as they are practically always less than a hundred elements, total.

As figure Fig. 3.9 shows, the layout specifications on the explicit form can contain arbitrary expressions for a given element in the matrix.

Fig. 3.9 specified the matrix $M = \begin{pmatrix} a & a-b & 0 \\ b-a & b & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

Whenever an element is referenced, it is replaced by the expression contained in that element. In theory, this allows us to only store the unique elements, in the above case a, b and c , but this has not been fully implemented yet.

Assume we multiply a matrix A with the above layout with a unit matrix. The evaluator will emit the code shown in Fig. 8.9.

```
C[0][0] = A[0][0] * 1;
C[0][1] = (A[0][0] - A[1][1]) * 1;
C[0][2] = 0;
C[1][0] = (A[1][1] - A[0][0]) * 1;
C[1][1] = A[1][1] * 1;
C[1][2] = 0;
C[2][0] = 0;
C[2][1] = 0;
C[2][2] = 1;
```

Figure 8.9: Unrolled explicit code

As we can see, the elements have been replaced by the expression contained within them. Also, the code contains some duplicated subexpressions. There is a common subexpression removal transformation for CodeBoost, not described in this thesis, that can be used to remove the duplicates in a later pipeline step.

After the code is emitted, the resulting layout is calculated. The multiplication is evaluated symbolically, and the result is stored as a new explicit layout.

As in the implicit case, both the resulting code and the layout are passed back to the traverser as `ATerms` after evaluation is completed.

8.4.1 Integration

At the end of evaluation, the traversal part receives a term describing the specialised code for the multiplication operator, and a term describing the layout for the resulting matrix.

The code is stitched into the program as a member function on the data type (either `Mesh` or `Tensor`). The layout is tagged onto this function, as a totem.

8.5 Shortcomings

The previously described transformations ultimately reduce the number of floating point operations necessary to perform simple arithmetic on various kinds of matrices.

A number of areas for further improvement exist:

- *storage patterns for parallelisation*; We have not taken the time to investigate data storage patterns more suited for parallel supercomputers. Given detailed knowledge about the matrix layout, we suspect there is an opportunity for significantly reducing the amount of data traffic between computing nodes.
- *cache optimisation*; The naïve storage pattern we employ does not take advantage of modern computer architectures' cache mechanism. By compacting the in-memory representation of our matrix structures, it is thought cache hits and performance would improve. Work done by Hogné Hundvebakke suggests that adaptively modifying the traversal order of the large mesh structures exhibits a statistically significant improvement [Hun02].
- *transformation functions*; Automatically exploiting non-trivial symmetries in the matrices where one part (say lower half after LU decomposition) is a transformed version of the other (say upper half), and the transformation

function is sufficiently cheap (multiplication by -1), we could reduce the number of required floating point operations further.

- *optimising the remaining operators*; the Sophus matrix abstractions have additional operations not described in this chapter. Some of these, such as the traversal function `map`, should benefit greatly from exploiting the sparseness patterns described by the matrix layouts.

Once all operators are known by the transformation and partially evaluated, we are may finally change the internal representation freely, and only store the unique elements of a matrix.

As it is now, we have made some experiments with generating specialised `Mesh` classes with internal data representations that store the minimum amount of data required; If the lower half of the mesh is known to be zero, it need not be stored.

As the layout propagates and changes, new, specialised `Mesh` classes are generated. The problem is that the current implementation of the transformation only knows how to generate specialised version of a few of the `Mesh` operators. When other, unknown operators are invoked, such as `minus`, un-specialised code that assumes a standard layout is invoked, which results in data corruption.

8.6 Application

We do not yet apply the advanced transformation described above to the Sophus library. Therefore, we do not have any hard benchmarking evidence from real, live programs.

What we do know, is that the transformation results in code equivalent to the hand-optimised specialisations of the `apply` function in the Seismod solver loop (Fig. 2.1).

We can therefore expect roughly the same runtime characteristics as the hand-optimised code, as the CodeBoost generated code is virtually identical.

If changes in the formulation of the problem results in new matrix layouts, it is a trivial matter to regenerate a specialised `apply` function.

One is therefore free to program and reason about general solutions without loss of performance.

8.7 Summary

This chapter has described a set of related transformations which improve the running time performance of elementary matrix algebra by partially evaluating matrix operations at boosting time.

The transformations will consult hand-specified layout information for the matrix variables, and insert specialised code for operations on these variables.

As long as the data processed by the specialised program corresponds to the layout information provided at compile-time, the two programs are semantically equivalent.

Comparing our generated code with the hand-written code, we see that we can expect to have the same run-time performance.

Chapter 9

Further work

Sometimes, science has this nasty habit of telling you that the more you uncover, the less you know. We appear not to have been spared, either.

The following sections will detail some of our thoughts that have cropped up during the process of completing our work.

We first discuss details around the matrix description language and opportunities for new Sophus-related transformations. Next, we consider a few suboptimal aspects of CodeBoost, such as inadequate error handling.

Finally, we make a few suggestions on how to improve some non-technical parts of the development process of CodeBoost.

9.1 Extending Mold-L

The Mold-L matrix specification language cannot easily describe arbitrary layouts on medium to large matrices. In particular, the implicit form as described in Section 7.3 is not powerful enough to describe all common matrix patterns.

We would like to cater for the following patterns: diagonal, upper, lower, symmetrical, anti-symmetrical, unit, checkerboard.

We could do this by adding each pattern as a special case in the matrix specialisation code, but that would mean a combinatorial explosion of $totalcases = specialcases \times operators$, which is less than desirable.

However, going the case-by-case route is probably worthwhile to gain the necessary insight into how to generalise the algorithms properly. The route should be coupled with applying the transformations to real-world programs, preferably ones where the matrix access patterns are known and both generic and hand-optimised versions of the code exist.

Which leads us to another question. Currently, the dichotomy between implicit and explicit modes is present due to the need for expanding the complete

matrix structures in-memory at boosting-time. We tried some naïve compactification techniques (see Chapter 8), but the running times when dealing with huge matrices proved prohibitive.

It may be the case that selecting more advanced data structures and applying some memoization tricks will vindicate the compactification idea. If so, the implicit/explicit dichotomy can hopefully go away and we will be able to use the same transformation core for both `Tensor` and `MeshSF` objects.

If it is deemed too time-consuming to go the route through partial-evaluation of the matrix operators, the technique outlined in Section 5.2.4 should be simple to implement for C++ in CodeBoost as well. This would require us to have semi-complete “template” code for the various cases, which brings us back to the problem of combinatorial explosion.

However, it may be the case that there are few enough matrix patterns to justify this simpler approach, at least for a given application.

9.2 Further augmenting the Sophus library

There are many apparently minor interrelations between the Sophus data types not currently captured by our transformation collection. This ranges from simple algebraic identities such as `Mesh t; minus(minus(t)) == t;` to arbitrarily complex algebraic expressions.

Adding algebraic simplification rules for such cases probably makes for a nice introduction to CodeBoost for new contributors. Some of these simplification rules can be written using the user-defined rules described in [Bag03], while others are probably so complex that they need the full CodeBoost framework, and therefore need to be a full-fledged transformation.

As described in Section 6.4, we do not properly differentiate over `Meshes` and `Tensors` yet. Getting low and dirty with the various approaches to program derivation is probably material for an entire thesis by itself.

We have so far only optimised some of the algebraic tensor operators with respect to the matrix layout. Adding optimised code for the remaining operators (such as `minus`, `shift`) is probably also worthwhile, and should be relatively simple.

The generated code for the matrix specialisations should eventually be based on the `map-construct` instead of loops. This could not be done at the present moment, as the complete semantics of `map` operators over tensors and meshes had not been fully specified yet.

Interestingly enough, one of our minor problems is that there are too few programs written using the Sophus library. This means we do not get the chance to test the robustness of our transformations on a decently sized body of code.

9.3 Robust error handling

One of the major obstacles remaining before CodeBoost can be used outside the research group is graceful error handling. The framework itself mostly handles errors well enough, but some of the transformations we provide do not.

Detecting semantic errors can be very tricky. For instance, the symbolic differentiator assumes that functions like `pow` and `sqrt` are defined on the data types it differentiates on. Adding proper error-checking may easily clutter the transformation and make it more resistant to change.

Usually, it is preferable to detect the errors early on, in order to emit sensible error messages that can explain to the user why a given transformation is not applied properly. It is the transformation writer’s job to insert instructive error and warning messages.

This job has not been taken seriously enough with all of our current transformations. We initially aim for error messages and warnings following the style of regular compilers, possibly also including with a pretty-printed excerpt of the offending code.

However, there are a few stumbling blocks for this:

1. The AST does not contain information about the line numbers in the original source code.
2. The pretty-printed source may be confusingly dissimilar to the original source code.
3. The faulty code may be the result of a previous transformation step, where the line numbers no longer correspond with the original source code.

The crucial point here is handling line numbers sensibly. If the error cannot be traced back to the user’s original code, there is little chance for him to fix the flawed code.

This tracking problem may become very complex. Consider the code in Fig. 9.1.

```

template<T> T foo(T x) { return x.generatedOp(); }
void bar() { Mesh<float> x,y; y = foo(x); }
```

Figure 9.1: Calling generated code

Assume `Mesh::generatedOp()` is supposed to be generated by CodeBoost. If `foo()` is inlined at transformation step 1 and `Mesh::generatedOp()` fails to generate in transformation step 2, there is no easy way to emit an error that says “`x.generatedOp()` could not

be generated in function `template<T> T foo(T)`”, as we no longer see `foo()`.

Keeping track of where the terms move throughout the transformation pipeline is complex, but it has been done in existing compilers, such as MIPSpro.

One solution may be to employ a relatively new invention in Stratego, called term annotations. We can add metadata to each term, and have it propagate along the term throughout the transformations. This metadata would contain line numbers and possibly information about the original scope.

To avoid maintenance hassles, we need to figure out a way to separate the error handling from the transformation rules and traversal strategies in a nice manner.

Some solutions have already been tried out inside the CodeBoost core library. One simple technique, used in `src/sig/cpp-check.r`, is to generate special error terms to signal errors. When a strategy fails, it will return an error terms containing a relevant error message and possibly some extra error code or information. The calling strategy may attempt error handling, or further propagate the error up the call chain. If the error makes it to the top level, the error is caught, the error message is emitted and the transformation fails.

9.4 Improving CodeBoost running times

When boosting large bodies of code, CodeBoost may spend minutes performing its transformations. Some of the transformations (the matrix transformations in particular) can be quite sluggish and require up to a minute for a given expression, on today’s computers.

For prototyping research code, this is not a severe obstacle. In our case, the perceived impact has been lessened by running on fast hardware.

For more widespread adoption of CodeBoost, however, it may become necessary to improve the running times by optimising the transformations themselves, and reworking the CodeBoost pipeline somewhat.

We currently have three approaches to improving the running time performance:

1. *remove pipes*; instead of using intermediate files or pipes between each pipeline stage, we could maintain one, in-memory AST which the modules worked on. Each module would be a shared library, loaded at run-time by a central dispatching unit. As long as the pipeline stages are written in Stratego, this should be fairly trivial. For non-Stratego pipeline stages, we still have the ability to use pipes.
2. *optimise individual transformations*; the most irksome transformations may benefit from some hand-optimisation, once their semantics and functional-

ity has been settled. As “premature optimisation is the root of all evil”¹, we have not done that yet, as our experimentation phase on the transformations’ semantics is still ongoing.

3. *improve Stratego compiler*; as the Stratego compiler matures, the performance of its resulting code may also be expected to improve slightly. We do not expect any big leaps in performance here, though.

While developing CodeBoost, it was not CodeBoost’s own running time that proved most annoying. The Stratego compiler itself is actually the limiting factor in the edit-compile-run loop of the development cycle of CodeBoost. Compiling the `diff.r` module currently takes around 19 seconds on dual 1.666GHz x86 computer. Compiling all of CodeBoost on the same hardware measures in at about 10 minutes.

For rapid prototyping, it would be nice to have an interpreter for Stratego code, to allow for very quick experimentation. Older versions of the Stratego bundle came with a semi-functional interpreter, but it has been deprecated.

9.5 Documenting and promoting CodeBoost

While we do not think it is generally usable at the present time, we ultimately want people outside our research group to use CodeBoost. We also desire the benefits from outside input and peer-review.

To this end, we have developed CodeBoost as an open-source project. From personal experience with other open-source projects, we have a few ideas on how to foster a community around Codeboost, outlined below.

9.5.1 Release early, release often

Generally, open-source software projects that appear active are more likely to attract outside developers. Additionally, involving the users in the development process helps counteracting the Ivory Tower-effect, whereby a few academically minded wizards concoct hair-brained schemes not suited for the real world.

By making frequent releases and starting the release cycle early on when not all things are set in stone, input from outside sources have a better chance of affecting the course of development, thus giving the users a say.

At present, we are in the lucky situation that a sizable portion of the coordinate-free community use the Sophus library. Their experiences and complaints are filtered through Magne Haveraaen, the Sophus designer, before reaching the CodeBoost development team.

¹Originally from Tony Hoare, restated by Donald P. Knuth

We then get the distilled versions of the problems, along with well-considered suggestions for how to solve them. This allows us to focus on implementing transformations that solve concrete shortcomings.

While inviting external parties into this development process may give novel insights, there is also the risk of diluting the original goals.

For now, we offer information on CodeBoost in the form of papers and a conceptual preview. As the tool matures, we expect to offer pre-compiled packages for various platforms, and tutorials with examples, to attract outside users.

9.5.2 Central hub for development

By having one, central place for users to get documentation, new releases, report bugs, get in touch with the developers and discuss between themselves, one is better able to foster a community.

In our case, we already have a rudimentary web presence [OSB03]. It would be fairly easy to extend with a bug-tracker and a mailing-list.

However, to have a decent shot at attracting external parties, we will probably have to solve most of the following tasks:

- *documenting the standard library*; the CodeBoost standard library contains a lot of convenience functions useful when writing transformations. These need to be documented, and small examples should accompany some of them.
- *more regression testing*; whenever we change the CodeBoost library or some of its transformations, we can no longer be certain things do not break. To alleviate this, we try to keep our suite of regression tests up to date, but this has been falling behind lately.
- *examples*; we need a few test-cases to allow interested parties to download CodeBoost and toy with it. As our transformations are currently tied to the Sophus library, we probably need a very cut-down version of Sophus for these examples.

9.6 Summary

To ensure continued development on CodeBoost, getting up to speed for new contributors should be easy. This primarily means we need more and better documentation. Additionally, the current CodeBoost team should remain available to answer tricky questions.

The danger with bringing in new developers is always a dilution of the project's focus. It must be emphasised that CodeBoost is geared towards the numerical community, particularly towards programs written in the algebraic style.

We admit that CodeBoost is currently only a research tool. It is not production-ready, and that getting to a production-ready state will probably take years of directed effort, as the ground we cover is sometimes both complex and new.

The sum of our experience tells us that CodeBoost is powerful enough to do most of the transformations we have wanted to do, but that some extensions (particularly those outlined in the previous sections) are in order. Fortunately, extending CodeBoost itself has proven to be easy enough, given adequate knowledge of the Stratego language.

We feel it would be a shame if the effort put into the current CodeBoost version should end with slow bit rot on an inactive CVS repository in an obscure part of the Internet. We have therefore proposed a rudimentary plan for raising external awareness for CodeBoost, with the hopes of attracting external contributors.

Allowing external parties to build directly on our work, without having to laboriously re-implement the basic framework, is one of the reasons why CodeBoost was developed openly in the first place.

Chapter 10

Conclusion

This chapter summarises our experiences with implementing transformations with CodeBoost.

The chapter touches on the development process and some implementation-specific details.

In this thesis, we wanted to demonstrate a few aspects of CodeBoost, namely:

- *flexibility*; we show the flexibility and power available in the framework.
- *user-control*; we investigate the practical aspects of having the user control the application and aggressiveness of the transformations, instead of just naïvely applying them to the entire body of code.
- *extensibility*; we marry optimisation and programming techniques from the fields of partial evaluation, program derivation and specialisation, aspect-oriented programming with the high-level clarity of coordinate-free numerics.
- *optimisation*; To investigate the feasibility of programming general abstractions in an algebraic programming style without penalties to performance.

In our experience, the CodeBoost framework has proved to be pleasantly flexible.

The symbolic differentiator described in Chapter 6 shows that we can implement transformations for symbolic manipulation of non-trivial complexity.

The implementation of the differentiator follows the paradigm of strategic rewriting rules closely.

For cases where this paradigm is less suited, we have demonstrated that we can resort to more traditional paradigms, such as imperative programming. Due to the pipelined nature of CodeBoost, we may insert a transformation step written

in any language. As an alternative, an external program may be invoked at any time, from within a regular transformation. The partial evaluator part of the matrix transformation is an example of external invocation.

In general, very complex transformations are possible. It has been suggested that complex symbolic evaluation could be performed by established tools such as Mathematica or Maple. One would invoke the external mathematical package at suitable inside the transformation.

We have not pursued this path, because problems requiring that complex symbolic evaluation would in all likelihood result in a brittle transformation.

As for user control, experience with regular compilers tells us that tuning a given transformation to a particular program, or part of a program, is usually worthwhile, i.e. adjusting the default degree of loop-unrolling and inlining limits.

The user can control the application order of the individual transformations. In addition, the following tuning facilities incorporated available to our transformations:

- *command line parameters*; switches and simple data may be passed to the transformation, independently of the source code it operates on.
- *configuration files*; configuration files referenced from inside the source code can be used to load arbitrarily complex auxiliary data used at boosting time.
- *totems*; language constructs can be tagged with flags or auxiliary data loaded from configuration files.

It turns out that these mechanisms exhibit some overlap. For instance, one can use both totems and command line parameters to easily pass simple flags or switches to the transformation.

A significant drawback with the current implementation is that the transformation itself must handle the cases above differently, even though they all specify the same flag.

Overall, the degree of control collectively offered by the above mechanisms has proven more than adequate. Most transformations will only expose a few switches for tuning, but we have shown that should the need for complex, source-specific input to the transformation arise, it can be handled by the framework.

Regarding extensibility, we have experienced that adding new functions to the framework library, or adding new library modules entirely, is rather easy. The only major stumbling block in extending the library is the lack of reference documentation for the existing code.

To demonstrate the potential for optimisation, we show a transformation performing partial evaluation of some matrix operations (general matrix multiplication, point-wise addition, subtraction, multiplication and division). Given a

description of the matrix layouts, our optimiser generates code identical to the hand-optimised code.

As long as we only specialise the code for these operations, integrating the optimiser into the Sophus numerical library is fairly easy. Once we decided to specialise the internal data representation, in order to optimise the memory usage, we sort of crashed and burned.

The matrix abstractions in the Sophus library have numerous operations defined on them, some which traverse the internal data and some which operate on single elements. They all make assumptions about the internal data representation, as they work directly on it. We ended up needing to generate specialised versions of all these operations if we were to make changes to the representation.

As we started down the road of specialising all relevant operations, the transformation became large and difficult to maintain. Multiple attempts at splitting it up were made, but none successful.

Fortunately, progress on the Sophus library solved our problems. The library is being rewritten so that all traversal of the internal data will be done using map traversal function.

Changing the internal data representation is then only a matter of specialising the map function, and a very few point-wise element access functions.

Surprisingly, developing transformations with CodeBoost proved to be trickier than initially thought. Writing a transformation for CodeBoost is a laborious affair, compared to the simpler TigerBoost framework.

One reason for this is probably personal. The edit-compile-run cycle is very long, which is anathema to the way I usually work. Compiling a single transformation for CodeBoost takes approximately 30 seconds on a high-end workstation. For the smaller TigerBoost framework, compilation time is significantly shorter.

Another aspect is that generating new terms for the C++ AST is hard. As Fig. 2.3 indicates, the AST for even simple expressions is large. The primary obstacle is the type information and signatures. Although CodeBoost supplies auxiliary strategies for building and modifying signatures and types, the process of constructing a semantically correct term is time-consuming, and usually involves at least some debugging.

The debugging process is also a bit immature, as debugging tools for Stratego are non-existent. Debugging a faulty AST usually amounts to manual inspection, as exhibited in Fig. 3.12.

Alternatively, shotgun debugging may be employed¹, and has proved highly effective.

In total, we have found that augmenting libraries using CodeBoost is feasible, but currently far from easy enough. We can achieve the goal of an algebraic

¹Inserting debug statements across the program to narrow down the faulty code

style, separation of concerns as well as performance, at the cost of transformation complexity.

Developing a transformation is in general too tricky for CodeBoost to be another tool in the regular library developer's toolbox. Luckily, a simpler approach exists. The user-defined rewrite rules described in [Bag03] are much easier for a library developer to write and maintain. They are sufficiently powerful to handle most kinds of expression simplification, which is very important in the algebraic style.

Writing full-fledged CodeBoost transformations should be reserved for central library abstractions, or cases where the transformation significantly increases performance or usability of an abstraction.

Appendix A

Differentiation rules

This chapter lists the differentiation rules we have implemented for the transformation described in Chapter 6.

$$\frac{d}{dx}[f(x) * g(x)] = f'(x) * g'(x) + f(x) * g'(x) \quad (\text{A.1})$$

$$\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x) \quad (\text{A.2})$$

$$\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x) \quad (\text{A.3})$$

$$\frac{d}{dx}[f(x)/g(x)] = \frac{f'(x) * g(x) - f(x) * g'(x)}{g(x)^2} \quad (\text{A.4})$$

$$\frac{d}{dx}[f(x)^c] = c * f(x)^{c-1} * f'(x) \quad (\text{A.5})$$

$$\frac{d}{dx}[e^{f(x)}] = e^{f(x)} * f'(x) \quad (\text{A.6})$$

$$\frac{d}{dx}[\log f(x)] = \frac{1}{f(x)} * f'(x) \quad (\text{A.7})$$

Figure A.1: Elementary differentiation rules

$$\frac{d}{dx}[\sin f(x)] = \cos f(x) * f'(x) \quad (\text{A.8})$$

$$\frac{d}{dx}[\cos f(x)] = -\sin f(x) * f'(x) \quad (\text{A.9})$$

$$\frac{d}{dx}[\tan f(x)] = (1 + \tan^2 f(x)) * f'(x) \quad (\text{A.10})$$

$$\frac{d}{dx}[\sqrt{f(x)}] = \frac{1}{2\sqrt{f(x)}} * f'(x) \quad (\text{A.11})$$

$$\frac{d}{dx}[\sinh f(x)] = \cosh f(x) * f'(x) \quad (\text{A.12})$$

$$\frac{d}{dx}[\cosh f(x)] = \sinh f(x) * f'(x) \quad (\text{A.13})$$

$$\frac{d}{dx}[\operatorname{sech} f(x)] = -\operatorname{sech}^2 f(x) * f'(x) \quad (\text{A.14})$$

$$\frac{d}{dx}[\arctan f(x)] = \frac{1}{1 + f(x)^2} * f'(x) \quad (\text{A.15})$$

$$\frac{d}{dx}[\arcsin f(x)] = \frac{1}{\sqrt{1 - f(x)^2}} * f'(x) \quad (\text{A.16})$$

$$\frac{d}{dx}[\arccos f(x)] = -\frac{1}{\sqrt{1 - f(x)^2}} * f'(x) \quad (\text{A.17})$$

$$\frac{d}{dx}[c^f(x)] = c^f(x) * \ln c * f'(x) \quad (\text{A.18})$$

$$\frac{d}{dx}[f(y)] = 0 \quad (\text{A.19})$$

$$\frac{d}{dx}[g(x)] = D \langle g, 1 \rangle (x) \quad (\text{A.20})$$

$$\frac{d}{dx}[c] = 0 \quad (\text{A.21})$$

Figure A.2: Elementary differentiation rules (cont.)

Appendix B

Installing CodeBoost

This chapter will explain how to download, compile and install CodeBoost and accompanying utilities.

Getting CodeBoost up and running should be no more difficult than compiling and installing any other regular Unix program. We use the GNU automake tools, and our source is available as a regular tarball.

Apart from CodeBoost itself, we will need to install the Stratego compiler.

B.1 Downloading

The first package we need is Stratego. Head over to www.stratego-language.org and get the 0.8.1 release from their download section. You may try a later release, but this is what we have tested with. Remember to install the ATerm library first, as Stratego depends on it.

Proceed by getting the CodeBoost 0.2.2 tarball from the download section on www.codeboost.org. You may try a later version, but the baseline for this thesis is 0.2.2.

You will also need our patched version of OpenC++. Get the latest version available on the CodeBoost download page.

While you can also get the sources from CVS, we do not recommend this, as the CVS tree may be in an unstable state.

B.2 Compiling

Issue the following commands to install Stratego (the `$` signifies the prompt; it should not be entered):

```
$ tar xzf stratego-0.8.1.tar.gz
```

```
# cd stratego-0.8.1
# ./configure --prefix=<your desired basepath>
# make all install
```

Now we need to install the OpenC++ compiler. To compile, issue:

```
# tar xzf openc++.tar.gz
# cd openc++/src/Unix
# make -f Makefile.Linux\footnote{There are {\tt Makefile}s for
other platforms as well, named {\tt Makefile.<platform>}.}
```

A program named `occ` will be generated. Put this somewhere in your path.

At this point, verify that both `occ` and `sc` are available by trying to run them.

If they are not, verify that they both are in your path.

Proceed by installing CodeBoost:

```
# tar xzf codeboost-0.2.2.tar.gz
# cd codeboost-0.2.2
# ./configure --prefix=<your desired basepath>
# make all install
```

At this point, provided no errors occurred, CodeBoost is installed and ready to run.

B.3 Testing

To get an immediate feel for CodeBoost, we have prepared a few elementary examples. Get the `codeboost-examples` tarball from the download page on www.codeboost.org.

Unpack it, verify that `codeboost` is in your path (issue `which codeboost`), and you're ready to go.

The examples are conveniently arranged by the transformation they are demonstrating. For a very trivial demonstration, issue `codeboost -t simplify2 simplify2/test-1.C`. CodeBoost will output the transformed code on `stdout`¹.

As explained in Section 3.2.3, sometimes it makes sense to apply the transformations *A* and *B* in sequence as *A* may have generated code suitable for optimisation by *B*.

A small example of this can be observed by first doing:

¹You may experience a few warnings on `stderr`. These are usually harmless

```
# codeboost -t diff diff/test-1.C
# codeboost -t diff -t simplify2 diff/test-1.C
```

Observe the difference in output from the two commands. The latter will result in a simpler algebraic expression.

The file `examples/README` explains in detail how to play with the provided example code.

B.4 Summary

For regular Unix system administrators and programmers, the installation procedure of CodeBoost is pretty straightforward. However, we would like to offer distribution-specific packages as a convenience to our users.

Part of the nuisance of installing CodeBoost is installing the auxiliary tools it requires. Some distributions have already started to include Stratego². As Stratego matures, we expect the others to follow, thus mitigating this problem somewhat.

We plan on extending the `codeboost-examples` with juicy demonstrations as we keep on adding new transformations to the CodeBoost framework.

²Gentoo Linux

Bibliography

- [ABB⁺00] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, and James W. Demmel. *Lapack Users' Guide*. Software, Environments and Tools, 9. SIAM, Philadelphia, USA, 2000.
- [App98] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
- [Aul90] Bertram A. Auld. *Acoustic fields and waves in solids, 2nd ed revised*. Krieger Publishing Company, 1990.
- [Bag03] Otto Skrove Bagge. Codeboost: A framework for transforming c++ programs. Master's thesis, University of Bergen, Norway, 2003.
- [BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
- [BF01] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Reading, Mass.: Addison-Wesley, 2001.
- [BF02] Moshe Bar and Karl Franz Fogel. *Open Source Development with CVS, 2nd ed*. The Coriolis Group, 2002.
- [BHV01] Otto Skrove Bagge, Magne Haveraaen, and Eelco Visser. Code-Boost: A framework for the transformation of C++ programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, 2001.
- [BHW97] James M. Boyle, T.J. Harmer, and V.L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, Boston, 1997.

- [Boy70] James M. Boyle. A transformational component for programming language grammar. Technical Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, July 1970.
- [BvdDH⁺01] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001*, pages 365–370. Springer-Verlag, 2001.
- [CDK⁺00] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufman Publishers, 2000.
- [CEG⁺] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T.L. Veldhuizen. Generative programming and active libraries. In *Proceedings of the 1998 Dagstuhl-Seminar on Generic Programming(1998)*, Lecture Notes in Computer Science.
- [CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, Asilomar, Pacific Grove, CA, September 1996.
- [Chi96] Shigeru Chiba. Openc++ programmer’s guide for version 2. Technical Report SPL-96-024, Xerox PARC, 1996.
- [Cod00] LLC CodeSourcery. g++ internal representation. <http://gcc.gnu.org/onlinedocs>, August 2000.
- [DHH98] T.B. Dinesh, Magne Haverdaen, and Jan Heering. An algebraic programming style for numerical software and its optimisation. Technical Report SEN-R9844, CWI, Amsterdam, Netherlands, 1998.
- [EV03] et. al. Eelco Visser. stratego – strategies for program transformation. <http://www.stratego-language.org/>, March 2003.
- [FJH⁺01] Helmer André Friis, Tor Arne Johansen, Magne Haverdaen, Hans Munthe-Kaas, and Åsmund Drottning. Use of coordinate-free numerics in elastic wave simulation. *Applied Numerical Mathematics*, 39:151–171, 2001.

- [GHW00] P.W. Grant, M. Haveraaen, and M.F. Webster. Coordinate free programming of computational fluid dynamics problems. Technical Report CSR 2-2000, University of Wales, Swansea, 2000.
- [GL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins Series in the Mathematical Sciences. Johns Hopkins University, dec 1996.
- [GL99] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Domain-Specific Languages*, pages 39–52, 1999.
- [GM97] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1997.
- [Gol90] Charles F. Goldfarb. *The SGML Handbook*. Oxford: Oxford University Press, 1990.
- [Gus01] Fred G. Gustavson. New generalized matrix data structures lead to a variety of high-performance algorithms. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software*, volume 188 of *IFIP Conference Proceedings*. Kluwer, 2001.
- [Hav03] Magne Haveraaen. the saga project. <http://www.ii.uib.no/saga/>, March 2003.
- [HMMK92] Magne Haveraaen, V. Madsen, and Hans Munthe-Kaas. Algebraic programming technology for partial differential equations. In *Proceedings of Norsk Informatikk Konferanse (NIK)*. Trondheim, Norway: Tapir, 1992.
- [Hun02] Hogne Hundvebakke. A study of mesh-traversal patterns for the sophus library and runtime variations on sgi origin (norwegian). Master’s thesis, University of Bergen, Norway, 2002.
- [JAH01] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Reading, Mass.: Addison-Wesley, 2001.
- [JGS93] Neil D. Jones, Carsten K. Gmard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

- [Kar98] Michael Karasick. The architecture of montana: An open and extensible programming environment with an incremental c++ compiler. In *International Symposium on Foundations of Software Engineering*, pages 131–142, November 1–5 1998.
- [KC00] Ulrich W. Eisenecker Krzysztof Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Reading, Mass.: Addison-Wesley, 2000.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th European Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
- [LVV02] Ralf Lämmel, Eelco Visser, and Joost Visser. The essence of strategic programming. March 2002. (Draft).
- [MdJ03] Joost Visser Merijn de Jonge, Eelco Visser. xt – a bundle of program transformation tools. <http://www.program-transformation.org/twiki/bin/view/Tools/XT>, March 2003.
- [NIG95] Mark Nelson and Jean loup Gailly. *The Data Compression Book*. M&T Books, New York, 1995.
- [OSB03] Karl Trygve Kalleberg Otto Skrove Bagge. codeboost – a c++ transformation framework. <http://www.codeboost.org/>, March 2003.
- [Par93] Terence J. Parr. Sorcerer reference. In *Language translation using PCCTS and C++: A Reference Guide*, pages 161–199. Automata Publishing Company, 1993.
- [PK02] Kambiz Salari Patrick Knupp. *Verification of Computer Codes in Computational Science and Engineering*. Discrete Mathematics and Its Applications. CRC Press, October 2002.
- [SGML01] Sibylle Schupp, Douglas P. Gregor, David R. Musser, and Shin-Ming Liu. User-extensible simplification—type-based optimizer generators. In Reinhard Wilhelm, editor, *International Conference on Compiler Construction*, Lecture Notes in Computer Science, 2001.

- [SL998] The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.
- [Sta97] Richard Matthew Stallman. *GNU Emacs manual : updated for Emacs version 20.1*. Boston: Free Software Foundation, 1997.
- [TBSM00] J. Paoli T. Bray and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. <http://www.w3.org/TR/2000/REC-xml-20001006>, October 2000.
- [vdBdJKO00] M.G.J. van den Brand, H.A de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. In *Software - Practice and Experience*, pages 30:259–291, 2000.
- [Vel95a] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.
- [Vel95b] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.
- [Vis00] E. Visser. The stratego tutorial. Technical Report Technical Documentation, 0.5, Institute of Information and Computing Sciences, Utrecht University, 2000.
- [Vis01a] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
- [Vis01b] Eelco Visser. Tiger in stratego: An exercise in compilation by transformation. Technical report, 2001.