On Measuring and Optimizing the Performance of Parametric Polymorphism

(Spine Title: On the Performance of Parametric Polymorphism)
(Thesis Format: Monograph)

by

Laurentiu Dragan

Graduate Program
in
Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
September, 2007

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Chief Advisor

Dr. Stephen M. Watt

Advisory Committee

Examining Board

Dr. Mark Giesbrecht

Dr. Marc Moreno Maza

Dr. Olga Veksler

Dr. David Jeffrey

The thesis by
Laurentiu Dragan

entitled
On Measuring and Optimizing the Performance of Parametric Polymorphism

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Date

Chairman of Examining Board

Abstract

On Measuring and Optimizing the Performance of Parametric Polymorphism

Laurentiu Dragan

Doctor of Philosophy

Graduate Department of Computer Science

The University of Western Ontario

2007

With the introduction of support for "generic" style programming in mainstream languages, it is possible to write generic algorithms by using parametric polymorphism. Parametric polymorphism can be expressed with templates in C++, generics in Java and C# and dependent types in Aldor. Generic code is not as fast as specialized code, because the compilers usually do not make any optimizations based on the specific properties of the data used by the generic algorithm. In order to make generic code appealing, the performance of generic code must be similar to that of specialized code.

First goal of this thesis is to understand what is the performance penalty for using generic code for scientific computing. Ideally, programmers should be able to write expressive programs while compilers should do the optimization. Most benchmarks implement algorithms as efficiently as possible by specializing the code for particular data sets and even performing some hand optimizations in the source code to eliminate the impact of wether certain optimizations are performed. We have implemented SciGMark, a benchmark with both generic and specialized code that shows us the difference in execution time between a highly optimized version and a generic version.

The results obtained by SciGMark show that generic code lags far behind hand-specialized code.

The second goal of this thesis is to provide optimization ideas that would allow writing generic code with less performance penalty. For this, we propose a compiler optimization called *type tower*. The type tower optimization is split into two parts: code specialization and data layout optimization. We have evaluated this optimization for the Aldor programming language because it has the most general support for parametric polymorphism, compared to C++, C#, and Java.

The code specialization optimization specializes the generic types by producing clones of the original type and then transforming the clone to replace the variable type parameter with constant values. This allows optimizations that are not possible in a general context. This optimization alone produces important speedups at the cost of an increased code size.

The data layout optimization relies on code specialization optimization and improves performance by rearranging the data representation such that types composed from several smaller types are grouped together in a larger aggregated type. It is then possible to optimize the operations belonging to this specialized type to produce better memory behavior by reducing the number of memory allocations performed.

The thesis presents tools that are able to measure the performance overhead imposed by the use of generic code. We have shown that a price must be paid for using parametric polymorphism with the current generation of compilers. However, the cost can be substantially reduced by the proposed optimizations.

Keywords: parametric polymorphism, generics, dependent types, compiler optimization, specialization, partial evaluation, generic code benchmark

Acknowledgments

First, I would like to thank my supervisor, Dr. Stephen M. Watt who gave me the chance to work on this remarkable project and who was always there to guide every step of my work.

Thanks to people in the Ontario Research Centre for Computer Algebra for many great discussions. I would like to thank Ben, Cosmin and Oleg for their comments on earlier drafts.

Most of all, I am grateful to my wife for her support and understanding during this thesis. Her love and support helped me go through the hardest of times. Thank you Magdalena, I could not have done it without you.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Importance of Parametric Polymorphism

Parametric polymorphism, a language feature that allows programs to be written using type parameters, is now supported by several modern programming languages such as Ada, Aldor, C++, C# and Java. In C++ the language constructs supporting parametric polymorphism are known as "templates" and in Java, C# and Ada, they are known as "generics".

Good examples of the utility of generic libraries are provided by the C++ Standard Template Library (STL) and the Boost libraries [1]. There are many computer algebra libraries using parametric polymorphism: the NTL library for number theory [59], the LinBox library for symbolic linear algebra [25], the Sum-IT library for differential operators [9] and the BasicMath library [41], to restrict attention to just a few. In Java and C#, type-safe collections use generics. The feasibility of generic code for scientific computing has been investigated in [28, 67].

Previous studies [28, 51, 67] have shown that modern implementations of C++, C# and Java now have sufficient performance for traditional scientific computing. As scientific computing evolves to take greater advantage of modern programming language features, we must understand which of these can be implemented sufficiently efficiently for numerically intensive codes. In this thesis we start by understanding what are the costs associated with the use of parametric polymorphism. Once the costs have been identified, we propose solutions to reduce the cost of using parametric polymorphism.

In some programming languages such as Java or C#, the generics were provided as a mechanism to implement generic collections. This can be seen from the difficulties encountered when trying to use generics to create generic algorithms. In other languages such as C++ and Aldor, generics are very powerful and allow implementation of generic algorithms.

1.2 The Cost of Parametric Polymorphism

Although parametric polymorphism has been widely accepted in symbolic computation, and for "container" libraries in object-oriented computing, it has not yet been widely adopted in numerically intensive computation. One of the prerequisites for its adoption in this context is a clear understanding of its cost.

To address this question we have developed a benchmark for generics in scientific computing. We call it the "SciGMark" benchmark because it is an extension of the well known SciMark benchmark [51] using generics.

The benchmark suite contains various language implementations of a number of tests. The first version of SciGMark contains all of the tests from SciMark for Aldor, C++, C# and Java in both specialized and generic form. We have added polyno-

mial multiplication and matrix multiplication with a recursive data representation to represent some types of scientific computation missing from the original suite.

Initially, Java's performance was unacceptable for numerically intensive computation. To encourage performance improvement of Java for numerical computation, benchmarks where created to measure the performance relative to higher-performance languages such as Fortran and `C`. By running the Scimark benchmark on new Java virtual machines one can observe a dramatic increase in performance to the point that Java implementations can be comparable to `C` and, in some cases, even faster. We anticipate that the same performance evolution could occur for generics.

With parametric polymorphism available in certain mainstream languages, such as C++, Java and C#, we foresee an increased reliance on generic code. To support this, compilers must be able to optimize generic code to an acceptable level. We therefore need benchmarks to measure the performance of compilers in this area. We hope that the SciGMark benchmark will help in this regard.

1.3 Improving the Performance of Parametric Polymorphism

The implementation of parametric polymorphism admits many optimizations. It should be possible, in principle, to see similar performance for code that uses parametric types and hand-specialized code, provided the compiler is able to perform suitable code transformations. In their current state, however, the compilers we tested fall far short of achieving this. We believe that providing a benchmark for generics in scientific computing can help improve this situation.

The nature of scientific computing can place different emphasis on the performance of generics than other programming styles. In most object-oriented programming, objects are created and then, their state is modified through the invocation of a series of methods. In mathematical computing, expressions tend to be more functional, with objects being short-lived and unmodified. The optimization of generic code must take this into account.

We have chosen the Aldor programming language to evaluate our proposed optimizations because the Aldor support for parametric polymorphism is the most general compared to C++, C#, and Java. Generic programming in Aldor is less cumbersome than in C++, and more flexible than in C# or Java. Moreover, Aldor allows for types to be constructed at runtime making its parametric polymorphism a more interesting setting for optimizations.

The Aldor programming language is particularly well suited for programs related to computer algebra. The language provides extensive generic libraries for mathematical objects. This makes it possible to construct types that are based on compositions of generic types that lead to what we call "type towers". One such example could be:

```
DenseUnivariatePolynomial(Complex(Fraction(BigInteger)))
```

Such "towers" of types lead to performance degradation if the compiler is not able to deal gracefully with their optimization.

Generic code cannot be fully optimized in generic form, so we have used partial evaluation as a method to specialize the code. However, unlike general partial evaluators, we applied our optimization in the special setting of parametric types. This has permitted us to create an efficient implementation of the specializer. The current implementation performs full specialization, constructing specialized versions for each type instantiation. For places where only partial information is available, a par-

tial specialization of the type is performed. Optimization by code specialization was studied in general by some interprocedural methods like cloning [17]. Customization is another code specialization technique used by the SELF programming language where possible values for the type parameters are guessed and specializations based on the guessed values are created during the compilation phase. At run-time, particular specializations are picked from the ones already created [14]. A different approach is used in partial evaluators, where the specialization is done based on some inputs to the program. Partial evaluation for object-oriented languages has been presented in [56, 57]. Specialization of virtual calls in object-oriented languages has also been studied in [20].

Code specialization has two drawbacks: the code size is increased and the compilation process takes more time to finish because of the increased number of domains that have to be compiled.

To measure the performance of the type tower optimization, some simple tests were designed that showed significant improvements. However, the simple tests were not representative of real world programs. As such, SciGMark was used to measure the performance obtained for algorithms used in scientific computation. After using SciGMark, it was clear that there was still room for improvement.

In creating the SciGMark benchmark suite, we identified two sources for the performance gains achieved by converting from the generic model to the specialized model. One of the aspects was code specialization, and other was data representation specialization. It was clear that by *replacing a generic data type* with a specialized form, further improvements could be obtained.

Data representation specialization for object-oriented programming languages has been studied in [21]. Their proposed method was to fuse the objects that have a one

to one relation between the parent and the child. The articles also prove that, for this particular case, fusing the objects preserves the semantics of the program.

Data specialization optimization analyzes the data inlining in the case of parametric polymorphism, and the particularities that might simplify the data flow analysis in this case.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. The second chapter presents background information related to parametric polymorphism and compiler optimizations. It then describes the Aldor compiler, in particular, with its run-time system and intermediate language. The third chapter presents the SciGMark benchmark suite that is used to measure the compiler's optimization power with respect to parametric polymorphism. The fourth chapter describes the code specialization optimization implemented for the Aldor compiler and some additional tests performed in programming languages such as C++, C# and Java. The fifth chapter presents an analysis of the data specialization optimization implemented for Aldor compiler. Finally, the sixth chapter summarizes and presents some final conclusions.

Chapter 2

Background and Related Work

2.1 Parametric Polymorphism

This section makes a few observations about the implementation of generics in the languages and compilers we tested, and on how this affected the implementation of the tests.

Polymorphism, in the context of programming languages, means that same code can be used with different types. If a functions accepts for its arguments, or result, values of different types, it is said to be polymorphic. Polymorphic types are types whose operations are applicable to values of more than one type [12]. Strachey [60] divided the type polymorphism into two distinct categories: *ad-hoc* polymorphism and *parametric* polymorphism. Cardelli and Wegner [12] later refined the polymorphism categories into *universal* and *ad-hoc*. The universal polymorphism is further divided into *parametric* and *inclusion*, while ad-hoc consists of *overloading* and *coercion*. The *ad-hoc polymorphism* is obtained when a function works, or appears to work, on several different types (which might not exhibit a common structure) and may behave in unrelated ways for each type. We are not concerned in this thesis about this

kind of polymorphism. *Parametric polymorphism* is obtained when a function works uniformly on a range of types: these types normally exhibit a common structure.

Parametric polymorphism was introduced as early as CLU [37] and ML [40, 39] programming languages more than three decades ago, and has now become important in mainstream languages. For example, it is supported in Ada, C++, C#, Java and Modula-3.

Parametric polymorphism can be implemented in different ways, each of which has different overhead trade-offs. Currently, there are two main approaches: the "homogeneous" approach and the "heterogeneous" approach.

The *heterogeneous* approach constructs a special case class for each different use of type parameters. For example, with `std::vector` from the C++ STL, one can construct `std::vector<int>` and `std::vector<double>`. Because C++ uses the heterogeneous approach, two distinct classes are generated for the above cases: one with the type parameter replaced by `int` and one with it replaced by `double`. This approach duplicates the code of the `std::vector` generic class and produces different specialized compiled forms.

This approach has the benefit that the compiled code is specialized, and therefore fast. The drawback is that object code can be bulky, with many different versions of each class. Bulky object code can cause problems due to space constraints at any level of the memory hierarchy. Furthermore, much more sophisticated system software is required if generics are to be instantiated at run time.

The *homogeneous* approach uses the same instance of code for all uses of the type parameters. Specialized behavior is achieved through inheritance polymorphism of variables belonging to the parameter type. Java uses this approach by "erasing" type information and using the `Object` class instead of the specialized form. Instances of `Object` are cast back to the target class whenever necessary. This method has

Listing 2.1: Generic swap function in the Ada programming language.

```
1  -- Function declaration
2  generic
3    type Element_Type is private;
4  procedure Generic_Swap(Left, Right : in out Element_Type);
5
6  -- function definition
7  procedure Generic_Swap(Left, Right : in out Element_Type) is
8    Temporary : Element_Type;
9  begin
10   Temporary := Left;
11   Left := Right;
12   Right := Temporary;
13 end Generic_Swap;
```

an overhead comparable to that of subclassing, and the code size is comparable to the non-generic version. For example, `Vector<Integer>` will be transformed to a `Vector` that contains `Object` type values, and the compiler will check if the type of the elements inserted into the collection are objects of type `Integer` (or a subclass). This approach allows the same code to be used for all instances of `Vector`, therefore `Vector<Double>` is also just `Vector`.

In some languages such as C# and Java, it is possible to specify a class or an interface as a *bound* that declares the operations that are allowed on the type parameter. For example, let us suppose that we have an interface `Addable` which declares the `add` operations. If one constructs a collection `Aggregate<T extends Addable>`, it is possible to create a `sum` operation that adds all the elements in the collection. This would not be possible with a type `Aggregate<T>` since `Object` class does not have an `add` operation.

Aldor uses a variation on the homogeneous approach. Instead of obtaining the specialized behavior from sub-classing polymorphism, Aldor has types that are first-

Listing 2.2: Instantiating the generic swap function.

```
1  procedure Swap is new Generic_Swap(Integer);
2  procedure Swap is new Generic_Swap(Float);
```

class values, allowing it to use types like any other variable. This provides excellent flexibility and the ability to construct domains at runtime. However, the cost of this implementation is decreased execution speed. This is even more visible when the parameters of the domains are other parametric domains, leading to a deeply nested type (or type tower). Because of the way Aldor is implemented, extensive type towers affect the performance.

2.1.1 Ada

Ada uses generics to implement parametric polymorphism. Implementation details of Ada generics are provided in, [8].

An example of generic programming is given by the generic swapping function that is presented in Listing 2.1. To create a generic package, one has to begin the package with the keyword "generic" and a list of generic formal parameters. The list of formal parameters is like the list of parameters in a procedure declaration. For the implementation, the names declared in the generic formal parameters list can be used as types inside the package.

To be able to use a generic package, we have to create a real package from the generic version. This process is called *instantiation*, and the result is called an *instance*. These are big words for a simple concept. An example of two Swap procedure instances instantiated from the generic one can be seen in Listing 2.2.

Listing 2.3: An usage example for Ada generics.

```
1 with Generic_Swap;
2 procedure Tswap is
3  procedure Swap is new Generic_Swap(Integer);
4  A, B : Integer;
5 begin
6  A := 5;
7  B := 7;
8  Swap(A, B);
9 end Tswap;
```

Listing 2.2 overloads the `Swap` function with two definitions by instantiating the generic function, `Generic_Swap`, with two types: `Integer` and `Float`. Listing 2.3 shows an example of usage of the `Generic_Swap` function.

2.1.2 Aldor

Aldor [27, 70, 74] was designed as an extension language for the Axiom computer algebra system [31]. It became a general purpose programming language that placed an emphasis on the uniform handling of functions and types, and less emphasis on a particular object model. However, even though Aldor has a general purpose library, it also comes bundled with computer algebra specific libraries, showing its affinity towards computer algebra. The Aldor programming language was designed to be efficient and to offer support for optimizations. Many optimizations were already implemented in the Aldor compiler, making it possible to write code that had comparable efficiency to C or Fortran compilers.

The rich type system made Aldor a good candidate to test type-related optimizations ideas. This section gives a brief introduction to the Aldor programming language, with emphasis on its capabilities related to generic types.

Listing 2.4: Aldor example of dependent types.

```
1 suml(R: ArithmeticType, l: List R): R == {
2     s: R := 0;
3     for x in l repeat s := s + x;
4     s
5 }
```

Functions as first class values are a characteristic of functional programming languages. However, Aldor goes beyond that and uses types as first class values. When types and functions are first class values they can be created and manipulated just like any other values. This allows the rich relationships between mathematical structures to be modeled efficiently.

An unusual feature of Aldor is its pervasive use of *dependent types* [49]. This allows the *type* of one subexpression to depend on the *value* of another. It also allows normal functions to provide parametric polymorphism, see Listing 2.4. In this example, the variable `l` is a list that contains elements of type `R`. `R` is also used to give the return type of the function.

The type system in Aldor is organized on two levels: *domains* and *categories*. The categories represent the types of domains, and they declare the operations that can be performed on an implementing domain. They are analogous to interfaces of Java or C#.

Cardelli and Wegner [12] introduced the notion of *bounded quantifiers* for types. They presented universal, existential and subtyping quantifiers. Later, Canning et al. [11], introduced the notion of *F-bounded polymorphism*. The F-bounded polymorphism allows for the bound to be a function of the bounded variable. An example of F-bounded polymorphism given by Oderski and Wadler in Pizza [46] is the `Pair` class:

Listing 2.5: Aldor implementation of `Ring interface`.

```
1  define IRing: Category == with {
2     +        : (%, %) -> %;
3     a        : (%, %) -> %;
4     clone    : %      -> %;
5     newArray : int    -> Array %;
6  };
7
8  define IMyComplex(E: IRing): Category == IRing with {
9     create: (E,E) -> %;
10    getRe : %     -> E;
11    setRe : (%,E) -> ();
12    getIm : %     -> E;
13    setIm : (%,E) -> ();
14 };
15
16 define MyComplex(E: IRing): IMyComplex(E)
17 == add {
18    Rep == Record(r: E, i: E);
19    (t:%) + (o:%): % == per [rep(t).r+rep(o).r, rep(t).i+rep(o).i];
20    a(t:%, o:%): % == per [a(rep(t).r,rep(o).r), a(rep(t).i+rep(o).i)];
21 }
```

```
class Pair<elem implements Ord<elem>>
```

Dependent types are similar to the bounded polymorphism provided by Java and C#, but they allow better type checking for generic types. In bounded polymorphism, as seen in object-oriented languages, the type instantiation can be the bound or any sub-class of the bound. With dependent types the type instantiation must be exactly the type used as parameter. For instance, in Listing 2.4, if `R` is the type `Integer`, then the list `l` will contain only `Integer` elements.

In the following sections, we continue by presenting the features of Aldor as we describe the construction of a new type representing complex numbers. We shall use the same example for the C++, C# and Java programming languages. In Listing 2.5, a new category `IRing` is defined. In Aldor, primitive categories are created by using

the keyword `with`. The operations available from the `IRing` category are `+`, `a`, `clone`, and `newArray`. The type of a symbol is introduced after the colon. For example, "`+`" has the function type `(%,%)->%`. The `%` symbol has a special meaning in Aldor, namely the type of the current domain.

Categories can be extended by specifying the base category and new exports. This is done by placing a category before the `with` keyword and putting the new operations inside the `with` block. In our example, the `IMyComplex` category extends the `IRing` category with methods: `create`, `getRe`, `setRe`, `getIm` and `setIm`.

A primitive domain is defined using an `add` statement. Domains can be used either as packages or as abstract data types. As packages, they represent a collection of named values (functions, types or values), and as abstract data types they represent a distinguished type and a collection of related exported named values. In the `MyComplex` domain definition, the type of the domain is `IMyComplex`. This means that `MyComplex` should provide implementations for all the operations declared in `IMyComplex`.

Throughout this thesis we use domains as abstract data types.

When used as abstract data types, domains contain a representation for the distinguished type. The representation is denoted by the constant `Rep`, which is the internal representation of the type. The public view of the data type is accessible via the name `%`. To convert between the public and private view of the data type, Aldor offers two macros:

$$per : Rep \to \%$$

$$rep : \% \to Rep$$

Domains can be passed as parameters to functions. In the definition of `MyComplex`, a type parameter `E` of type `IRing` is provided which represents the real and imaginary

part of the complex number. The type of the elements can be of any type that satisfies the category `IRing`. This means that the operations declared in `IRing` can be called on the real and imaginary parts of `IMyComplex`. The representation of the `MyComplex` domain is a record with two fields `r` and `i`, both of type `E`.

`DoubleRing` is a wrapper of the `DoubleFloat` type which implements the category `IRing`. The `DoubleRing` domain can be used as a parameter for `MyComplex` by using the instantiation:

```
MyComplex(DoubleRing)
```

This construction is similar to the use of generics in Java and C# or templates in C++.

2.1.3 C++

In C++, parametric polymorphism is achieved by using templates [61, 62]. C++ templates are implemented using a macro engine that expands templates to generate the specialized code. When the parameter is instantiated, the compiler checks and generates the resulting code. This approach allows more straightforward code specialization and compile-time optimization.

C++ does not support bounded polymorphism. That is, it does not allow any qualifications on the types used as a parameters in the templates. As a consequence compilers cannot check parametric code for correctness until the template is instantiated. Type checking is thus deferred to the moment of instantiation. Whenever a template is instantiated, the actual definition of the template is specialized by replacing the template parameter with the actual instance. For example, to create a container `std::vector` with elements of type `int`, we would write: `std::vector<int>`. At this point, the code for `std::vector` is duplicated, and the parameter of the `std::vector` template is replaced by `int`. The specialized code is then checked by the compiler.

The lack of proper support for bounded polymorphism in C++ is compensated by clever tricks using the template specialization technique to restrict the generic code in special cases. One such technique is called SFINAE which is an acronym for "substitution failure is not an error". The idea of SFINAE is to use template specialization to provide a function definition only for some type parameters. For the types that are forbidden, but which provide the operations required by the template, a template specialization is provided that produces a compilation error. Particular cases can be used to implement bounded polymorphism. This requires template metaprogramming to check the polymorphism constraints explicitly. We do not find this to be a completely satisfactory implementation.

In C++, it is the programmer's responsibility to instantiate the template with a type that has the required operations. Unfortunately, because it is not possible to type check the generic code, the error messages are usually misleading. In addition, if the generic code is not properly tested and documented instantiations may reveal problems in the code only at a later stage of code development.

Another issue with the templates from C++ is the requirement that the sources of the templates be made available to the instantiating code. This requirement forces the programmer to provide the implementation in the source form in the header files. Not all software vendors wish to do this.

The `IRing` class for C++ can be implemented as shown in Listing 2.6. C++ does not have interfaces, but similar semantics can be obtained with the use of abstract classes with all methods being pure virtual. This declaration would force all the calls to be virtual, reducing the performance of the C++ implementation. To avoid this, the classes corresponding the interfaces in other programming languages (the classes starting with the letter I) do not use virtual functions.

Listing 2.6: IRing class in C++.

```
template <typename T>
class IRing {
public:
        void ae(const T& o);
        T operator+(const T& o) const;
        T a(const T& o) const;
        . . .
};

template <typename C, typename E>
class IComplex : IRing<C> {
public:
        C create(E re, E im) const;
        E getRe() const;
        void setRe(E re);
        E getIm() const;
        void setIm(E im);
};

template<typename R> class Complex : IComplex<Complex<R>,R> {
        R re, im;
public:
        Complex(R re = 0, R im = 0)     {this->re = re; this->im = im;}
        Complex<R> operator+(const Complex<R>& o) const
        {
                Complex<R> t;
                t.re = re + o.getRe(); t.im = im + o.getIm();
                return t;
        }
        . . .
};
```

The `IRing` class is parametric because the C++ language does not offer the % concept as seen in the Aldor programming language. Inside the definition of the `IRing` class, the type of the actual class which implements the `IRing` operations is not known. For this reason, the class that implements the `IRing` uses itself as a parameter for the `IRing` class. This construction allows `IRing` to have more precise signatures for its operations. For example, if `IRing` is not parametric then the functions must use the type `IRing` and the implementing class must also use `IRing` to override it. The `operator+` function from `IRing` would be declared as: `IRing operator+(IRing o)` instead of the declaration from Listing 2.6. The overriding from implementing class `Complex` would have to be `IRing operator+(IRing o)` as well. This new declaration requires a cast which may lead to runtime errors, by deferring the type checking from compile-time to run-time.

The `IComplex` class (Listing 2.6) extends the `IRing` class with operations: `create`, `getRe`, `setRe`, `getIm`, and `setIm`. Like `IRing`, an extra type parameter is required to pass the type of the implementing class. The class `Complex`, from Listing 2.6, shows an example of implementation of complex numbers using parametric polymorphism.

The initial intention of the templates was to provide better support for generic programming. Later, it was discovered that with template specialization, it was possible to write code that was evaluated by the C++ compiler while expanding the templates. Veldhuizen [65] explained how to use templates to perform static computation. An example of compile-time (static) computation that was presented in [65] is given in Listing 2.7. The code presented in Listing 2.7 computes X^Y at compile time and the constant `z` will have the value 125 when the executable code is produced. This computation with templates is called template meta-programming.

Listing 2.7: Example of templates used to perform the computation of X^Y.

```
template<int X, int Y>
struct ctime_pow {
        static const int result = X * ctime_pow<X, Y-1>::result;
};

//Base case to terminate recursion
template<int X>
struct ctime_pow<X,0> {
        static const int result = 1;
};

//Example use:
const int z = ctime_pow<5,3>::result; // z = 125
```

2.1.4 C#

The implementation of generics in C# is described by Kennedy and Syme [34, 38, 75]. Like Java, C# is compiled to an intermediate language. The intermediate language is executed by the runtime environment known as the Common Language Runtime (CLR). The CLR is similar to the Java virtual machine. The whole software development framework is called the .NET framework.

C# generics are implemented at the CLR level. The advantage of the C# implementation of generics, compared to the Java implementation, is that type information is retained at runtime, making optimizations by a just-in-time compiler possible. This also avoids some of the restrictions imposed by generics as implemented in Java. For example, in C#, unlike Java, it is possible to create instances whose type is of the type parameter.

C# provides a language-level differentiation between heap allocated and stack allocated objects. This gives the programmer the choice between small objects that have fast stack allocation and slower copy time, or larger heap allocated objects that

Listing 2.8: C# implementation of `Ring type`.

```
1  public interface IRing<T> {
2     T a(T other_elem);
3     void ae(T other_elem);
4     T clone();
5     T newInstance();
6     T[] newArray(int size);
7  }
8
9  public interface IComplex<C, E>: IRing<C> where E: IRing<E> {
10    C create(E re, E im);
11    E getRe();
12    void setRe(E re);
13    E getIm();
14    void setIm(E im);
15 }
16
17 public struct Complex <R>: IComplex<Complex<R>,R>
18                                  where R: IRing<R> {
19    public Complex<R> a(Complex<R> o) {
20     return new Complex<R>(re.a(o.re), im.a(o.im));
21    }
22 }
```

are manipulated by reference. The stack-allocated objects are constructed using the keyword `struct`, while the heap-allocated objects are constructed using the keyword `class` [30]. C# also allows the use of basic types as type parameters. In contrast to Java, in C# it is possible to use `Complex<double>` instead of `Complex<DoubleRing>`.

CLR uses a mixed approach by implementing heterogeneous parametric polymorphism for basic types (similar to C++) and a homogeneous approach for reference types (similar to Java). When it is necessary for a value type to behave like an object, it is boxed by allocating a new object on the heap and copying the value into the newly allocated object [36].

In C#, there are two types of collections: generic and non-generic. Non-generic

collections store all the elements as references with type `object`. This can lead to heterogeneous collections that are not type safe. The alternative, generic collections, introduced in version 2.0 of .NET, use parametric polymorphism. They are type safe and exhibit better performance than non-generic collections. Any stack-allocated object that is stored in a non-generic collection is automatically boxed by the compiler, leading to decreased performance compared to using classes. The use of generic collections together with stack-allocated objects might increase performance, since no auto-boxing is performed for generic collections. For this reason, types are implemented as stack-allocated structures, allowing the CLR to optimize the structure's allocation. An example similar to the previous one, this time for C# generics, is presented in Listing 2.8.

2.1.5 Java

There were several different proposals to introduce generics for Java. One of the first proposed approach was to use virtual types as described by Thorup [64]. A different extension called "Pizza" was proposed by Odersky and Wadler [46]. Pizza extended Java with parametric polymorphism, higher-order functions and algebraic data types. The Pizza compiler implemented both heterogeneous and homogenous approach for translating generic types into Java types and the conclusion was that in Java the heterogenous approach did not produce big speedups while increasing the size of the code [45].

Other proposed approaches to implement generics in Java were presented by Bank, Myers and Liskov in [4]. They proposed two new instructions `invokewhere` and `invokestaticwhere` to the Java bytecode to deal with bounded polymorphism.

Another idea used a hybrid approach: one erased class plus one thin wrapper for each parameterized class is NextGen [13]. Viroli and Natali [68, 69] propose a method to carry the type information for parametric types at run-time, and in order to avoid the computational overhead, create instances at load-time. This has a small impact on the overall performance.

Version 5 of Sun's Java compiler uses the GJ [6, 7] approach to implement generics. An advantage of this approach is that it does not require changes to the virtual machine and has no additional performance penalty beyond the subclassing polymorphism commonly used in Java. The resulting code is the same as in the version that does not use generics. However, in order to keep the virtual machine compatible, some functionality and potential speedup has been sacrificed. An example of such restriction is the inability to construct a new object that has the class of the type parameter. This design was used by Sun in their compiler and is now part of Java.

Generic programming in Java performs static type checking on the type parameters and then erases the type information of the type parameters. After the erasure, the specialized behavior is obtained by the sub-classing polymorphism. For example, a `List <Integer>` is actually transformed by the compiler to `List` that contains `Object`-type elements. In Java, `Object` is the super-type of all types, so any object is of type `Object`. By replacing the actual type with `Object`, the type information is *erased*. This implementation is called the *erasure technique*. Elements are cast down to `Integer` when extracted from the `List`. This homogeneous approach uses the same representation for all instantiations, and does not increase the code size.

One problem using generics in Java is that one cannot instantiate parameters using primitive types, such as `int`, `float`, `char` and so on. The problem is partially addressed using wrapper classes and "auto-boxing", a language feature for automatic conversion between primitive types and reference types. Another problem is that the

Listing 2.9: Java implementation of `Ring` type.

```
interface IRing <T> {
   T a(T other);
   void ae(T other);
   T clone();
   T[] newArray(int size);
   T newInstance();
}
interface IComplex<C, E extends IRing<E>> extends IRing<C> {
   create(E re, E im);
   E re();
   void setRe(E re);
   E im();
   void setIm(E im);
}
```

type information is lost at compile time, making it impossible to construct a new instance of the type parameter [35]. There are two ways to solve this problem: by creating a factory object to produce new instances or by using reflective features of the language, such as `Class.newInstance`.

Examples of generic code in Java can be seen in the Listing 2.9.

Java uses bounded polymorphism by specifying the class or interface that is a super-type of the actual type parameter. When the instance of the type is erased, it is erased to the specified super-type, giving the compiler information about what methods can be invoked on the type parameter. This way it is possible to compile the class without knowing the actual instantiation that will be used. By using the `final` keyword, one could ensure that a class will not be extended. However, there is no way to specify as a requirement a final interface as a parameter to the type parameter.

The code that uses the `IRing` and `IComplex` interfaces is presented in Listing 2.10. By explicitly specifying the `Complex` class as a type parameter to `IComplex`,

Listing 2.10: Java implementation of the `Complex` type.

```
1 public class Complex <R extends IRing<R>>
2 implements IComplex<Complex<R>,R> {
3    public Complex<R> a(Complex<R> o) {
4      return new Complex<R>(re.a(o.re()), im.a(o.im()));
5    }
6 }
```

the Java type inference will issue fewer warnings and the code will be cleaner, because `IComplex` interface declare the "`a`" method as `Complex a(Complex a)`. If `IRing` and `IComplex` would not have known the `Complex` type, the method to be overridden would have been `IRing a(IRing a)`. This second case can be made to work, but it issues many unchecked warnings. It also requires some runtime type casts from `IRing` to the type in which the method is defined, leading to possible runtime exceptions.

To work around Java shortcomings, we specified a `newInstance` method to create new values of the parameter type. Each program had to store a sample value belonging to the parameter type and use it to construct new values later on. We could have used reflective features, but reflective features are slower than normal method invocation.

2.1.6 Maple

Maple offers the possibility to create modules. It is also possible to have functions that return modules. Thus, it is possible to implement parametric polymorphism. Another possibility of using generic code from Maple is to access Aldor through an interface with the help of the ALMA framework [44]. This approach would bring the benefits of the extensive support for generic programming and high performance provided Aldor to Maple. However, one would be also interested in the consequences

Listing 2.11: Maple Example

```
IMod := proc (p)
    module()
        export zero,
               `+`, `*`, `=`,
               convertIn, convertOut;
        zero:= 0;
        `+` := (a,b) -> a+b mod p;
        `*` := (a,b) -> a*b mod p;
        `=` := (a,b) -> evalb(a = b);
        convertIn := proc(n)
            if not type(n, 'integer') then
                error("Bad number")
            end if;
            n mod p
        end;
        convertOut := v -> v;
    end module
end proc:
```

of using parametric polymorphism in Maple, so we describe in the following how this can be achieved by using only Maple.

An example can be seen in Listings 2.11, 2.12, 2.13 and 2.14. We have created a parametric type representing a modular integer `IMod` (Listing 2.11), and a parametric domain `Poly` using a dense representation (Listings 2.12 and 2.13). `Poly` uses as parameter an object of type Ring. `IMod`, is an implementation of such an object. Unfortunately, in Maple is not possible to specify the type of the parameter if the parameter has a user defined type as is the case with modules. The types export functions like `=`, `+`, `*`, `convertIn`, `convertOut`; `Poly` provides also the operations `pcoef` and `pdegree` (these names were use to avoid collision with standard `coeff` and `degree` functions. This example shows the powerful mechanism provided by the Maple modules to write generic algorithms.

Listing 2.12: Maple example (The polynomial).

```
Poly := proc (x) proc(R)
     module()
         export zero, `+`, `*`, `=`, convertIn, convertOut, pcoef, pdegree;
         local fixDegree;
         zero := [];
         # Drop leading coefficients equal to zero.
         dropZeros := proc(p)
              local i, dtrue, d;
              d := pdegree(p);
              dtrue := -1;
              for i from d to 0 by -1 do
                  if not R:-`=`(R:-zero, pcoef(p,i)) then
                      dtrue := i;
                      break
                  end if
              end do;
              if d = dtrue then p else [op(0..dtrue+1, p)] end if
         end proc:
         `+` := proc(p,q)
              local i, d;
              d := max(pdegree(p), pdegree(q));
              dropZeros([seq(R:-`+`(pcoef(p,i), pcoef(q,i)), i=0..d)]);
         end proc:
         `*` := proc(p, q)
             local a, i, j, pd, qd;

             if p = zero or q = zero then return zero end if;

             pd := pdegree(p); qd := pdegree(q);
             a := array(0..pd+qd);
             for i from 0 to pd + qd do a[i] := R:-zero(); end do;
             for i from 0 to pd do
                 for j from 0 to qd do
                     a[i+j] := R:-`+`(a[i+j],
                                      R:-`*`(pcoef(p,i), pcoef(q,j)));
                 end do
             end do;
             [seq(a[i], i = 0..pd+qd)]
         end proc:
```

Listing 2.13: Maple example (The polynomial Part 2).

```
        `=` := proc(p,q)
            local i;
            if pdegree(p) <> pdegree(q) then return false end if;
            for i from 0 to degree(p) do
                if not R:-`=`(pcoef(p,i), pcoef(q,i)) then
                    return false
                end if
            end do;
            true
        end proc:
        convertIn := proc(w0)
            local i, w;
            w := collect(w0,x);
            dropZeros(
                [seq(R:-convertIn(coeff(w,x,i)), i=0..degree(w,x))])
        end proc;
        convertOut := proc(p)
            local i, dp;
            dp := pdegree(p);
            add(R:-convertOut(pcoef(p,dp-i)) * x^(dp-i), i = 0..dp)
        end proc;
        pdegree := p -> nops(p) - 1;
        pcoef := proc(p,i)
            if i+1<=nops(p) then p[i+1] else R:-zero end if
        end proc:
    end module
end proc end proc:
```

Listing 2.14: Maple Example (Usage example).

```
Px := Poly(x);
Py := Poly(y);
Pz := Poly(z);

pm := Px(Py(Pz(IMod(17))));

a0   := 2*x^2+4*y^2+3*z^2+(2*x*y*z)+y*(4*x+5*z);
a    := pm:-convertIn(a0);
oa   := pm:-convertOut(a);
o2a := pm:-convertOut(pm:-`+`(a,a));
oaa := pm:-convertOut(pm:-`*`(a,a));
```

2.2 Compiler Optimizations

2.2.1 Interprocedural Analysis and Optimization

Modularity is very important for programming languages because it allows creation of well-designed programs that are easier to understand and maintain. Modular programs separate the program into several procedures. Each procedure presents an interface and acts as black box by hiding the implementation details inside its body and offering some functionality which is described by its name. When needed, the caller selects a procedure based on name and provides some arguments that follow the rules of the interface. This separation of the problem into smaller sub-problems is a desired feature from the software engineering point of view. However, this approach makes it hard for a compiler to perform some optimizations.

Most compilers perform intraprocedural code analysis and optimizations. These optimizations have been thoroughly studied [3, 42]. Unfortunately, these optimization cannot send the information across procedure calls, and the compiler cannot make any assumptions about the data that has been passed in through the interface. As

described in [17], traditionally there are two approaches to solve this problem:

- locally expand the callee at the call site, procedure called *inlining* or *procedure unfolding* and then perform the regular intraprocedural optimizations
- perform interprocedural analysis and optimization.

Interprocedural analysis uses the calling structure of the program to construct and transmit the information between call sites. The interprocedural optimization consists of a sequence of phases that do control-flow analysis, data-flow analysis, alias analysis, and code transformations. Intraprocedural optimizations perform similar steps, but unlike interprocedural the scope of the analysis is restricted to the procedure.

Both solutions proposed to extend the scope of the optimizations beyond function boundaries have disadvantages. The inline expansion can lead to code growth, and as a direct consequence, to increased compilation time [18]. Code growth is a real problem with embedded devices where memory resource is limited. Other problems with inline expansion include: possible instruction cache misses, increased pressure on register allocation for local variables. In the cases of recursive function calls, is not even possible to perform function inlining. The interprocedural analysis can be rather expensive to implement and, at the end, results are not always worth the implementation effort and the increased compilation time.

Richardson and Ganapathi studied the effectiveness of interprocedural optimizations in comparison to procedure integration [52, 53, 54]. They have shown that procedure integration is very effective, but it increased the compilation time by a factor of 10. On the other hand, simple intraprocedural analysis was not very efficient, and the complexity of the compiler is increased considerably by the interprocedural analysis support. There is evidence that interprocedural analysis is more valuable for parallel architectures [10].

Control flow analysis constructs a program call graph. The other important aspect of the interprocedural analysis is the data flow analysis. This analysis provides information about the data manipulation, but unlike the intraprocedural analysis this analysis crosses the procedure barrier.

Interprocedural analysis is useful for optimizations like: procedure integration, constant propagation, register saving between calls, replacing call by value with call by reference for parameters that are not modified by the called procedure, using interprocedural data-flow information to enhance intraprocedural information.

2.2.2 Partial evaluation

The process of fixing one or more arguments of a program, to produce a residual program is called *partial evaluation*. Beckman [5] introduces the term *partial evaluation* as program specialization. In analysis, projection or restriction means fixing a parameter of a function of n variables, obtaining a function of $n-1$ variables. In logic, the same procedure is called *currying*.

There are many applications of partial evaluation in pattern recognition, computer graphics by ray tracing, neural network training, database queries, spreadsheet computation, scientific computing, and discrete hardware simulation. But one of the main applications is compiler generation and code compilation.

A graphic representation of a partial evaluator is shown in Figure 2.1 (adapted from [32]). One can see that the partial evaluator takes as input the source program and some input `in1` and produces a specialized program as a result. The specialized program takes as input the resulted inputs to the program and produces the output.

The first input is considered to be static, because once it is used by the partial

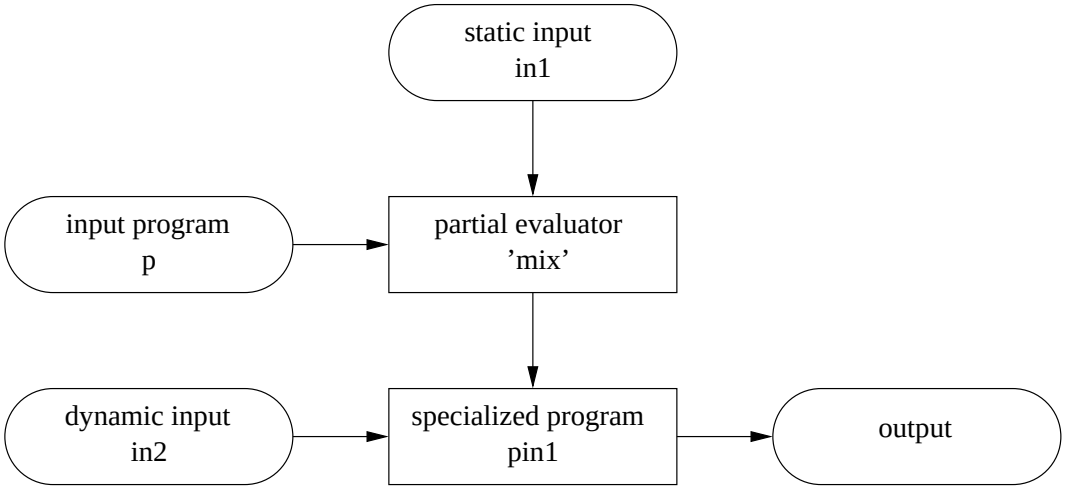

Figure 2.1: A partial evaluator.

evaluator it cannot be changed. The rest of the inputs are dynamic, since they can change at run-time in the residual program `pin1`.

Static inputs are inputs that are either given as constants for a subset of problems that can be solved by the generic program `p`. The static inputs can also change, but that requires a re-specialization of the program `p` for each change. This means that static inputs should be selected from the input data that does not vary as often as the rest. The advantage of a partial evaluator is that the time to produce the specialized form and the time to execute the specialized program on the dynamic input takes less than executing the program `p` on all input data as variable.

Partial evaluation relies on some techniques to specialize the program. These techniques are: symbolic evaluation, unfolding function calls, and program point specialization.

The example presented in Listing 2.15 can be specialized by applying partial evaluation techniques for a value "5" of the formal parameter `a`. The resulting code is presented in the Listing 2.16. The symbolic evaluation detected that the condition in the if statement is false for the input value of five, and therefore the body of if

Listing 2.15: Generic version of a function using Aldor programming language syntax.

```
1 f(a:AldorInteger, b:AldorInteger): AldorInteger == {
2     if (a > 10) then {
3         for i in 1 .. a repeat b := b + 2 * i;
4     }
5     2 * a + b;
6 }
```

Listing 2.16: Specialized version of a function using Aldor programming language syntax.

```
1 f__5(b:AldorInteger): AldorInteger == {
2     10 + b;
3 }
```

statement becomes dead code which is eliminated. The value of expression `2*a + b` evaluates to `10+b` by evaluating the subexpression `2*a`.

The example presented above uses only the symbolic evaluation part of the partial evaluation. There are no function calls in the body of `f` to be unfolded or specialization points.

The code simplification from the previous example can be performed by intraprocedural analysis and optimization once the specialization of the function has been performed. Function specialization can be performed only by partial evaluation or the interprocedural analysis.

In order to analyze the code, partial evaluators use either "concrete interpretation" or "abstract interpretation" of the code.

Abstract interpretation is an estimation of some properties of code without actually evaluating it. For example, one can find out if the the expression `5 * 2` is odd or even, by concrete interpretation which would evaluate the expression to `10` and then

determining whether the result is odd or even. With abstract interpretation, the expression is not evaluated, but the result is inferred, knowing that multiplication of an odd and an even number is always even. Abstract evaluation is even more important when the value of one of the operands is not known. If the given expression is `x * 2` it is possible to infer that the result is even.

For imperative languages such as `C`, a two-level execution solution is presented in [32]. They propose a simple language, called *Core `C`*. The Core `C` language is a "syntactic desugaring" of the `C` language which contains only a very simple set of instructions and types. The standard `C` language is translated to Core `C` and the resulting program is analyzed.

In Figure 2.2, one can see two parallel definitions for some non-terminals. The non-terminals starting with two belong an extended *two-level* Core `C` language, while the other ones belong to the regular Core `C` language.

Intuitively, all the underscored statements and expressions require code generation by the partial evaluator. The underscored statements and expressions are dynamic, since they cannot be evaluated statically, and the regular statements and expressions are static, since they can be evaluated statically.

The partial evaluator executes a two level Core `C` program. The dynamic statements and expressions are first reduced and then the residual code code corresponding to the reduced value is generated. The static expressions are evaluated. During the execution, all the variables and parameters are computed and stored to be able to optimize the code.

For function calls that need specialization, if an already existing specialization is found, that specialization is shared, otherwise the function is specialized by a recursive call of the executing program.

⟨2CC⟩ ::= ⟨2decl⟩* ⟨2fundef⟩*
⟨2decl⟩ ::= ⟨2type⟩ id ⟨2typespec⟩ | decl
⟨decl⟩ ::= ⟨type⟩ id ⟨typespec⟩
⟨2type⟩ ::= *base* | ⟨2type⟩ * | **struct** id { ⟨2decl⟩+ } | type
⟨type⟩ ::= *base* | ⟨type⟩ * | **struct** id { ⟨2decl⟩+ }
⟨2typespec⟩ ::= ⟨2typespec⟩ [const] | typespec
⟨typespec⟩ ::= ϵ | ⟨typespec⟩ [const] | typespec
⟨2fundef⟩ ::= ⟨2type⟩ fid (⟨2decl⟩*) { ⟨2decl⟩* ⟨2stmt⟩+ }
⟨fundef⟩ ::= ⟨type⟩ fid (⟨decl⟩*) { ⟨decl⟩* ⟨stmt⟩+ }
⟨2stmt⟩ ::= ⟨stmt⟩
| lab : **expr** ⟨2exp⟩
| lab : **return** ⟨2exp⟩
| lab : **goto** lab
| lab : **if** (⟨2exp⟩) lab lab
| lab : **call** id = fid (⟨2exp⟩*)
⟨stmt⟩ ::= lab : **expr** ⟨exp⟩
| lab : **return** ⟨exp⟩
| lab : **goto** lab
| lab : **if** (⟨exp⟩) lab lab
| lab : **call** id = fid (⟨exp⟩*)
⟨2exp⟩ ::= ⟨exp⟩ | **lift** ⟨exp⟩
| **struct** ⟨2exp⟩ . id
| **index** ⟨2exp⟩ [⟨2exp⟩]
| **indr** ⟨2exp⟩
| **addr** ⟨2exp⟩
| **unary** uop ⟨2exp⟩
| **binary** ⟨2exp⟩ bop ⟨2exp⟩
| **ecall** eid (⟨2exp⟩*)
| **alloc** (id)
| **assign** ⟨2exp⟩ = ⟨2exp⟩
⟨exp⟩ ::= **cst** const | **var** id
| **struct** ⟨exp⟩ . id
| **index** ⟨exp⟩ [⟨exp⟩]
| **indr** ⟨exp⟩
| **addr** ⟨exp⟩
| **unary** uop ⟨exp⟩
| **binary** ⟨exp⟩ bop ⟨exp⟩
| **ecall** eid (⟨exp⟩*)
| **alloc** (id)
| **assign** ⟨exp⟩ = ⟨exp⟩

Figure 2.2: Two level Core C presented in [32].

For the dynamic return statement, the expression is reduced and the value of the residual return is used.

2.3 The Aldor Compiler

The Aldor compiler is able to generate stand-alone executables, object libraries in native format for the target operating system, portable byte code libraries and C or Lisp sources. Aldor source code is transformed into FOAM, the intermediate language representation. All the optimizations are done at the FOAM level, and then the FOAM code is translated to native code for the OS or is left as is to be compiled or interpreted on the target operating systems [26, 72].

The compilation stages for the Aldor compiler are presented in Figure 2.3. The include phase includes all the files specified by the `#include` directives. The scan phase converts the text stream into a stream of tokens. The parsing phase performs a semantic analysis and converts the tokens stream into a parse tree. The macro expansion is performed on the parse tree. The result of all these phases is a parse tree. The parse tree is normalized, checked for correctness and, in the scope-binding phase, the scopes of the variables are deduced. The result of these phases is a an abstract syntax tree.

Type inference is performed on the resulting abstract syntax tree to verify the correctness of the program with respect to type usage. The result of type inference phase is a semantic form (`Sefo`) which is an abstract syntax tree annotated with type information. The symbol table (`Stab`) is also filled with information about each known symbol and its meaning. In the next phase, an intermediate code is generated. The Aldor intermediate code is FOAM (First Order Abstract Machine). The result of this phase is an intermediate FOAM representation of the source program.

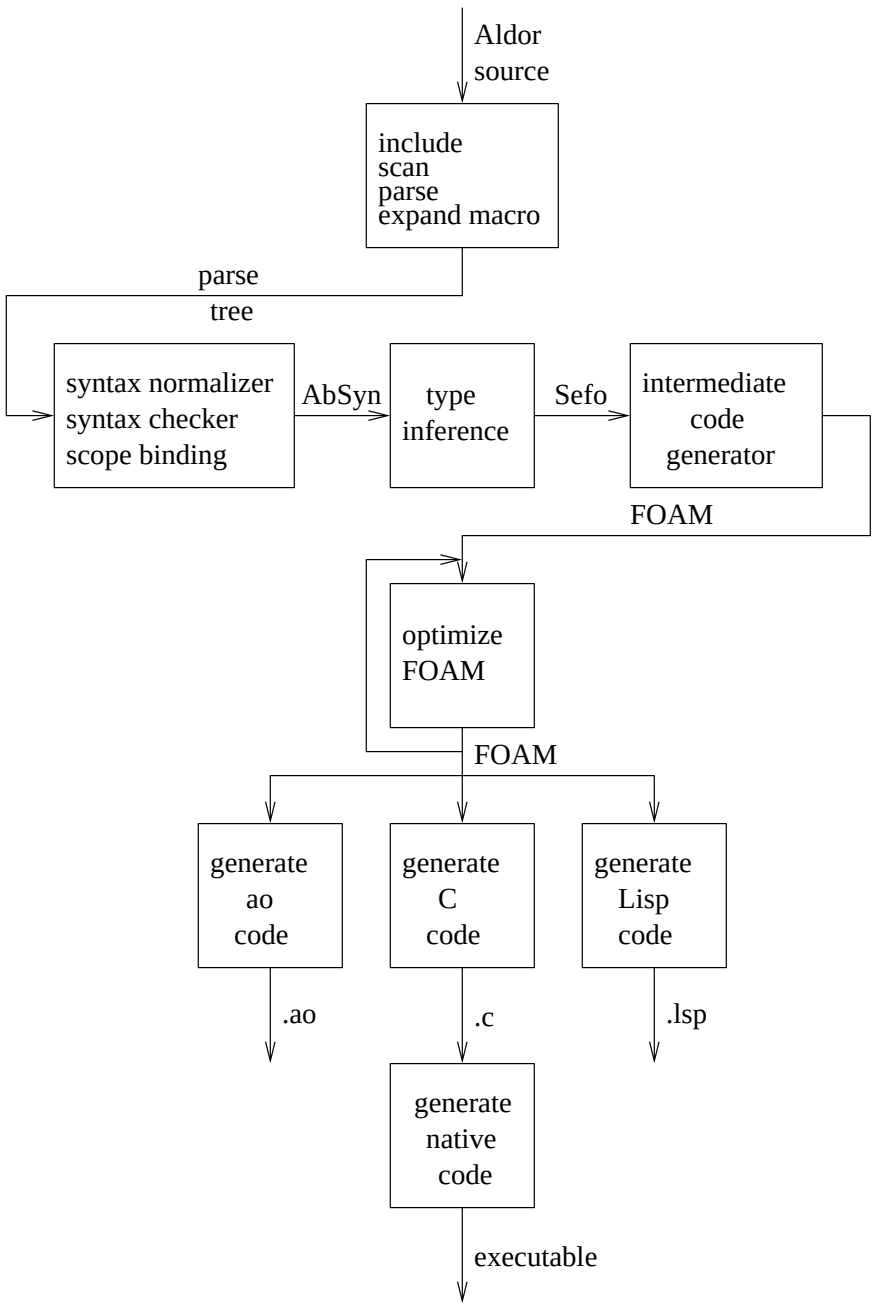

Figure 2.3: Compilation stages for the Aldor compiler. Each compilation stage is presented in boxes and the result of each stage is shown on the arrow connecting the stages. Parse tree is a tree representation of the source code. AbSyn is the abstract syntax tree, a normalized parse tree. Sefo is the semantic form where all the nodes in the abstract syntax tree has a type annotation. First Order Abstract Machine (FOAM) is the intermediate language used by the Aldor compiler.

The next phase is optimization, where FOAM-to-FOAM transformations are performed to produce optimized code that is platform independent. The optimizations done at this stage are architecture independent. Depending on the level of optimization, several passes are performed.

The next stage is code generation. The Aldor compiler is able to generate Aldor portable intermediate code in the form of `ao` files. The `ao` files can be interpreted by the Aldor compiler, since the Aldor compiler also includes an interpreter. It can also produce `C` code or `Lisp`. A `C` compiler can be invoked to produce native executables from the `C` source files produced by the Aldor compiler.

The compiler contains several key data structures. Their internal names are presented in the following:

- `AbSyn` (abstract syntax tree) is the tree constructed by the semantic analysis
- `Sefo` (semantic form) is an abstract syntax tree annotated with type and meaning
- `Syme` (symbol meaning) contains information about the symbols used in the Aldor program. It is the most used data structure throughout the compiler.
- `TForm` (type form) represents the type information constructed by the type inference. The type inference phase executes passes over the syntax tree: bottom-up and top-down. In the bottom-up phase the possible types of each node are propagated from leaves to the root, and in the top-down phase, the unique type restriction is enforced.
- `Foam` is intermediate code representation

The abstract syntax tree resembles very closely the source code. An abstract syntax tree containing annotations for usage information and inferred unique types

```
if (a<b) then
        min := a
else
        min := b
print << min;
```

Figure 2.4: Small Aldor code sample which computes the minimum.

```
0: (If (BCall SIntLT (Loc 0) (Loc 1) 2)
1: (Set (Loc 2)(Loc1 ))
   (Goto 3)
2: (Set (Loc 2) (Loc 0))
3: (CCall Word (Lex 0 0) (Loc 2))
```

Figure 2.5: The corresponding FOAM code for the code sample.

is ready to be translated into the intermediate code. The leaves contain symbol meanings and some symbols are linked to their position in the source file.

The code generation searches for function definitions, "deep" identifiers usage (i.e. identifiers that are not defined inside the local function) and then creates the FOAM code for the top level definitions and top level references lexical variables (i.e. variables that are used in a deeper lexical level than the their definition) and global variables. Function values are represented by closures (i.e. binding of the function code together with the execution environment).

Optimization tools used by the Aldor compiler include flow graphs and data flow analysis. The flow graphs are directed graphs where the nodes are basic blocks and the edges represent the links between the basic blocks. Basic block are sequences of FOAM instructions that have exactly one entry point and one exit point. This means that there are no branches inside the basic block. All code branches are represented by the edges in the flow graph. An example of an Aldor code that computes the minimum value between two variables can be seen in Figure 2.4. The FOAM code constructed by the Aldor compiler for the code presented in Figure 2.4 is presented in Figure 2.5. The labels in Figure 2.5 correspond to the beginning of each basic block. The flow graph resulted from the FOAM code presented in Figure 2.5 can be seen in Figure 2.6. This example is presented in [26].

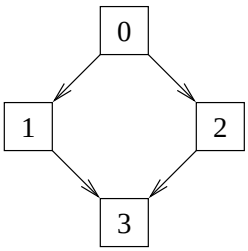

Figure 2.6: A flow graph example.

Based on the information gathered, several optimizations can be implemented. The Aldor compiler provides the following optimizations: copy propagation, constant folding, hash folding, peep-hole, common subexpression elimination, procedure integration, control flow optimization, dead code elimination, environment merging.

Copy propagation replaces values of several definitions with the same value by a single value, for example: `l1:=l0; l2:=l1; foo(l2)` is replaced with `foo(l0)`. After this, if the values of `l2` is never used, the assignment and the variable can be eliminated.

Constant folding computes the values of simple operations applied to constant values. For example, `a := 3*5` will be computed as `a:=15`. Hash folding is similar in scope, but knows how to deal with run-time function calls that extract the hash of a domain. Hash folding eliminates calls to `domainGetHash!`, which is a runtime function call to retrieve the hash of the domain.

Peep hole optimization applies simple code transformations that are limited in scope to only a small piece of code. For example, applying known properties of boolean algebra.

Common sub-expression elimination tries to remove redundant operations. For example, `l0:=t1*t1+t1*t1` is replaced by `l1:=t1*t1;l0:=l1+l1`.

Procedure integration is the most important aspect of the Aldor compiler. One of the main issues with optimizations is that procedure calls limit the scope of the optimization. If there is no interprocedural analysis, the optimization information constructed in a procedure cannot be passed to the called procedures. Procedure integration can alleviate this problem by locally unfolding the code of the callee in the caller. In Aldor, functions are represented by function closures which have a higher execution time cost since they are runtime objects. Inlining makes the calls faster.

Control flow optimization reorganizes the basic blocks to eliminate unnecessary jumps. This is important after procedure integration because the code of the callee is appended at the end of the caller which might produce an unnecessary jump.

Dead code elimination removes the parts of the program that are never called or the variables that are never used.

Environment merging replaces heap-allocated object by local variables so that heap allocation, which is an expensive operation, is never performed. This is done for the variables that do not escape the local function.

All these optimizations make the Aldor compiler a powerful one. Therefore, carefully written Aldor code is comparable in performance with the equivalent program written in the `C` programming language.

2.4 The Aldor Run-Time Domain Representation

In Aldor, domains and categories are run-time objects and, as a consequence, they can be created only when they are actually needed. To support this kind of behavior a run-time system is provided. The implementation of the run-time system can be found in the implementation file `runtime.as`. The run-time system offers other run-

time services as well, like debugging hooks, but its most important part is dealing with domains and categories.

The implementation details of the Aldor domains is important for the optimization presented here. Since there is no documentation available that describes these details, this section will present a short description of the run-time system support for the Aldor domains.

Aldor domains are designed to operate in an environment of mixed run-time systems. To achieve this, there are three domains that represent the *run-time Aldor domain*. These three domains are: `Domain`, `DomainRep` and `DispatchVector`.

The role of `Domain` is to offer a top level domain representation, independent of the run-time, and provide information about the hash codes of the domains and their exported functions. Internally, it uses `DomainRep` as the native Aldor run-time domain and `DispatchVector` for the exported functions. The operations implemented by the `Domain` domain can be seen in the Listing 2.17.

As stated before, `DomainRep` is the native domain representation, which deals with particularities of the Aldor run-time domains. One such particularity is that Aldor domains are implemented in a lazy fashion (i.e. they are created only they are needed.) From the study of the `DomainRep`, it can be seen that Aldor domains are created in two stages:

1. Construct minimal information to be able to retrieve the domain instantiation by:

    - calling the function `domainAddHash!` to fill in the domain hash (an integer value number to uniquely identify the domain)
    - calling the function `domainAddNamFn!` to fill the name of the domain

Listing 2.17: Domain interface

```
+++ Domain is the top-level domain representation, designed to operate in
+++ an environment of mixed runtime systems.  The domain consists of
+++ a pointer to the domain's native representation, and a vector of
+++ functions for accessing it.  Currently only "get" and "hash" functions
+++ are required.
Domain: Conditional with {
    new:            DomainRep -> %;
        ++ new(dr) creates a new domain by wrapping
        ++ a dispatch vector around a DomainRep.
    newExtend:        DomainRep -> %;
        ++ extend(dr) creates a new domain by wrapping
        ++ the dispatch vector for extensions around a DomainRep.
    getExport!:        (%, Hash, Hash) -> Value;
        ++ getExport!(dom, name, type) gets an export from a domain,
        ++ given the hash codes for its name and type.  Takes a hard
        ++ error on failure.
    getExportInner!:    (%, %, Hash, Hash, Box, Bit) -> Box;
        ++ getExportInner!(dom, name, type, box, skipDefaults)
        ++ Fetch an export from the given domain, putting the result
        ++ in the box.  It won't look in category default packages if
        ++ if skipDefaults is true.  Returns nullBox on failure.
    getHash!:        % -> Hash;
        ++ getHash!(dom) returns the hash code for a domain.
    testExport!:        (%, Hash, Hash) -> Bit;
        ++ testExport!(dom, name, type) tests for an
        ++ export with the given name and type in the domain
    getName:        %->DomainName;
        ++ getName(dom) returns the name of a domain
    inheritTo:         (%, Domain)->Domain;
        ++ returns an object suitable for being a parent of dom2
        ++ This function is so that A# can have a single function
        ++ for both computing the hashcodes of types and initialising
        ++ a domain.  Really ought to be expunged.
    makeDummy:     () -> %;
        ++ specialized domain creators
    fill!:        (%, %) -> ();
    reFill!:    (%, DispatchVector, DomainRep) -> ();
    domainRep:    % -> DomainRep;
}
```

- creating a function closure `addLevel1` to fully construct the domain, with all its exports

2. Completely construct the domain by calling the function `addLevel1` which fills in all the domain data

This implementation allows Aldor domains to delay instantiation until the function from the domain is actually called. The interface of the `DomainRep` domain can be seen in Listing 2.18.

The last part of the run-time domain implementation is the `DispatchVector` which deals with run-time independent representation of the domain's exported functions. This is an implementation independent way to extract domain information such as: name, hash or exported functions. The interface of the `DomainRep` domain can be seen in Listing 2.19.

Categories use a similar mechanism as the as the one described for domains. The domains that deal with Aldor run-time categories are: `CatObj`, `CatRep` and `CatDispatchVector`, described below. Their interfaces are given in Listings 2.20, 2.21 and 2.22 respectively.

`CatObj` has a similar functionality to `Domain`. The actual interface to the `CatObj` is presented in Listing 2.20.

`CatRep` is the run-time representation of Aldor categories with a functionality similar to `DomainRep`. The interface to this domain can be seen in Listing 2.21.

The domain that does the dispatching for categories is `CatDispatchVector`. It is similar in function to the domain `DispatchVector`. The interface for `CatDispatchVector` can be seen in Listing 2.22.

To eliminate the duplication of duplicate domains, a cache containing the already created domains is used. The way the cache is used can be seen in Listings 2.36 and

Listing 2.18: DomainRep interface.

```
+++ DomainRep defines the run-time representation of axiomxl domains.
+++ Initially domains only hold a function which, when called, fills in
+++ the hash code for the domain, and sets another function. When this
+++ function is called, the parent and export fields are set, and any
+++ code from the "add" body is run. Domains cache the last few lookups.
DomainRep: Conditional with {
    new:            DomainFun % -> %;
        ++ new(fun) creates a new domain.
    prepare!:       % -> ();
        ++ prepare!(dom) forces a domain to fully instantiate.
    addExports!:    (%, Array Hash, Array Hash, Array Value) -> ();
        ++ addExports!(dom, names, types, exports)
        ++ sets the exports fields of a domain.
    addDefaults!:   (%, CatObj, Domain) -> ();
        ++ addDefaults!(dom, defaults, domain) sets the default package
        ++ for a domain. Additional arg is the wrapped domain.
    addParents!:    (%, Array Domain, Domain) -> ();
        ++ addParents!(dom, parents) sets the parent field of a domain.
    addHash!:       (%, Hash) -> ();
        ++ addHash!(dom, hash) sets the hash field of a domain.
    addNameFn!:     (%, ()->DomainName) -> ();
        ++ addName!(dom, name) sets the naming fn  of a domain.
    get:            (%, Domain, Hash, Hash, Box, Bit) -> Box;
        ++ get(dom, name, type, box, skipDefaults) Fetch an export from the
        ++ given domain, putting the result in the box.
    hash:           % -> Hash;
        ++ hash(dom) returns the hash code for a domain.
    getExtend:      (%, Domain, Hash, Hash, Box, Bit) -> Box;
        ++ get(dom, name, type, box, skipDefaults)
        ++ Fetch an export from an extended domain.
    hashExtend:     % -> Hash;
        ++ hashExtend(dom) returns the hash code for extended domains.
    extendFillObj!: (%, Array Domain) -> ();
        ++ fills extend info
    axiomxlDispatchVector: () -> DispatchVector;
        ++ axiomxlDispatchVector() creates the dispatch vector for domains.
    extendDispatchVector:  () -> DispatchVector;
        ++ extendDispatchVector() creates the dispatch vector for extended
        ++ domains.
    dummyDispatchVector:   () -> DispatchVector;
}
```

Listing 2.19: `DispatchVector` interface.

```
+++ Structure containing a domain's protocol for getting exports and
+++ producing hash codes.  This is in a separate structure to accomodate
+++ mixed runtime environments.
DispatchVector: Conditional with {
    new:        (DomNamer, DomGetter, DomHasher, DomInheritTo) -> %;
        ++ new(get, hash) constructs a dispatch vector.
    getter:     % -> DomGetter;
        ++ getter(dv) returns the getter function.
    hasher:     % -> DomHasher;
        ++ hasher(dv) returns the hash code function.
    namer:      %->  DomNamer;
        ++ namer(dv) returns the function giving the name of a domain
    tag:        % -> Int;
    reserved:   % -> Reserved;
    inheriter:  % -> DomInheritTo;
}
```

Listing 2.20: CatObj

```
+++ CatObj is the top-level category representation.
CatObj: Conditional with {
    new:            CatRep -> %;
        ++ new(cr) creates a new category by wrapping
        ++ a dispatch vector around a CatRep.
    getDefault!:    (%, Domain, Hash, Hash, Box) -> Box;
        ++ getDefault!(cat, pcent, name, type, box)
        ++ Find a default from the given category,
        ++ putting the result in box.  Returns nullBox on failure.
    getParent:      (%, Int) -> %;
        ++ getParent(cat, i) finds the i-th parent of cat.
    parentCount:     % -> Int;
        ++ returns the # of parents of the category
    build:          (%, Domain) -> %;
    name:           % -> DomainName;
    hash:           % -> Hash;
    makeDummy:      () -> %;
    fill!:          (%, %) -> ();
    reFill!:        (%, CatDispatchVector, CatRep) -> ();
}
```

Listing 2.21: CatRep

```
+++ CatRep defines the run-time representation of axiomxl categories.
CatRep: Conditional with {
    new:        (CatRepInit %, ()->Hash, ()->DomainName) -> %;
        ++ new(fun) creates a new category.
    prepare!:    % -> ();
        ++ prepare!(cat) forces a category to fully instantiate.
    addExports!:    (%, Array Hash, Array Hash, Array Value) -> ();
        ++ addExports!(cat, names, types, exports)
        ++ sets the exports fields of a category.
    addHashFn!:    (%, ()->Hash) -> ();
    addParents!:    (%, Array CatObj) -> ();
        ++ addParents!(cat, pars) set the parent field of a category.
    addNameFn!:     (%, ()->DomainName) -> ();
        ++ addName! sets the name of an category.
    axiomxlCatDispatchVector: () -> CatDispatchVector;
        ++ axiomxlCatDispatchVector() creates the dispatch vector
        ++ for axiomxl categories.
    dummyDispatchVector: () -> CatDispatchVector;
        ++ creates the dispatch vector for unfilled categories
}
```

Listing 2.22: CatDispatchVector

```
+++ Structure containing a category's protocol for getting parents/hash
+++ codes.
CatDispatchVector: Conditional with {
    new:        (CatNamer, CatBuilder, CatGetter, CatHasher,
                 CatParentCounter, CatParentGetter) -> %;
        ++ new(build, get, hash, parentCount, parentGet)
        ++ constructs a category dispatch vector.
    builder:    % -> CatBuilder;
        ++ builder(cat) returns the building function of the category.
    getter:        % -> CatGetter;
        ++ getter(cat) returns the getter function of the category.
    hasher:        % -> CatHasher;
        ++ hasher(cat) returns the hasher function of the category.
    namer:        % -> CatNamer;
        ++ returns the naming function of the category.
    parentCounter:    % -> CatParentCounter;
        ++ parentCounter(cat) returns the #parents function.
    parentGetter:    % -> CatParentGetter;
        ++ parentGetter returns the getParent function.
}
```

2.37. The cache increases the performance significantly and it uses runtime functions `rtCacheExplicitMake`, `rtCacheCheck` and `rtCacheAdd`.

An extract of the most important functions that operate on domains is presented in Listings 2.23 and 2.24.

The implementation of the runtime systems can be found in `runtime.as` which is part of the libfoam library, the core library of the Aldor compiler.

2.5 FOAM Intermediate Code Representation

First Order Abstract Machine (FOAM) is an intermediate programming language used by the Aldor compiler. Any Aldor program is compiled into FOAM [73]. Aldor code is translated in FOAM and then all optimizations are produced by FOAM-to-FOAM transformations.

The main designed goals of FOAM are:

- well-defined semantics, independent of platform
- efficient mapping to Common Lisp and ANSI C
- easy manipulation

Each Aldor source file is compiled to a FOAM unit. A program may be formed by several FOAM units. Each program must contain at least one unit, and the program starts by calling the *first* program in the FOAM unit corresponding to the current file being compiled. The first program is called with a empty (NULL) environment as the initial environment.

Each FOAM unit is formed by a list of declarations, and a list of definitions. The declaration part is formed by a list of formats. Each format is a list of lexical

Listing 2.23: Run-time functions dealing with domain manipulation.

```
1  -- Functions for creating and enriching axiomxl domains.
2  domainMake:    DomainFun(DomainRep) -> Domain;
3    ++ domainMake(fun) creates a new lazy domain object.
4  domainMakeDispatch:  DomainRep -> Domain;
5    ++ domainMakeDispatch(dr) wraps a dispatch vector
6    ++ around a DomainRep.
7  domainAddExports!: (DomainRep, Array Hash, Array Hash, Array Value)->();
8    ++ domainAddExports!(dom, names, types, exports)
9    ++ Set the exports of a domain.
10 domainAddDefaults!: (DomainRep, CatObj, Domain) -> ();
11   ++ domainAddDefaults!(dom, defaults, dom)
12   ++ Sets the default package for a domain.
13 domainAddParents!: (DomainRep, Array Domain, Domain) -> ();
14   ++ defaultsAddExports!(dom, parents)
15   ++ Set the parents of a default package.
16 domainAddHash!: (DomainRep, Hash) -> ();
17   ++ domainAddHash!(dom, hash) sets the hash code of a domain.
18 domainAddNameFn!: (DomainRep, ()->DomainName)->();
19   ++ sets the domains naming function
20 domainGetExport!: (Domain, Hash, Hash) -> Value;
21   ++ domainGetExport!(dom, name, type)
22   ++ Gets an export from a domain, given the hash codes for
23   ++ its name and type.  Takes a hard error on failure.
24 domainTestExport!: (Domain, Hash, Hash) -> Bit;
25   ++ domainTestExport!(dom, name, type)
26   ++ returns true if the given export exists in dom
27 domainHash!: Domain -> Hash;
28   ++ domainHash!(dom) returns the hash code from a domain.
29 domainName: Domain -> DomainName;
30   ++ domainName returns the name of a domain
31 domainMakeDummy: () -> Domain;
32 domainFill!: (Domain, Domain) -> ();
```

Listing 2.24: Run-time functions dealing with domain manipulation (Continued).

```
1  -- Functions for creating and enriching axiomxl categories.
2  categoryAddParents!:  (CatRep, Array CatObj, CatObj) -> ();
3    ++ categoryAddExports!(dom, parents, self)
4    ++ Set the parents of a default package.
5    ++ additional arg is for uniformity
6  categoryAddNameFn!: (CatRep, ()->DomainName) -> ();
7    ++ Sets the name of a category.
8  categoryAddExports!: (CatRep, Array Hash, Array Hash, Array Value) -> ();
9    ++ categoryAddExports!(dom, names, types, exports)
10   ++ Set the exports of a category.
11 categoryMake: (CatRepInit(CatRep),()->Hash,()->DomainName)->CatObj;
12   ++ Constructing new cats
13 categoryBuild:    (CatObj, Domain) -> CatObj;
14 categoryName:    CatObj -> DomainName;
15   ++ Returns the name of a category
16 categoryMakeDummy:    () -> CatObj;
17 categoryFill!:    (CatObj, CatObj) -> ();
18
19 -- Utility functions called from code generation.
20 noOperation:    () -> ();
21   ++ Do nothing --- used to clobber initialisation fns.
22 extendMake:    DomainFun(DomainRep) -> Domain;
23   ++ extendMake(fun) creates a new lazy extend domain object;
24 extendFill!:  (DomainRep, Array Domain) -> ();
25   ++ adds the extendee, extender pair to an extension domain
26 lazyGetExport!:    (Domain, Hash, Hash) -> LazyImport;
27   ++ creates a lazy function to retrieve the export
28 lazyForceImport:   LazyImport->Value;
29   ++ forces a get on the lazy value
30 rtConstSIntFn: SingleInteger->(()->SingleInteger);
31   ++ Save on creating functions.
32 rtAddStrings: (Array Hash, Array String) -> ();
33   ++ Adds more strings to the list of known exports
34 domainPrepare!:    Domain -> ();
35   ++ initializes a domain.
```

Listing 2.25: A simple FOAM example.

```
(Unit
  (DFmt
    (DDecl
      LocalEnv
      (GDecl Clos "phd1" -1 4 0 Init)
      (GDecl Clos "noOperation" -1 4 1 Foam)
      (GDecl Clos "phd1_Dom2_635009245" -1 4 0
        Foam)
      ...
    (DDecl
      Consts
      (Decl Prog "phd1" -1 4)
      (Decl Prog "Dom1Cat" -1 4)
      ...
    (DDecl LocalEnv (Decl Clos "m" -1 4))
    (DDecl LocalEnv (Decl Word "p" -1 4))
    ...
    (DDecl
      LocalEnv
      (Decl Word "TextWriter" -1 4)
      ...
```

Listing 2.26: A simple FOAM example (Continued).

```
(DDef
  (Def
    (Const 0 phd1)
    (Prog
      ...
      NOp
      ...
      (DDecl Params)
      (DDecl
        Locals
        (Decl Clos "" -1 4)
        ...
      (DFluid)
      (DEnv 14)
      (Seq
        (CCall NOp (Glo 5 runtime))
        (Set
          (Glo 0 phd1)
          (Glo 1 noOperation))
        ...
  (Def
    (Glo 0 phd1)
    (Clos (Env 0) (Const 0 phd1)))
  (Def
    (Glo 2 phd1_Dom2_635009245)
    (Cast Clos (Nil)))
  ...
```

elements that exist in a scope that forms an environment. The declaration contains formats for globals, constants, fluid scope variables, local environments, non-local environments. The definition part consists of program definitions and initializations of locally defined global variables.

An example of FOAM code is presented in Listings 2.25 and 2.26. Due to the fact that the FOAM code is usually long, in the presented example there are only some samples from the various categories of FOAM constants. Figures 2.25 and 2.26 show a unit with samples from declarations and definitions. One can see the LISP-like syntax and the tree structure. The FOAM operation is the first operation after opening the bracket, followed by a number of arguments indented one level more than the current operation. Also the first lexical presented is `phd1` which is also the name of the file and the name of the program that contains the initialization code for the current file. The Aldor source code used to generate the FOAM program is presented in Listing 2.27.

An interesting aspect of the FOAM capabilities is the support for function closures. Closures are objects that bind together the code of the function with its environment. The advantage of the closures is that they can be executed anywhere in the program, even if the environment required by the function is not accessible at the point of execution.

In FOAM, the type for the closure is `Clos` and the syntax of the closure constructor is "(`Clos` *env prog*)". The execution environment can be constructed relative to the lexical environment of the current function (by using "(`Env` *level*)") or extracted from another closure (by using "(`CEnv` *clos*)"). The program is either a reference to a program definition "(`Const` *idx*)" or extracted from a different closure by using "(`CProg` *clos*)". In the example presented ifn Figures 2.25 and 2.26, the global variable with index `0` is a closure with the current environment and the program index `0`.

The FOAM instruction to delimit programs is `Prog`. It must be supplied a list of formal parameters described by a `DDecl` instruction, a list of fluid variables (variables that have dynamic binding) described by a `DFluid`, a list of referenced environments described by `DEnv` and the sequence of instructions described by `Seq` instructions.

For function calls there are four instructions:

1. (`BCall` *opId* e_0 ... e_{n-1}) - builtin call which is used for FOAM builtin operations. `BCall` is equivalent to (`OCall` *type* (`BVal` *opId*) (`Env` -1) e_0 ... e_{n-1})

2. (`CCall` *type clos* e_0 ... e_{n-1}) - closed call which is used for calling the closure objects. `CCall` is equivalent to (`OCall` *type* (`CProg` *clos*) (`CEnv` clos) e_0 ... e_{n-1})

3. `OCall` *type fun env* e_0 ... e_{n-1} - open calls are calls that explicitly reference the program and the environment. The `OCall` is equivalent to (`PCall` `FOAM_Proto_Foam` *type fun env* e_0 ... e_{n-1})

4. `PCall` *proto type fun env* e_0 ... e_{n-1} - protocol calls are the most general form of call. The calling protocols defined by FOAM are: `FOAM_Proto_Foam`, `FOAM_Proto_Other`, `FOAM_Proto_Init`, `FOAM_Proto_C`, `FOAM_Proto_Lisp`, `FOAM_Proto_Fortran`.

The Aldor programming language uses lexical scoping. Instructions that create new lexical scopes are: `where`, `+->`, `with`, `add`, `for` *var* `in`, and applications e.g. `Record(i: Integer == 12)` [74]. In FOAM, each lexical scope is associated to a `DDecl` which contains a list of symbols declared at that lexical level and their types. For each FOAM program, the environments referenced are declared in `DEnv`. There are four different types of lexically scoped references in FOAM:

- (`Glo` *index*) - global variables

Listing 2.27: Simple example of Aldor domains

```
#include "axllib.as"
import from SingleInteger;
SI ==> SingleInteger;
Dom1Cat: Category == with { m: SI -> SI; };
Dom1: Dom1Cat == add {
        m (g: SI) : SI == { import from SI; g := g + 1; }
}
Dom2(p: Dom1Cat): Dom1Cat == add {
        m (x: SI) : SI == {
                import from SI;
                for i in 1..2 repeat { x := m(x)$p + 1; } --$
                x;
        }
}
import from Dom2 Dom1;
print << m(0) << newline;
```

- (`Loc` *index*) - local variables
- (`Lex` *level index*) - lexicals. Lexical variables can be nested. The level specifies the number of levels up the lexical levels stack.
- (`Par` *index*) - function parameters

A complete description of the grammar for the FOAM intermediate language can be found in the FOAM language specification [73]. Since presenting the FOAM structure is not the goal of this thesis, the needed information will be provided along when presenting the optimizations made.

2.6 Domain representation in FOAM

As presented in Aldor User Guide [74], domains are environments providing collections of exported constants. Exported constants are types, functions or other values.

According to this definition, domains are, in a way, similar to classes from object-oriented programming languages. This representation is reviewed in detail here, as this is important to understand the optimizations presented later.

In order to understand how domains are represented in FOAM, we start by presenting the source code of a simple example. This example creates a category `Dom1Cat` as a type which exports a function `m`. The function `m` takes an integer value as an argument and returns another integer value as a result. Next, two domains are defined, namely `Dom1` and `Dom2`. Both of them implement the `Dom1Cat` category. `Dom1` is a regular domain, and `Dom2` is a parametric one. The parameter of `Dom2` is a domain of type `Dom1Cat`. This declaration allows the construction `Dom2(Dom1)`. This is the simplest example that creates domains in Aldor. An example which only creates `Dom1` would not be enough to illustrate the construction method of parametric domains, since their construction is slightly different from the construction of regular domains. The complete source code is presented in Listing 2.27.

The example presented in Listing 2.27 will be used in the following to describe how to create the regular and parametric domains, access the exports of a domain, and how to access functions from the library.

2.6.1 Creating a domain

The FOAM function which corresponds to the file level scope is the definition with index 0. In this function, one can see the call presented in Listing 2.28 which defines the global with index 3 to be a new domain created with the help of the run-time system by calling `domainMake`. As explained in Section 2.4, the Aldor domains are lazy and they are constructed in two stages. The first stage is initiated by calling the function `domainMake` with a closure as argument.

Listing 2.28: Domain construction function.

```
1 (Def
2   (Glo 3 phd1_Dom1_774680846)
3   (CCall
4     Word
5     (Glo 23 domainMake)
6     (Clos (Env 0) (Const 2 addLevel0))))
```

The next step in understanding how a domain is constructed is to follow the definition `(Const 2 addLevel0)` given in Listing 2.29. The role of `addLevel0` functions is to set the domain name function and hash. The domain object is given as a parameter to the closure. The returned value is another closure that represents the second stage in domain creation.

The actual construction of the domain is performed by `(Const 3 addLevel1)`. The `addLevel1` functions will add the exports of the domain. Listings 2.30 and 2.31 present the FOAM code for constructing the domain `Dom1`.

The first thing to note in line 22 is the declaration of the lexical level associated with this domain which in this case is the format number 7. Each domain has an associated lexical level. The declaration of the lexical level for `Dom1` is presented in Listing 2.32. The closure `m` (on line 34) is the function `m` implemented in `Dom1`. The lexical named `%` is a reference to the type that contains the definition of the exported symbol. A similar concept exists in object-oriented programming languages when the keyword *this* is used. In object-oriented languages, *this* represents a reference to the current object. In Aldor, the type of *this* is the equivalent of `%`. The `SInt` type lexical is the hash code of the domain. Each domain has its own hash code. The hash code is the value used to identify the domain at run-time.

In Listings 2.30 and 2.31, lines 25–27 construct three arrays to hold the information

Listing 2.29: The first stage of domain construction

```
1  (Def
2    (Const 2 addLevel0)
3    (Prog
4      (DDecl Params (Decl Word "domain" -1 4))
5      (DDecl Locals)
6      (DFluid)
7      (DEnv 0 14)
8      (Seq
9        (CCall
10         Word
11         (Glo 15 domainAddNameFn!)
12         (Par 0 domain)
13         (CCall Word (Glo 13 rtConstNameFn) (Arr Char 68 111 109 49)))
14       (CCall
15         Word
16         (Glo 16 domainAddHash!)
17         (Par 0 domain)
18         (SInt 285110261))
19       (Return (Clos (Env 0) (Const 3 addLevel1))))))
```

for retrieving the exports of the domain. In the instruction corresponding to the line 29, a `Domain` type object is created from the `DomainRep`. The next step is to populate the lexical environment of the domain as seen in the lines 32–34. The three arrays and their sizes are encapsulated in three records in lines 15–25. This encapsulation is done because the run-time system uses the data type `Array` which is represented by such a record. The line 26 uses the run-time system to add the functions to the domain. The next instruction adds to this domain the defaults that might have been defined in the category. Finally, all the exports of the domain are added to the three arrays: the name hash code, the type hash code and the closure.

The hash code of the type is computed by combining the hashes of the domains that form the signature of the functions. In this case the function `m` has the signature `SingleInteger -> SingleInteger`. The hash code of the type of `m` is computed

Listing 2.30: The second stage of domain construction.

```
(Def
  (Const 3 addLevel1)
  (Prog
    (DDecl
      Params
      (Decl Word "domain" -1 4)
      (Decl Word "hashcode" -1 4))
    (DDecl
      Locals
      (Decl Word "%" -1 4)
      (Decl SInt "" -1 4)
      (Decl Arr "" -1 8)
      (Decl Arr "" -1 5)
      (Decl Arr "" -1 5)
      (Decl Word "" -1 4)
      (Decl Rec "" -1 5)
      (Decl Rec "" -1 5)
      (Decl Rec "" -1 5)
      (Decl Word "" -1 4)
      (Decl SInt "" -1 4))
    (DFluid)
    (DEnv 7 4 14)
    (Seq
      (Set (Loc 1) (SInt 1))
      (Set (Loc 2) (ANew SInt (Loc 1)))
      (Set (Loc 3) (ANew SInt (Loc 1)))
      (Set (Loc 4) (ANew Word (Loc 1)))
      (Set (Loc 5) (ANew Bool (SInt 3)))
      (Set
        (Loc 9)
        (CCall Word (Glo 18 domainMakeDispatch) (Par 0 domain)))
      (Set (Lex 0 1 %) (Loc 9))
      (Def (Lex 0 3) (Par 1 hashcode))
      (Def (Lex 0 0 m) (Clos (Env 0) (Const 4 m)))
      (Def
        (Loc 10)
        (BCall
          SIntPlusMod
          (CCall SInt (Glo 7 domainHash!) (Lex 2 3 SingleInteger))
          (BCall
            SIntShiftUp
```

Listing 2.31: The second stage of domain construction (Continued).

```
            (BCall
              SIntAnd
              (BCall
                SIntPlusMod
                (CCall SInt (Glo 7 domainHash!) (Lex 2 3 SingleInteger))
                (BCall
                  SIntShiftUp
                  (BCall SIntAnd (SInt 51489085) (SInt 16777215))
                  (SInt 6))
                (SInt 1073741789))
              (SInt 16777215))
            (SInt 6))
          (SInt 1073741789)))
      (Set (Loc 6) (RNew 5))
      (Set (RElt 5 (Loc 6) 0) (Loc 1))
      (Set (RElt 5 (Loc 6) 1) (Loc 1))
      (Set (RElt 5 (Loc 6) 2) (Cast Word (Loc 2)))
      (Set (Loc 7) (RNew 5))
      (Set (RElt 5 (Loc 7) 0) (Loc 1))
      (Set (RElt 5 (Loc 7) 1) (Loc 1))
      (Set (RElt 5 (Loc 7) 2) (Cast Word (Loc 3)))
      (Set (Loc 8) (RNew 5))
      (Set (RElt 5 (Loc 8) 0) (Loc 1))
      (Set (RElt 5 (Loc 8) 1) (Loc 1))
      (Set (RElt 5 (Loc 8) 2) (Cast Word (Loc 4)))
      (CCall NOp
        (Glo 17 domainAddExports!)
        (Par 0 domain)
        (Loc 6)
        (Loc 7)
        (Loc 8))
      (CCall
        Word
        (Glo 19 domainAddDefaults!)
        (Par 0 domain)
        (Glo 4 phd1_Dom1Cat_735503011)
        (Loc 9))
      (Set (AElt SInt (SInt 0) (Loc 2)) (SInt 200150))
      (Set (AElt SInt (SInt 0) (Loc 3)) (Loc 10))
      (Set (AElt Word (SInt 0) (Loc 4)) (Cast Word (Lex 0 0 m)))
      (Return (Par 0 domain)))))
```

Listing 2.32: The declaration of the lexical level associated to `Dom1`.

```
(DDecl
  LocalEnv
  (Decl Clos "m" -1 4)
  (Decl Word "%" -1 4)
  (Decl Word "%%" -1 4)
  (Decl SInt "" -1 4))
```

Listing 2.33: The algorithm to combine two hashes.

```
CombineHash(h1, h0):
  (BCall SIntPlusMod
    h1
    (BCall SIntShiftUp (BCall SIntAnd h0 16777215) 6)
    1073741789)
```

starting at line 35 in Listing 2.30 and continued in Listing 2.31. The constants are the same for all operations, and the formula to combine two hashes is given in Listing 2.33.

The exported functions are defined next in the list of function definitions.

Every domain has a similar structure. They all contain the `addLevel0` and `addLevel1` functions and the list of the exported functions. Some domains will include some helper functions like function for producing the domain name in case the domain the more than one argument.

Any function imported from other domains using the `import from` statement and used by the program will be included in the lexical environment of the importing domain.

Listing 2.34: The closure of the parametric domain producing function.

```
1   (Def
2     (Glo 2 phd1_Dom2_635009245)
3     (Clos (Env 0)
4     (Const 5 Dom2)))
```

Listing 2.35: Instantiating `Dom2` by calling `Dom2(Dom1)`.

```
1 (Def
2   (Lex 0 25 dom)
3   (CCall
4     Word
5     (Glo 2 phd1_Dom2_635009245)
6     (Glo 3 phd1_Dom1_774680846)))
```

2.6.2 Creating a parametric domain

The Aldor source code presented in Listing 2.27 provides an example of a parametric domain. A parametric domain is actually a function that produces a non-parametric domain as a result when called. In order to learn how such domains are created, one must look to the FOAM code associated with the given file (i.e. the first program definition.) First, a closure of the domain creating function is produced as seen in Listing 2.34. Later on, the closure is used to produce new domains by instantiating its parameters with other domains. For example, the domain `Dom2(Dom1)` is produced by the statement `import from Dom2(Dom1)`. The corresponding FOAM code can be seen in Listing 2.35. The domain-producing function corresponding to `Dom2` from the example can be seen in Listings 2.36 and 2.37.

First thing to note about this function is that it uses a special lexical environment for the parameters (see line 12 in Listing 2.36). In this case, the lexical environment is the environment with format number 12. The declaration can be seen in Listing 2.38. Each field corresponds to the parameter with the same index.

Listing 2.36: The domain producing function `Dom2`.

```
(Def
  (Const 5 Dom2)
  (Prog
    (DDecl Params (Decl Word "p" -1 4))
    (DDecl
      Locals
      (Decl Arr "" -1 8)
      (Decl Rec "" -1 8)
      (Decl Word "" -1 4)
      (Decl Word "" -1 4))
    (DFluid)
    (DEnv 12 14)
    (Seq
      (Set (Lex 0 0 p) (Par 0 p))
      (Set (Loc 0) (ANew Word (SInt 1)))
      (Set (Loc 1) (RNew 8))
      (Set (RElt 8 (Loc 1) 0) (SInt 1))
      (Set (RElt 8 (Loc 1) 1) (Loc 0))
      (If (BCall BoolNot (BCall PtrIsNil (Cast Ptr (Lex 1 21)))) 1)
      (Set
        (Lex 1 21)
        (CCall Word (Glo 24 rtCacheExplicitMake) (SInt 15)))
      (Label 1)
      (Set (AElt Word (SInt 0) (Loc 0)) (Lex 0 0 p))
```

Listing 2.37: The domain producing function `Dom2` (Continued).

```
      (Set
        (Values (Loc 2) (Loc 3))
        (MFmt
          9
          (CCall
            Word
            (Glo 25 rtCacheCheck)
            (Lex 1 21)
            (Cast Word (Loc 1)))))
      (If (Loc 3) 0)
      (Set
        (Loc 2)
        (CCall
          Word
          (Glo 23 domainMake)
          (Clos (Env 0) (Const 6 addLevel0))))
      (Set
        (Loc 2)
        (CCall
          Word
          (Glo 30 rtCacheAdd)
          (Lex 1 21)
          (Cast Word (Loc 1))
          (Loc 2)))
      (Label 0)
      (Return (Loc 2)))))
```

Listing 2.38: Lexical environment for parameter instances.

```
(DDecl LocalEnv (Decl Word "p" -1 4))
```

The Dom2 program checks for the existence of the domain in the domains cache and uses it from there if the domain is already created. Otherwise, domainMake is used to create a new domain, which is placed in the cache.

The lexical environment of the Dom2 is given by format number 11 (see Listing 2.39.) The lexical environment of Dom2 is very similar to the lexical environment of Dom1, but it also includes the functions and domains imported in lines 9 and 10.

Listing 2.39: The lexical environment corresponding to Dom2.

```
1  (DFmt
2    ...
3   (DDecl
4     LocalEnv
5     (Decl Clos "m" -1 4)
6     (Decl Word "%" -1 4)
7     (Decl Word "%%" -1 4)
8     (Decl SInt "" -1 4)
9     (Decl Clos "m" -1 4)
10    (Decl Word "dom" -1 4))
```

The parametric domains are implemented by a function that stores the parameters into a separate lexical environment and then constructs the resulting domain like any other regular domain. The presence of the cache is only for optimization reasons and does not affect in any way the functionality.

2.6.3 How to Use Domain Functions

After constructing the domain, and setting its exports, the exported symbols of the domain can be used. To be able to use an exported lexical from a domain inside another domain, it is necessary to get the export from the former domain. The procedure is presented in Listing 2.40.

To set the lexical variable m, one should make use of the rtDelayedGetExport! function and to provide it with the exporting domain, the hash code of the name

Listing 2.40: Access to domain exports

```
(Def
  (Lex 0 19 m)
  (CCall
    Clos
    (Glo 10 stdGetWordRetWord0)
    (CCall
      Word
      (Glo 9 rtDelayedGetExport!)
      (Lex 0 25 dom)
      (SInt 200150)
      (BCall
        SIntPlusMod
        (CCall
          SInt
          (Glo 7 domainHash!)
          (Lex 0 3 SingleInteger))
        (BCall
          SIntShiftUp
          (BCall
            SIntAnd
            (BCall
              SIntPlusMod
              (CCall
                SInt
                (Glo 7 domainHash!)
                (Lex 0 3 SingleInteger))
              (BCall
                SIntShiftUp
                 (BCall
                   SIntAnd
                   (SInt 51489085)
                   (SInt 16777215))
                 (SInt 6))
               (SInt 1073741789))
             (SInt 16777215))
           (SInt 6))
         (SInt 1073741789)))))
```

Listing 2.41: Use the domain export after it was imported.

```
(CCall
  Word
  (Lex 0 19 m)
  (Cast
    Word
    (CCall
      Word
      (Glo 21 lazyForceImport)
      (Lex 0 12 \0)))))
```

and the hash code of the type. The returned value of `rtDelayedGetExport!` is a closure. Having the closure allows execution of the code in any environment, because the creation environment of the closure is bound with the code. However, due to the lazy nature of Aldor, an extra instruction is required to ensure that the environment of the called function is initialized. The required FOAM instruction is `EEnsure`.

Once all this is set up, the function can be called in the same way as any other closure (see Listing 2.41.)

This code is placed in the initialization part of a file and it is executed once therefore, it should not affect the performance.

2.6.4 Library access from FOAM

The FOAM code is given and compiled in units. Each unit corresponds to the compilation of a file. Using the `include` directive in Aldor causes code from other files to be locally expanded and becomes part of the current compilation unit. In some cases the code is compiled into a library. Every library contains initialization code In order to initialize the libraries, the corresponding units are initialized before the initialization of the current unit. The units that should be initialized are flagged with

Listing 2.42: Initialization of library units.

```
(GDecl Clos "char" -1 4 1 Init)
(GDecl Clos "textwrit" -1 4 1 Init)
(GDecl Clos "segment" -1 4 1 Init)
(GDecl Clos "sinteger" -1 4 1 Init))
```

Listing 2.43: Initialization of library units.

```
(Def
  (Lex 0 3 SingleInteger)
  (CCall Word (Glo 12 rtLazyDomFrInit) (Loc 0) (SInt 0)))
```

the `Init` protocol in the declaration part of the current unit (see Listing 2.42.)

After the initialization, the domains defined in other units can be imported by using the run-time function `rtLazyDomFrInit` (see Listing 2.43).

2.7 Benchmarks

A benchmark is a standard program that run on different platforms to measure the performance. There are two categories of benchmarks with respect to complexity: *microbenchmarks* and *macrobenchmarks*.

Microbenchmarks measure the performance using simple operations and are usually created to tests only the issue that needs measuring. The macrobenchmarks are more elaborate programs that solve more complex problems simulating closer the conditions of a real application.

By using the same program on different, but compatible, platforms allows the comparison between the two platforms. For example, there are benchmarks that measure the floating point performance a CPU (central processing unit), or the input/output

performance of a disk drive.

There are also software benchmarks, where programs are compiled with different compilers and their performance is evaluated, or same queries are run on same databases but on different database management systems. These benchmarks measure the performance of the compilers, or database management systems respectively.

Both current computer architectures and programs generated by compilers are complex, making performance analysis very difficult by consulting the specification details and the resulted program. Therefore, standard tests were developed that can be run on different architectures and have the results compared between them. For compilers, same or similar programs are run through different compilers and the performance of the resulted programs allows comparison of the compilers.

For the hardware benchmarks the standard benchmark is SPEC [19]. For the software benchmarks there is no standard benchmark suite.

Chapter 3

Benchmarking Generic Code

3.1 Introduction

The goal of optimizations is to improve performance of a program. The resulting program is not usually optimal in a mathematical sense, but merely improved. This thesis studies improvements that can be made to compilers for languages such as Aldor with support for generic programming. One important aspect of optimization is to quantify the effects of the optimization by measuring the speedup of the resulting code.

Performance can be measured in two ways. One way is to analyze the generated code and compare it to the original. While this method can show the exact differences produced by the optimization, it cannot quantify the difference in the execution time. Understanding the performance difference requires understanding all the interaction between all the parts of the program and all the interactions between the program and the underlying architecture that should execute the program.

The second way to measure the performance is to write programs and simply run them and measure their execution time. With the second approach it is much easier

to see the results and has the advantage that it takes into account all the variables that are involved in the execution of the program.

In our specific case, the main problem is the performance of generic code. To our knowledge, no macro-benchmark existed to test the performance of generic code. To be able to measure the performance of compilers with respect to generic code, a new benchmark had to be created that used generic code and this resulted in the creation of SciGMark a "generic" version of SciMark 2.0 a popular benchmark for numerical computation.

The results presented here have been published in [23].

3.2 Benchmarks

In computing, benchmarks are tools to evaluate the performance of different systems. With respect to component being measured, there are two types of benchmarks, namely hardware and software.

The hardware benchmarks are programs that measure the performance of different computer architectures aspects. For example, a benchmark might execute only floating-point operations to measure the performance of the CPU with respect to floating-point operations. Other benchmarks might test the performance of the input/output systems or in executing certain applications.

The software benchmarks measure the performance of different software systems. This is the case for database management systems. Same queries are used for databases with identical entries in difference database management systems and the results are compared against each other.

If the same input program can be processed by two different compilers, the resulted programs can be measured for different parameters like execution time or size and

then compared with each other. The comparison will provide a quantification of the performance difference between the compilers used.

Another popular software benchmarking technique is to implement same algorithms in different programming languages and compare the resulted programs to measure the performance difference by using a different programming language.

Programs that are used to measure the performance fall into two categories: programs that only exhibit the characteristics that must be optimized; and programs that solve real problems. The first kind are called micro-benchmarks and the second kind are called macro-benchmarks. The advantage of micro-benchmark is that they measure only the implemented optimization, showing a possible upper limit of the strength of the optimization, while the macro-benchmarks show how useful this optimization can be when linked together with the rest of the optimizations in real-world scenarios. There are many benchmarks that test the performance of different compilers, but there is no standard benchmark suite to be used.

Benchmarks are produced in two different ways: by using an existing application or by creating programs specially designed to measure certain aspects of the system. There are two different targets for these kinds of benchmarks, application benchmarks show what to expect from the system under a real load, and the synthetic benchmarks provide information about different components of the systems.

Real programs can be any program that user wants to run on the system, and this provides the performance under that type of load. For most applications, it is hard to measure the difference and some benchmarks suites have been created and these are used instead of the real applications. The suites use real applications or parts of them to perform their tests. An example of macro-benchmark is SPEC [19].

The synthetic benchmarks such as Whetstone/Dhrystone are based on statistics to measure the frequencies of operations and then use the frequency in designing the

benchmark. Other micro-benchmarks are created ad-hoc for different purposes. They are usually small that measure only a single aspect that is interesting for the problem.

A set of micro-benchmarks could give a simple overview of the expected performance, but a greater consideration should be given to the application benchmarks.

3.3 Motivation

While there are many aspects of a compiler that are important [55], for scientific computation one important aspect is the execution speed of the generated program. This is a function of the programming language and of the quality of the optimizations performed by the compiler. Performance evaluation is necessary to evaluate the quality of a compiler.

Due to the complexity of the generated code and the interaction intricacies between the program and the advanced architectures, it is not feasible to evaluate the quality of the resulted program by analyzing the generated code. The solution is to use a suite of benchmarks to compare the performance between the unoptimized and the optimized code.

Generic code is not usually as fast as hand written specialized code and benchmarks traditionally tried to offer the best possible implementation for the given programming language. Although, in theory, it is possible to optimize generic code, using different methods for different programming languages implementations, the compilers tested by us did not do a very good job of trying to optimize the generic code. This means that even if the programming language provides support for parametric polymorphism, the benchmark implementation would try to avoid using polymorphism to produce the fastest possible implementation.

To our knowledge, the only previous benchmark that tested parametric polymorphism is Stepanov's abstraction penalty benchmark to measure the performance of C++ compilers for compiling templated code used in STL [33, 43]. Stepanov's benchmark is very limited in its scope, and a more thorough benchmark is useful to see how a compiler behaves under a more complex situation than just parametric calls.

Since previous benchmarking suites did not tackle this problem a new benchmark needed to be created. SciMark 2.0 is a popular benchmark for numerical computation. The reason for choosing SciMark was the availability of source code and the fact that it has been ported also to C. We wanted to target more programming languages to measure their performance under heavy use of generic code, and having the code for more programming languages helped speedup the development. Due to the popularity of SciMark, we later found a version for C# which is very similar to the C# version included in SciGMark. The benchmark was started as a macro-benchmark to measure the performance of the Aldor compiler. Later on, other popular programming languages with support for parametric polymorphism such as C++, C# and Java were included in the benchmark suite.

SciGMark implements generic versions of all the tests included in the original SciMark. Additionally, we implemented some more tests to broaden the spectrum of possible use of generics. SciGMark benchmark suite contains a slightly modified and extended version of SciMark, and the generic versions of the tests included in the extended SciMark, to allow easy comparison between generic and specialized code.

3.4 SciMark

SciMark [29, 51] is a Java benchmark for scientific and numerical computing, which has later been ported to C and C#. It measures several computational kernels and

reports the score in Mflops (Millions of floating point operations per second).

SciMark measures the following computational kernels for floating point performance: fast Fourier transform, Jacobi successive over-relaxation, Monte Carlo integration, sparse matrix multiplication, and dense LU factorization.

3.4.1 Fast Fourier Transform (FFT)

The Fast Fourier transform (FFT) kernel performs a one-dimensional FFT transform. This kernel exercises double arithmetic, shuffling, non-constant memory references and trigonometric functions.

The algorithm of the transform is presented as Algorithm 1.

3.4.2 Jacobi Successive Over-Relaxation (SOR)

Jacobi successive over-relaxation (SOR) on a 100x100 grid exercises typical access patterns in finite difference applications, for example, solving Laplace's equation in 2D with Dirichlet boundary conditions. The algorithm exercises basic "grid averaging" memory patterns, where each `A(i,j)` is assigned an average weighting of its four nearest neighbors.

This is presented in Algorithm 2.

3.4.3 Monte Carlo Integration

This Monte Carlo integration approximates the value of π by computing the integral of the quarter circle $y = \sqrt{1 - x^2}$ on $[0, 1]$. It chooses random points within the unit square and computes the ratio of those within the circle. The algorithm exercises random-number generators, synchronized function calls, and function inlining.

This is presented in Algorithm 3.

Algorithm 1 The FFT algorithm implemented in SciMark 2.0.

Input: data: array of double values, n: size of data, direction: $\{-1, 1\}$
Output: data: array of double values updated inplace

```
for i := 0 to n - 1 do
  ii := 2 * i; jj := 2 * j
  k := n / 2
  if i < j then
    (data[ii], data[ii+1]) ↔ (data[jj], data[jj+1])
  end if
  while k <= j do
    j := j - k
    k := k / 2
  end while
  j := j + k
end for
dual := 1
for bit := 0 to log n do
  w := (1, 0)
  θ := direction * π / dual
  s := sin θ
  t := sin(θ/2)
  s2 := 2 * t^2
  for b := 0 to n - 1 by 2 * dual do
    i := 2 * b
    j := 2 * (b+dual)
    wd := (data[i], data[i+1])
    (data[j],data[j+1]) := (data[i],data[i+1]) - wd
    (data[i],data[i+1]) := (data[i],data[i+1]) + wd
  end for
  for a := 1 to dual - 1 do
    w := (re(w) - s*im(w) - s2*re(w), im(w) + s*re(w) - s2*im(w)) {w = e^(iθ) w}
    for b := 0 to n - 1 by 2 * dual do
      i := 2 * (b + a)
      j := 2 * (b + a + dual)
      z1 := (data[j], data[j+1])
      wd := w * z1
      (data[j],data[j+1]) := (data[i],data[i+1]) - wd
      (data[i],data[i+1]) := (data[i],data[i+1]) + wd
    end for
  end for
  dual := 2 * dual
end for
```

Algorithm 2 The SOR algorithm implemented in SciMark 2.0.

Input: G: matrix of double values of size M × N, ω: double
Output: G

```
for i := 1 to M - 2 do
  for j := 1 to N - 2 do
    G[i,j] := ω / 4 * (G[i-1,j]+G[i+1,j]+G[i,j-1]+G[i,j+1]) + (1 - ω) * G[i,j]
  end for
end for
```

Algorithm 3 The Monte Carlo integration algorithm implemented in SciMark 2.0.

Input: N: number of samples
Output: an approximation of π

```
inside := 0
for i := 1 to N - 1 do
  x := random number
  y := random number
  if x*x + y*y <= 1 then
    inside := inside + 1
  end if
  return 4 * inside / N
end for
```

3.4.4 Sparse Matrix Multiplication

Sparse matrix multiplication uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references.

This is presented in Algorithm 4.

The sparse matrix is represented in compress-row format. If the size of the matrix is $M \times N$ with nz nonzeros, then the `val` array contains the nz nonzeros values, with its i-th entry in the column `col[i]`. The integer vector `row` is of size $M + 1$ and `row[i]` points to the beginning of the i-th row in the `col` array.

Algorithm 4 The matrix multiplication algorithm implemented in SciMark 2.0.

Input: val: non-zero values, col: indexes in val, row: indexes where each row starts in col, x: the vector used for multiplication
Output: y := val * x

```
M := size(row) - 1 {M is the number of rows in the matrix}
for r := 0 to M - 1 do
  sum := 0
  for i := row[r] to row[r+1] do
    sum := sum + x[col[i]] * val[i];
  end for
  y[r] := sum
end for
```

3.4.5 Dense LU Matrix Factorization

Dense LU matrix factorization computes the LU factorization of a dense 100x100 matrix using partial pivoting. This exercises linear algebra kernels (BLAS) and dense matrix operations. The algorithm is the right-looking version of LU with rank-1 updates.

This is presented in Algorithm 5.

3.4.6 Other Aspects of SciMark 2.0

Stopwatch

There is a stopwatch to measure the execution time. The operations offered by the stopwatch are `start`, `stop`, `reset`, `resume` and `read`.

Random number generator

SciMark 2.0 implements a random number generator to avoid measuring the performance of the standard number generator. The initialization algorithm is presented in Algorithm 6 and the algorithm that generates the random numbers is presented in Algorithm 7.

Algorithm 5 The LU factorization algorithm implemented in SciMark 2.0.

Input: A: matrix of size M × N
Output: A: LU factorized, pivot: pivot vector for reordering of rows

```
for j := 0 to min(M, N) do
  jp := j
  for i := j+1 to M-1 do
    if ‖A[i, j]‖ > ‖A[j, j]‖ then
      jp = i
    end if
    pivot[j] := jp
    if A[jp,j] = 0 then
      return error {zero pivot}
    end if
    if jp <> j then
      row(j) ↔ row(jp)
    end if
    if j < M - 1 then
      for k := j + 1 to M - 1 do
        A[k,j] := A[k,j] / A[j,j]
      end for
    end if
    if j < min(M, N) - 1 then
      for ii := j+1 to M-1 do
        for jj := j+1 to N-1 do
          A[ii,jj] := A[ii,jj] - A[ii,j]*A[j,jj]
        end for
      end for
    end if
  end for
end for
```

Algorithm 6 The random number generator initialization implemented in SciMark 2.0.

Input: seed
Output: initialized random number generator

```
m2 := 2^16
jseed := min(||seed||, 2^31 − 1)
if jseed rem 2 == 0 then
  jseed := jseed - 1
end if
k0 := 9069 rem m2
k1 := 9069 / m2
j0 := jseed rem m2
j1 := jseed / m2
for iloop := 0 to 16 do
  jseed := j0 * k0
  j1 := (jseed / m2 + j0 * k1 + j1 * k0) rem (m2 / 2)
  j0 :- jseed rem m2
  m[iloop] := j0 + m2 * j1
end for
i := 4
j := 16
```

Algorithm 7 The random number generator implemented in SciMark 2.0.

Input: an intialized random number generator
Output: a pseudo-random number

```
k := m[i] - m[j]
if k < 0 then
  k := k + 2^31 − 1
end if
if i = 0 then
  i := 16
else
  i := i - 1
end if
if j = 0 then
  j := 16
else
  j := j - 1
end if
k / (2^31 − 1)
```

Kernels Manager

All kernels are controlled by a special manager. For each algorithm there exists a function that sets up the environment, times the function and runs the kernel in a tight loop. If the execution time does not exceed a predetermined resolution time (the value used by SciMark 2.0 is two seconds), the number of iterations is increased and the kernel is run again for the new number of iterations. When the resolution time is exceeded, the time is divided by the number of estimated floating point operations and divided by 10^6 to report the result in MFlops.

Main Entry Point

The main entry point runs all the kernels with some predefined constants. There are two sets of constants: a small dataset and a large dataset. The small dataset is very small so that data should fit into the CPU cache memory to ignore the effects of main memory access. The large dataset is much larger to include the effects of main memory as well.

3.5 SciGMark

Although there are many benchmarks for different aspects of programming languages, to our knowledge, there is no benchmark that tests the performance of generic code except Stepanov's abstraction penalty benchmark [33], which is a micro-benchmark. To be able to measure the performance of optimizations related to generic code, we needed a benchmark to compare the generic code against the fastest implementation.

We consider SciMark to be a good implementation of several real algorithms specialized for floating point numbers. So we have created a generic version of the same algorithms in SciGMark [23] and compared them against the results offered by

SciMark. The results are reported in MFlops (millionfloating point operations per second) so they can be directly comparable with those of SciMark.

3.5.1 New Kernels Included in SciGMark

SciGMark contains also the enhanced version of SciMark to allow easy comparison between generic and high specialized code. The enhanced version of SciMark added two kernels (polynomial multiplication with dense representation, and matrix inversion of sparse matrices using "quadtree" representation) to the kernels already present in SciMark.

Dense Polynomial Multiplication

The algorithm for dense polynomial addition is presented in Algorithm 8, and the algorithm for dense polynomial multiplication is presented in Algorithm 9.

Besides addition and multiplication, the dense polynomials offer operations for building new instances of polynomials equivalent to "0" and "1".

For SmallPrimeField, the algorithms are too simple to be presented here. They consist of modular operations on integer values. All programming languages used for SciGMark have built-in support for modular operations.

Recursive Matrix Inversion

Another useful construct in generic programming is recursive types. As an example of this type of programming recursive matrices were implemented using the a recursive data structure in the form of "quadtree matrices". Suppose there is a matrix M:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Algorithm 8 The dense polynomial addition algorithm implemented in SciGMark.

```
Input:  a,b: dense polynomials
Output:  c := a + b
  ac := coefficients(a)
  bc := coefficients(b)
  rc := new array(max(degree(a), degree(b)) + 1)
  i := 0
  while i <= degree(c) and i <= degree(b) do
    rc[i] := ac[i] + bc[i]
    i := i + 1
  end while
  while  i <= degree(a) do
    rc[i] := ac[i]
    i := i + 1
  end while
  while  i <= degree(b) do
    rc[i] := bc[i]
    i := i + 1
  end while
  return  new polynomial(rc)
```

then A, B, C, D are also block matrices. If the size of M is $n \times m$, then size of each sub-matrix is $n/2 \times m/2$.

The idea behind the quad matrices is to split the matrix in four sub-matrices and represent the matrix by its four blocks. All block type algorithms work very well with this representation. An example of such algorithm for a matrix multiplication is the Strassen matrix multiplication algorithm presented in [2].

The Strassen algorithm presents a way to multiply 2×2 matrices with only seven multiplications and 15 additions. An entry in the matrix can be either a number or another sub-matrix. This algorithm works very nicely on recursive data structures like "quadtree matrices". This data representation is good for memory locality, and provides reasonable memory use for dense and sparse matrices.

Algorithm 9 The dense polynomial multiplication implemented in SciGMark.

```
Input:  a,b: dense polynomials
Output:  c := a * b
  if a or b = 0 then
    return  polynomial(0)
  end if
  if a = 1 then
    return  b
  end if
  if b = 1 then
    return  a
  end if
  ac := coefficients(a)
  bc := coefficients(b)
  rc := new array(degree(a) + degree(b) + 1)
  for i := 0 to degree(c) do
    rc[i] := 0
  end for
  for i := 0 to degree(a) do
    t := ac[i]
    if t <> 0 then
      for j := 0 to degree(b) do
        rc[i+j] := rc[i+j] + t * bc[j]
      end for
    end if
  end for
  return  new polynomial(rc)
```

Using this representation, a new kernel performing matrix inversion was implemented in SciGMark. A simple algorithm for block matrix inversion is given by Aho, Hopcroft and Ullmann in [2]. The algorithm is recursive and is presented in formula 3.1.

$$\begin{aligned} M^{-1} &= \begin{bmatrix} I & -A^{-1}B \\ 0 & I \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & S_A^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix} \\ &= \begin{bmatrix} A^{-1} + A^{-1}BS_A^{-1}CA^{-1} & -A^{-1}BS_A^{-1} \\ -S_A^{-1}CA^{-1} & S_A^{-1} \end{bmatrix} \end{aligned} \tag{3.1}$$

where $S_A = D - CA^{-1}B$ is the Schur complement of A in M. There is a dual way to compute the inverse as seen in formula 3.2.

$$\begin{aligned} M^{-1} &= \begin{bmatrix} I & 0 \\ -D^{-1}C & I \end{bmatrix} \begin{bmatrix} S_D^{-1} & 0 \\ 0 & D^{-1} \end{bmatrix} \begin{bmatrix} I & -BD^{-1} \\ 0 & I \end{bmatrix} \\ &= \begin{bmatrix} S_D^{-1} & -S_D^{-1}BD^{-1} \\ -D^{-1}CS_D^{-1} & D^{-1} + D^{-1}CS_D^{-1}BD^{-1} \end{bmatrix} \end{aligned} \tag{3.2}$$

Both formulae 3.1 and 3.2 require that sub-matrix A or D be invertible. An example of invertible matrix where both A and D are non-invertible is presented in [71]:

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

It is clear that a naive matrix inversion, as presented in formula 3.1 or formula 3.2, cannot be performed in all cases. One solution to this problem is to use pivoting, but that solution is hard to implement with block matrices since it might require some data to be moved between blocks. Watt proposed a more elegant solution, in [71], that only requires block operations. For the real case, the inverse can be computed as $(M^T M)^{-1} M^T$, where M^T is the transpose of the matrix, and the meaning of $M^T M$ is computed using 3.1 or 3.2. This method requires two additional matrix multiplications at top level. However, it is possible to use the regular simple block matrix inversion as presented in 3.1 or 3.2, and only use the solution proposed in [71] when a singular block is encountered.

The algorithm for the simple matrix inversion is presented as Algorithm 10. The `safeInvert` procedure is the safe matrix inversion algorithm.

The algorithm for the safe matrix inversion is presented in Algorithm 11.

There is also a dual to Algoritm 10, which uses the formula 3.2. The inversion function tries to apply first the algorithm from 10, if that one fails, tries to apply the dual and finally, if they don't work the safe version is applied.

Algorithm 10 The `invert` algorithm for block matrix (as presented in formula 3.1).

Input: m: square matrix represented as a quad-tree
Output: $mi := m^{-1}$

if $m_{11}^{-1} = 0$ **then**
  return safeInvert(m)
end if
$d := m_{22} - m_{21}m_{11}^{-1}m_{12}$
if $d = 0$ **then**
  return safeInvert(m)
end if
$mi_{11} := m_{11}^{-1} + m_{11}^{-1}m_{12}d^{-1}m_{21}m_{11}^{-1}$
$mi_{12} := -m_{11}^{-1}m_{12}d^{-1}$
$mi_{21} := -d^{-1}m_{21}m_{11}^{-}1$
$mi_{22} := d^{-1}$
return mi

Algorithm 11 Algorithm `safeInvert` for safe block matrix inversion.

Input: M: square matrix represented as a quad-tree
Output: $Mi := M^{-1}$

$mtm_{11} := m_{11}^T m_{11} + m_{21}^T m_{21}$
$mtm_{12} := m_{11}^T m_{12} + m_{21}^T m_{22}$
$mtm_{21} := (m_{11}^T m_{12} + m_{21}^T m_{22})^T$
$mtm_{11} := m_{12}^T m_{12} + m_{22}^T m_{22}$
$mtm := (mtm_{11}, mtm_{12}, mtm_{21}, mtm_{22})$
return $mi := mtm^{-1}m^T$ {Invert M^TM instead of M}

3.5.2 From SciMark 2.0 to SciGMark

When creating SciGMark, two opposite approaches were used. For the algorithms implemented in SciMark 2.0, a generic version version was created by creating new interfaces that provided the basic operations required by the algorithms. Then implementations of those types were provided. Finally, a generic form of the kernel was created that used the same algorithm as the original SciMark algorithm, but using the operations provided by the generic type.

For most algorithms, it was possible to create types that represent the semantics of the data. For example, the fast Fourier transform uses complex numbers, but the algorithm represents a complex number by two `double` values. The generic form creates the type `Complex`. Similarly, the recursive matrix inversion uses a matrix type.

To separate the types from their implementation, all operations are performed through interfaces. The UML diagram of the interfaces and their relationship is presented in Figure 3.1. The interfaces used are described below.

The `ICopiable` interface ensures that a new instance can be created given an existing one. This was necessary to overcome the type parameter instantiation problem which exists in the implementation of generics in the Java programming language.

The `IRing` interface provides the basic arithmetic operations: addition, subtraction, multiplication, and division. It also provides operations to create the `0` and `1` elements.

The `IInvertible` interface offers the `invert` operation.

The `ITrigonometric` is a convenience interface that implements trigonometric operations. The specialized FFT algorithm uses sin and cos to compute the recurrence $\omega = e^{i\theta}\omega$. These operations are grouped in the `ITrigonometric` interface.

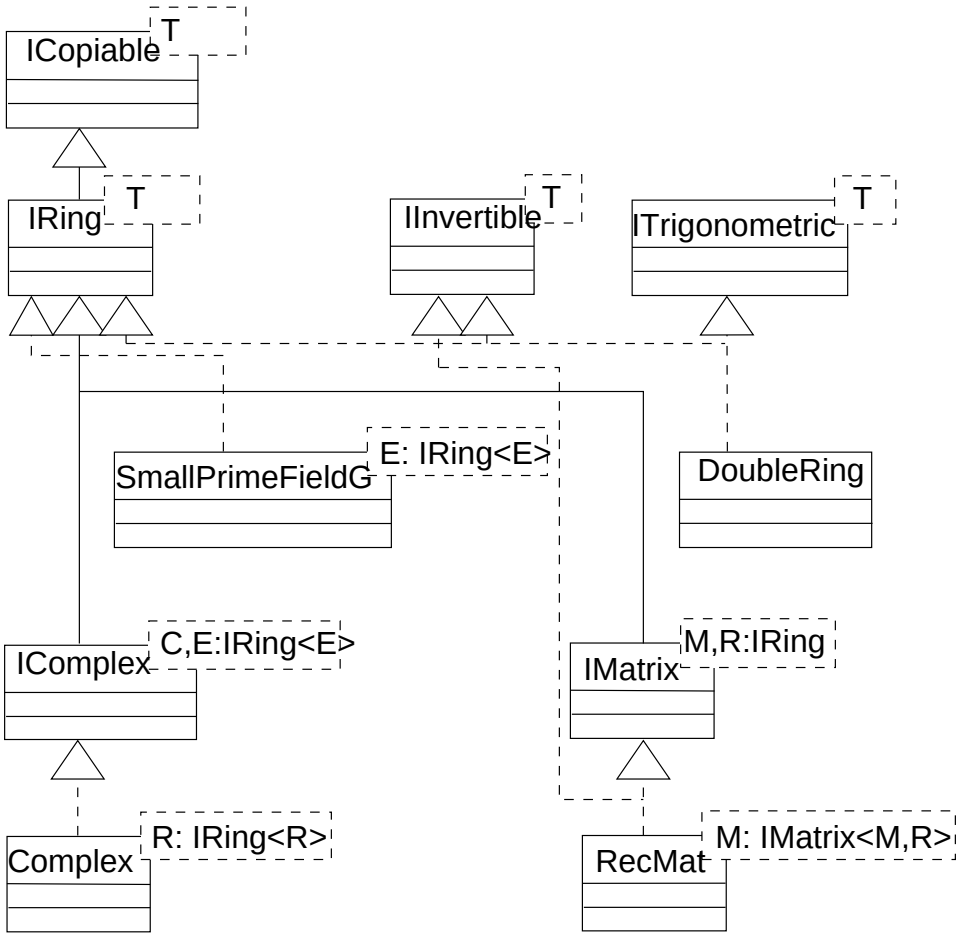

Figure 3.1: The UML diagram of the interfaces and implementation examples for `Complex` and `RecMat`.

The interface `IComplex` builds on top of `IRing` with operations specific to complex numbers: `create` to create new complex numbers given two values of type `R`, `re` extracts the real part of the complex number, `setRe` sets the real part of the complex number, `im` extracts the imaginary part of the complex number, `setIm` sets the imaginary part of the complex number.

Similarly, the interface `IMatrix` adds operations specific to matrices: `get` gets a value at the coordinates i and j, `set` sets the values at coordinates i and j, `t` performs a transposition of the matrix, `getRows` retrieves the number of rows and `getCols` retrieves the number of columns.

The simplest data type is `DoubleRing`. As seen in Figure 3.1, `DoubleRing` implements the `IRing`, `IInvertible` and `ITrigonometric` interfaces. The `DoubleRing` type is just a simple wrapper on top of `double` basic data type used by SciMark 2.0.

We are aware that comparing boxed objects in the form of `DoubleRing` (which is nothing more than a wrapper on top of double precision floating point type) with a primitive type like double introduced a large overhead. However, the current Java language specification does not allow use of primitive types as type parameters. This is not the case for C++ or C#, but for the sake of uniformity we used similar approach for all languages involved in the test. C++ is able to remove the overhead of `DoubleRing` when stack allocated objects are used. For C# it is possible to specify primitive types as type parameters, but it is not possible to describe their operations through our interfaces, so we define our own data types, which are structures instead of classes for efficiency reasons.

The next level is the `Complex` data type, which uses another type parameter, namely the type of the real and imaginary part. The complex number can use any of the types that implement the `IRing` interface. In some cases the use of complex numbers is implicit in SciMark, as is the practice in much scientific computation. For

example, the FFT test uses an array of $2n$ `double`s to represent n complex numbers. In our generic kernel, we replaced this use with `Complex <DoubleRing>` class. This explicit form is more likely to be used in practice with generic codes.

For the generic version of the Monte Carlo algorithm it was only necessary to replace the `double` basic type with `DoubleRing`. There is no other useful change that can be performed to make it more generic.

To generalize successive over-relaxation, sparse matrix multiplication and LU factorization, the `double` primitive type was only replaced by the `DoubleRing` class and use the operations through the `IRing` interface. A more complete implementation for these kernels would have been to provide generic versions of `Matrix` and `Vector` corresponding to sparse and dense cases.

Since generic polynomial arithmetic uses type parameters in another representative manner, we decided to include polynomial multiplication as a test in SciGMark. To make comparisons, it was necessary to provide both specialized and generic implementations. Both versions used a dense polynomial representation with coefficients from a prime field. For the polynomial multiplication we started with the generic version and then created the specialized version by inlining the modular operations. The generic polynomial multiplication test used the generic class `DensePolynomial <R>`. The parameter `R` was instantiated with the `SmallPrimeField` class, which would perform the modular calculations within its arithmetic methods. This test made significant use of memory allocation to create the polynomial objects.

Another high level data type example is `RecMat`. It implements the `IMatrix` and the `IInvertible` interfaces. This means any operation available in `IMatrix` and `invert` can be performed in `RecMat`. It takes as type parameters another `IMatrix` for block sub-matrices, and a `IRing` for the elements of the matrix. The type `IRing` is used by the `get` and `set` methods that treat the matrix as a non-recursive matrix.

Since `RecMat` implements `IMatrix` interface it can use itself as a type parameter, allowing constructions such as:

```
RecMat<RecMat<...>,DoubleRing>,DoubleRing>
```

where for each recursive layer added another power of 2 is added to the size of the matrix. This means that if there are k levels of `RecMat`, the size of the matrix will be $2^k \times 2^k$ and for the next layer, the new size will be $2^{k+1} \times 2^{k+1}$. Using only `RecMat`, the type construction cannot stop, so a new type was created `Matrix2x2` which a 2×2 matrix. Using `Matrix2x2` a 4×4 matrix can be constructed using:

```
RecMat<Matrix2x2<DoubleRing>,DoubleRing>
```

There is also a `Matrix1x1` which extends `DoubleRing` with matrix type operations, but the performance using `Matrix1x1` is far worse than `Matrix2x2`. Therefore, only `Matrix2x2` was used in the tests.

3.5.3 SciGMark for Aldor

With the Aldor programming language interfaces are implemented using categories and classes are implemented using domains. Parametric polymorphism can be obtained in Aldor using functors that produce new domains using type parameters. For example `Complex` domain is implemented in Listing 3.1.

All operations used are declared in categories, and the example for the complex case is presented in Listing 3.2.

In the generic version, the basic arithmetic operations are used from the domain `E`, which is of type `IRing`.

Listing 3.1: Implementation of Complex parametric type in Aldor.

```
define MyComplex(E: IRing): IMyComplex(E) == add {
        Rep == Record(r: E, i: E);

        create(re: E, im: E): % == per [re,im];
        getRe(t:%): E == rep(t).r;
        setRe(t:%, re: E): () == rep(t).r := re;
        getIm(t:%): E == rep(t).i;
        setIm(t:%, im: E): () == rep(t).i := im;
        (t:%)+(o:%): % == per [rep(t).r+rep(o).r, rep(t).i+rep(o).i];
        (t:%)-(o:%): % == per [rep(t).r-rep(o).r, rep(t).i-rep(o).i];
        (t:%)*(o:%): % == {
                rt := rep(t);   ro := rep(o);
                per [ rt.r*ro.r - rt.i*ro.i, rt.r*ro.i + rt.i*ro.r];
        }
        (t:%)/(o:%): % == {
                rt := rep t; ro := rep o;
                denom := ro.r*ro.r + ro.i*ro.i;
                per [rt.r*ro.r+rt.i*ro.i/denom, rt.i*ro.r-rt.r*ro.i/denom];
        }
        coerce(t:%): double == rep(t).r::double;
        (t:TextWriter)<< (d:%): TextWriter == t<<rep(d).r<<"+i*"<<rep(d).i;
        ...
};
```

Listing 3.2: Implementation of Complex category in Aldor.

```
define IMyComplex(E: IRing): Category == IRing with {
        create: (E,E) -> %;
        getRe: (%) -> E;
        setRe: (%,E) -> ();
        getIm: (%) -> E;
        setIm: (%,E) -> ();
        test : () -> ();
};
```

In the specialized version, the standard Aldor library provides `DoubleFloat` for double precision floating point numbers. The `DoubleFloat` type uses a boxed value to represent floating point values (the actual representation is `Record(float:DFlo)`).

To achieve maximum performance, in SciGMark, the type used was `DFlo` which the representation of an unboxed double precision floating point value.

To be able to work with this type, a specialized version of arrays was implemented for double values called `PAD`. There is another version for arrays of integer values, called `EPA`. All these implementation use the operations available in the `Machine` domain. This was done to achieve the maximum performance possible for specialized cases.

The kernels were implemented using a domain for each kernel, and also some domains were used for the `Stopwatch`, `Random`, kernel manager, and main entry point. As an example of generic and specialized implementations, the fast Fourier transform code is included in the appendix A.1 and A.2. While the Aldor programming language is not an object-oriented programming language and the implementation could have been done with top level functions, this approach was used to have an uniform implementation among all programming languages.

3.5.4 SciGMark for C++

C++ supports object-oriented programming. Classes are supported directly. C++ does not have the notion of interface, but abstract classes can be used instead of interfaces. C++ also supports parametric polymorphism with the *templates* mechanism.

In C++, the use of stack allocated objects is much faster than the heap allocated objects. Code similar to typical Java, where each object is heap allocated, will sometimes result in worse performance for C++ than Java, because Java has a highly

Table 3.1: C++ micro-benchmark: STL vector versus C-style arrays.

Container	Parameter	Iterator	Time (s)
vector	double	no	9.8
vector	double	iterator	9.8
array	double	no	9.9
array	double	double *	9.9
vector	Double	no	9.9
vector	Double *	no	34.4
vector	Complex<Double>	no	18.5
vector	double,double	no	17.5

optimized memory allocator. Therefore, stack allocated objects for the C++ version of SciGMark tests have been used.

To decide which implementation to use for our benchmark, we wrote a micro-benchmark to check if there is a significant difference between arrays and STL `vector` (Table 3.1). The two containers used are `vector` and primitive arrays. The elements of the containers are of basic type `double`, or a wrapper class `Double`. It can be observed that the use of iterators makes no difference. The results were obtained using GNU C++ compiler `gcc` version 3.3.

Based on the simple tests from Table 3.1, which show that the performance of `std::vector` is close to primitive arrays, and because the C++ standard template library is well supported by most of the C++ compilers, we decided to use the collections available from C++ instead of basic arrays provided by plain C.

Another optimization problem, worth mentioning, was the use of another file for the `Double` class. If `Double` was defined in a separate file, the C++ optimizer did not inline the code of `Double` into the caller. So we had to either put the definition together with the declaration in the header file, or to define the `Double` class in the same file as the caller code.

An implementation example for the `IComplex` interface is presented in Listing 3.3.

Listing 3.3: `IComplex` interface in C++.

```
template <typename C, typename E>
class IComplex : IRing<C> {
public:
        virtual C create(E re, E im) const = 0;
        virtual E getRe() const = 0;
        virtual void setRe(E re) = 0;
        virtual E getIm() const = 0;
        virtual void setIm(E im) = 0;
};
```

Listing 3.4: `Complex` class implemented in C++.

```
template<typename R> class Complex : IComplex<Complex<R>,R> {
        R re, im;
public:
        Complex(R re = 0, R im = 0)     {this->re = re; this->im = im;}
        Complex<R> create(R re, R im) const {return Complex<R>(re,im);}
        Complex<R> copy() const {return Complex<R>(re.copy(), im.copy());}
        std::vector<Complex<R> > newArray(int size) const
        {
                return std::vector<Complex<R> >(size);
        }
        R getRe() const {return re;}
        void setRe(R re) {this->re = re;}
        R getIm() const {return im;}
        void setIm(R im) {this->im = im;}
        Complex<R> operator+(const Complex<R>& o) const
        {
                Complex<R> t;
                t.re = re + o.getRe(); t.im = im + o.getIm();
                return t;
        }
        Complex<R> operator-(const Complex<R>& o) const
        {
                Complex<R> t;
                t.re = re - o.getRe(); t.im = im - o.getIm();
                return t;
        }
```

Listing 3.5: `Complex` class implemented in C++.

```
  Complex<R> operator*(const Complex<R>& o) const
  {
    Complex<R> t;
    t.re = re * o.re - im * o.im;  t.im = re * o.im + im * o.re;
    return t;
  }
  Complex<R> operator/(const Complex<R>& o) const
  {
    Complex<R> t;
    R den = o.re*o.re+o.im*o.im;
    t.re = (re*o.re+im*o.im)/den;  t.im = (im*o.re-re*o.im)/den;
    return t;
  }
  double coerce()  {return re.coerce();}
  std::string toString()
  {
    std::stringstream str;
    str<<re.toString()<<"+i*"<<im.toString();
    return str.str();
  }
  ...
};
```

C++ templates do not support bounded polymorphism. As a consequence it is not possible to specify more precisely the type of the parameter. One of the reasons for having a specified bound on the type parameters is a clearer understanding of the requirements for the implementation of the type parameter. With the current approach the only way to specify the interface is through human language in documentation. Another reason for having the bounds specified explicitly would greatly improve the type checking in the generic type leading to improved error messages. The parameter name is replaced in a macro expansion approach in the resulted code after the instantiation of a parameter with a type value. In Listing 3.3, an instantiation like:

```
IComplex<Complex, DoubleRing>
```

would replace all occurrences of `C` with `Complex` and all occurrences of `E` with `DoubleRing`. After the substitution the new class is type checked for correctness.

The class `Complex` which implements the `IComplex` interface is presented in Listings 3.4 and 3.5.

The generic version overloaded the basic arithmetic operators in `DoubleRing` and all other classes that implement the `IRing` interface, hence all operations are generic. For the specialized case all operations are done on basic type `double` for maximum performance.

Classes were created for `Stopwatch`, `Random`, kernel manager and main entry point. C++ supports programming without objects and templates can also be applied to functions, but for uniformity we used the same approach as in Java and C# that only allow code to exist in objects.

3.5.5 SciGMark for C#

C# has an interesting implementation for generics. It offers both homogeneous and heterogeneous approach of implementing the generics. For stack allocated types it uses a heterogeneous implementation like C++ and for heap allocated objects it uses a homogeneous approach like Java.

The stack allocated objects are basic types and user defined types that are declared with the keyword `struct`. In C# instantiations of classes declared with the keyword `class` heap allocated. The syntax for `struct` and `class` is identical, just as in C++. One problem with stack allocated objects in C# is that the compiler can decide to allocate them on heap if they are stored in collections [38]. In particular, if stack allocated objects are stored in collections that require heap allocated objects, the stack allocated objects are automatically "boxed". The boxing procedure transforms stack allocated into heap allocated objects. Kennedy and Syme [34] propose a mechanism to specialize the code for reference types as well. Their proposal also shares the specializations between "compatible" instantiations. It is not clear if this proposal was implemented in the commercial .NET framework. Generic collections do not box stack allocated objects.

For `Complex` interface, the source code is presented in Listing 3.6. The value type `Complex` that implements `IComplex` is presented in Listing 3.7. As seen in Listing 3.7, `struct` was used to create value type objects. This was done to use the specialization of data implemented for value type objects.

The generic version creates corresponding methods for each of the basic arithmetic operations. Even though the C# programming language allows operator overloading, the operators cannot be used in interfaces, which means they cannot be called through an interface. Since SciGMark uses the types through their interface, we had to use

Listing 3.6: `IComplex` interface implemented in C#.

```
1 public interface IComplex<C, E> : IRing<C> where E: IRing<E> {
2     C create(E re, E im);
3     E getRe();
4     void setRe(E re);
5     E getIm();
6     void setIm(E im);
7 }
```

usual function names such as: `a` instead of `+`, `s` instead of `-`, `m` instead of `*` and `d` instead of `/`.

For the specialized case all operations are done on basic type `double` for maximum performance.

Just as in the original SciMark, classes were created for `Stopwatch`, `Random`, kernel manager and main entry point.

3.5.6 SciGMark for Java

In Java, like in C#, generics are a new addition to the language. Unlike, C#, the implementation of generics in Java has some problems due to backward/forward compatibility requirements imposed by Sun. Forward compatibility requires that programs written for current virtual machine to be run on older virtual machines. Due to this restriction, the virtual machine could not include some extra bytecode instructions that were required to properly support the generics.

We tried to use different optimization hints for the Java compiler to help it produce faster code. In our experiments, we discovered that by using the current release of the server version of the just-in-time compiler from Sun, it is not important to the overall performance to use modifiers like static, or final. Primitive arrays were used instead

Listing 3.7: `IComplex` interface implemented in C#.

```
1  public struct Complex <R> : IComplex<Complex<R>,R>      where R: IRing<R>
2  {
3    private R re;
4    private R im;
5    public Complex(R re, R im) {this.re = re; this.im = im;}
6    public Complex<R> create(R re, R im) {return new Complex<R>(re,im);}
7    public Complex<R> copy() {return new Complex<R>(re.copy(), im.copy());}
8    public R getRe() {return re;}
9    public void setRe(R re) {this.re = re;}
10   public R getIm() {return im;}
11   public void setIm(R im)       {this.im = im;}
12   public Complex<R> a(Complex<R> o) {
13     return new Complex<R>(re.a(o.re), im.a(o.im));
14   }
15   public Complex<R> s(Complex<R> o) {
16     return new Complex<R>(re.s(o.re), im.s(o.im));
17   }
18   public Complex<R> m(Complex<R> o) {
19     return new Complex<R>(re.m(o.getRe()).s(im.m(o.getIm())),
20                           re.m(o.getIm()).a(im.m(o.getRe())));
21   }
22   public Complex<R> d(Complex<R> o) {
23     R d = o.re.m(o.re).a(o.im.m(o.im));
24     return new Complex<R>(re.m(o.getRe()).a(im.m(o.getIm())).d(d),
25                           im.m(o.getRe()).s(re.m(o.getIm())).d(d));
26   }
27   public double coerce() {return re.coerce();}
28   public override String ToString() {
29     return re+"+i*"+im.ToString();
30   }
31   ...
32 }
```

Container	Parameter	Time (s)
array	double	1
array	D	20
Vector	D	26

Table 3.2: Java micro-benchmark: Vector versus arrays.

of collection classes such as `Vector`, because of the performance loss. `Vector` uses synchronization mechanism to implement thread safe collections and this is one of the reasons why `Vector` is slow compared with primitive arrays. A non-synchronized version of list collection is `ArrayList` which performs better than `Vector` for single threaded access. One simple test that uses `Vector` and arrays as containers and performs an element by element multiplication shows a 25% performance decrease in performance. Parameter `D` in table 3.2 is a simple wrapper for `double` basic type.

The erasure technique used to implement generics in Java created some problems that complicated the implementation of generic algorithms. In this technique, the type parameter is replaced with the type bound if it exists, or if there is no type bound, the parameter is replaced with `Object`. For example `Complex` class in Java is implemented as seen in Listing 3.8. Due to the erasure of the type, the actual type used to instantiate the type is unknown and this makes it impossible to create new objects of the type used as parameter.

The `IComplex` interface is presented in Listing 3.9.

Java does not allow operator overloading, therefore the generic version creates corresponding methods for each of the basic arithmetic operations: `a` instead of `+`, `s` instead of `-`, `m` instead of `*` and `d` instead of `/`.

For the specialized case all operations are done on basic type `double` for maximum performance. Java was the initial implementation for the SciMark, and all the supporting classes were reused from there.

Listing 3.8: `Complex` implementation in Java.

```
public class Complex <R extends IRing<R>>
       implements IComplex<Complex<R>,R> {
  private R re;
  private R im;
  public Complex(R re, R im) {this.re = re; this.im = im;}
  public Complex<R> create(R re, R im) {return new Complex<R>(re,im);}
  public R re() {return re;}
  public void setRe(R re) {this.re = re;}
  public R im() {return im;}
  public void setIm(R im) {this.im = im;}
  public Complex<R> a(Complex<R> o) {
    return new Complex<R>(re.a(o.re()), im.a(o.im()));
  }
  public Complex<R> s(Complex<R> o) {
    return new Complex<R>(re.s(o.re()),im.s(o.im()));
  }
  public Complex<R> m(Complex<R> o) {
    return new Complex<R>(re.m(o.re()).s(im.m(o.im())),
                          re.m(o.im()).a(im.m(o.re())));
  }
  public Complex<R> d(Complex<R> o) {
    R denom = o.re().m(o.re()).a(o.im().m(o.im()));
    return new Complex<R>(re.m(o.re()).a(im.m(o.im())).d(denom),
                          im.m(o.re()).s(re.m(o.im())).d(denom));
    }
  public double coerce() {return re.coerce();}
  public String toString() {return new String(re + "+i*" + im);}
}
```

Listing 3.9: `IComplex` interface implementation in Java.

```
interface IComplex<C, E extends IRing<E>> extends IRing<C> {
  C create(E re, E im);
  E re();
  void setRe(E re);
  E im();
  void setIm(E im);
  C fromPolar(E modulus, E exponent);
}
```

Listing 3.10: Implementation of a generic type in Maple.

```
MyGenericType := proc(R)
    module ()
        export f, g;
        # Here f and g can use u and v from R.
        f := proc(a, b) foo(R:-u(a), R:-v(b)) end;
        g := proc(a, b) goo(R:-u(a), R:-v(b)) end;
    end module
end proc:
```

3.5.7 SciGMark for Maple

Parametric polymorphism can be achieved in Maple using module-producing functions. The basic mechanism is to write a function that takes one or more modules as parameters and produces a module as its result. The module produced uses operations from the parameter modules to provide abstract algorithms in a generic form. An example can be seen in Listing 3.10. The results obtained here have been presented in [24].

We investigated two ways to use this basic idea to provide generics in Maple:

- The first method — the "object oriented" (OO) approach — represented each value as a module. This module had a number of components, including fields (locals or exports) for the data and for the operations supported. Each value would be represented by its own constructed module.

- The second method — the "abstract data type" (ADT) approach — represented each value as some data object, manipulated by operations from some module. One module was shared by all values belonging to each type, and the module provided operations only. The data was free-standing.

We produced two Maple versions of SciGMark: one for each of these approaches.

Listing 3.11: Implementation of the double wrapper.

```
1  DoubleRing := proc(val::float)
2      local Me;
3      Me := module()
4          export v, a, s, m, d, gt, zero, one,
5                 coerce, absolute, sine, sqroot;
6          v  := val;
7          a  := (b) -> DoubleRing(Me:-v + b:-v);
8          s  := (b) -> DoubleRing(Me:-v - b:-v);
9          m  := (b) -> DoubleRing(Me:-v * b:-v);
10         d  := (b) -> DoubleRing(Me:-v / b:-v);
11         gt := (b) -> Me:-v > b:-v;
12         zero    := () -> DoubleRing(0.0);
13         one     := () -> DoubleRing(1.0);
14         coerce  := () -> Me:-v;
15         absolute:= () -> DoubleRing(abs(v));
16         sine    := () -> DoubleRing(sin(v));
17         sqroot  := () -> DoubleRing(sqrt(v));
18     end module:
19     return Me;
20 end proc:
```

In both the OO and ADT versions, the SciGMark kernels use numerical operations from a generic parameter type. For the concrete instantiation of this parameter, we created the module "`DoubleRing`" as a wrapper for floating point. The code for the OO version of `DoubleRing` is presented in Listing 3.11.

This version simulates the object-oriented model by storing the value and the operations in a module. Each call to `DoubleRing` produces a new module that stores its own value. The exports `a`, `s`, `m` and `d` correspond to addition, subtraction, multiplication and division. We chose these names, rather than `‘+‘`, `‘-‘`, `‘*‘` and `‘/‘`, since Maple's support for overloading basic operations is rather awkward and we were not producing a piece of code for general distribution. The last two functions, `sine` and

Listing 3.12: The ADT version of double wrapper.

```
1  DoubleRing := module()
2      export a, s, m, d, zero, one,
3             coerce, absolute, sine, gt, sqroot;
4      a  := (a, b) -> a + b;
5      s  := (a, b) -> a - b;
6      m  := (a, b) -> a * b;
7      d  := (a, b) -> a / b;
8      gt := (a, b) -> a > b;
9      zero := () -> 0.0;
10     one  := () -> 1.0;
11     coerce   := (a::float) -> a;
12     absolute := (a) -> abs(a);
13     sine     := (a) -> sin(a);
14     sqroot   := (a) -> sqrt(a);
15 end module:
```

`sqroot`, are used only by the FFT kernel to replace complex operations and to test the correctness of the results.

The code for the ADT version of `DoubleRing` is given in Listing 3.12.

It can be seen that this approach does not store the data; it provides only the operations. As a convention, one must coerce the float type to the representation used by the module. In this case the representation used is exactly float (as can be seen from the coerce function). The DoubleRing module is created only once when the module for each kernel is created.

Each SciGMark kernel exports an implementation of its algorithm and a function to compute the estimated floating point instruction rate. Each of the kernels is parametrized by a module, `R`, that abstracts the numerical type. An example of this structure is presented in Listing 3.13.

The high-level structure of the implementation is the same in both the OO and ADT generic cases. The detailed implementations of the functions in the module are

Listing 3.13: The generic version of the FFT algorithm.

```
gFFT := proc(R)

    module()
        export num_flops, transform, inverse;
        local transform_internal, bitreverse;

        num_flops := ...;
        transform := proc(data::array) ... end proc;
        inverse := proc(data::array) ... end proc;
        transform_internal := proc(data, direction) ... end proc;
        bitreverse := proc(data::array) ... end proc;
    end module:

end proc:
```

Model	Code
Specialized	`x*x + y*y`
Object-Oriented	`(x:-m(x):-a(y:-m(y))):-coerce()`
Abstract Data Type	`R:-coerce(R:-a(R:-m(x,x), R:-m(y,y)))`

Table 3.3: Differences in implementation of specialized and generic code

different, however. An example of the same piece of code in all three cases is shown in Table 3.3. One can see that the specialized version makes use of the built-in Maple operations. In this case, the values use Maple's native floating point representation. The other two versions make use of exported operations from `R`, which in our case is given by `DoubleRing`. The object-oriented model uses a module instance to obtain the operations associated with the data. One can see that in the object-oriented model the variables are themselves modules and are used to find the operations. On the other hand, the abstract data type model uses a module for the operations that is not connected to the data in any explicit way. In the abstract data model, the parameter passed in to the kernel module is the same for all operations on all data.

We tested the kernels described in Section 3.5. These were implemented in the same way in Maple as in the other languages. In particular, we did not make use of Maple's own arithmetic to treat complex values and matrices as single objects. By doing this, and by taking tests where the parameter values were relatively light weight (floating point numbers), we hoped to expose the worst case performance of generics.

3.6 Results

The following tables show the performance both for generic code, and hand-written specialized code. The entries in the tables are given in *MFlops*. The tests were performed on a Pentium IV 3.2 GHz with 1MB of cache and 2GB RAM. The operating system was Windows XP SP2. The compilers used were: Cygwin/gcc 3.4.4 for C++, Sun Java JDK 1.6.0 for Java, Microsoft.NET v2.0.50215 for C#, and version 1.0.3 for Aldor.

For C++, SciGMark was run on other compilers, too. The two compilers tested were: Microsoft Visual C++ 2005 and Intel C++ compiler version 10. The Intel compiler produced a ratio between generic and specialized of approximately 2×, and the Microsoft compiler obtained a ratio of approximately 4.5× speedup. The results are consistent with those produced by the GNU compiler. Therefore, the GNU C compiler was representative.

The results are presented for both small and large datasets for each programming language individually. In the small dataset, the size of each test is reduced such that all the data required by each kernel is small enough to fit in the processor's cache memory to avoid measuring the cache misses penalty. At the other extreme, the large dataset has bigger size of working data such that it is very unlikely that all the data could fit in the processors's cache, therefore cache misses will occur.

The absolute values of the result for each programming languages are not as important as the ratio because the goal of this benchmark was to provide a comparison between code that is specialized by hand and the code produced by the compiler for the generic case.

The testing procedure ran the benchmark for each compiler three times and reported the average value obtained. The variations between results of each iteration were less than 2%. The data uses random values and the arrays are filled in a linear fashion.

It is also outside the scope of this benchmark to compare the languages with each other since some compilers, such as the one for Java and the one for C++ are able to use the extended instructions sets and perform automatic vectorization of the code, while C# and Aldor do not do this.

Kernel	Generic	Specialized	Ratio	Size
Fast Fourier Transform	5	259	51.8	1024
Successive Over Relaxation	43	488	11.4	100x100
Monte Carlo	85	165	1.9	$n \times t_{integrate} \geq t_{min}$
Sparse Matrix Multiplication	18	295	16.4	N=1000, nz=5000
LU factorization	21	342	16.3	100x100
Polynomial Multiplication	9	81	9.0	N=40
Recursive Matrix Inversion	7	28	4.0	N=16x16
Composite	27	237	8.8	

Table 3.4: Performance of generic and specialized code in the Aldor programming language for the small dataset. The values are presented in MFlops.

Kernel	Generic	Specialized	Ratio	Size
Fast Fourier Transform	3	41	13.7	1048576
Successive Over Relaxation	39	446	11.4	1000x1000
Monte Carlo	85	164	1.9	$n \times t_{integrate} \geq t_{min}$
Sparse Matrix Multiplication	16	188	11.8	100000x1000000
LU factorization	23	247	10.7	1000x1000
Polynomial Multiplication	10	87	8.7	100
Recursive Matrix Inversion	7	26	3.7	128x128
Composite	26	171	6.6	

Table 3.5: Performance of generic and specialized code in the Aldor programming language for the large dataset. The values are presented in MFlops.

3.6.1 Results in Aldor

The results obtained for the SciGMark benchmark using the Aldor programming language on the small dataset is presented in Table 3.4. For the large dataset the results are presented in Table 3.5.

From Tables 3.4 and 3.5, one can see that the Aldor programming language suffers from a significant performance penalty when generic code is used. The difference is smaller for the larger dataset, because the CPU has to spend extra cycles on accessing the data, which in this case is big enough to cause cache misses. For the large dataset the useful work is divided between algorithm and memory access and this reduces the effect of a less than optimal code for the generic case. For the cases where ratios

Kernel	Generic	Specialized	Ratio	Size
Fast Fourier Transform	247	649	2.6	1024
Successive Over Relaxation	273	558	2.0	100 x 100
Monte Carlo	131	212	1.6	$n \times t_{integrate} \geq t_{min}$
Sparse Matrix Multiplication	453	783	1.7	1000x5000
LU factorization	610	868	1.4	100x100
Polynomial Multiplication	135	649	4.8	40
Recursive Matrix Inversion	130	127	1.0	16x16
Composite	283	478	1.7	

Table 3.6: Performance of generic and specialized code in the C++ programming language using the small dataset. The values are presented in MFlops.

Kernel	Generic	Specialized	Ratio	Size
Fast Fourier Transform	72	47	0.7	1048576
Successive Over Relaxation	277	529	1.9	1000 x 1000
Monte Carlo	130	212	1.6	$n \times t_{integrate} \geq t_{min}$
Sparse Matrix Multiplication	249	271	1.1	100000x1000000
LU factorization	266	307	1.2	1000x1000
Polynomial Multiplication	137	47	0.3	100
Recursive Matrix Inversion	129	134	1.0	16x16
Composite	180	235	1.3	

Table 3.7: Performance of generic and specialized code in the C++ programming language using the large dataset. The values are presented in MFlops.

are larger, the algorithms perform more computations in the generic ring `R` and also allocate many temporary objects as intermediate values.

3.6.2 Results in C++

The results obtained for the SciGMark benchmark using the C++ programming language on the small dataset are presented in Table 3.6. For the large dataset the results are presented in Table 3.7.

From Tables 3.6 and 3.7, the C++ programming language suffers very little compared to the other languages tested when using generic code. Due to the heterogeneous nature of the templates, all code is specialized by the compiler automatically,

Kernel	Generic	Specialized	Ratio	Size
Fast Fourier Transform	37	240	6.5	1024
Successive Over Relaxation	65	418	6.4	100x100
Monte Carlo	24	62	2.6	$n \times t_{integrate} \geq t_{min}$
Sparse Matrix Multiplication	67	426	6.4	1000x5000
LU factorization	59	395	6.7	100x100
Polynomial Multiplication	43	305	7.1	40
Recursive Matrix Inversion	47	124	2.6	16x16
Composite	49	281	5.7	

Table 3.8: Performance of generic and specialized code in the C# programming language on the small dataset. The values are presented in MFlops.

Kernel	Generic	Specialized	Ratio	Size
Fast Fourier Transform	23	34	1.5	1048576
Successive Over Relaxation	64	400	6.3	1000x1000
Monte Carlo	23	62	2.7	$n \times t_{integrate} \geq t_{min}$
Sparse Matrix Multiplication	73	329	4.5	100000x1000000
LU factorization	60	333	5.6	1000x1000
Polynomial Multiplication	44	421	9.7	100
Recursive Matrix Inversion	46	124	2.7	128x128
Composite	48	244	5.1	

Table 3.9: Performance of the generic and specialized code in the C# programming language using the large dataset. The values are presented in MFlops.

and the expected results are that generic code is closest in performance to the specialized code when compared with the other languages tested. As in other cases, the large data set reduces the difference.

3.6.3 Results in C#

The results obtained for the SciGMark benchmark using the C# programming language on the small dataset is presented in Table 3.8. For the large dataset the results are presented in Table 3.9.

From Tables 3.8 and 3.9, the C# programming language suffers more than C++, but still not as much as the other programming languages included in this test. This might

Kernel	Generic	Specialized	Ratio	Size
Fast Fourier Transform	38	423	11.0	1024
Successive Over Relaxation	26	788	30.7	100x100
Monte Carlo	14	69	4.9	$n \times t_{integrate} \geq t_{min}$
Sparse Matrix Multiplication	53	307	5.8	1000x5000
LU factorization	34	856	24.9	100x100
Polynomial Multiplication	38	148	3.8	40
Recursive Matrix Inversion	30	36	1.2	16x16
Composite	33	374	11.2	

Table 3.10: Performance of generic and specialized code in the Java programming language using the small dataset. The values are presented in MFlops.

Kernel	Generic	Specialized	Ratio	Size
Fast Fourier Transform	3	44	14.2	1048576
Successive Over Relaxation	16	703	45.1	1000x1000
Monte Carlo	13	69	5.2	$n \times t_{integrate} \geq t_{min}$
Sparse Matrix Multiplication	27	208	7.6	100000x1000000
LU factorization	21	228	10.9	1000x1000
Polynomial Multiplication	51	151	3.0	100
Recursive Matrix Inversion	32	31	1.0	128x128
Composite	23	205	8.8	

Table 3.11: Performance of generic and specialized code in the Java programming language using the large dataset. The values are presented in MFlops.

be due to the hybrid approach used in the .NET virtual machine just-in-time compiler that specializes basic data types and stack allocated types whenever possible. Our implementation tried to use stack based objects instead of reference objects wherever possible. As in other cases, the large data set reduces the difference.

3.6.4 Results in Java

The results obtained for the SciGMark benchmark using the Java programming language on the small dataset is presented in Table 3.10. For the large dataset the results are presented in Table 3.11.

From Tables 3.10 and 3.11, one can observe that Java also suffers a performance penalty from the usage of generic code. It is almost comparable with Aldor which also uses a homogeneous approach to implement parametric polymorphism. The results show that some specialization should help Java improve the performance. As in other cases, the large data set reduces the difference.

3.6.5 Results in Maple

The results of running SciGMark in Maple 10 are presented in Table 3.12. The benchmark was run on a Pentium 4 processor with 3.2 GHz, 1MB cache and 2 GB RAM. The operating system used was Linux, Fedora Core 4.

The results show that abstract data type model is very close in performance to the specialized version. The ratio between abstract data type and specialized versions is roughly 1.3. This means there is not strong justification, based on performance alone, to avoid writing generic algorithms in Maple. We should point out two situations, however, that require special consideration: The first is that with several nested levels of generic construction the compounding of the performance penalty may become significant. The second consideration is that some Maple procedures obtain their performance from an evaluation mode, `evalhf`, that treats hardware floats specially. Our investigation assumes `evalhf` is not being used.

The last column of Table 3.12 shows the results for the object-oriented model. This model tries to simulate as closely as possible the original SciGMark test, given the language features offered by Maple. This model constructs many modules during the benchmark, leading to a significant performance degradation. The ratio between object-oriented and specialized versions is 9.9; that is, the generic OO code is about one order of magnitude slower than the specialized code. This shows that this ap-

Test	Specialized	Abstract Data Type	Object Oriented
Fast Fourier Transform	0.123	0.088	0.0103
Successive Over Relaxation	0.243	0.166	0.0167
Monte Carlo	0.092	0.069	0.0165
Sparse Matrix Multiplication	0.045	0.041	0.0129
LU factorization	0.162	0.131	0.0111
Composite	0.133	0.099	0.0135
Ratio	1.0	1.3	9.9

Table 3.12: SciGMark MFlops in Maple 10

proach to writing generic code might be avoided in Maple. If generic object-oriented code is truly required for some application, it would be worthwhile to explicitly separate the instance-specific data values from a shared-method module. Then values would be composite objects (e.g. lists) with one component being the shared module.

The performance penalty for generic code should not discourage writing of generic code, but rather encourage compiler writers to think harder about optimizing generic code constructs. Generic code is useful, they provide a much needed code reuse that can simplify the libraries. An example of such optimization has been proposed by specializing the type according to the particular parameter used when constructing the type, as mentioned in [22].

3.7 Conclusions and Future Research

As expected, the results show a rather big difference between generic and specialized code, showing that there is room for improvement in the way compilers handle generic code. Before we developed SciGMark, there was no tool to measure how much slower the code becomes when making heavy use of generics. The good thing about

benchmarks is that they generally show the weak points of the compilers motivating the developers to produce better compilers that generate more efficient code.

There are certain benefits that a generic programming style provides, including improved modularity, improved maintainability and re-use, and decreased duplication. In a mathematical context, writing programs generically also helps programmers operate at a higher, more appropriate level of abstraction. With these potential benefits, it is important to understand whether there are opposing reasons that preclude use of this style. We have made a quantitative assessment of the performance impact of using a generic programming style in Aldor, C++, C#, Java and Maple.

The use of generic code code impacts differently the performance of the code in various programming languages. There is a wide range of performance penalties.

C++ produced rather efficient generic code with a ratio of only 1.3 or 1.7 between generic and specialized. This result was expected due to nature of the implementation of templates in C++.

The next in the performance list was C# which was showing a ratio of 5.7 for the small dataset or 2.7 for the large dataset. C# uses a hybrid approach toward generics implementation, doing specialization for stack allocated objects and code sharing for reference objects.

The next on the performance gap between generic and specialized code was Aldor. The Aldor programming language uses a homogeneous approach similar with the one present in Java. Unlike Java, the parametric polymorphism in Aldor is not converted to sub-classing polymorphism, but the performance of the memory allocation domain runtime representation and function lookup in Aldor could be the reasons of the gap between the two tested cases. One solution to this problem is to avoid heap wherever possible to close the gap. This contrasts with the results obtained by Stanford bench-

mark that showed generic Aldor code runs at similar speeds to specialized `C` code. The implementation of the Stanford benchmark did not implement any domains and did not try to use a generic approach. The difference shows the importance of full optimization of generics.

The last in order on the performance list was Java which offers only the code sharing approach, where the polymorphism is actually sub-classing polymorphism. The results obtained for Java are 11.3 for small dataset and 8.8 for the large one. We can see here that homogeneous approach chosen for generics implementation in Java exhibits a large performance gap and we believe that Java could benefit a lot by specializing the polymorphic code, and not rely on sub-classing polymorphism.

We have found that writing generic code using parametric polymorphism and abstract data types does not introduce an excessive performance penalty in Maple. We believe that this is in part due to the fact that Maple is interpreted and there is little overall optimization. Even specialized code executes function calls for each operation. Carefully written generic code and code that is not excessively generic can do well in Maple environment. We suggest that it would be worthwhile to consider modifications to the Maple programming language to make generic programming easier.

Writing code in Maple that tries to simulate the sub-classing polymorphism provided by object-oriented languages such as Java, can be very expensive in Maple. The code written using this approach can be an order of magnitude slower compared to the specialized code.

The benchmark in its current state, only measures the speed performance. Most of the times, and especially for numerically intensive applications this is the most important variable. However, this is not always the case. Usually, there is a trade-

off between code speed and code size when trying to optimize and this might be a direction for future development. Simple code duplication becomes a problem with recursive types. For example the recursive matrix representation could produce huge code increase for larger sizes of the matrix.

Chapter 4

Automatic Domain Code Specialization

4.1 Introduction

The usual way to enhance the performance of a given program is to profile the code and then optimize the bottlenecks. The optimization phase is performed by the programmers taking into account factors such as target architecture, and particular properties of the problem to be solved. These two factors must be coupled with the programmer's skill in both of them. Based on these facts, the resulting final product may attain the best possible implementation given the circumstances.

Programming languages started with simple instructions and no abstractions of the hardware, making them very hard to program with. In order to simplify the programming task and to increase the productivity of the programmers new abstractions were introduced. Abstractions such as structure, functions, objects helped producing more complex programs with reduced programming complexity. However, abstractions do not come for free. For each abstraction there is a cost associated, which

can be reduced, or in some cases even removed by using very capable compilers. Parametric polymorphism is one such abstraction.

This chapter proposes an optimization technique that will alleviate the effects of generic code use transparently for the programmer. The solution proposed is to create specialized versions of the domains based on the instantiations discovered by interpreting the FOAM.

The experimental results show that an increase up to an order of magnitude faster is possible in some cases. This is possible because in those cases the specialized code can be reduced to simple basic operations rather than operations on generic data.

The results presented in this chapter have been presented in [22].

4.2 Motivation

The Aldor programming language has very good support for generic programming. Due to this fact, the libraries provided with Aldor are highly generic and by using these libraries is possible to create complicated types. Deeply nested types are types formed by composing several generic type constructors. An example of such deeply nested type can be obtained with the following construction:

```
Vector(Matrix(Polynomial(Complex(DoubleFloat))))
```

The above expression tells us that we have a vector whose elements are matrices, the elements of the matrix are polynomials with complex coefficients. Also, the real and imaginary part of complex number are represented by floating-point numbers. This is a fully defined deeply nested type. Even though `Vector`, `Matrix`, `Polynomial` and `Complex` are parametric type constructors, the resulting type is not parametric anymore. Because in each case, the type parameter has been instantiated by a con-

crete type: `DoubleFloat` for `Complex`, `Complex(DoubleFloat)` for `Polynomial` and so on.

Constructions such as the one presented above are not very common, but types with three or four nested levels are commonly encountered. The depth of the nested type is given by the number of generic type constructors that are used in the construction. The number of type constructors at the same level does not increase the depth of the type.

Now, let us assume that one would call an operation from the `Vector` domain. For example, multiply by a constant. This means each element of the list should be multiplied by the constant. The elements are matrices, therefore it will call a function to multiply a matrix by a constant from the `Matrix` domain. Each element contained by the matrix is a polynomial, so the matrix multiplication must invoke the constant multiplication operation from the polynomial domain. All these function calls require many frame activations and clean up whenever a function from a domain is called, because for every function call the context has to be switched. This introduces an overhead that can be avoided by specializing the domain. Furthermore, after specializing the operations of the domain, it is possible to optimize the resulted operation using other intra-procedural optimizations which are already implemented in the compiler.

Because Aldor allows dynamic types, it means that we may not know one domain constructor from the deeply nested domain tower at compile-time. We may have situations like these:

```
Vector(Matrix(Polynomial(Complex(       X))))

X     (Matrix(Polynomial(Complex(Fraction))))

Vector(Matrix(X         (Complex(Fraction))))
```

where `X` is an unknown domain at the compile-time. In situations like these, the compiler only specializes the part that is constant. For example in the construction `Vector( Matrix( Polynomial( Complex(X))))`, the compiler specializes `Vector( Matrix( Polynomial( Complex)))` to a new type. This type will still be parametric, but the operations from `Vector` call operations from `Matrix` directly, instead of calling operations from a generic type. The specialization is applied recursively on the resulted type with the `Polynomial` argument. This happens for all constant parametric types, and the variable ones are skipped.

We have chosen to implement our optimizations in Aldor because it offers many benefits such as: excellent support for generic programming, runtime creation of domains, homogeneous implementation of domains, advanced compiler optimizations useful for our optimization. C++ is already efficient with respect to generics by implementing templates in a similar fashion to macro processors. C# and Java do not offer a good support for writing generic algorithms in the sense of Standard Template Library from C++ by separating the data from the generic algorithms.

4.3 Domain Code Specialization

As shown previously, generic functions and functors in Aldor are implemented using homogeneous approach. What makes this more interesting in the Aldor setting is that domains may be created at runtime by functions. This means that parametric domain share the same code among all instances. While this is very flexible, the performance is not optimal because certain optimizations cannot be performed due to the requirement that the shared code must be polymorphic, so it cannot take advantage of any specific properties of the instantiation.

Listing 4.1: `Ring` category

```
1 Ring: Category == with {
2         +: (%, %) -> %;
3         *: (%, %) -> %;
4         0: %;
5         <<: (TextWriter, %) -> TextWriter;
6 }
```

4.3.1 Example

To illustrate the homogeneous approach used by the Aldor compiler to implement the domains, let us start with a simple example of a polynomial multiplication.

First, we define a `Ring` category that will specify the operations available. The `Ring` category declares a type that performs the arithmetic operations: `+` and `*`. It also has a `0` value to produce a new value and an operation to display on screen by using the operator `<<` (see Listing 4.1.)

Next, we create the domain that defines operations on complex numbers. The `Complex` domain is a non-parametric domain of type Ring. It implements all the operations declared in `Ring` and has some extra operations like the `complex` constructor, and two getter functions `real` and `imag` to extract the real and imaginary part of the complex number, respectively.

The representation used by the Complex domain is a pair of integer numbers enclosed in a record.

This domain would normally be parametrized by the type of the elements contained in the pair, but for simplicity they were set to integer type.

The polynomial domain is a parametrized domain. One can see that it expects a type parameter of type `Ring` and produces a domain `Polynomial` also of type `Ring`. Besides the Ring operations, the `Polynomial` domain provides two constructors.

Listing 4.2: Generic version of domain for complex operations

```
MI ==> MachineInteger;

Complex: Ring with {
        complex: (MI, MI) -> %;
        real: % -> MI;
        imag: % -> MI;
} == add {
        Rep == Record(re: MI, im: MI);
        import from Rep;
        complex(r: MI, i: MI): % == per [r, i];
        real(t: %): MI == rep(t).re;
        imag(t: %): MI == rep(t).im;
        (a: %) + (b: %): % == {
                complex(rep(a).re+rep(b).re, rep(a).im+rep(b).im);
        }
        (a: %) * (b: %): % == {
                ra := rep.a; rb := rep.b;
                r := ra.re*rb.re-ra.im*rb.im;
                i := ra.re*rb.im+ra.im*rb.re;
                complex(r, i);
        }
        0: % == complex(0, 0);
        (w: TextWriter) << (t: %): TextWriter == {
                w << rep.t.re << "+i*" << rep.t.im;
        }
}
```

Listing 4.3: Generic version of domain for polynomial operations.

```
1  MI ==> MachineInteger;
2  import from MI;
3
4  Polynomial(C: Ring): Ring with {
5          poly: MachineInteger  -> %;
6          poly: Array(C) -> %;
7  } == add {
8          Rep == Array(C);
9          import from Rep;
10         import from C;
11
12         poly(size: MachineInteger): % == {
13                 res: Rep := new(size);
14                 i:=0@MI;
15                 while i < size repeat { res.i := 0@C; i := i + 1; }
16                 per res;
17         }
18
19         poly(a: Array(C)): % == {
20                 res: Rep := new(#a);
21                 i := 0@MI;
22                 while i < #a repeat { res.i := a.i; i := i + 1; }
23                 per res;
24         }
```

The internal representation used for the polynomial objects is a dense representation using an `Array`.

Using the parametric polynomial domain presented in Listings 4.3 and 4.4 the complex domain presented in Listing 4.2, one could write a program that performs an addition between two polynomials as in Listing 4.5.

With the homogeneous approach the code for the `Polynomial` domain is parametric and cannot take any advantage of the fact that complex numbers are represented by a pair of integer numbers. As a consequence the code cannot be optimized in any way inside the `Polynomial` domain with respect to particularities of `Complex` domain.

Listing 4.4: Generic version of domain for polynomial operations (Continued).

```
        (a: %) + (b: %): % == {
                c: Rep := new(#rep(a));
                i := 0@MI;
                while i < #rep(a) repeat {
                        c.i := rep(a).i + rep(b).i;
                        i := i + 1;
                }

                per c;
        }

        (a: %) * (b: %): % == {
                i := 0@MI;
                c: Rep := new(#rep(a)+#rep(b)-1);
                while i < #c repeat { c.i := 0@C; i := i + 1; }
                i := 0@MI;
                while i < #rep(a) repeat {
                        j := 0@MI;
                        while j < #rep(b) repeat {
                                c(i+j) := c(i+j) + rep(a).i * rep(b).j;
                                j := j + 1;
                        }
                        i := i + 1;
                }

                per c;
        }

        0: % == {poly(1);}

        (w: TextWriter) << (t: %): TextWriter == {
                w << "[";
                i := 0@MI;
                while i < #rep(t) repeat {
                        w << rep(t).i << " ";
                        i := i + 1;
                }
                w << "]";
        }
}
```

Listing 4.5: Addition between two polynomials.

```
1  import from Polynomial(Complex);
2  import from Complex;
3  import from Array Complex;
4  c1 := complex(1,1);
5  c2 := complex(2,2);
6  arr: Array Complex := new(2);
7  arr.0 := c1; arr.1 := c2;
8  a := poly(arr);
9  b := poly(arr);
10 c := a + b;
```

For the unoptimized case, the function calls to functions defined in domains can be seen in Figure 4.1. The call to `+` from `Polynomial(Complex)` (labeled with (1) in Figure 4.1) is unknown at compile-time. The closure for `+` is created in the initialization part of the program, namely program index 0 in the current unit. The function call is performed in another function. Since the Aldor compiler only optimizes code inside functions it will not be able to unfold locally to code of the function from `Poly`. Similarly, the call to `+` performed inside the `+` function defined in the `Poly` domain (labeled with (2) in Figure 4.1) will defer the function discovery to the run-time.

Due to the heavily generic nature of the libraries used by Aldor, this would mean a poor performance for the Aldor code. Since the Aldor compiler does not perform interprocedural analysis, the only possible alternative to improve the code is to aggressively unfold the functions at the call site and try to perform the local optimizations on the local function. This works quite well in some cases. So, the way the current compiler works is described by Algorithm 12.

The localization of the implementation for a closure is not easy. To be able to optimize closures, the current optimizer of the Aldor compiler uses information constructed by the previous stages of the compiler, namely type inference, to find the

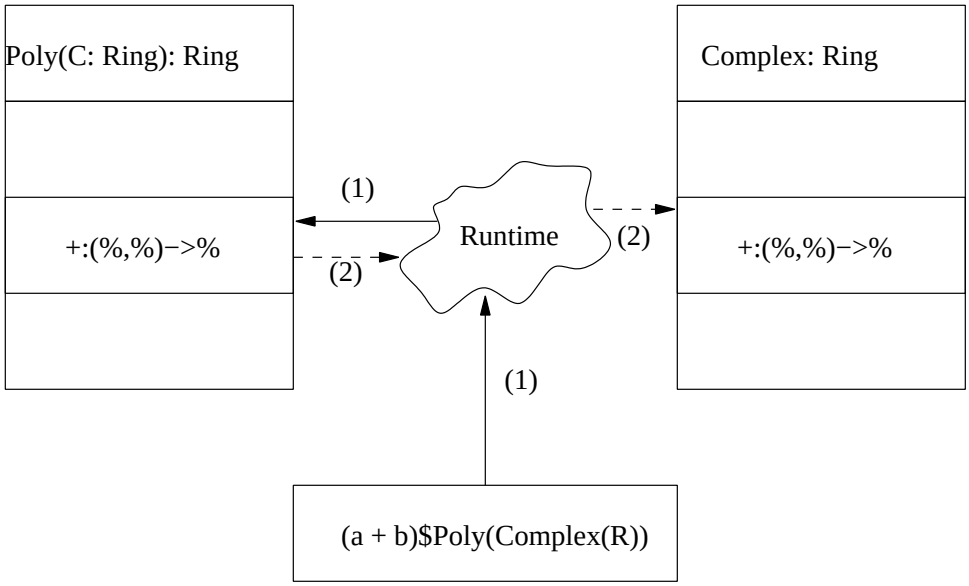

Figure 4.1: Dynamic dispatch of function calls used by unoptimized code.

Algorithm 12 The inline algorithm used by the Aldor compiler.

Input: : FOAM of the current compilation unit
Output: : FOAM with procedural integration performed

```
q := create priority queue
inlined := true
for all p ∈ programs(unit) do
  while inlined do
    inlined := false
    for all e ∈ expression(p) do
      if type(e) = CCall or OCall then
        set priority based on size and number of call sites in current procedure
        enqueue(q, e)
      end if
    end for
    while not empty(q) do
      c := dequeue(q)
      if size(p) < max_size then
        inline call c
        inlined := true
      end if
    end while
  end while
end for
```

actual implementation of the `+` operator. For every call to closures, the operator of the function call is annotated with additional information about the call (e.g. the complete type, if known, and the location of the implementation.) If there is enough information at compile time to find the implementation, then the code is unfolded locally at the call site. This local unfolding happens recursively, as long as the target of the call is known and the local function did not exceed a certain limit.

This approach has the potential to produce an optimal solution for cases when the whole domain is fully expanded locally. An example of such domain declaration is `Polynomial(Complex)`. This construction is not parameterized in the function that contains the code from Listing 4.5. However, the drawback to this approach is that no optimization is performed on the innermost function calls, should the compiler decide that the `+` from `Polynomial` is not worth inlining. A reason to stop inlining a function is that the size of the calling function becomes too large. Also, if there are several calls to a function from parametric domain, the function might not be deemed worthy to be inlined, and again the whole optimization inside the domain is impossible.

Yet another case when the existing optimization will not work is in all those cases when the function cannot be decided at compile time. In these cases, the calling function cannot unfold the callee locally, and the callee cannot be optimized.

4.3.2 Proposed Optimization for Domain Functions

We see that many opportunities exist for optimization of parametric polymorphism and that the Aldor compiler seems a suitable test bed for their implementation.

With all these bad cases for the current optimizer, it was clear that a better

solution was desirable. The techniques we propose will try to overcome all these problems and optimize in the different possible cases.

The proposed solution, in its general form, is to use specialization as a technique to produce versions of the domains that are no longer parametric.

Partial evaluation is program specialization [32]. Program specialization is a very effective optimization technique. Partial evaluators specialize programs as a whole rather than functions. They take as input a program and some of the input and evaluate the parts they can, producing a residual program that takes only the rest of the input and produces the same result, only faster.

In Aldor, domains are homogeneous, which means that the same domain definition can be used with different types at runtime. Cooper [17] used a technique called *cloning* to improve the interprocedural optimization for functions, by cloning the body of the function and setting one or more arguments of the function to a constant value. This allowed a more powerful optimization on the cloned procedure, because it had some variables replaced by constants. We extend this approach by cloning the entire domain according to the instantiations used throughout the program. This allows us to replace the type variables with constants and perform partial evaluation on the cloned domain.

From an abstract point of view our type specialization is partial evaluation of type producing functions. In practical terms, we take a FOAM program in the representation used by the Aldor compiler and we specialize it according to some type instantiations. We specialize only those types that can be statically determined *i.e.* the types that are constant at compile-time. Whenever a new type is constructed based on some parametric type, we produce a new *specialized type*. This optimization is called *type tower* optimization.

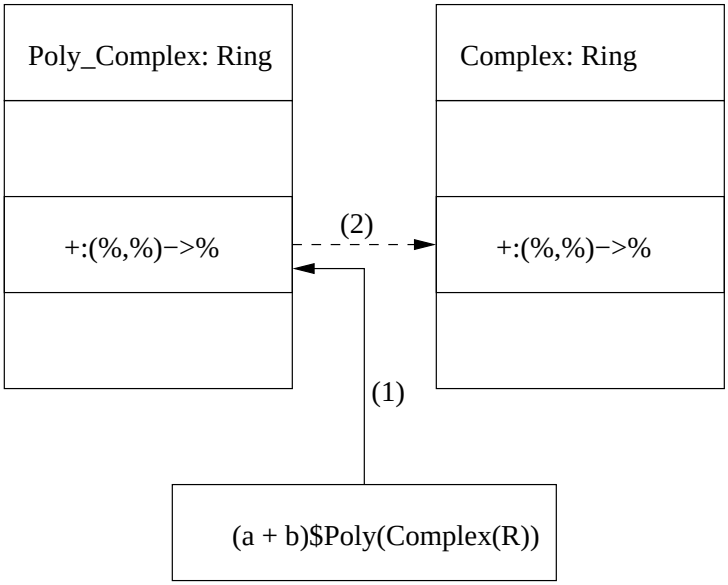

Figure 4.2: After domain specialization function calls are not dynamically dispatched.

The technique used is similar to the two-level language used for partial evaluation that not only specializes the type constructor, but it also specializes all the exports of that domain, effectively specializing the code of the domain constructed by the type constructor. This creates operations of that specialized domain as monolithic operations that are more efficient. Figure 4.2 depicts the new structure of the resulted code after the specialization. The specialized domain `Polynomial_Complex` is not parametrized anymore and as a consequence, it does not need the dynamic dispatch to find the implementation of `+` from `Complex` (as it can be seen from the link labeled (2) in Figure 4.2). Also the `+` from `Polynomial` is also directly linked as shown from link labeled (1) in Figure 4.2.

The overhead of the domain creation is significant, but it only happens once when the first function exported by the domain is invoked. However, the speedup does not come from domain creation overhead, rather from the aggressive optimization of the specialized functions.

This kind of optimization is not possible with the optimizations currently present in the compiler. This optimization requires an interprocedural analysis to pass the information between callers and domain producing functions.

One problematic point in performing optimizations is the function call. This happens in our case when domains are created by instantiating parametric domains. The information about instantiation does not cross the function call boundary. However, there are two approaches to solve this problem: first is to inline everything into the caller and optimize everything there, and the second is to perform interprocedural analysis and send some information across the call boundary.

As we have seen before, the Aldor compiler uses the first approach, but it is limited to thoses cases where the domain information is complete, or when the procedure integration is performed.

The disadvantage of data flow analysis from the interprocedural analysis is that even if the information is sent to the parametric domain the optimization is still limited. To produce a more powerful optimization, a different technique was proposed by Cooper, Hall and Kennedy [17]. They propose to create new clones for the functions based on the data flow analysis. In the cloned functions, one variable is replaced by a constant and the resulted code is optimized, usually with better results than the generic form of the function.

The solution proposed here uses a similar approach by creating a clone. Unlike the simple cloning procedure, our optimizer clones *all* the operations exported by the domain. This effectively clones the whole type. The cloning of the type simplifies the analysis greatly and is able to send the information quicker to all exports of the parametric domain.

A lot of work has been done to perform type-based optimizations (see section 4.6), which is very useful for object-oriented programming languages where virtual

function calls are expensive. Moreover, type specialization optimizations have been performed for functional programming languages like ML for primitive types. The optimization proposed for the Aldor compiler is a lot simpler to implement and than a general interprocedural analysis.

The types are checked by the compiler in the type checking phase, so any type construction is correct. Therefore, by specializing the type there is no danger to change the behavior of the program.

The compilation will always terminate, because it does not try to evaluate the type constructors. For those cases where the complete type cannot be determined at compile time, only the part that is clearly determined is specialized in a manner similar to the one described in the section 4.2. The part that cannot be determined is left in parametric form and no special optimization is performed on it.

This optimization is restricted to those cases where the type is fully or partly known at compile time. For those types that are dynamically constructed at run-time, as is the case with some functions able to generate new types at run-time, it is not possible to optimize in this manner.

4.3.3 Finding Domain Constructing Functions and Getting Domain Related Information

All Aldor code optimizations are performed by manipulating the FOAM code. Analyzing the FOAM code is much simpler than analyzing the Aldor source code. As mentioned earlier, the domains in Aldor are run-time entities and they have no special representation in FOAM. Therefore, the first step in the optimization is to reconstruct partial type information.

There are two possible ways to reconstruct that information: use the information inferred by the type inference phase of the Aldor compiler, or use knowledge about the runtime implementation and reconstruct the type information from the FOAM by interpreting it.

At first sight it seems that the first approach might be simpler. Unfortunately, complete information is not available for compiled modules. The full type information can be provided for some variables, but changing code without a thorough understanding of the operating procedures can lead to subtle errors that very hard to track. This is not to say that it cannot be used at all. This approach was used to deal with optimizing code that resides in the already compiled library.

The second approach is a safe alternative. Retrieving the information was done using a minimal FOAM abstract interpreter. This interpreter detects the run-time values of the variables and fills the parameters of the functions for function calls. The execution starts from the program index `0` in the current unit. All the code for preparing the environment of the file happens inside that program. The code related to initialization of environments defined in other units from the library happens in other places and that will be discussed later when dealing with the library code optimizations.

Domain related operations are performed using run-time function calls such as: `domainMake` and `rtDelayGetExport`. The FOAM interpreter must be able to understand the run-time function call to reconstruct domain related information. So for every run-time function there is an equivalent in the interpreter. The version from the interpreter is not always the same with the version from the Aldor run-time system.

Listing 4.6: C language representation of important run-time values

```
struct val {
        enum VTag  tag;
        Foam       *orig;
        Foam       foam;
        union {
                AInt   i;
                Clos   clos;
                Dom    dom;
                Dtor   dtor;
                String str;
                Arr    arr;
                Rec    rec;
                Env    env;
        };
};
```

FOAM Values

The first step in implementing the interpreter was to create a representation for important run-time values produced by the FOAM code. For this reason, the structure `Val` was introduced, which can be seen in Listing 4.6. FOAM expressions can have various values. For a full description of all the possible values please refer to [73]. For the purpose of the domains optimization, a restricted set of supported values was chosen. The values can be seen in the `union` part of the data structure. The `union` simulates a polymorphic type in `C`, a much nicer approach is to use sub-classing in object-oriented programming languages.

The protocol of operation for the `Val` data structure is to construct a new `Val` object with an initial FOAM expression, placed in `orig` and to try to evaluate the expression in a given run-time context. Partial values that can be evaluated are placed in the `foam` field, while values that can be completely evaluated to one of the known types is placed in the corresponding value types. Operations on `Val` type objects

Listing 4.7: C language representations of FOAM arrays and records.

```
1 struct arr {
2         AInt size;
3         Val  *v;
4 };
5
6 struct rec {
7         AInt fmt;
8         Val  *v;
9 };
```

should only be performed through the Val interface methods, methods starting with a twrV... prefix.

Arrays and Records

Arr and Rec have been introduced to deal with FOAM types used in the construction of the domain. As seen in the domain implementation in FOAM (section 2.6,) all exports of the domain are represented by the Array domain. The domain is internally represented by a Rec and an Arr objects.

These types needed support in the interpreter and therefore two data structures corresponding to each were introduced. The Rec is an exact map of Rec data type from FOAM, where each Rec has a format declaration, similar to the class declaration, while the Rec object is an actual instance of that declaration. For the arrays, the representation contains also the size of the array. This information is not stored in the FOAM Arr, but it was added in the interpreter to be able to use arrays for functions parameters list.

Listing 4.8: C language representation of closures

```
struct clos {
        Lib     lib;
        AInt    idx;
        AInt    orig_idx;
        AInt    crtLine;
        Foam    prog;
        Arr     loc;
        Arr     par;
        Env     env;
        Clos    caller;
        Bool    evaluated;
        Val     retVal;
        AIntList jumpedLabel;
};
```

Closures

All domain exports are stored as closures in the domain's environment. This is one of the main types handled by the FOAM interpreter. The closure binds together the code of the function and the execution environment of that function. A special type was designed to deal with programs related issues. The representation of closures can be seen in Listing 4.8.

Closures can be defined in different compilation units, therefore the definition index is not enough to uniquely identify the a closure. The field `lib` stores the originating library for non-local closures. For local closures `lib` is `NULL`.

The next two fields `idx` and `orig_idx` identify the definition of the closure. The second index, `orig_idx` is used when a function is cloned to check if two cloned functions come from the same origin.

The following set of fields `loc` and `par` hold the values of the local variables and parameters of the closure. They are represented by `Arr`.

The closure's environment is stored in `env`, and the lexical level stack is constructed by saving the caller closure in `caller`. The `evaluated` flag is used to check if the closure has already been evaluated. The return value of the function is stored in `retVal`.

The last field, `jumpedLabel` is a list of labels that were jumped, as explained below. This was introduced to allow breaking out of some loops when the loop condition cannot be evaluated. A simple scanning of the sequence of instructions from the special functions is not enough to properly determine the value of the variables. This happens because some values are constructed out of sequence. To solve this problem, the FOAM interpreter had to jump to different labels based on the condition of some flags. In some cases, the value of the condition is not known because it requires the evaluation of all the initialization code from imported library units. If it is not possible to evaluate a condition, and the condition creates an infinite loop, the jump is not performed second time. This extra condition for jumping, means the FOAM interpreter is not a general purpose interpreter. Because of this, code that constructs domains based on values computed at run-time cannot be properly evaluated. The code in domain constructing functions does not contain loops, but regular function can use conditions to select exports from domains. This generates a warning from the compiler, and will compute the domain information in a regular function instead of the domain constructing function. This means that the current FOAM interpreter cannot safely be applied to any FOAM function.

Each closure can be evaluated using `twrClosEvalBody`. During the evaluation all FOAM expression that are present in the initialization programs are handled. At the end of the evaluation, the result is stored in `retVal`.

Most domain related services are implemented in the runtime system of Aldor.

The interpreter should be able to handle those functions to reconstruct domain information. The functions supported by the FOAM interpreter are:

- `rtDelayedGetExport!` – this function takes as arguments a domain, the hash code of the name, and the hash code of the type. If successful, it will return the closure corresponding to the export from the requested domain. In the run-time system, this function is a lookup in the lists of exports of the domain. In the interpreter, it was extended to return the export if and only if there is only one function with the given name hash. If the function is overloaded and a second function with the same name hash is found then the function will fail.

- `domainAddHash!` – this function sets the hash of the domain, and calls the corresponding function from `Dom`.

- `domainAddNameFn!` – this function sets the function that returns the name of the domain. If the domain is parametric, the name can only be known at run-time. The name of the domain is not important for domain discovery; it is only used for debugging information.

- `domainAddExports!` – this function sets the exports of the domain. First the three lists (name hashes, type hashes and closures) are created, next they are stored in the corresponding `Dom` object.

- `domainGetExport!` – this is handled by the same function that handles the delayed exports.

- `domainMake` – this function creates a run-time Aldor domain, or correspondingly a `Dom` object in the FOAM interpreter.

- `domainFill!` – this function does nothing in the interpreter. It should fill the information in the run-time domain.
- `domainHash!` – this function retrieves the hash of the domain. It is used to construct the type hash, when domains are used in the signatures of the function.
- `domainMakeDispatch` – this function created the dispatch vector at run-time. This required due to the two layers of the Aldor domains. It does not affect in any way the interpreter.
- `domainName` – retrieves the name of the domain.
- `rtSingleParamNameFn` – this is a special form of the naming function for domain that take only one argument as parameter.
- `namePartFrString` – used when concatenating names to form the domain name with more than one argument.
- `namePartConcat` – concatenates the names of the parameters to the name of the domain.
- `rtConstNameFn` – constructs a function that returns a constant name for a domain.
- `rtLazyDomFrInit` – this function is used to initialize domains from other compilation units.
- `rtCacheAdd` – this function adds parametric domains to cache to be retrieved faster. The interpreter needs to handle this function to retrieve the domain that is being created.

Listing 4.9: Unoptimized call

```
1 (CCall Word (Lex 1 24 *) (Loc 1 a) (Loc 0 b))
```

- `lazyGetExport!` – this is used for domains imported from other compilation units. They are ignored by the interpreter, since the standard Aldor compiler is able to optimize them.

- `rtDelayedInit!` – this function is used to initialize domtors imported from other compilation units.

- `stdGet` – these are just wrapper functions for closures used by the run-time system. They are handled only to retrieve the closures called by them.

Function dispatch uses hash code values to identify specific functions. For parameteric domains, the hashes are computed at run-time. By specializing the domains, the hash codes of the domains become known at compile time. As a result, the hashes of the function can be computed. In order to compute the hashes of the functions, an evaluator for basic calls (`BCall`) had to be implemented. The `BCall` evaluator only evaluates those basic calls that are used to compute the values of the hashes, this means only integer values and only the functions described in the algorithm for combining hashes (see Listing 2.33.)

Another function of the `Clos` "class" is to optimize the code. The function `twrClosCCallInline` optimizes the code by replacing `CCall` instructions with `OCall` instructions. The difference between `CCall` and `OCall` is that closure calls perform calls on already constructed closures, while open calls provide the implementation details in the call. For example, a call to the multiplication function is invoked; the corresponding closure call is presented in Listing 4.9.

Listing 4.10: Optimized call

```
1 (EEnsure (CEnv (Lex 1 24 *)))
2 (OCall Word (Const 33 *) (CEnv (Lex 1 24 *)) (Loc 1 a)(Loc 0 b))
```

The optimized call, obtained by replacing the closed call with an open call, is presented in Listing 4.10. One can see that the open call provides also the constant `(Const 33 *)`, which is the index in the current unit of the definition. The environment used for the open call is the same environment as for the closed call.

The open call presented in Listing 4.10 can easily be inlined by the Aldor compiler optimizer since it provides all the required information to perform the local unfolding. This method relies on the existing heuristics of the the Aldor compiler to decide if a function unfolding is worth performing.

Simply replacing the `CCall` instructions with `OCall` instructions, is not enough. It is additionally necessary to make sure the necessary environment has been created. Normally this is done when the function is obtained dynamically from the domain, but when it is inlined another step is needed. An extra `EEnsure` instruction has to be inserted (see Listing 4.10.) That instruction is necessary because a call to a closure call will ensure the environment is properly setup before calling the function, while an open call uses an already setup environment for the call. If the function is called inside a loop, the `EEnsure` instruction will slow the execution. To avoid executing the instruction inside a loop, it is moved outside of the loop.

Domains

Since Aldor domains are run-time objects, their representation in FOAM is `Word`, which is the representation for any reference object. But for the actual domain, a special representation was needed for the domains. Therefore, a new structure was

Listing 4.11: C language representation of Aldor domains

```
1  struct dom {
2          String name;
3          Val    vName;
4          Val    vHash;
5          Clos   al0;
6          Clos   al1;
7          Rec    expNames;
8          Rec    expTypes;
9          Rec    expCloses;
10         Syme   syme;
11 };
```

created that represents the Aldor domains, namely `Dom` (see Listing 4.11.)

The fields `vName` and `vHash` are the name and hash of domain as constructed by the run-time functions. The field `name` is just a convenience function for debugging purposes.

The domain constructor functions are stored in the fields `al0` and `al1`. The `addLevel0` function only sets the hash of the domain, the name and the closure for lazy initialization `addLevel1`. The second function `addLevel1` fills in all the information for the domain.

Next three fields are the exports of the domain. Each export has an entry in each list. Here the representation is of type `Rec` because that is the corresponding representation for `Array`. The `Array` data structure stores the size of the array. `expNames` contains the hashes of the names, `expTypes` contains the hashes of the types, and `expCloses` contains the closures of the exports.

The last field is the meaning of the domain. This is used for imported domains, for which there is no implementation available in the current compilation unit. To avoid, processing all the compilation units from the libraries, the already constructed

information is retrieved from `syme`. Once the domain is cloned, the local copy is analyzed just like any local domain.

When a new domain is constructed, the domain constructor functions `addLevel0` and `addLevel1` are evaluated and domain information is filled in.

The only function of the domain is to optimize the specialized the domain. This is perfomed by `twrDomSpecialize2` function, which calls `twrClosOptimize` on all the exports of the domain. The function `twrClosOptimize` is not an interpreter, it just scans the regular domain exports and optimizes them by replacing closed calls with open calls for all calls that have been made constants by the domain specialization.

Domtors

Functions that produce types as values are sometimes called functors. In tower type optimization implementation, the names we have used is *domtors*. In this thesis the two names will be used interchangeably and they mean the same thing.

Another type identified as special was `Dtor`, the representation for domtors (functors), functions that create Aldor domains. At the FOAM level, they are called *getter* functions, and are treated differently by the optimizer. This kind of objects are the main focus of our optimization.

The structure that offers support for this type is presented in Listing 4.12. Domtors are closures that create parametric domains based on the arguments provided to them. The domtor stores the parameters used to create a type in a special environment which is used by the resulted Aldor domain. After storing the parameters of the domain, it checks a runtime cache for an already created instance. If an instance is found, the cached values is used, otherwise it creates a new domain by using the runtime function call `domainMake`. For a more detailed description of parametric domains see section 2.6.2.

Listing 4.12: C language representation of Domtors.

```
struct dtor {
  Clos clos;
  Bool isImp;
  Dom  dom;
  Syme syme;
};
```

Listing 4.13: Unspecialized domtor

```
(Def (Lex 0 28 dom)
  (CCall Word
    (Glo 4 polyg_Poly_300556761)
    (Glo 5 polyg_Complex_754210471)))
```

The field `clos` is the closure corresponding to the domtor. If the domtor is not local, there is no closure, the field `clos` has the value `NULL`, the field `isImp` has the value `true`, and the field `syme` contains the compiler provided meaning for the domtor.

When a specialization is performed, the code of the domtor from the library is copied locally. After the copy, a new domtor object is created for the local copy of the domtor. The new domtor points to the new closure. The local copy is performed by the extension of the Aldor optimizer.

After the cloning, the cloned domain is specialized by calling `twrDomSpecialize2`.

In the final step, the original call is replaced by a call to the specialized domtor. An example of a call to a generic domtor is shown in Listing 4.13.

After the cloning procedure, the call is changed to use the newly cloned domain as seen in Listing 4.14. In this case a closure is constructed locally with the environment from the original domtor, and with a new implementation.

Listing 4.14: Specialized domtor

```
(Def (Lex 0 28 dom)
  (CCall Word
    (Clos (CEnv (Glo 4 polyg_Poly_300556761)) (Const 27 |Poly(Complex)|))
    (Glo 5 polyg_Complex_754210471)))
```

Listing 4.15: `C` language representation of Unit.

```
struct unit {
    Foam        unit;    /* the original foam */
    FoamBox     globals; /* globals */
    Arr         glo;     /* globals values*/
    FoamBoxList formats; /* all formats */
    ClosList    clos;    /* Assoc list of inlined constants */
    FoamList    exts;    /* definitions of externals */
    DtorList    dtors;
};
```

Units

Another data structure is `Unit` (see Listing 4.15), which corresponds to the entire FOAM `Unit`. It is meant to represent the whole compilation unit and to also provide an interface between the optimization and the FOAM.

At the initialization stage, the FOAM unit is analyzed and broken down into smaller blocks. The original unit is saved in `unit`. The global variables are saved into the fields `globals` and `glo`, the first one contains the declarations and the second one the values.

All declarations are stored in the field `formats` and the list of programs in the `clos` field. These closures cannot be properly evaluated because they contain a `NULL` environment. `Clos` was used to store the implementation of the local programs without introducing a new data type for the `Prog` FOAM instruction.

The `exts` are just the initialization code for the local global variables. They are restored in the finalization code of the optimization.

The last field `dtors` is a list of known domtors, and their specialization. This list is used to check if a specialization exists already. Finding an existing specialization is done by constructing a new specialization based on some instantiation code, and checking if an equivalent specialization already exists in `dtors`. In case such a specialization is found, the new specialization is dropped and the existing one is used instead. This is implemented in `twrUnitFindSpecialization2`.

The unit manages the values of the global variables, and manipulates FOAM code by adding the code of the functions from the library.

The Code Specialization Algorithm

Algorithm 13 is the top-level code specialization algorithm. The algorithm starts with program index 0 by evaluating each instruction. For each expression that is understood by the evaluator, the corresponding action is taken. The runtime calls supported by the evaluator are presented in Section 4.3.3. For each evaluation a `Val` object is created with either the partially evaluated FOAM code, or a special object for arrays, records, closures, domains and domtors.

Definitions store values. Function calls evaluate the operand and the arguments and based on the operand, some actions are performed. Builtin calls are evaluated. The builtin calls present in program 0 or `addLevel0` and `addLevel1` are used to compute the hash codes for names and types. `If`s and `Goto`s are used to evaluate the code when the code is not in sequence. A list of already jumped labels is stored to avoid jumping to labels that were already processed. The return statement sets the value retured by the function. This is important for domtors whose return value is a domain.

The processing of the domain constructor functions `addLevel0` and `addLevel1` is done when domains are created by calling `domainMake`.

Other regular functions are not processed for domain specialization because domains that are constant are created by the compiler in program 0 or `addLevel1`, domains that use variables are created inside regular functions, but those domains cannot be specialized.

Domtor are cloned and then specialized. The details for cloning and specialization are given in the following sections.

4.3.4 Cloning

While trying to evaluate the special functions, the interpretor looks for domtor (functor) instantiations. Domtor instantiation happens by calling the closure that contains the code of the domtor with some type arguments.

The interpreter evaluates, first, the values of the arguments and then proceeds to clone the functor. The cloning is performed with the help of an extension written for the Aldor inliner. The cloning makes distinction between the code that is local to the compilation unit, and the code that is imported from the library. The case of the library domains is explained in Section 4.3.7.

The cloning procedure is performed by the domtor's function `twrDtorClone`, which makes use of the `twrUnitAddDtorFrLib`.

Cloning domains or functions from the library was not supported by the original Aldor compiler. To solve this problem, we extended the original inliner module to allow cloning of domains and functions. `twrUnitAddDtorFrLib` uses the main entry

Algorithm 13 Top level code specialization algorithm

Input: FOAM tree of the current compilation unit
Output: optimized FOAM tree of the current compilation unit

```
p := prog(0)
for all e ∈ expression(p) do
  jumpedLabels := empty_list
  if tag(e) is Def or Set then
    v1 := get_reference(lhs)
    v2 := evaluate(rhs)
    store(v1, v2)
  else if tag(e) is CCall then
    evaluate(operand(e))
    for all a ∈ args(e) do
      evaluate(a)
    end for
    if op is runtime call then
      evaluate(e)
    else if op is dtor then
      nd := clone dtor(op, args(e))
      specialize exports of nd
    else
      ignore call
    end if
  else if tag(e) is Return then
    set the return value
  else if tag(e) is If then
    evaluate condition and label
    if condition true then
      jump to label {also adds l to jumped_list}
    end if
  else if tag(e) is Goto then
    l := evaluate label(e)
    if l ∉ jumped_labels then
      jump(l) {also adds l to jumped_list}
    end if
  else if tag(e) is BCall then
    evaluate(e)
  else
    ignore expression
  end if
end for
```

point to the inliner extension, namely `inlCloneProg`. Algorithm 14 shows the steps taken to clone the whole domain.

Algorithm 14 Domain code cloning.

```
make a copy of the program
if the program is not local to the unit then
   copy the declarations of the non local environments to the local declaration
   flag declaration as non local
   adjust DEnv to local declaration
end if
for all line of code in the copied program do
   if the code creates a closure then
      clone the code of the closure
      fix lexical references to the local copies
   end if
end for
```

After the cloning procedure is complete, the new domtor is specialized. Then the call to the old implementation is updated to call the newly created clone.

Preliminary results presented in [22] showed that most of the speedup is obtained by specializing the innermost domains. For example, let us suppose that we have a deeply nested type `Dom4(Dom3(Dom2(Dom1)))`, most of the speedup will be obtained by specializing `Dom2(Dom1)`. On the other hand, specializing `Dom4(Dom3(X))` will not produce a significant speedup. Due to these results, the specialization is done in pairs of two starting from the innermost to the outermost domain.

4.3.5 Specializing the Functions in the Cloned Domtor

After the clone process is complete the resulted domtor is specialized. Domains are not specialized in any way since they do not contain any type related variables.

The specialization consists of updating all the exports to perform open calls instead of closed calls. This information will help the standard Aldor compiler inliner

to find the code of the target function and to inline it, if it chooses to do so. The final inline decision is still left to the regular inliner to prevent the code explosion.

This specialization is much better than one performed by the unmodified compiler because it allows optimizations to be performed even on inner parts of the parametric domains, which it was not possible before.

The domain constructors are not modified because they construct the domain environment. The current transformation of a closed call to an open call happens according to Algorithm 15.

The actual program is more complicated because there can be more closure calls in the same statement, which means the `OCall` can replace only an expression of a statemnt. The `EEnsure` statement is a statement and cannot be placed inside an expression evaluation, so it must be inserted before the current instruction.

Algorithm 15 Domain exports specialization.

Input: f: FOAM code, c: CCall to be specialized
Output: f: updated FOAM code

```
t := type(c)
e := (CEnv c)
for all a ∈ arguments(c) do
  na(i) := a
end for
f(i) := (OCall t e na(0) ... na(n-1))
f(i-1) := (EEnsure e)
```

The insertion of the environment ensure statement (`EEnsure`) is necessary because open calls do not make sure the environment is created at the first call. The lack of these statements leads to hard to track errors when trying to run the generated code.

The introduction of environment ensure statements had an unexpected downside: bad performance. Even though they are simple checks, they cannot be optimized away and if they happen inside loops, they can be pretty expensive.

The compiler is able to move the loop invariants outside loops, but it does not move the environments ensure because they have side effects. So for our particular case, the environments ensure statements were moved as much as possible to the beginning of the functions. This would affect the laziness of the evaluation for the environments creation, but this does not affect the output of the program.

The specialized types will preserve the signature of all the exported symbols, so they can be used instead of the original call without affecting the caller's code.

4.3.6 Avoiding Multiple Copies of the Same Specialization

It is not uncommon to encounter several equivalent instantiations. While ignoring this problem would not affect the performance of the resulted program, the performance of the compiler will be drastically reduced.

The tower types optimization tries to find if an equivalent specialization has already been constructed. If so, that specialization is used instead of needlessly constructing a new one.

The equivalence between specialization is translated into equivalence between function calls. To test for equivalence between two function calls, one has to make sure that the original call is performed to the same parametric domain and all the values of the parameters are the same between an existing specialization and the intended one.

4.3.7 Specializing the Libraries

Gathering information about the domains is not easy. It requires interpreting the initialization part of all the units that are referenced in the declaration part. The units that must be initialized before the current one are declared with a `GDecl` in the

`DDecl` part of the current unit. The name of the unit is specified as the identifier of the global declaration and the protocol is set to `Init`.

Another approach to retrieving the domain information is to use the information constructed during the type checking phase. As mentioned before, this is possible for the whole tower type optimization, but code modification which changes type semantics cannot be easily reflected in the existing type information. However, retrieving type information about the code already compiled in the libraries is possible.

In order to explain how the type information is stored in the compiler a short overview of the Aldor compiler data structures is required.

The main structures that deal with symbols are: `Syme` and `TForm`. Each symbol used in the Aldor source code has a corresponding `Syme` after its meaning (syme comes from **sy**mbol **me**aning) was inferred. The `Syme` contains all the information about known symbols. One such piece of information is the type corresponding the symbol. That piece of information is given by the `TForm` (or **t**ype **form**).

Tower type optimization does not optimize the domains that are defined in other compilation units and are referred in the current unit, but it still extracts the name of the domain to be able to check the equivalence of specializations. The initialization of library domains happens by using `rtLazyDomFrInit`. When this happens, the initialization function is evaluated and the name of the domain is extracted. Then the symbol table is used to look up the domain and the whole information is retrieved in the `syme` field of the `dom` structure.

For domtors, the problem is different because they are locally copied and specialized. By not cloning locally the library domtors the resulted optimization will be weaker, because the code in the library will not be optimized. By performing the local cloning, the compilation time increases, but only to allow more optimizations to be performed. The run-time function dealing with library domtors is `rtDelayedInit`.

Listing 4.16: Example showing a domain that is constructed based on run-time values.

```
1  define Cat: Category == with {m: SI -> SI;}
2  define Dom1: Cat == add {m(i:SI): SI == i+1;}
3  define Dom2(p:Cat): Cat == add {m(i:SI): SI == {...}}
4  ParDom(S: Cat): with {method:() -> Cat;} == add {method():Cat == S;}
5  main(): () == {
6          import from SingleInteger;
7          i := 5;
8          if i = 6 then
9                  d == method()$ParDom(Dom1); --$
10         else
11                 d == method()$ParDom(Dom2 Dom2 Dom1); --$
12         print << m(1)$d << newline; --$
13 }
```

The interpreter equivalent of this run-time function will evaluate the code of the initialization function and retrieve the name of the domtor. The name is used to construct a partial information domtor that will be completely processed as soon as it is cloned locally.

4.3.8 Domains Not Known at Compile Time

It is not possible to know the domains at compile time. A simple example when such a domain is not known at compile time is presented in Listing 4.16. In this example, if the condition for the if statement cannot be evaluated by the compiler, the code executed for `m` is not known, so it will not be optimized.

However, because two *constant* domain constructors were used `ParDom(Dom1)` and `ParDom(Dom2 Dom2 Dom1)`, the tower type optimization can construct two specializations for `ParDom` and optimize the code in `Dom2 Dom2 Dom1`. As a result, the function `m` from the parameter `d` will be optimized. The code cannot be optimized by bringing the code into `main`, and it also cannot be optimized inside `Dom2`.

Listing 4.17: Domain instance is constructed in a regular function.

```
1  ParDom(S: Cat): with {
2          method:() -> Cat;
3          m: SI -> SI;
4  } == add {
5          method():Cat == S;
6          m(i:SI):SI== m(i)$S; --$
7  }
8  main(): () == {
9          import from SingleInteger;
10         i := 6;
11         if i = 5 then
12                 d == ParDom(Dom1);
13         else
14                 d == ParDom(Dom2 Dom2 Dom1);
15         print << m(1)$d << newline; --$
16 }
```

Another example is given in Listing 4.17. Notice in this example that `d` calls directly the domain constructor. In cases like this, the Aldor compiler would call the domtor function inside function `main` instead of the environment setup function (which in this case would be program index 0). Because of this, the tower type optimization is not performed. It is possible to deal with this type of code by analyzing all functions. Domains constructed with parametric values in regular functions cannot be optimized due to the variable value of the parameter.

4.4 Performance Results

4.4.1 Tests on Different Aspects

We want both to make sure that the implementation works correctly and to measure the performance of different usage models. For this reason several simple tests were

Test1	Original	Optimized	Ratio
Execution Time (ms)	10484	5234	2.0
Code Size (bytes)	22345	25652	1.2
Compile Time (ms)	703	766	1.1

Table 4.1: The results for Test1.

developed to check the implementation of the automatic code specialization. The tests also perform some lengthy operations to be able to use them as performance measurements as well.

A brief description of each test will follow.

Test1

Test1 is a simple test to check the correctness of the tower type optimization. The test creates two domain `A` and `D`, both implementing `methodA`. A parametric domain `B` uses `A` and `D`. This example selects a method `methodB` from `B` whose signature varies only in type. This is used by the speculative mechanism of the optimization, to select a method when only the name is known and there is no overloading. The complete code is presented in A.3.

The results obtained for this case are presented in Table 4.1.

Test2

Test2 is very similar to test1, except that it tests if including code from another source will make a difference, and instead of having all the code in a function `main`, it puts all the code in outside any function. For this reason, the original optimization performs very poorly (original optimizer does not inline code into environment setup functions.) Our optimization does not inline either, but the specialized form of the code is much faster.

Test2	Original	Optimized	Ratio
Execution Time (ms)	7140	1125	6.4
Code Size (bytes)	21381	24947	1.2
Compile Time (ms)	547	734	1.3

Table 4.2: The results for Test2.

Test3	Original	Optimized	Ratio
Execution Time (ms)	10375	5234	2.0
Code Size (bytes)	26445	32721	1.2
Compile Time (ms)	657	844	1.3

Table 4.3: The results for Test3.

The results obtained for this case are presented in Table 4.2.

Test3

Test3 follows the structure of test1, but it makes one of the parameters a parametric domain instead of a regular one.

The results obtained for this case are presented in Table 4.3. The executed code is identical to Test1, so the timings are similar, the code size is bigger because of the extra specializations. The compilation time is increased due to code specialization.

Test4

Test4 is different from all previous tests. It tries to deal with deeply nested types. For this reason, four domains are created, from `Dom1` to `Dom4`. `Dom1` in the only non-parametric domain. The specialization is of the form `Dom4(Dom3(Dom2(Dom1)))`. Each domain calls the inner domain in a loop. The function `m` from `Dom4` is not placed in any function, rather it is called from the top level. The complete code is presented in A.4.

The results obtained for this case are presented in Table 4.4. The performance is different because the original optimizer is not able to optimize code at the file level,

Test4	Original	Optimized	Ratio
Execution Time (ms)	2438	47	51.9
Code Size (bytes)	15312	18423	1.2
Compile Time (ms)	500	641	1.3

Table 4.4: The results for Test4.

Test5	Original	Optimized	Ratio
Execution Time (ms)	8375	8375	1.0
Code Size (bytes)	16785	19897	1.2
Compile Time (ms)	485	641	1.3

Table 4.5: The results for Test5.

while the tower type optimizer produces an optimized domain at the file level which does not need to be optimized at the file level anymore.

Test5

Test5 uses a structure similar to Test4 (see A.4), but the type tower is constructed inside a function. The results obtained for this case are presented in Table 4.5. Both cases here show identical result since the original optimizer was able to specialize the code as well as our tower optimization. The relocation of the code inside a function allowed the original optimizer to optimize the code as well as tower optimizer.

Test6

Test6 creates a parametric domain `Dom3(p)`, which uses internally type `Dom25(p)`. It then calls `m` from `Dom25(p)`. In the main program, `p` is instantiated with `Dom2(Dom1)`. The complete code is presented in A.5.

The results obtained for this case are presented in Table 4.6. The difference in performance is due to the fact that the original optimizer can only inline from `Dom3`, but nothing more, while the tower optimizer inlines everything. The parametric type `Dom25(p)` is store into a new type `Rep`. The optimizer is not able to compute the value

Test6	Original	Optimized	Ratio
Execution Time (ms)	26375	63	418.7
Code Size (bytes)	14092	17326	1.2
Compile Time (ms)	484	625	1.3

Table 4.6: The results for Test6.

Test7	Original	Optimized	Ratio
Execution Time (ms)	5109	1844	2.8
Code Size (bytes)	21904	27997	1.3
Compile Time (ms)	594	844	1.4

Table 4.7: The results for Test7.

of type `Rep` and it cannot inline from `p`. For this reason, the resulting code is a chain of function calls from inner domains that are expensive. The tower type optimization produces specialized forms of type for each pair of type instantiations which lead to a fully optimized `Dom3` (no function calls) which is used from the main function. A simple experiment that replaces the type `Rep` with a macro, immediately resolves the problem and the original optimizer is able to compute the domain instantiation at the macro expansion place.

Test7

Test7 is a more complicated version of Test6, where `Dom6(d)` is parametric and the internal type used is `Dom5(Dom4(d)` and the instantiation of `d` is `Dom2(Dom1)`. The complete code is presented in A.6.

The results obtained for this case are presented in Table 4.7. The result obtained for Test7 cannot be improved by replacing a type definition with a macro constructor.

Test8

Test8 is a more complicated version of Test7, where `Dom6(d)` is parametric and the internal type used is `Dom5(Dom4(d(Dom2(Dom1))))` and the instantiation of `d` is `Dom3`.

Test8	Original	Optimized	Ratio
Execution Time (ms)	5329	1891	2.8
Code Size (bytes)	23328	29573	1.3
Compile Time (ms)	594	890	1.5

Table 4.8: The results for Test8.

Polyg	Original	Optimized	Ratio
Execution Time (ms)	16468	8329	2.0
Code Size (bytes)	34671	45585	1.3
Compile Time (ms)	984	1110	1.1

Table 4.9: The results for Polyg.

This time `d` is a functor, not a domain. The complete code is presented in A.7.

The results obtained for this case are presented in Table 4.8.

Polyg

Polyg is a test that performs polynomial multiplication on polynomials with complex coefficients.

The results obtained for this case are presented in Table 4.9. The difference comes from the fact that the original optimizer does not inline the code for the complex multiplication in `main`.

4.4.2 Testing with SciGMark

The next step in testing the performance of the tower type optimization is to run the optimization on the SciGMark benchmark suite. The results for each kernel are reported separately.

FFT	Original	Optimized	Ratio
Operations (MFlops)	3	5	1.7
Code Size (bytes)	148359	191146	1.3
Compile Time (ms)	5130	8531	1.7

Table 4.10: The results for fast Fourier transform.

SOR	Original	Optimized	Ratio
Operations (MFlops)	34	37	1.1
Code Size (bytes)	75229	83807	1.1
Compile Time (ms)	2323	3321	1.4

Table 4.11: The results for successive over-relaxation.

Fast Fourier Transform

In Table 4.10, one can see that an increase of performance of 67% is obtained, with an increase of 28% in code size. The generic version of FFT algorithm uses two levels of tower types as it can be seen from the following instantiation used: `GenFFT (DoubleRing, MyComplex (DoubleRing))`.

Successive Over Relaxation

In Table 4.11, one can see that a increase of performance is small and also the code increase is not significant. The generic version of successive over relaxation algorithm uses only `DoubleRing` as argument and the code is completely inlined by the original optimizer.

Monte Carlo Algorithm

In Table 4.12, one can see that a increase of performance is significant while the code increase is not significant. The generic version of Monte Carlo algorithm does not use any parameter, however, it uses random numbers that use an array of integers.

MC	Original	Optimized	Ratio
Operations (MFlops)	83	175	2.1
Code Size (bytes)	66669	68041	1.0
Compile Time (ms)	1949	2254	1.2

Table 4.12: The results for Monte Carlo.

MM	Original	Optimized	Ratio
Operations (MFlops)	10	10	1.0
Code Size (bytes)	78746	84909	1.1
Compile Time (ms)	2486	3256	1.3

Table 4.13: The results for matrix multiplication.

The code is not completely inlined by the original optimizer, as can be seen from the difference in performance.

Matrix Multiplication for Sparse Matrices

In Table 4.13, one can see that there is no significant increase in performance and the code is also only slightly increased. The generic version of matrix multiplication algorithm uses `DoubleRing` as parameter. The code is not completely inlined by the original optimizer, and the code is fully inlined by tower type optimization, yet the difference in performance is not significant.

LU Factorization for Dense Matrices

In Table 4.14, one can see that there is no significant increase in performance. The generic version of lu factorization algorithm uses `DoubleRing` as parameter. The code is completely inlined by both the original optimizer, and the tower type optimization, and the difference in performance is small.

LU	Original	Optimized	Ratio
Operations (MFlops)	13	14	1.1
Code Size (bytes)	87114	98245	1.1
Compile Time (ms)	2979	4503	1.5

Table 4.14: The results for LU factorization.

PM	Original	Optimized	Ratio
Operations (MFlops)	14	18	1.3
Code Size (bytes)	108048	126050	1.2
Compile Time (ms)	3727	5022	1.4

Table 4.15: The results for polynomial multiplication.

Polynomial Multiplication for Dense Polynomials

In Table 4.15, one can see that there is a significant increase in performance without too much code expansion. The generic version of polynomial multiplication algorithm uses `SmallPrimeField` as a parameter.

Matrix Inversion Using Quad-tree Matrices

In Table 4.16, one can see that there is a significant increase in performance and also in the code size. The reason for code increase is the recursive nature of the matrix representation; a new domain is created for each doubling of matrix size. The generic version of polynomial multiplication algorithm uses itself of `Matrix2x2` as parameter.

RM	Original	Optimized	Ratio
Operations (MFlops)	5	11	2.3
Code Size (bytes)	118144	337654	2.7
Compile Time (ms)	4229	18408	4.4

Table 4.16: The results for matrix inversion using quad-tree representation.

4.5 Applicability to Other Programming Languages

In this section we discuss the code specialization optimization in the context of other programming languages such as: C++, C#, and Java. For these programming languages no modification to the compilers has been performed. All the results are produced by hand specialized code.

4.5.1 C++

This optimization does not apply to C++, because it already uses heterogeneous approach to implement templates. In C++, the generic code is only syntactically checked when a library is compiled. When an instantiation is created, the instantiation value replaces the formal type parameter similar to the macro expansion procedure and the code is type checked and compiled. Most compilers will not detect equivalent instantiations and will produce unnecessary code.

Due to the heterogeneous nature of the implementation, the code can be optimized in the specialized form. This implementation has also drawbacks because the code of the template has to be available at compile time, this means that libraries must supply the generic code.

The C++ programming language also offers the possibility to write specializations for templates (called "traits"). The specialized templates take precedence over the normally specialized code. As a consequence of this the templates mechanism provided by C++ is very powerful leading to the ability to write code that is evaluated by the compiler at compile time, moving the execution from run-time to compile time. This has been presented in template meta-programming [66].

An experiment tried on SciGMark shows that simple specialization of the templates in C++ does not produce any improvement (see Table 4.17.)

SciGMark (C++)	Generic	Specialized	Ratio
Fast Fourier Transform	141	141	1.0
Successive Over Relaxation	147	145	1.0
Monte Carlo	45	45	1.0
Sparse Matrix Multiplication	153	152	1.0
LU Factorization	182	182	1.0
Polynomial Multiplication	92	92	1.0
Recursive Matrix Inversion	28	29	1.0
Composite	113	112	1.0

Table 4.17: Comparison between generic and specialized code in C++. The results are reported in MFlops.

SciGMark (C#)	Generic	Specialized	Ratio
Fast Fourier Transform	37	44	1.2
Successive Over Relaxation	65	67	1.0
Monte Carlo	13	13	1.0
Sparse Matrix Multiplication	67	67	1.0
LU Factorization	59	60	1.0
Polynomial Multiplication	27	30	1.1
Recursive Matrix Inversion	47	44	0.9
Composite	45	47	1.0

Table 4.18: Comparison between generic and specialized code in C#. The results are reported in MFlops.

4.5.2 C#

C# uses a hybrid approach to generics implementation. In C#, the reference type objects (class instantiations) are implemented using a homogeneous approach. However, in C# the type information is not erased like in Java.

The generic code is supported by the CLR virtual machine, which can perform optimizations that make use of the type information. For the value objects (basic types, and stack allocated objects), C# uses a heterogeneous approach similar to C++ templates.

As seen in Table 4.18, there is no difference between the generic code and the specialized code version.

4.5.3 Java

Until version 5, Java did not offer any support the generic code. The preferred type of polymorphism for Java was, like in any object-oriented programming language, the sub-classing polymorphism. The advantages of a stricter type checking made possible by parametric polymorphism, recent versions of Java introduced "generics", with support for bounded parametric polymorphism.

The erasure technique has some drawbacks that make generic algorithms more complicated than necessary. One example of such complication is the inability to construct an instance of an object of the type parameter.

This homogeneous approach makes it interesting for our optimization. In Java, calls performed through interfaces are more expensive than direct calls to methods from classes, so will be helpful to specialize the code and see how this code specialization optimization helps the Java virtual machine. The code is hand specialized using the same algorithm as the one presented for code specialization. The results, in MFlops, of specialized and generic versions of the kernels from SciGMark benchmark can be seen in Table 4.19. It is interesting to note also that the tests with more generic code (fast Fourier transform, polynomial multiplication, and recursive matrix inversion) show a higher difference between the generic and the specialized versions.

The problem of optimizing the generic code in Java is a little more complex than in the other languages. Java compilers typically do not perform any optimization at compile time. They only produce bytecode which is later optimized at runtime by the just-in-time compiler. This approach puts the pressure on code specialization to run quickly. Another problem is the erasure technique used in Java, which completely removes the type information for generic types. We believe that both these problems can be solved. To preserve type information the Java annotation mechanism can be

SciGMark (Java)	Generic	Specialized	Ratio
Fast Fourier Transform	38	46	1.2
Successive Over Relaxation	48	46	1.0
Monte Carlo	47	55	1.2
Sparse Matrix Multiplication	70	79	1.1
LU Factorization	51	52	1.0
Polynomial Multiplication	55	89	1.6
Recursive Matrix Inversion	47	58	1.3
Composite	51	61	1.2

Table 4.19: Comparison between generic and specialized code in Java. The results are reported in MFlops.

used. Annotations can be inserted with in the code for each use of `new` from a parameterized type. When `new` operation is encountered a new specialization of the class can be generated if needed by the just-in-time compiler. The just-in-time compiler does not need to analyze the code to reconstruct the type information because classes are preserved by the virtual machine. This can improve the code specialization efficiency.

4.6 Related Work

Procedure cloning was used by Cooper [17]. Procedure cloning analyzes the code and produces a new clone for different values of the arguments of the functions. Finally, equivalent clones are merged together. The advantage of this method over standard interprocedural analysis and optimization is that the cloned procedures have some of the arguments as constants which permits the compiler to better optimize the code. Our optimization applies the principle of cloning to user defined domains. It does not require a complete interprocedural analysis to detect possible values. Tower optimization works with type values and the type is reconstructed using run-time type information.

Similar to procedure cloning, Chambers used the "customization" technique [14] for SELF, a dynamically typed object-oriented programming language, to create specialized versions procedures. The compiler predicts some types and creates specialized versions of the method for each predicted type. At run-time the proper specialization is selected based on the run-time type information. Customization guesses some values and compiles specialized version that might not be used, the specialization is selected at run-time. Tower type optimization reconstructs the run-time type information at compile time and specializes only the statically known types.

Type directed procedure cloning was studied by Plevyak in case of dynamic dispatch used in the Java programming language [48]. They create clones and use a selection algorithm similar to customization to select at run-time between the clones.

There has been considerable research done for type specialization in statically typed functional programming languages like SML, Caml and Haskell. In these programming languages, the data is represented using a certain size, usually the word size, and if the data does not fit in this size, the data is boxed. Operations on boxed data is more expensive and different optimizations were implemented for Standard ML compiler [47, 58, 63].

General program specialization through partial evaluation has been studied for object-oriented languages, such as Java, by Schultz in [56, 57]. He uses two level languages to formalize the partial evaluation and uses a simplified Java-like language called Extended Featherweight Java. The partial evaluator specializes only base type values, while object instantiations are not modified. Tower type optimization tries to optimize the user defined parametric types, not the base types.

4.7 Conclusions and Future Research

This chapter presented an optimization that it is easy to implement and the analysis can be performed fast. The speedup depends greatly on the code being optimized. A small part of the speedup comes from the elimination of the function call. Most of the speedup comes from the ability to locally expand the code in the caller function and perform more powerful local optimization, which in some cases can be impressive. We have seen here that improvements of up to several times faster are possible.

The cost paid for this optimization is an increased code size and compilation time. The increases are big only for many instantiations of the same type. For static compilers, the increase in compilation should not be a problem.

This optimization is not limited to the Aldor compiler. Any language that uses a homogeneous approach to implement generic types could benefit from this optimization as seen from the Java tests.

This optimization is also necessary to be able to perform other optimizations related to user defined types. The data representation specialization needs the code to be specialized first so it could change the code to use the new data representation.

Chapter 5

Automatic Domain Data Specialization

5.1 Introduction

With the help of SciGMark, we noticed that a significant speedup was obtained by modifying the data representation. The change consisted of replacing the `DoubleRing` wrapper with the basic type `double`. This modification is called unboxing and it has been studied for functional languages.

Boxing and unboxing was necessary in functional languages to allow parameter passing for data types whose representation size is larger than machine word. An example of boxed data type is the double precision floating point data type which must be boxed. However, boxing is only limited to replacing the basic types with corresponding reference objects.

A more general optimization that deals with inlining objects was presented by Dolby and Chien in [21]. The idea is to fuse two objects together when there is a one-to-one relationship between them.

Another optimization that can be performed for objects that do not escape the scope of a function is to allocate such object on the stack instead of on the heap [16]. This optimization is already implemented in the Aldor compiler as the "environment merge" optimization. The environment merge optimization replaces all of the fields of a record with local variables. The advantage of this optimization is that it completely eliminates the cost of object allocation for objects that are only used inside one function.

This chapter analyzes an extension of the environment merge optimization. We show that the representation of the whole domain can be merged in cases of parametric polymorphism. A high level introduction to this optimization was presented in [23].

5.2 Automatic Domain Data Specialization

5.2.1 Example

To illustrate how data representation specialization works, we shall use the polynomial multiplication example that was presented in chapter 4.

The implementation of the `Ring` category and the `Complex` domain can be seen in Figure 5.1. The `Ring` category declares a type that has operations such as addition, multiplication, the ability to construct a neutral element and the ability to be displayed on screen. The `Complex` domain is an example of a domain that implements a `Ring` type by providing implementations for all of the operations declared in `Ring`.

In Figure 5.1, the `Complex` domain is not a parametric domain. The data representation for the real and imaginary parts is the domain `MachineInteger`. This is only due to the fact that we shall perform the code and data specialization by hand, without any help from an automatic system. In this case, the type of `Complex` is a

Listing 5.1: The `Ring` type.

```
Ring: Category == with {
    +: (%, %) -> %;
    *: (%, %) -> %;
    0: %;
    <<: (TextWriter, %) -> TextWriter;
}

MI ==> MachineInteger;

Complex: Ring with {
    complex: (MI, MI) -> %;
    real: % -> MI;
    imag: % -> MI;
} == add {
    Rep == Record(re: MI, im: MI);
    import from Rep;
    complex(r: MI, i: MI): % == per [r, i];
    real(t: %): MI == rep(t).re;
    imag(t: %): MI == rep(t).im;
    (a: %) + (b: %): % ==
        complex(rep(a).re+rep(b).re,
                rep(a).im+rep(b).im);
    (a: %) * (b: %): % == {
        ra := rep.a; rb := rep.b;
        r := ra.re*rb.re-ra.im*rb.im;
        i := ra.re*rb.im+ra.im*rb.re;
        complex(r, i);
    }
    0: % == complex(0, 0);
    (w: TextWriter) << (t: %): TextWriter == {
        w << rep.t.re << "+i*" << rep.t.im;
    }
}
```

Listing 5.2: Implementation of a generic polynomial of type `Ring`.

```
Polynomial(C: Ring): Ring with {
    poly: Array(C) -> %;
} == add {
    Rep == Array(C);
    import from Rep; import from C;
    poly(size: MachineInteger): % == {
        res: Array(C) := new(size);
        i:=0@MI;
        while i < size repeat {
            res.i := 0@C; i := i + 1;
        }
        per res;
    }
    (a: %) + (b: %): % == {
        c: Array(C) := new(#rep(a));
        i := 0@MI;
        while i < #rep(a) repeat {
            c.i := rep(a).i + rep(b).i;
            i := i + 1;
        }
        per c;
    }
    0: % == {poly(1);}
    ...
}
```

`Ring` extended with some extra operations such as `complex`, `real` and `imag`. This means that `Complex` is a *sub-type* of `Ring`. Percent (`%`) is the representation of *this* type, similar to `this` in object-oriented programming languages. The two macros `per` and `rep` are used to convert between the `%` type (the external type) and the `Rep` type (the internal representation). The rest of the code is self explanatory, and the implementation of the operations is a straight forward implementation of the definition of addition and multiplication for complex numbers.

The Aldor code for the corresponding `Polynomial` domain can be seen in Listing 5.2.

Listing 5.3: Polynomial multiplication.

```
1 import from Polynomial(Complex);
2 sz == 10000;
3 a := poly(sz);
4 b := poly(sz);
5 d := a * b;
```

The `Polynomial` domain is also a sub-type of `Ring`. It is formed by augmenting the `Ring` type with an operation to construct the object. This time the domain is parametric, requiring the type of the coefficients of the polynomial. The internal representation, given by `Rep`, is `Array(C)`. Here, `Array` is another parametric domain that implements a generic collection similar to arrays in other programming languages. In the interest of brevity, we only provided the implementation of the addition between two polynomials here. The multiplication and the display functions are straightforward to implement.

The Aldor programming language uses type inference to determine the type information when it is not provided. However, in some cases the type of the expression is ambiguous and the disambiguating operator must be used. For example, we specified that `i` gets the value `0` from `MachineInteger` by using `0@MI`. Possible candidates were `0` from `C` or from `Polynomial`. The rest of the code is easy to understand.

An example usage of the polynomial operations can be seen in Listing 5.3. The program creates two polynomials with all coefficients 0, of degree 9999 and then it multiplies them. This example is not interesting from the mathematical point of view, but it merely tries perform numerous function calls inside the polynomial object and the most expensive operation provided is multiplication.

The polynomial in Listing 5.2 uses a dense representation, which means it is an array of size $d + 1$ where d is the degree of the polynomial. Each entry in the array

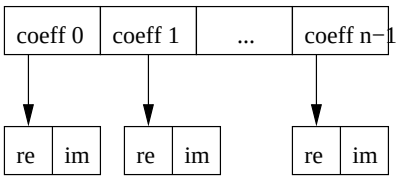

Figure 5.1: Data representation for polynomial with complex coefficients (before specialization)

is the coefficient corresponding to each monomial. If the polynomial is a generic type accepting different algebraic types for its coefficients, complex numbers can be used as polynomial coefficients. Furthermore, the complex type could be implemented as a generic type accepting any values that can be added and multiplied. The real and imaginary part of the complex number could be implemented with integer numbers or floating point numbers. A domain such as this would be created in Aldor using `Polynomial(Complex(Integer))` and the data representation is an array similar to that shown in Figure 5.1.

One should note that for each operation on complex numbers, a new object is allocated on the heap and heap allocation is an expensive operation. The code specialization optimization presented in chapter 4 replaces the generic `Polynomial` domain with the domain presented in Listings 5.4 and 5.5.

By specializing the code, and unfolding the functions from `Complex` into `Polynomial` it was possible to remove the temporary objects created during operations such as

```
c(i+j) := c(i+j) + rep(a).i * rep(b).i.
```

However, the objects that are stored in the array collection, namely `c(i)`, must still be allocated because the array is an array of references to real complex objects.

A specialized version of the array must be constructed in order to further improve the data specialization. The specialized version should allocate all of the necessary

Listing 5.4: Code specialized polynomial.

```
Polynomial__Complex: Ring with {
    poly: MachineInteger  -> %;
    poly: Array(C) -> %;
} == add {
    Rep == Array__Complex;
    import from Rep;
    import from Complex;
    poly(size: MachineInteger): % == {
        res: Rep := new(size);
        i:=0@MI;
        while i < size repeat { res.i := 0@C; i := i + 1; }
        per res;
    }
    poly(a: Array(C)): % == {
        res: Rep := new(#a);
        i := 0@MI;
        while i < #a repeat { res.i := a.i; i := i + 1; }
        per res;
    }
```

space at once for all the data. The FOAM intermediate language offers a special data type to deal with arrays of objects in an efficient manner. The data type is `TrailingArray`. The advantage of trailing arrays is that all the elements are allocated at once together with the array.

According to the Aldor User Guide [74], trailing arrays are an aggregate data type consisting of two parts: a header and an array of objects. The type is represented as a single block of memory.

```
TrailingArray(size:Integer, (re:  Integer, im:  Integer))
```

The representation of the polynomial that results from specializing it for complex numbers with integer coefficients can be seen in Figure 5.2. This new representation has much better memory locality, eliminates some indirections and, most importantly, eliminates the need to allocate objects on the heap one by one.

Listing 5.5: Code specialized polynomial (Continued).

```
    (a: %) + (b: %): % == {
        c: Rep := new(#rep(a));
        i := 0@MI;
        while i < #rep(a) repeat {
            c.i := rep(a).i + rep(b).i;
            i := i + 1;
        }

        per c;
    }
    (a: %) * (b: %): % == {
        i := 0@MI;
        c: Rep := new(#rep(a)+#rep(b)-1);
        while i < #c repeat { c.i := 0@C; i := i + 1; }
        i := 0@MI;
        while i < #rep(a) repeat {
            j := 0@MI;
            while j < #rep(b) repeat {
                c(i+j) := c(i+j) + rep(a).i * rep(b).j;
                j := j + 1;
            }
            i := i + 1;
        }

        per c;
    }
    0: % == {poly(1);}
    (w: TextWriter) << (t: %): TextWriter == {
        w << "[";
        i := 0@MI;
        while i < #rep(t) repeat {
            w << rep(t).i << " ";
            i := i + 1;
        }
        w << "]";
    }
}
```

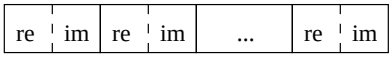

Figure 5.2: Data representation for polynomial with complex coefficients (after specialization)

For the specialized version, the generic type `C` will be replaced with `Complex`, according to the instantiation given on the first line of Listing 5.3. The specialized implementation can be seen in Listings 5.6 and 5.7. One can note that the parametric domain `Polynomial` has become `Polynomial__Complex` and the internal representation has been changed to `TrailingArray`. In addition, `Cross` is used instead of `Record` for complex numbers. The specialized case presented in Listing 5.6, performs both specializations: code and data representation. The code is specialized by creating a new domain `Polynomial__Complex`, copying the operations from `Complex` into `Polynomial__Complex`, and finally inlining the code of complex addition and multiplication into polynomial addition and multiplication. The data representation is changed from `Record` to `TrailingArray`.

`ComplexStack`, the stack based version of `Complex`, can be seen in Listing 5.8. The domain's exports could also be copied into the parent domain, namely `Polynomial`. One may note here that the representation of the domain is no longer necessary. All of the fields of the complex numbers are passed through the parameter list. While this seems like an increase in stack use, if the functions from `ComplexStack` are integrated into their caller, they can actually use the local variables of the caller instead of passing them as parameters. In our case the inlined function will actually access the elements directly from the trailing array.

This example showed us a way to replace the heap allocation of each individual complex object with only one allocation performed at initialization. This approach is much faster than allocating each object individually. Moreover, for each updating

Listing 5.6: Specialized polynomial representation.

```
Polynomial__Complex: Ring with {
    poly: MI  -> %;
    poly: Array(Complex) -> %;
} == add {
    T == Cross(re: MI, im: MI);
    Rep == TrailingArray(MI, (MI, MI));
    import from Rep, T;
    poly(size: MI): % == {
        i:MI := 1;
        res: TrailingArray(s:MI,(re:MI,im:MI)) := [size, size, (0,0)];
        while i <= size repeat {
            res(i, re) := 0;
            res(i, im) := 0;
            i := i + 1;
        }
        per res;
    }
    poly(a: Array(Complex)): % == {
        import from Array Complex;
        import from ComplexStack;
        res: TrailingArray(s:MI,(re:MI,im:MI)) := [#a, #a, (0,0)];
        i:MI := 1;
        while i <= #a repeat {
            res(i, re) := real(a(i-1));
            res(i, im) := imag(a(i-1));
            i := i + 1;
        }
        per res;
    }
    (a: %) + (b: %): % == {
        local ra: TrailingArray(s:MI,(re:MI,im:MI)) := rep(a);
        local rb: TrailingArray(s:MI,(re:MI,im:MI)) := rep(b);
        res: TrailingArray((s:MI),(re:MI,im:MI)) := [ra.s, ra.s, (0,0)];
        i:MI := 1;
        while i <= ra.s repeat {
            (res(i,re),res(i,im)):=(ra(i,re),ra(i,im))+(rb(i,re),rb(i,im));
            i := i + 1;
        }
        return per res;
    }
```

Listing 5.7: Specialized polynomial representation (Continued).

```
    (a: %) * (b: %): % == {
        local ra: TrailingArray(s:MI,(re:MI,im:MI)) := rep(a);
        local rb: TrailingArray(s:MI,(re:MI,im:MI)) := rep(b);
        res: TrailingArray((s:MI),(re:MI,im:MI)) :=
                            [ra.s+rb.s-1, ra.s+rb.s-1, (0,0)];
        i:MI := 1;
        while i <= res.s repeat {
            res(i, re) := 0;
            res(i, im) := 0;
            i := i + 1;
        }

        i := 1;
        while i <= ra.s repeat {
            j:MI := 1;
            while j <= rb.s repeat {
                c(i+j) := c(i+j) + rep(a).i * rep(b).j;
                (res(i, re),res(i,im)) := (res(i, re),res(i,im))
                    + (ra(i,re),ra(i,im)) * (rb(j, re),rb(j,im));
                j := j + 1;
            }
            i := i + 1;
        }
        return per res;
    }
    0: % == poly(1);
    (w: TextWriter) << (t: %): TextWriter == {
        import from C;
        w << "[";
        i:MI := 1;
        local r: TrailingArray(s:MI, (re:MI, im:MI)) := rep(t);
        while i <= r.s repeat {
            w << (r(i, re), r(i,im)) << " ";
            i := i + 1;
        }
        w << "]";
    }

}
```

Listing 5.8: Complex domain using stack based allocation instead of heap.

```
ComplexStack: Ring with {
    complex: (MI, MI) -> %;
    real: (MI,MI) -> MI;
    imag: (MI,MI) -> MI;
} == add {
    T == Cross(MI, MI);
    --Rep == Record(re: MI, im: MI);
    import from Rep;
    complex(r: MI, i: MI): T == (r, i);
    (a: T) + (b: T): T == {
        --These two lines work around a bug in the compiler when
        --a new type is declared as Cross.  Macros can be used
        --but they crash the compiler.
        aa: Cross(MI, MI) := a;
        bb: Cross(MI, MI) := b;
        (ar, ai) := aa;
        (br, bi) := bb;
        complex(ar+br, ai+bi);
    }
    (a: T) * (b: T): T == {
        aa: Cross(MI, MI) := a; bb: Cross(MI, MI) := b;
        (rar, rai) := aa; (rbr, rbi) := bb;
        r := rar*rbr-rai*rbi;
        i := rar*rbi+rai*rbr;
        complex(r, i);
    }
    zero: T == complex(0, 0);
    (w: TextWriter) << (t: T): TextWriter == {
        tt: Cross(MI, MI) := t;
        (r, i) := tt;
        w << r << "+i*" << i;
        w
    }
}
```

of the elements contained in the array a new memory allocation must be performed, which is an unwanted overhead.

5.2.2 Data Representation Change Problems

Data specialization cannot be performed in all the cases. When dealing with data representation modification, it is critical to make sure that all accesses to the original data are handled properly.

The access to the data can be done in two different ways: use of the interface functions (functions exported by the domain) or accessing the data directly through the domain representation.

If only the interface functions are used to manipulate objects of that domain, then data layout changes will be handled by the functions exported by the domain. Even if the domain representation is not exported, it is still possible to construct other data structures equivalent to the representation of the current domain and then use the `pretend` keyword to access the data. In these cases, the behaviour of the program will be different in its unoptimized and optimized forms. Therefore the optimization will not preserve the semantics of the program. In such improbable cases, there are two possible solutions: either do not perform the optimization, or change the code in every location in the program that access the data directly. Another, not very good, option is to reconstruct the data layout when such accesses are detected. Unfortunately, both solutions that were proposed require a complete analysis of the whole program. Therefore, the simple analysis performed for code specialization is not enough and the partial evaluator should be extended to deal with the entire program.

Aliasing happens when two different variables point to the same memory location. In this case, any modification to one of the variables will also modify the value of the

other. Aldor uses pass-by-value semantics for basic data types and pass-by-reference semantics for records and arrays (or pass-by-value of the pointer to the object). This means that aliases are created when simple assignments are used for values of type `Rec` or `Arr`. In these cases, replacing the reference objects by their fields and passing them as parameters will not make the changes visible in the aliased variable. The only way to solve this problem is to detect all aliases and change the code according to the data representation transformation. This analysis is complex and would slow the compiler considerably. As a result, the data specialization optimization will only be applied to objects that are either immutable, meaning that their state does not change, or if the objects are mutable (i.e. they contain functions that alter some of the fields of the domain representation), no mutating function is called on the objects stored in the specialized data type. For immutable objects creating a new instance of the contained objects does not affect the result in any way, since the data is just a copy used for reading.

5.2.3 Domain Interface Preservation

Data specialization optimization is performed at the domain level. The changes are performed in the domain, but the interface of the domain must be preserved. The interface preservation, means that the operations exported by the domain must be the same.

Interface preservation, for the operations that do not contain object belonging to the type parameter, is not a problem. The `%` type does not impose any problems either because it will always be treated by the current domain which can handle the new representation. The only type that should be handled different is the type parameter.

All the operations that use as argument or as return value object of the type pa-

rameter must reconstruct the expected type. The reconstruction consists of allocating a new object and copying the corresponding fields from the specialized representation into the newly allocated object.

To avoid allocation and copying we propose an extension to FOAM. The extension consists of introducing the concept of a value record. In the existing FOAM specification, the records are represented as references. The `C` programming language, offers structures which can contain other structures. It would be useful to have something similar in FOAM instead of only structures containing references to structures. For more details please see section 5.2.4. This solution allows us to return the inner structure corresponding to the type parameter without memory allocation and copy. However, this approach still does not solve the problem of aliased mutable objects. It does not create aliases for type parameter, but aliases already created outside the generic type cannot be detected only from analyzing the current domain.

Another approach to solving the problem of accessing the type parameter in the new representation is to replace the type parameter by another type called `Reference(T)` where `T` is the type parameter. This new type would provide two operations: `value` to extract the object of type `T` and `set` to set the value of the object of type `T`. The type `Reference(T)` has a similar behavior to pointers. The advantage of this approach is that the Aldor compiler could specialize the type `Reference(T)` and its operations such that they reflect the new data layout. The disadvantage is that it still does not solve the aliasing problem for objects that are updated from outside the domain and it requires a whole program change to replace the use of `T` with `Reference(T)`. For different data specializations the operations `value` and `set` must have different implementation to reflect the different layout of `T` in each specialization.

None of the solutions presented here solve the problem of aliasing. They deal with the interface between domains, one with extra allocation and copy, other by changing the intermediate language, and the final one with a simple whole program transformation. We have chosen to implement the first solution since it is the easiest to implement and even if performs extra object allocation, this should not occur very often.

5.2.4 Data Representation Change in Cloned Types

A significant increase of performance can be obtained by replacing heap allocated objects with stack allocated objects. There are difficulties with applying this optimization because of the object aliasing that can occur. However, in some cases (when the objects are immutable) the problem of aliasing does not exist and objects can safely be replaced with local objects on the stack. This is the type of optimization that we performed when we specialized the polynomial multiplication test.

The Aldor compiler already offers an optimization called environment merging which replaces the heap allocated objects with local variables if the objects do not escape the scope of the function. This optimization removes the heap allocation. Our proposed specialization goes a step further by merging not only environments that are local to a function but also merging environments for the whole specialized domain.

Environment Merge Optimization

This optimization replaces heap allocated objects with local variables for those records that do not escape the scope of the function.

An example of this optimization is presented in Listing 5.9. The listing presents an addition between two floating point numbers. In Aldor, all double precision floating

Listing 5.9: Addition between two floating point numbers in Aldor.

```
(Set (Loc 1) (RNew 8))
(Set (Relt 8 (Loc 1) 0) (DFlo 2.5))
(Set (Loc 0) (RNew 8))
(Set (Relt 8 (Loc 1) 0) (DFlo 3.5))
(Set (Loc 2) (RNew 8))
(Set (Relt 8 (Loc 1) 0)
  (BCall DFloPlus
    (RElt 8 (Loc 1) 0)
    (RElt 8 (Loc 0) 0)))
```

point numbers are boxed because the arguments of the domain parameters cannot exceed the size used for the `Word` data type. Any boxing requires construction of a heap allocated object that stores one field of `DFlo` type. In Listing 5.9, three local variables are created that must be allocated on heap. All `RNew` operations are expensive and should be avoided.

Since the floating point values are store in local variables, they cannot be seen from anywhere else in the program except the current function. Therefore, it is possible to replace the fields of the record with index 8 with local scalars. The result can be seen in Listing 5.10, where local variables `Loc 3`, `Loc 4`, and `Loc 5` replace the only field of the record with index 8 for the different instances of variables `Loc 0`, `Loc 1`, and `Loc 2` respectively.

Listing 5.11 presents the final form of the FOAM code for the addition of two double precision floating point numbers after dead code elimination and constant propagation have been performed.

The code presented in Listing 5.11 is much faster than the code from Listing 5.9. Unfortunately, if `(Loc 2)` is replaced by `(Lex 2 0)` the variable escapes the local function and the last memory allocation cannot be removed.

Listing 5.10: Addition between two floating point numbers in Aldor with local expansion of the fields from record 8.

```
(Set (Loc 1) (RNew 8))
(Set (Loc 4) (DFlo 2.5)
(Set (Loc 0) (RNew 8))
(Set (Loc 3) (DFlo 3.5)
(Set (Loc 2) (RNew 8))
(Set (Loc 5)
  (BCall DFloPlus
    (Loc 3)
    (Loc 4)))
```

Listing 5.11: Optimized version of floating point addition in Aldor.

```
(Set (Loc 2)
  (BCall DFloPlus
    (DFlo 2.5)
    (DFlo 3.5)))
```

Domain Representation Optimization

The idea behind this optimization is to incorporate the data structure associated with the inner domain into the data structure of the outer domain. All values produced by the inner domains are copied into the already allocated space of the outer domain. If operations from the inner domain are unfolded into operations from the outer domain, then the emerge optimization will detect that the memory allocation is only used to store temporary values before copying them into memory allocated by the outer domain. As a result, it is not necessary to allocate memory for the inner domain.

The purpose of this optimization is to reduce the number of memory allocations or remove them for objects belonging to the inner domain type. A direct consequence of reducing the number of allocations is reduced stress on the garbage collector. In

addition, overall memory usage is decreased by removing the pointers to the reference objects.

In FOAM, every domain is represented by a lexical environment. The lexical environment contains all of the symbols defined inside the domain. If the domain contains a representation, then the `Rep` keyword is be used to define the representation. According to the Aldor User's Guide `Rep` is the internal/private view of the domain, while `%` represents the public view of the domain. According to these definitions, it should not be possible to modify the representation of the domain from the outside, but as mentioned before, `pretend` can be used to bypass the type checking system.

Domain representation specialization requires changing the format of the internal representation. Similar to environment merging for functions, we propose to replace the representation of the inner domains (which are represented as type `Word` in the `Rep` of the outer domain) with all of the fields of the `Rep` of the inner domain. If any of the fields is of type `Arr`, which is a generic array, it will be replaced by a trailing array, type `TR`, whose trailing part is formed from the fields of the `Rep` of the inner domain.

It should be noted here that a more elegant solution to this problem is to have support at the FOAM level of value-typed records. This can be achieved easily by extending the current FOAM language to support "inlinable" structures. The proposed change is to use one of the unused fields of the `Decl` instruction, namely `symeIndex`, and use it as a flag to specify which values of type `Rec` should be stored inline. This solution simplifies the process of accessing some fields of the outer domain as a representation of the inner domain. Without making changes to the FOAM intermediate language, memory must be allocated first and then fields must be copied one-by-one in the newly allocated space. After modifying the FOAM, the `RElt` instruction can directly return the subtype, with a simple change to the FOAM implementation.

Unoptimized	Optimized
`ANew` type size	`TRNew` fmt size
`AElt` type arr index	`TRElt` fmt tr index field$_{1,\ldots,n}$
`RNew` child	`NOp`
`RElt` child field	`RElt` parent_record offset + field
`RElt` child field	`RElt` parent_trarray offset + field

Table 5.1: Translation between unoptimized to optimized code for statements inside functions.

Similarly, the trailing array element `TRElt` can be changed to return the whole record as the stored type instead of accessing the data field by field. This new inlined record can be implemented in all of the back-ends supported by the Aldor compiler as well as in the interpreter.

In FOAM the instructions which deal with arrays and records are:

- for arrays: `ANew` (to create new arrays) and `AElt` (to access elements of the arrays)
- for records: `RNew` (to create new records) and `RElt` (to access fields of the record)

Transformation Algorithm

The proposed translation scheme is presented in Table 5.1. One can see that the arrays are replaced by their specialized form using trailing arrays, and the fields of the child domain are integrated into the parent domain. This translation must only be applied to the inner domains. The topmost domain must still use a reference type for its representation.

To preserve the interface of the functions, the returned values are reconstructed from the locally stored fields. This reconstruction requires allocation of new objects and copying the data into the new memory location. This method fails when the returned object is used to modify the state of the object stored in the specialized

domain. The solution works only for immutable cases, where in order to change the value of a domain, a new instance is created with the new values. If already created objects are modified instead of creating new ones, this optimization will not improve anything and it is not compatible with mutable objects, so it should not be attempted. Using the modified FOAM language, instead of creating a new instance, a set of fields can be returned as a record using the inlined `Decl` extension.

Algorithm 16 is used to optimize the code. This algorithm is called from the code specialization algorithm which recursively applies the specialization from the innermost to the outermost domain parameters.

Algorithm 16 The data specialization algorithm.

Input: domain object after code specialization
Output: updated domain object and translation table computed

```
tr := construct translation table based on % and Rep of child (Table 5.1)
for all e ∈ exports(dom) do
  locally unfold the code of the export with code from inner domain
  for all i ∈ instructions(e) do
    for all exp ∈ expression(i) do
      if any of the exp ∈ tr then
        if exp is used for writing then
          replace with optimized form
        else
          r := construct an object of the type required by the call
          copy from outer Rep into r
        end if
      end if
    end for
  end for
end for
```

For immutable objects, fields are only written to when newly allocated objects have their initial values set before they are returned.

Algorithm 16 can be integrated with the code specialization algorithm and data

specialization can be performed together with code specialization rather than as a separate phase.

The outer domains and other programs should be checked to ensure that they do not use any expressions with data that points to the domain that was specialized. If such expressions are detected, a translation can be attempted using the translation table computed. If it is not possible to then translate the whole domain, the data representation should be reverted to code specialization only.

Record Specialization

An example of a representation which uses a record is the `Complex` domain:

```
Rep == Record(re:  MachineInteger, im:  MachineInteger)
```

The Aldor compiler generates the FOAM code presented in Listing 5.12 in the domain's initialization function `addLevel1`.

The operations that work with the representation of the domain use a record which is declared in the format declaration section. The corresponding declaration of the `Rep` is given in Listing 5.13. In the record declaration, the fields are stored as values of type `Word`, which is the type corresponding to a domain object and represents a reference to a domain object. From this record declaration alone, it is not possible to extract the actual types used as fields of the array. This is where the domain initialization function proves useful. The type `Record` is treated specially by the Aldor compiler. The code presented in Listing 5.12 shows that the type information of the fields of a record are stored at run-time. This allows the data specialization optimization to retrieve the domain used as the type for the fields of the record.

In the domain initialization function, `addLevel1`, there is no correlation between the record used to store the values and the domain constructor function `Record`

Listing 5.12: FOAM code for domain representation.

```
(Set (Lex 0 2 Rep) (CCall Word (Glo 23 domainMakeDummy)))
(Def (Loc 7) (Lex 0 2 Rep))
(Set (Loc 8) (ANew Word (SInt 2)))
(Set (Loc 9) (RNew 8))
(Set (RElt 8 (Loc 9) 0) (SInt 2))
(Set (RElt 8 (Loc 9) 1) (Loc 8))
(Set (AElt Word (SInt 0) (Loc 8)) (Lex 2 1 MachineInteger))
(Set (AElt Word (SInt 1) (Loc 8)) (Lex 2 1 MachineInteger))
(Def
  (Lex 0 2 Rep)
  (CCall
    Word
    (Clos
      (CEnv (Lex 2 0 Record))
      (Const 33 |Record(MachineInteger,MachineInteger)|))
    (Cast Word (Loc 9))))
(CCall NOp (Glo 24 domainFill!) (Loc 7) (Lex 0 2 Rep))
(Set (Lex 0 2 Rep) (Loc 7))
(Set (AElt SInt (SInt 0) (Loc 0)) (SInt 316169058))
(Set (AElt SInt (SInt 0) (Loc 1)) (SInt 850925108))
(Set (AElt Word (SInt 0) (Loc 2)) (Lex 0 2 Rep))
```

Listing 5.13: The record declaration corresponding to `Rep`.

```
(DDecl
  Records
  (Decl Word "re" -1 4)
  (Decl Word "im" -1 4))
```

Listing 5.14: The `Pair` domain.

```
Pair(d: ComplexType): with {
        pair: (SI, SI) -> %;
        first: %-> d;
        second: %-> d;
} == add {
        Rep == Record(f: d, s: d);
        import from Rep;
        pair(p:SI, q:SI):% == per [complex(p,q),complex(q,p)];
        first(t:%): d == rep(t).f;
        second(t:%): d == rep(t).s;
}
```

Listing 5.15: The record declaration of the representation of `Pair`.

```
(DDecl
  Records
  (Decl Word "f" -1 4)
  (Decl Word "s" -1 4))
```

`(MachineInteger, MachineInteger)`. A special function was created to compute the format of the record used to store the representation of the domain. The function investigates the functions that return the type `%` and finds the FOAM type corresponding to that returned value.

In order to explain how record specialization works, we need to extend the example with a new parametric domain `Pair`. The Aldor code for the `Pair` domain is presented in Listing 5.14. The domain accepts a parameter `d` representing the type of the stored elements. In this case, the type of the parameter `d` is `ComplexType`. The representation of the `Pair` domain is a record with two fields of type `d`. A posible instantiation of `Pair` type would be `Pair(Complex)`. The corresponding representation declaration of the `Pair(Complex)` is presented in Listing 5.15.

Listing 5.16: The *specialized* record declaration of the representation of `Pair`.

```
(DDecl
  Records
  (Decl Word "re" -1 4)
  (Decl Word "im" -1 4)
  (Decl Word "re" -1 4)
  (Decl Word "im" -1 4)))
```

Listing 5.17: The record declaration of the representation of `Pair`.

```
(DDecl
  Records
  (Decl Rec "f" 1 9)
  (Decl Rec "s" 1 9))
```

The specialized representation will replace the declaration presented in Listing 5.15 with the representation presented in Listing 5.16.

Listing 5.17 shows the specialized representation when the extended `Decl` instruction is used. The changes begin with the type of each field from a generic domain reference `Word` to a `Rec` which represents the representation of the domain corresponding to that field. In addition, a flag's value has changed from value -1 to 1 to signal that the record should be inlined. Finally, a format is specified which represents the format of the record to be inlined.

After the new representation is constructed, a mapping scheme from the old type to the new type is constructed as well. The mapping scheme translates a field from the original record into a field and a format in the target format. For example, the translation map for our pair example is presented in Table 5.2. The format corresponding to the representation of the complex domain is format number nine. Using this translation scheme, a record access instruction `(RElt 12 (Loc 0) 0)` will be translated into `(RElt 50 (Loc 0) 0)` and `(RElt 50 (Loc 0) 1)`. Format number

Field Index	Target Field	Target Format
0	0	9
1	2	9

Table 5.2: Mapping scheme for fields of the old format into the new format of `Pair`.

Assignee	Container	Format	Field Index
`(Loc 2)`	`(Loc 0)`	12	0
`(Loc 1)`	`(Loc 0)`	12	1

Table 5.3: Connection between fields of the container and variables containing the record to be inlined.

12 is the original representation of the `Pair` while format number 50 is the new *specialized* representation. The target format is used to know the target record from which the original field was expanded. By convention, if the value of the target format is 0, the field has not been expanded. This is useful because the instruction that stores the inner domain value into the outer should be completely removed, since all its fields have been copied into the expanded fields of the outer domain representation.

Before proceeding with the translation some new information must be gathered from the function that will be changed. A new table is constructed which connects expressions of the inner domain type to the fields of the outer domain fields. The code for the `pair` function is presented in Listing 5.18. This code checks to see which variables are stored in which fields of the specialized domain. For Listing 5.18, local variable `(Loc 2)` is stored in field index `0` of `(Loc 0)` and local variable `(Loc 1)` is stored in field index `1` of `(Loc 0)`. The complete analysis table is presented in Table 5.3.

With all the above information constructed, the exported function of the specialized domain is translated according to the following rules:

- any `RNew` with a format corresponding to the old domain representation will be changed to have the format corresponding to the new representation.

Listing 5.18: The `pair` function.

```
(Def
  (Const 23 pair)
  (Prog
    0
    1
    Word
    4
    1155
    36
    1
    0
    (DDecl Params (Decl Word "p" -1 4) (Decl Word "q" -1 4))
    (DDecl
      Locals
      (Decl Rec "" -1 12)
      (Decl Rec "" -1 9)
      (Decl Rec "" -1 9))
    (DFluid)
    (DEnv 4 13 4 4 4)
    (Seq
      (Set (Loc 0) (RNew 12))
      (Set (Loc 2) (RNew 9))
      (Set (RElt 9 (Loc 2) 0) (Par 0 p))
      (Set (RElt 9 (Loc 2) 1) (Par 1 q))
      (Set (RElt 12 (Loc 0) 0) (Loc 2))
      (Set (Loc 1) (RNew 9))
      (Set (RElt 9 (Loc 1) 0) (Par 1 q))
      (Set (RElt 9 (Loc 1) 1) (Par 0 p))
      (Set (RElt 12 (Loc 0) 1) (Loc 1))
      (Return (Loc 0)))))
```

- any `RElt`, with a format corresponding to the inner domain representation, will be changed to have a format of the new container, the expression of the new container and the field computed as an offset in the new container. This translation uses information from Tables 5.2 and 5.3.

- any `RElt`, on the left hand side of an assignment, with a format corresponding to the specialized domain, is replace by a `(Nil)` statement.

- any `RElt`, on the right hand side of an assignment statement, with a format corresponding to the specialized domain, is replace by constructing a new element of the original field type. Then the values from the specialized domain representation are copied into the newly allocated object. The allocation is not necessary if the extended `Decl` instruction is implemented in FOAM.

Once the transformations presented previously are performed, the compiler's optimizer can eliminate all of the unnecessary instructions producing the final code for the `pair` function presented in Listing 5.19. One can see that the resulting `pair` function eliminated the allocation of the contained objects.

Array Specialization

Arrays are similar to records. They both require memory allocation, and any update to their fields requires the creation of a new object and storage of the new value in the corresponding position, while the old value will be freed by the garbage collector. This is the only way to update values stored in an array of immutable objects. If the objects are mutable, the object could be retrieved from the array, and its state could be changed. For mutable objects. This optimization does not apply to mutable objects.

Listing 5.19: The *specialized* version of the `pair` function.

```
(Def
  (Const 23 pair)
  (Prog
    0
    1
    Word
    4
    1155
    22
    1
    0
    (DDecl Params (Decl Word "p" -1 4) (Decl Word "q" -1 4))
    (DDecl Locals (Decl Rec "" -1 12))
    (DFluid)
    (DEnv 4 13 4 4 4)
    (Seq
      (Set (Loc 0) (RNew 50))
      (Set (RElt 50 (Loc 0) 0) (Par 0 p))
      (Set (RElt 50 (Loc 0) 1) (Par 1 q))
      (Set (RElt 50 (Loc 0) 2) (Par 1 q))
      (Set (RElt 50 (Loc 0) 3) (Par 0 p))
      (Return (Loc 0)))))
```

Listing 5.20: Polynomial of complex coefficients.

```
1  Polynomial(d: ComplexType): with {
2          poly: SI -> %;
3          get: (%,SI) -> d;
4          set: (%,SI,d) -> d;
5  } == add {
6          Rep == PA(d);
7          import from Rep;
8          poly(size:SI):% == per new(size);
9          get(t:%,i:SI):d == rep(t).i;
10         set(t:%,i:SI,v:d):d == set!(rep(t), i, v);
11 }
```

Changing the representation of arrays requires the use of another data structure offered by the Aldor programming language, namely `TrailingArray`. This data structure does exactly what we need: it creates an array where each entry in the array is a sequence of fields from the record, and then retrieves the fields one by one. In trailing arrays, there is no need to allocate the memory for each element, rather the elements are stored inside the trailing array.

An example of a domain which uses an array of complex numbers for the coefficients of a polynomial is presented in Listing 5.20. The representation of `Polynomial` is `PA(d)`. The parametric domain `PA` is a simple primitive array whose representation is `Arr` (`Arr` is the FOAM type used for arrays).

The FOAM code for the initialization of the representation of the `Polynomial` domain is presented in Listing 5.21. One can see that the representation is just the domain `Arr` and that there is no specification of the real type used to store the elements.

To find the actual domain used to store the elements of the array, the exported functions of the `Polynomial` domain are searched for FOAM array access instructions,

Listing 5.21: Representation of `Polynomial` domain.

```
1 (Set (Lex 0 6 Rep) (CCall Word (Glo 17 domainMakeDummy)))
2 (Def (Loc 6) (Lex 0 6 Rep))
3 (Def (Lex 0 6 Rep) (Lex 3 8 Arr))
4 (CCall NOp (Glo 18 domainFill!) (Loc 6) (Lex 0 6 Rep))
5 (Set (Lex 0 6 Rep) (Loc 6))
6 (Set (AElt SInt (SInt 0) (Loc 0)) (SInt 316169058))
7 (Set (AElt SInt (SInt 0) (Loc 1)) (SInt 547382598))
8 (Set (AElt Word (SInt 0) (Loc 2)) (Lex 0 6 Rep))
```

`AElt`. For those values that are linked to the function parameters or the return type, the symbol meaning of the function is checked and the meaning of the array elements is extracted. That is compared with the meaning of the domain parameters. If a match is found, the domain parameter instance is returned.

Before starting the transformation, the `TrailingArray` domain constructing function must be retrieved from the library. In order to achieve this, a new global variable is created:

```
(GDecl Clos "sal_lang_TrailingArray_283473430" -1 4 1 Foam)
```

next, a lexical variable at the file level is introduced:

```
(Decl Word "TrailingArray" -1 4)
```

In the file initialization function (constant index 0), the lexical and global variables are initialized by the introduction of a new library initialization function, namely `sal_lang-init`, as seen in Listings 5.22 and 5.23. If the function `sal_lang-init` exists due to other domains from the same library being used in the current file, the existing version is modified so that is also initializes the value of the global introduced. The final step is to initialize the lexical variable introduced at the file level. All these changes are presented in the FOAM code extract presented in Listings 5.22 and 5.23.

Listing 5.22: Initializing `TrailingArray` from library.

```
(Unit
  (DFmt
    (DDecl
      Globals
      ...
      (GDecl Clos "sal_lang_TrailingArray_283473430" -1 4 1 Foam)
    (DDecl
      Consts
      ...
      (Decl Prog "sal_lang-init" -1 4)
    (DDecl
      LocalEnv
      ...
      (Decl Word "TrailingArray" -1 4))
  (DDef
    (Def
      (Const 0 t3)
      (Prog
        ...
        (DDecl Params)
        (DDecl
          Locals
          (Decl Clos "" -1 4)
          (Decl Clos "" -1 4)
          (Decl Clos "" -1 4)
          (Decl Arr "" -1 5)
          (Decl Arr "" -1 8)
          (Decl Rec "" -1 8)
          (Decl Rec "" -1 8))
        (DFluid)
        (DEnv 20)
        (Seq
          (CCall NOp (Glo 8 runtime))
          (Set (Glo 0 t3) (Glo 1 noOperation))
          (Def (Loc 0) (Clos (Env 0) (Const 26 sal_lang-init)))
```

Listing 5.23: Initializing `TrailingArray` from library.

```
        (Def
          (Lex 0 17 TrailingArray)
          (CCall
            Clos
            (Glo 29 stdGetWordWordRetWord0)
            (CCall Word (Glo 35 rtDelayedInit!) (Loc 0) (SInt 1))))
  (Def
    (Const 26 sal_lang-init)
    (Prog
      0
      2
      Word
      0
      1154
      18
      25
      0
      (DDecl Params (Decl SInt "idx" -1 4))
      (DDecl Locals)
      (DFluid)
      (DEnv 4 20)
      (Seq
        (If (Lex 1 14) 0)
        (Set (Lex 1 14) (Bool 1))
        (CCall NOp (Glo 39 sal_lang))
        (Label 0)
        (If (BCall SIntNE (Par 0 idx) (SInt 1)) 1)
        (Return (Glo 42 sal_lang_TrailingArray_283473430))
        (Label 1)
        (Return (Glo 19 sal_lang_Machine_915715331)))))
```

Listing 5.24: The specialized version of the representation of the `Polynomial` domain.

```
1           (Set (Lex 0 6 Rep) (CCall Word (Glo 17 domainMakeDummy)))
2           (Def (Loc 6) (Lex 0 6 Rep))
3           (Set (Loc 11) (ANew Word (SInt 1)))
4           (Set (Loc 12) (RNew 38))
5           (Set (RElt 38 (Loc 12) 0) (SInt 1))
6           (Set (RElt 38 (Loc 12) 1) (Loc 11))
7           (Set (AElt Word (SInt 0) (Loc 11)) (Lex 3 2 MachineInteger))
8           (Set (Loc 13) (ANew Word (SInt 2)))
9           (Set (Loc 14) (Cast Arr (RNew 38)))
10          (Set (RElt 38 (Loc 14) 0) (SInt 2))
11          (Set (RElt 38 (Loc 14) 1) (Loc 13))
12          (Set (AElt Word (SInt 0) (Loc 13)) (Lex 3 2 MachineInteger))
13          (Set (AElt Word (SInt 1) (Loc 13)) (Lex 3 2 MachineInteger))
14          (Def
15            (Lex 0 6 Rep)
16            (CCall
17              Word
18              (Lex 3 17 TrailingArray)
19              (Cast Word (Loc 12))
20              (Cast Word (Loc 14))))
21          (CCall NOp (Glo 18 domainFill!) (Loc 6) (Lex 0 6 Rep))
22          (Set (Lex 0 6 Rep) (Loc 6))
23          (Set (AElt SInt (SInt 0) (Loc 0)) (SInt 316169058))
24          (Set (AElt SInt (SInt 0) (Loc 1)) (SInt 547382598))
25          (Set (AElt Word (SInt 0) (Loc 2)) (Lex 0 6 Rep))
```

Once the initialization code is in place, the representation of the specialized domain must be changed. The changed representation is presented in Listing 5.24. This code creates a representation which uses a trailing array of the form:

```
Rep == TrailingArray(MI, (re:MI, im:MI))
```

where the types of the trailing part of the trailing array are copied from the representation of the domain used to instantiate the domain, which is also used as the type of the elements stored in the array.

Each trailing array uses values of type `TR`. Each `TR` type must be declared in the

Listing 5.25: The trailing array declaration.

```
1     (DDecl
2       TrailingArray
3       (Decl NOp "" -1 1)
4       (Decl Word "" -1 4)
5       (Decl Word "re" -1 4)
6       (Decl Word "im" -1 4))
```

declaration part of the unit. For our example, the declaration can be seen in Listing 5.25.

The final step in the translation between arrays and trailing arrays is to translate all of the exports of the specialized domain to the new representation. The change rules are:

- any `ANew` corresponding to type `%` is replaced by `TRNew` with a format index of the declaration presented in Listing 5.25
- any `AElt` used for storing elements of same type as the the type parameter will be replaced by the equivalent `TRElt` for each of the fields of the type parameter
- any `AElt` used for reading elements of same type as the the type parameter will be replaced by an allocation of the type parameter value followed by a copy of all the fields from the trailing array into the newly allocated memory

The `get` function for our example is presented in Listing 5.26. The specialized form of the `get` function is presented in Listing 5.27. This code introduces an extra memory allocation because it has to return an object of the type of its inner domain. The operations of the specialized domain which use internal objects belonging to the inner domain do not allocate a new variable for each operation that updates the values of the outer domain.

Listing 5.26: The FOAM code of `get` function from `Polynomial(Complex)`.

```
(Def
  (Const 33 get)
  (Prog
    0
    1
    Word
    4
    3203
    7
    1
    0
    (DDecl Params (Decl Word "t" -1 4) (Decl Word "i" -1 4))
    (DDecl Locals)
    (DFluid)
    (DEnv 4 16 4 4 4)
    (Seq
      (Return (AElt Word (Cast SInt (Par 1 i)) (Cast Arr (Par 0 t)))))))
```

Test	Original	Optimized	Ratio
Time (s)	119	8	15.0
Space (MB)	80	4	22.1

Table 5.4: Time and run-time memory improvement after hand specialization of polynomial multiplication.

5.3 Performance Results

The results of applying data specialization to the polynomial multiplication problem can be seen in Table 5.4. The results from Table 5.4 have been obtained by hand specializing the code. For the data representation optimization, objects created on the heap (mostly temporary objects resulting from arithmetic operations) are replaced by stack allocated objects or updating of the already allocated objects. This produces a decrease in memory usage.

All of the tests were performed using Aldor compiler version 1.0.3 under Fedora Core 5. The back-end C compiler used by the Aldor compiler was gcc 4.1.1. The

Listing 5.27: The specialized version of `get` function.

```
(Def
  (Const 33 get)
  (Prog
    0
    1
    Word
    4
    1155
    22
    1
    0
    (DDecl Params (Decl Word "t" -1 4) (Decl Word "i" -1 4))
    (DDecl Locals (Decl Rec "" -1 14))
    (DFluid)
    (DEnv 4 16 4 4 4)
    (Seq
      (Set (Loc 0) (RNew 14))
      (Set
        (RElt 14 (Loc 0) 0)
        (TRElt 37 (Cast Arr (Par 0 t)) (Cast SInt (Par 1 i)) 0))
      (Set
        (RElt 14 (Loc 0) 1)
        (TRElt 37 (Cast Arr (Par 0 t)) (Cast SInt (Par 1 i)) 1))
      (Return (Loc 0)))))
```

CPU was a Pentium 4 with a clock rate of 3.2 GHz, 1 MB cache and 2 GB of RAM. The actual hardware specification is not very important since we are only interested in the relative values presented in the ratio columns.

The specialized versions of the kernels from the SciGMark are good examples gains that can be obtained with data specialization. The data representation for the specialized versions of SciGMark use arrays of data that contain values unwrapped from `DoubleRing`. The data is only accessed through the class, so the optimization as the one presented in this chapter should be possible.

5.4 Applicability to Other Programming Languages

The compilers that we have tested for C++, C#, and Java do not perform the kind of optimization presented in this chapter.

For C++, we have seen the difference between a `std::vector` containing `Double`, a wrapper for the type `double`, and the same `std::vector` containing a pointer, e.g. `Double*`. In C++, the decision between these types is left to the programmer. There are some algorithms that perform more poorly after transforming heap allocated data into stack allocated data. This is mainly true for algorithms that rearrange data (sorting algorithms, matrix pivoting in jagged arrays). If the data size to be moved is large, it is much faster to rearrange the references to the objects than the whole objects.

C# offers the `struct` keyword that constructs a stack allocated object, rather than a heap allocated one. It is up to the programmer to choose between `struct` and `class`. As was the case for C++, replacing the data could be bad for some classes of algorithms without proper whole program analysis.

Java does not offer any way to distinguish between stack and heap allocation. All objects are allocated on the heap while all values of the basic types are allocated on the stack. To save values of the basic types inside collections, a wrapper has to be constructed for each value. This wrapping, called boxing, is performed automatically. However, operating on these collections requires boxing and unboxing the data which is time consuming. Work is in progress for the Java compiler to optimize away the allocations for temporary objects that do not escape the local scope of a function [16]. As of version 6.0, Sun's Java implementation still does not perform this optimization. It is also worth noting that this optimization will not optimize the representation of the objects that use collections. It will still be necessary to perform the boxing/unboxing operations for those cases.

5.5 Related Work

Object inlining has been presented by Julian Dolby and Andrew A. Chien in [21]. In their work they used the Concert compiler that performs extensive interprocedural analysis. The proposed optimization only fuses the objects that have a one-to-one relationship. This means that only objects that are created together and deleted together are fused. An example of such an object is a linked list that contains elements of a different class type. In this case, the elements are fused in the list.

A structure unfolding optimization was presented by Chen et al [15]. Several optimizations for C++ are described starting with agressive function inlining, indirect memory access folding using SSA based scalar optimizations for structures, and finally, structure folding. The final step replaces structures with their fields. This work performs some of the optimizations that we propose, by specializing the code locally in the function. This optimization is similar to the environment merge opti-

mization already present in the Aldor compiler. The difference is that a global SSA representation is necessary to track aliased variables.

A type-based compiler for standard ML used type information to optimize the code. In a usual Standard ML compiler, type information constructed by the front end of the compiler is used to verify the correctness of the program and then discarded. In the extension proposed by Zhong Shao and Andrew W. Appel in [58], their proposed type specialization refers to unboxing types and creating monomorphic functions.

Another ML compiler framework TIL presented in [63] proposed a typed intermediate language (TIL) with support for intensional polymorphism (the ability to dispatch to different routines based on the types of variables at runtime). The optimization does not eliminate the polymorphism at compile-time, leaving the functions polymorphic and selecting specialized behaviors based on the runtime time of the arguments. One interesting optimization, is the argument passing optimization. This optimization replaces the function arguments represented as records as multiple arguments containing the fields of the records. In contrast the optimization presented in this thesis, the specialization is not completed at compile-time, requiring the availability of an expressive type information representation of the data.

A more general framework for storing information about types which could later be used to produce efficient code for polymorphic labeled records was presented in [47]. The framework presented was only theoretical and there was no implementation mentioned.

Program specialization in Java using partial evaluation was presented Schultz in [56, 57]. He presented a two level language based on an extension of Featherweight Java (a simple object-oriented language based on Java), but in that language the author only specializes the base types. Also, the specialization does not deal with generic programming.

Virtual function calls, in C++, are very expensive. Object specialization is a way to replace virtual function calls with regular function calls. Porat et al [50] propose a code analysis technique to find possible types used as targets for the virtual call and then specialize according to this types. The complete analysis is NP-Hard so they approximate the set of types that are targets for a virtual a call. In particular, they restrict themselves to unique name (when there is only one target for the virtual call) and single type prediction (by predicting a most used type, specializing based on that type and leaving the virtual call for the rest of the types). Similar type specialization based on the most probable code execution path was proposed by Dean and al in [20]. This work applies to virtual call specialization, but not to generic programming. The type specialization method guesses some types and specializes for them while the optimization presented in this chapter that does the optimization based on the exact type constructor.

In C#, access to object fields is performed transparently through `get/set` methods [30]. This inspired the idea to use `Reference(T)` as a possible solution in section 5.2.3. The access to `T` can be transparently changed by the compiler with access through `Reference(T)`, where the operations to set and get the values can be different on a case by case basis.

5.6 Conclusions and Future Research

We have presented an optimization that can produce impressive speedups in particular cases, especially for algorithms that manipulate many small objects.

The optimization presented in this chapter is not as straightforward as the code specialization presented in chapter 4. A more complex framework had to be developed and it cannot be applied in all the cases. Some problems and their possible solutions

were discussed and an algorithm for implementing the optimization was presented. The limitation of the data specialization optimization is that it only works with immutable objects. This restriction was necessary to avoid the problems imposed by the aliasing of data. We also propose a simple extension to the Aldor intermediate language, FOAM, to allow records to be represented as inlined data instead of references to pointers to heap allocated objects. This extension would permit us to give access to the data contained in the specialized type, without reconstructing the original object. This is possible because the data representation of the types used to instantiate the generic type is not changed, it is only allocated inside the container type.

The results obtained for a hand optimized example were presented. They showed a dramatic increase in performance of up to 15 times execution time speedup and 20 times less memory use. This was expected since the number of allocations for temporary objects has been reduced dramatically.

The compiler implementation still needs improving to be able to handle more complex cases such as the ones seen in SciGMark.

We claim that this optimization could produce similar performance results in the specialized version of SciGMark due to the fact that SciGMark does not perform any special data accesses outside of the classes that implement the algorithms, and the difference in data representation between the generic and specialized versions is the wrapped data used to permit generic collections.

Chapter 6

Conclusions

The first objective of this thesis has been to develop a suitable framework to evaluate the performance of generic code. Such a framework must measure programs using parametric polymorphism in different ways as well as in combinations with various coding styles. This was achieved with SciGMark and comparing the timings of the generic code and the specialized code. To our knowledge, there was no benchmark available that evaluated a broad range of use of generics, so we created our own SciGMark.

SciGMark was an important tool in measuring the performance of the compilers with respect to the tower type optimization. Moreover, it helped in finding ways to improve the tower type optimization by changing the type representation together with the code specialization.

SciGMark is the first macro benchmark that uses generic code extensively and it is a very useful tool for measuring the compilers performance with respect to generic code.

The results provided by the benchmark have shown us that there is considerable room for improvement from the compiler optimization viewpoint. By examining

the generic and specialized code, we have found that type specialization helps in producing better code. Scientific code relies on rich mathematical models that are very well suited to generic model of programming. Extensive use of generic libraries can lead to type constructs that contain more levels of parametric types. We call these "deeply nested" parametric types. With the help of SciGMark, we have seen performance degradation of the generic code even when the type towers constructed are not deeper than two levels. We expect that deeply nested types will only increase the performance gap.

Another objective of the thesis was to present some of the proposed optimizations and to analyze their effect and limitations. Two optimizations methods were presented in the thesis. The code specialization optimizations was implemented for the Aldor programming language. A thorough analysis was performed on this optimization with different use cases. The results have shown that even with the powerful optimizations already implemented by the Aldor compiler, the generic code is still not close to its optimal performance.

Aldor programming language libraries are heavily generic and it is easy to create "deeply nested" types. Therefore, we have implemented an automatic code specializer to optimize the "deeply nested" generics for the Aldor programming language. The results obtained by the automatic code specializer are encouraging showing improvements between 1.05 and 3 times faster with cases where an order of magnitude faster was possible.

There is one disadvantage to code specialization, namely object code explosion. As a direct consequence of increased code size the compilation time will also be increase accordingly. This is the trade-off that has to be paid to increase the time performance.

Another important factor of the code specialization is that the optimization itself is very simple to implement and rather efficient.

The results produced by running SciGMark on code specialization optimization showed that there is still room for improvement. By examining the hand specialized code from the SciGMark, it was clear that the memory behavior needs to be improved. Therefore, we decided to extend the code specialization optimization with another optimization, namely data specialization.

The data specialization rearranges the data in the specialized domain to avoid allocation of intermediate values. We provided an implementation and showed results on some simple cases, but the implementation of the data specialization optimization could still be improved to handle more complex cases such as the SciGMark benchmark. The drawback of the data specialization optimization is that it only works with immutable objects. It can be extended to work on some mutable cases, but only with complex whole program analysis which complicates the compiler.

For the data specialization a speedup of 15 times was obtained for the polynomial multiplication. This result contains also the improvement resulted from code specialization. The most improvement is due to the reduced number of temporary objects created as a result of the use of trailing arrays and procedure integration that led to a 22 times decrease in memory consumption.

With the increased popularity of object-oriented programming languages we have seen an increased interest in optimizing subclassing polymorphism to the level where it is very easy to use virtual methods with very little performance penalty. However, for scientific algorithms parametric polymorphism seems more appealing because of the possibility to statically type check the type parameters making the code safer. The possibility to perform static type checking on type parameters offers better opportunities for compile-time optimizations, and we would like to contribute to the advancement of compiler technology in this area.

Appendix A

A.1 SciGMark Example

A complete listing of the SciGMark benchmark suite is available at `http://www.orcca.on.ca/benchmarks`.

This appendix lists the code for the fast Fourier transform kernel as implemented in Aldor. The generic and specialized forms of fast Fourier transform are presented in Listing A.1 and Listing A.2 respectively.

Listing A.1: Generic version of fast Fourier transform.

```
1
2  define GenFFTCat(R: Join(IRing, ITrigonometric), C: IMyComplex(R)):Category
3  == with {
4    constr    : (R, C) -> %;
5    numFlops  : int -> double;
6    transform : (%,Array C) -> ();
7    inverse   : (%,Array C) -> ();
8    test      : (%, Array C) -> R;
9    makeRandom: (%, int) -> Array C;
10   main      : () -> ();
11 };
12
13 define GenFFT(R:Join(IRing,ITrigonometric),C:IMyComplex(R)): GenFFTCat(R,C)
14 == add {
15   Rep == Record(num: R, c: C);
16   import from Rep, R, C, Array(C), String, int, double;
17   constr(n: R, c: C): % == {per [n, c];}
18   numFlops(n: int): double == {
19     Nd := n::double;
20     logN := log2(n)::double;
21     return (5.0 * Nd - 2.0) * logN + 2.0 * (Nd + 1.0);
22   }
23   transform(t:%,data: Array C): () == {transform__internal(t, data, -1);}
24   inverse(t:%,data: Array C): () == {
25     transform__internal(t, data, 1);
26     nd := #data;
27     norm := coerce(rep(t).c, 1.0 / nd::double);
28     for i in 0..nd-1 repeat data.i := data.i*norm;
29   }
30   test(this: %, data: Array C): R == {
31     import from Array C;
32     nd := #data;
33     cpy: Array C := newArray(nd)$C; --$
34     for i in 0..nd-1 repeat cpy(i) := copy(data(i));
35     transform(this,data);
36     inverse(this,data);
37     diff := coerce(rep(this).num, 0.0);
38     for i in 0..nd-1 repeat {
39       d := data.i; d := d-cpy(i);
40       setRe(d, getRe(d)*getRe(d));
41       setIm(d, getIm(d)*getIm(d));
42       diff := +(diff,getRe(d));
43       diff := +(diff,getIm(d));
44     }
45     diff := diff / coerce(rep(this).num, nd);
46     diff := sqrt(diff);
47     dispose(cpy);
48     diff;
49   }
```

```
50    makeRandom(t: %, n: int): Array C == {
51      data: Array C := newArray(n);
52      r: Random := Random(1);
53      for i in 0..n-1 repeat
54        data.i := create(coerce(rep(t).num, nextDouble(r)),
55          coerce(rep(t).num, nextDouble(r)));
56      data;
57    }
58    main(): () == {
59      import from DoubleRing;
60      c:MyComplex(DoubleRing) := create(DoubleRing(0.0), DoubleRing(0.0));
61      num := DoubleRing(0.0);
62      T == GenFFT(DoubleRing, MyComplex(DoubleRing));
63      import from T;
64      fft: T := constr(num, c);
65      n:int := 1024;
66      stdout << "n= " << n << " => RMS Error=" <<
67          test(fft,makeRandom(fft, n))::double << newline;
68    }
69    -- private
70    log2(n: int): int == {
71      log: int := 0;
72      k: int := 1;
73      while k < n repeat {log := log + 1; k := k * 2;}
74      if n ~= (shift(1,log)) then {
75        stdout << "FFT: Data length is not a power of 2!:"
76            << n << newline;
77        error("Exit");
78      }
79      log;
80    }
81    transform__internal(this:%, data: Array C, direction: int): () == {
82      if #data = 0 then return;
83      n := #data;
84      if n = 1 then return; -- Identity operation!
85      logn := log2(n);
86      bitreverse(data);
87      n1 := coerce(rep(this).num, 1.0);
88      n0 := coerce(rep(this).num, 0.0);
89      n2 := coerce(rep(this).num, 2.0);
90      w  := create(n1,n0);
91      theta__ := coerce(rep(this).num, 2.0 * direction::double * PI / 2.0);
92      dual := copy(n1);
93      FOR(bit:int:=0,bit<logn,{bit:=bit+1;me(dual,n2)}) {
94        setRe(w, n1); setIm(w, n0);
95        theta := theta__ / dual;
96        s := sin(theta);
97        t := sin(theta / 2.0::R);
98        s2 := t * t * 2.0::R;
99        FOR(b:int:=0,b<n,b:=b+2*integer(dual::double)::int) {
100         i := b; j := (b + integer(dual::double)::int);
```

```
101         wd := data.j; tmp := data.i; tmp := tmp-wd;
102         data.i := data.i+wd;
103       }
104       FOR(a:int:=1,a<integer(dual::double)::int,a:=a+1) {
105         nn := 1.0::R; nn := nn - s2;
106         tmp := create(nn, s);
107         w := w * tmp;
108         FOR(b:int:=0,b<n,b:=b+2*integer(dual::double)::int) {
109           i := b+a; j := b+a+integer(dual::double)::int;
110           z1 := data.j; wd := copy(w); wd := wd * z1;
111           data.j := data.i - wd; data.i := data.i + wd;
112         }
113       }
114     }
115   }
116   bitreverse(data: Array C):() == {
117     -- This is the Goldrader bit-reversal algorithm */
118     n: int := #data; nm1:int := n - 1; j: int := 0;
119     FOR(i: int := 0,i < nm1,i:=i+1) {
120       ii := shift (i, 1);
121       jj := shift (j, 1);
122       k := shift (n, -1);
123       if i < j then {tmp := data.i; data.i := data.j; data.j := tmp;}
124       while k <= j repeat {j := j - k; k := shift(k, -1);}
125       j := j + k;
126     }
127   }
128 };
```

Listing A.2: Specialized version of fast Fourier transform.

```
define FFT: with {
  num__flops: int -> double;
  transform : AD -> ();
  inverse   : AD -> ();
  test      : AD -> double;
} == add {
  import from AD;
  num__flops(n: int): double == {
    Nd := n::double; logN := log2(n)::double;
    return (5.0 * Nd - 2.0) * logN + 2.0 * (Nd + 1.0);
  }
  transform(data: AD): () == {transform__internal(data, -1);}
  inverse(data: AD): () == {
    transform__internal(data, 1);
    -- Normalize
    nd := #data;
    n := nd quo 2;
    norm := 1.0 / (n::double);
    FOR(i:int:=0,i<nd,i:=i+1) {data.i := data.i*norm}
  }
  test(data: AD): double == {
    nd := #data;
    copy: AD := new(nd);
    FOR(i:int:=0,i<nd,i:=i+1) {copy(i) := data(i)}
    -- Transform & invert
    transform(data);
    inverse(data);
    -- Compute RMS difference.
    diff := 0.0;
    FOR(i:int:=0,i<nd,i:=i+1){d := data(i)-copy(i);diff := diff+d*d}
    diff := diff/nd::double;
    diff
  }
  makeRandom(n: int): AD == {
    import from Random;
    nd := 2*n;
    data: AD := new(n);
    r := Random(1);
    FOR(i:int:=0,i<n,i:=i+1) {data.i := nextDouble(r)}
    data
  }
  -- private
  log2(n: int): int == {
    log:int := 0;
    k:int := 1;
    while k < n repeat { k := k * 2; log := log + 1; }
    if n ~= (shiftUp(1,log)) then {
      print << "FFT: Data length is not a power of 2!:" << n << newline;
      --flush!(print);
```

```
50        error("Exit");
51      }
52      log
53    }
54    transform__internal(data: AD, direction: int): () == {
55      if #data = 0 then return;
56      n := #data quo 2;
57      if n = 1 then return; -- Identity operation!
58      logn := log2(n);
59      -- bit reverse the input data for decimation in time algorithm
60      bitreverse(data);
61      -- apply fft recursion
62      -- this loop executed log2(N) times
63      FOR({dual:int:=1;bit:int:=0},bit<logn,{bit:=bit+1;dual:=dual*2}){
64        wReal := 1.0;
65        wImag := 0.0;
66        theta := 2.0 * direction::double * PI / (2.0 * dual::double);
67        s := sin(theta);
68        t := sin(theta/2.0);
69        s2 := 2.0*t*t;
70        -- a := 0
71        FOR(b: int := 0, b < n, b := b + 2 * dual) {
72          i := 2*b;
73          j := 2*(b + dual);
74          wdReal := data(j);
75          wdImag := data(j+1);
76          data(j)   := data(i)   - wdReal;
77          data(j+1) := data(i+1) - wdImag;
78          data(i)   := data(i)   + wdReal;
79          data(i+1) := data(i+1) + wdReal;
80        }
81        --  a := 1 .. (dual-1)
82        FOR(a:int:= 1,a<dual,a:=a+1) {
83          {
84            tmpReal := wReal - s*wImag - s2*wReal;
85            tmpImag := wImag + s*wReal - s2*wImag;
86            wReal := tmpReal;
87            wImag := tmpImag;
88          }
89
90          FOR(b: int := 0, b < n, b := b + 2 * dual) {
91            i := 2*(b + a);
92            j := 2*(b + a + dual);
93            z1Real := data(j);
94            z1Imag := data(j+1);
95            wdReal := wReal*z1Real - wImag*z1Imag;
96            wdImag := wReal*z1Imag + wImag*z1Real;
97            data(j)   := data(i)   - wdReal;
98            data(j+1) := data(i+1) - wdImag;
99            data(i)   := data(i)   + wdReal;
100           data(i+1) := data(i+1) + wdImag;
```

```
101         }
102       }
103     }
104   }
105   bitreverse(data: AD):() == {
106     -- This is the Goldrader bit-reversal algorithm */
107     n := #data quo 2;
108     nm1 := n - 1;
109     i:int := 0;
110     j:int := 0;
111     while i < nm1 repeat {
112       ii := shiftUp (i, 1);
113       jj := shiftUp (j, 1);
114       k := shiftDown (n, 1);
115       if i < j then {
116         tmpReal := data(ii);
117         tmpImag := data(ii+1);
118         data(ii) := data(jj);
119         data(ii+1) := data(jj+1);
120         data(jj) := tmpReal;
121         data(jj+1) := tmpImag;
122       }
123       while k <= j repeat {j := j - k;k := shiftDown(k, 1);}
124       j := j + k; i := i + 1;
125     }
126   }
127 }
```

A.2 Examples of Simple Tests for Code Specialization

This appendix contains the code for the simple tests presented in 4.4.1.

The first three tests: test1, test2 and test3 are based on the code presented in Listing A.3.

Listing A.3: Test4 implementation.

```
#include "axllib.as"
SI ==> SingleInteger; LOOP ==> 3000000;
define Base: Category == with {methodA: % -> ();methodA: (SI, SI, %) -> ();};
A: Base with {
        newA: SI -> %;
        coerce: SI -> %;
        coerce: % -> SI;
        mA: % -> ();
} == add {
        Rep ==> SI;
        import from Rep;
        newA(i: SI): % == per i;
        coerce(i: SI): % == per i;
        coerce(a: %): SI == rep a;
        methodA(a: %):() == {
                if (rep a) = 0 then {print << "A::methodA1" << newline;}
        }
        methodA(a: SI, b: SI, c: %):() == {
                if a = 0 then {print << "A::methodA2" << newline;}
        }
        mA(a: %): () == {print << "A::mA" << newline;}
}
D: Base with {
        newD: SI -> %;
        coerce: SI -> %;
        coerce: % -> SI;
        mD: % -> ();
} == add {
        Rep ==> SI;
        local l:SI := 1;
        newD(i: SI): % == per i;
        coerce(i: SI): % == per i;
        coerce(a: %): SI == rep a;
        methodA(a: %):() == {
                if (rep a)=0 then print << "D::methodA1 (" <<l<< ")" << newline;
        }
```

```
37          methodA(a: SI, b: SI, c: %):() == {
38                  if a = 0 then print << "D::methodA2" << newline;
39          }
40          mD(a: %): () == {print << "D::mD" << newline;}
41  }
42  B (V: Base, W: Base): with {
43          newB: V -> %;
44          methodB: V -> ();
45          methodB: W -> ();
46          methodB: (SI, W) -> ();
47          coerce: % -> V;
48  } == add {
49          import from V,W,SingleInteger;
50          Rep ==> V;
51          newB(i:V): % == per i;
52          coerce(c:%): V == rep c;
53          methodB(c: V): () == {for ii in 1..9*LOOP repeat {methodA(c)$V;}}
54          methodB(c: SI, x: W): () == {
55                  for ii in 1..9*LOOP repeat {methodA(c, c, x)$W;}
56          }
57          methodB(c: W): () == {for ii in 1..9*LOOP repeat {ttt(c);}}
58          ttt(p: W): () == {
59                  for ii in 1..9*LOOP repeat {methodA(p)$W;} print << newline;
60          }
61  }
62  main():() == {
63          default i: SI := 1;
64          default a: A;
65          default d: D;
66          default ba: B (A, D);
67          import from B(A,D);
68          a := newA(i);
69          d := newD(i);
70          ba := newB(a);
71          methodB(a);
72          methodB(d);
73          methodB(5, d);
74  }
75  main();
```

An interesting case is encountered in test4 and test5. The code implementation for test4 is given in Listing A.4. The only difference in test5 is that the last two lines from Listing A.4 are contained inside a funtion `main`. In test4, the call `m(1)` is not optimized at all, and `Dom4` cannot be optimized in generic form. In test5, the code was optimized in the context of function `main`.

Listing A.4: Test4 implementation.

```
#include "axllib.as"
int ==> SingleInteger;
out ==> print;
import from int;
define Dom1Cat: Category == with {m: int -> int;};
define Dom2Cat: Category == with {m: int -> int;};
define Dom3Cat: Category == with {m: int -> int;};
define Dom4Cat: Category == with {m: int -> int;};
define Dom1: Dom1Cat == add {m(g: int): int == g := g + 1;}
define Dom2(p: Dom1Cat): Dom1Cat == add {
        m(x: int) : (int) == {
                import from p;
                for i in 1..3000 repeat x := m(x)$p + 1;--$
                x;
        }
}
define Dom3(p: Dom1Cat): Dom1Cat == add {
        m(x: int) : (int) == {
                import from p;
                for i in 1..600 repeat x := m(x)$p + 1;--$
                x;
        }
}
define Dom4(p: Dom1Cat): Dom1Cat == add {
        m(x: int) : (int) == {
                import from p;
                for i in 1..500 repeat x := m(x)$p + 1;--$
                x;
        }
}
import from Dom4 Dom3 Dom2 Dom1;
out << m(1) << newline;
```

The example from Listing A.5 presents an exceptional speedup due to the fact that in `Dom3` the representation is a new type. If one would use a macro instead `Rep ==> Dom25 p`, both execution times become identical.

Listing A.5: Test6 implementation.

```
#include "axllib.as"
import from SingleInteger;
define DomCat: Category == with {m: (SingleInteger) -> (SingleInteger);};
define Dom1: DomCat == add {
        m (g: SingleInteger) : (SingleInteger) == {
                import from SingleInteger;
                g := next(g);
        }
        next (g: SingleInteger) : (SingleInteger) == g + 1;
}
define Dom2(p: DomCat): DomCat == add {
        m (x: SingleInteger) : (SingleInteger) == {
                import from SingleInteger, p;
                for i in 1..3000 repeat x := m(x)$p; --$
                x;
        }
}
define Dom25(p: DomCat): DomCat == add {
        m (x: SingleInteger) : (SingleInteger) == {
                import from SingleInteger,p;
                for i in 1..3000 repeat {x := m(x)$p;} -- $
                x;
        }
}
define Dom3(p: DomCat): DomCat == add {
        Rep == Dom25 p;
        import from Rep;
        m (x: SingleInteger) : (SingleInteger) == {
                import from SingleInteger;
                for i in 1..1000 repeat {x := m(x)$Rep;} --$
                x;
        }
}
main() :() == {import from Dom3 Dom2 Dom1;print << m(1) << newline;}
main();
```

Although the code from Test7 (Listing A.6) is similar to the one in Test6, the speedup is moderate. In this case the replacement of the type `BB` with a macro did not improve anything.

Listing A.6: Test7 implementation.

```
#include "axllib"
macro {SI == SingleInteger;}
define Dom1Cat: Category == with {m11: SI -> SI;}
define Dom1: Dom1Cat == add {
        m11(p:SI): SI == {for i in 1..100 repeat {p := p + 1;} p;}
}
define Dom2Cat: Category == with {m21: SI -> SI;}
define Dom2(d: Dom1Cat): Dom2Cat == add {
        import from d;
        m21(p:SI): SI == {for i in 1..1000 repeat {p := p + m11(p);} p;}
}
define Dom3Cat: Category == with {m31: SI -> SI;}
define Dom3(d: Dom2Cat): Dom3Cat == add {
        import from d;
        m31(p:SI): SI == {for i in 1..100 repeat {p := p + m21(p);} p;}
}
define Dom4Cat: Category == with {m41: SI -> SI;}
define Dom4(d: Dom3Cat): Dom4Cat == add {
        import from d;
        m41(p:SI): SI == {for i in 1..100 repeat {p := p + m31(p);} p;}
}
define Dom5Cat: Category == with {m51: SI -> SI;}
define Dom5(d: Dom4Cat): Dom5Cat == add {
        import from d;
        m51(p:SI): SI == {for i in 1..100 repeat {p := p + m41(p);}     p;}
}
define Dom6(d: Dom3Cat): with {m61: SI -> SI;} == add {
        BB == Dom5 Dom4 d;
        import from BB;
        m61(p:SI): SI == m51(p+10) + m51(p+20);
}
main ():() == {import from SI, Dom6 Dom3 Dom2 Dom1; print << m61(100) << newline;}
main();
```

Listing A.7 presents the complete implementation if Test8.

Listing A.7: Test8 implementation.

```
#include "axllib"
macro {SI == SingleInteger;}
define Dom1Cat: Category == with {m11: SI -> SI;}
define Dom1: Dom1Cat == add {
        m11(p:SI): SI == {for i in 1..100 repeat {p := p + 1;} p;}
}
define Dom2Cat: Category == with {m21: SI -> SI;}
define Dom2(d: Dom1Cat): Dom2Cat == add {
        import from d;
        m21(p:SI): SI == {for i in 1..100 repeat {p := p + m11(p);} p;}
}
define Dom3Cat: Category == with {m31: SI -> SI;}
define Dom3(d: Dom2Cat): Dom3Cat == add {
        import from d;
        m31(p:SI): SI == {for i in 1..100 repeat {p := p + m21(p);} p;}
}
define Dom4Cat: Category == with {m41: SI -> SI;}
define Dom4(d: Dom3Cat): Dom4Cat == add {
        import from d;
        m41(p:SI): SI == {for i in 1..100 repeat {p := p + m31(p);} p;}
}
define Dom5Cat: Category == with {m51: SI -> SI;}
define Dom5(d: Dom4Cat): Dom5Cat == add {
        import from d;
        m51(p:SI): SI == {for i in 1..1000 repeat {p := p + m41(p);} p;}
}
define Dom6(d: (Dom2Cat) -> Dom3Cat): with {m61: SI -> SI;} == add {
        BB == Dom5 (Dom4 (d (Dom2 (Dom1))));
        import from BB;
        m61(p:SI): SI == m51(p+10) + m51(p+20);
}
main ():() == {
        import from SI;
        BB == Dom3;
        import from Dom6(Dom3);
        print << m61(100) << newline;
}
main();
```

Bibliography

[1] Dave Abrahams, Carl Daniel, Beman Dawes, Jeff Garland, Doug Gregor, and John Maddock. Boost C++ libraries. `http://www.boost.org/`, Valid on 2007/07/23.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, 2007.

[4] Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for Java. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 132–145, New York, NY, USA, 1997. ACM Press.

[5] Lennart Beckman, Anders Haraldsson, Östen Oskarsson, and Erik Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[6] Gilad Bracha. Generics in the Java programming language. `http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html`, Valid on 2007/07/23.

[7] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[8] Gary Bray. Sharing code among instances of Ada generics. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 276–284, New York, NY, USA, 1984. ACM Press.

[9] Manuel Bronstein. SUM-IT: A strongly-typed embeddable computer algebra library. In *Proceedings of DISCO'96, Karlsruhe*, pages 22–33. Springer LNCS 1128, 1996.

[10] David Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 47–56, New York, NY, USA, 1988. ACM Press.

[11] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM Press.

[12] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[13] Robert Cartwright and Jr. Guy L. Steele. Compatible genericity with run-time types for the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference On Object-Oriented Programming, Systems, Languages, And Applications*, pages 201–215, New York, NY, USA, 1998. ACM Press.

[14] Craig Chambers and David Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 146–160, New York, NY, USA, 1989. ACM Press.

[15] Kaiyu Chen, Sun Chan, Roy Dz-Ching Ju, and Peng Tu. Optimizing structures in object oriented programs. *Interact*, 00:94–103, 2005.

[16] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA '99: Proceedings Of The 14th ACM SIGPLAN Conference On Object-Oriented Programming, Systems, Languages, And Applications*, pages 1–19, New York, NY, USA, 1999. ACM Press.

[17] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, 1992.

[18] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software - Practice and Experience*, 21(6):581–601, 1991.

[19] The Standard Performance Evaluation Corporation. SPEC benchmark. `http://www.spec.org/`, Valid on 2007/07/23.

[20] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 93–102, 1995.

[21] Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 345–357, New York, NY, USA, 2000. ACM Press.

[22] Laurentiu Dragan and Stephen M. Watt. Parametric polymorphism optimization for deeply nested types. In *Proceedings of Maple Conference 2005*, pages 243–259. Maplesoft, 2005.

[23] Laurentiu Dragan and Stephen M. Watt. Performance analysis of generics in scientific computing. In Daniela Zaharie, Dana Petcu, Viorel Negru, Tudor Jebelean, Gabriel Ciobanu, Alexandru Circotas, Ajith Abraham, and Marcin Paprzycki, editors, *Proceedings Of Seventh International Symposium On Symbolic And Numeric Algorithms For Scientific Computing*, pages 90–100. IEEE Computer Society, 2005.

[24] Laurentiu Dragan and Stephen M. Watt. On the performance of parametric polymorphism in Maple. In Ilias S. Kotsireas, editor, *Proceedings of Maple Conference 2006*, pages 35–42. Maplesoft, 2006.

[25] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LinBox: A generic library for exact linear algebra.

[26] Martin Dunstan. Aldor compiler internals II. `http://www.aldor.org/docs/reports/ukqcd-2000/compiler2-ukqcd00.pdf`, Valid on 2007/07/23.

[27] Ontario Research Centre for Computer Algebra. Aldor programming language project web page. `http://www.aldor.org/`, Valid on 2007/07/23.

[28] Jens Gerlach and Joachim Kneis. Generic programming for scientific computing in C++, Java, and C#. In Xingming Zhou, Stefan Jähnichen, Ming Xu, and Jiannong Cao, editors, *APPT*, volume 2834 of *Lecture Notes in Computer Science*, pages 301–310. Springer, 2003.

[29] Java Grande Forum Numerics Working Group. JavaNumerics. `http://math.nist.gov/javanumerics/#benchmarks`, Valid on 2007/07/23.

[30] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*, pages 317–326. Addison-Wesley, 2003.

[31] Richard D. Jenks and Robert S. Sutor. *Axiom The Scientific Computation System*. Springer-Verlag, 1992.

[32] Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation And Automatic Program Generation*, chapter 1 and 11. Prentice Hall, 1993.

[33] ISO/IEC JTC1/SC22/WG21. Technical report iso/iec 18015:2006 on C++ performance. Technical report, ISO/IEC PDTR 18015, August 2003.

[34] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime. In *PLDI '01: Proceedings Of The ACM SIGPLAN 2001 Conference On Programming language Design And Implementation*, pages 1–12, New York, NY, USA, 2001. ACM Press.

[35] Angelika Lagner. Java generics FAQs. `http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html`, Valid on 2007/07/23.

[36] Microsoft MSDN Library. Valuetype class. `http://msdn2.microsoft.com/en-us/library/system.valuetype.aspx`, Valid on 2007/07/23.

[37] Barbara Liskov. A history of CLU. `http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-561.pdf`, Valid on 2007/07/23.

[38] E. Meijer and J. Gough. Technical overview of the common language runtime, 2000.

[39] Robin Milner. How ML evolved. In *Polymorphism (The ML/LCF/Hope Newsletter) 1, 1*, 1983.

[40] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*, pages 73–74. The MIT Press, 1997.

[41] Marc Moreno-Maza. The BasicMath Aldor library. `http://www.nag.co.uk/projects/FRISCO.html`, Valid on 2007/07/23.

[42] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[43] Matthias Müller. Abstraction benchmarks and performance of C++ applications.

[44] Cosmin E. Oancea and Stephen M. Watt. Domains and expressions: An interface between two approaches to computer algebra. In *ACM International Symposium on Symbolic and Algebraic Computation ISSAC'05*, pages 261 – 269, 2005.

[45] Martin Odersky, Enno Runne, and Philip Wadler. Two ways to bake your Pizza - Translating parameterised types into Java. In *Generic Programming*, pages 114–132, 1998.

[46] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[47] Atsushi Ohori. Type-directed specialization of polymorphism. *Information and Computation*, 155(1–2):64–107, 1999.

[48] John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Languages and Compilers for Parallel Computing*, pages 566–580, 1995.

[49] Erik Poll and Simon Thompson. The type system of Aldor. Technical Report 11-99, Computing Laboratory, University of Kent at Canterbury, Kent CT2 7NF, UK, July 1999.

[50] Sara Porat, David Bernstein, Yaroslav Fedorov, Joseph Rodrigue, and Eran Yahav. Compiler optimization of C++ virtual function calls. In *Proceedings of the 2 nd Conference on Object-Oriented Technologies and Systems, (Toronto, Canada)*, pages 3–14, 1996.

[51] Roldan Pozo and Bruce Miller. SciMark2. `http://math.nist.gov/scimark2`, Valid on 2007/07/23.

[52] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis vs. procedure integration. *Inf. Process. Lett.*, 32(3):137–142, 1989.

[53] Stephen Richardson and Mahadevan Ganapathi. Interprocedural optimization: Experimental results. *Software: Practice and Experience*, 19(2):149–169, 1989.

[54] Stephen E. Richardson. *Evaluating interprocedural code optimization techniques.* PhD thesis, Stanford University, Stanford, CA, USA, 1991.

[55] Arch D. Robison. Impact of economics on compiler optimization. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10, New York, NY, USA, 2001. ACM Press.

[56] Ulrik P. Schultz. Partial evaluation for class-based object-oriented languages. *Lecture Notes in Computer Science*, 2053:173–198, 2001.

[57] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. *Lecture Notes in Computer Science*, 1628:367–391, 1999.

[58] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. *SIGPLAN Not.*, 30(6):116–129, 1995.

[59] Victor Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/doc/tour.html`, Valid on 2007/07/23.

[60] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000.

[61] Bjarne Stroustrup. A history of C++: 1979–1991. *ACM SIGPLAN Notices*, 28(3):271–297, 1993.

[62] Bjarne Stroustrup. *C++ Programming Language*. Addison-Wesley, 3rd edition, 1999.

[63] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[64] Kresten Krab Thorup. Genericity in Java with virtual types. *Lecture Notes In Computer Science*, 1241:444–461, 1997.

[65] Todd Veldhuizen. Using C++ template metaprograms. C++ Report Vol. 7 No. 4 (May 1995), pp. 36-43.

[66] Todd L. Veldhuizen. C++ templates as partial evaluation, 1998.

[67] Todd L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing In Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes In Computer Science. Springer-Verlag, 1997.

[68] Mirko Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 610–619, New York, NY, USA, 2001. ACM Press.

[69] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *OOPSLA '00: Proceedings Of The 15th ACM SIGPLAN Conference On Object-Oriented Programming, Systems, Languages, And Applications*, pages 146–165, New York, NY, USA, 2000. ACM Press.

[70] Stephen M. Watt. *Handbook Of Computer Algebra*, chapter Aldor, pages 265–270. Springer Verlag, 2003.

[71] Stephen M. Watt. Pivot-free block matrix inversion. In *Proc. 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, (SYNASC 2006)*, pages 151–155, 2006.

[72] Stephen M. Watt. Aldor compiler internals I. `http://www.aldor.org/docs/reports/ukqcd-2000/compiler1-ukqcd00.pdf`, Valid on 2007/07/23.

[73] Stephen M. Watt, Peter A. Broadbery, Pietro Iglio, Scott C. Morrison, and Jonathan M. Steinbach. Foam: A first order abstract machine version 0.35. `http://www.aldor.org/docs/foam.pdf`, Valid on 2007/07/23.

[74] Stephen M. Watt, Peter A. Broadbery, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. Aldor user guide. `http://www.aldor.org/`, Valid on 2007/07/23.

[75] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .NET common language runtime. *SIGPLAN Not.*, 39(1):39–51, 2004.

VITA

Name: Laurentiu Dragan

Born: Bucharest, Romania, 1975

Education:

- 2000 Master of Science in Electrical Engineering and Computer Science, "Politehnica" University of Bucharest, Romania.
- 1999 Bachelor with honors in Electrical Engineering and Computer Science, "Politehnica" University of Bucharest, Romania.

Awards:

- Western Graduate Research Scholarship (WGRS). The University of Western Ontario, 2005-2006
- Special University Scholarship (SUS). The University of Western Ontario, 2003-2005
- International Graduate Student Scholarship (IGSS). The University of Western Ontario, 2001-2004
- Presidential Scholarship for Graduate Studies (PSGS) The University of Western Ontario, 2001-2002
- Presidential Scholarship, "Politehnica" University of Bucharest, Romania, 1994-2000

List of Publications:

[1] L. Dragan and S. M. Watt. On the performance of parametric polymorphism in maple. In Ilias S. Kotsireas, editor, *Proceedings of Maple Conference 2006*, pages 35–42 Maplesoft, 2006.

[2] L. Dragan and S. M. Watt. Performance analysis of generics in scientific computing. In Daniela Zaharie, Dana Petcu, Viorel Negru, Tudor Jebelean, Gabriel Ciobanu, Alexandru Circotas, Ajith Abraham, and Marcin Paprzycki, editors, *Proceedings Of Seventh International Symposium On Symbolic And Numeric Algorithms For Scientific Computing*, pages 90–100. IEEE Computer Society, 2005.

[3] L. Dragan and S. M. Watt. Parametric polymorphism optimization for deeply nested types. In *Proceedings of Maple Conference 2005*, pages 243–259. Maplesoft, 2005.