Technical Report No. 06-18

Proceedings of the Second International Workshop on Library-Centric Software Design (LCSD '06)

JOSHUA BLOCH
JAAKKO JÄRVI (PROGRAM CO-CHAIRS)

ANDREAS PRIESNITZ
SIBYLLE SCHUPP (PROCEEDINGS EDITORS)

Department of Computer Science and Engineering

Division of Computing Science
CHALMERS UNIVERSITY OF TECHNOLOGY/
GÖTEBORG UNIVERSITY
Göteborg, Sweden, 2006

Technical Report in Computer Science and Engineering at
Chalmers University of Technology and Göteborg University

Technical Report No. 06-18
ISSN: 1652-926X

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, October 2006

Proceedings of the Second International Workshop on

Library-Centric Software Design (LCSD '06)

An OOPSLA Workshop

October 22, 2006

Portland, Oregon, USA

Joshua Bloch and Jaakko Järvi (Program Co-Chairs)

Andreas Priesnitz and Sibylle Schupp (Proceedings Editors)

Chalmers University of Technology
Computer Science and Engineering Department
Technical Report 06-18

Foreword

These proceedings contain the papers selected for presentation at the workshop Library-Centric Software Design (LCSD), held on October 22nd, 2006 in Portland, Oregon, USA, as part of the yearly ACM OOPSLA conference. The current workshop is the second LCSD workshop in the series. The first ever LCSD workshop in 2005 was a success—we are thus very pleased to see that interest towards the current workshop was even higher.

Software libraries are central to all major scientific, engineering, and business areas, yet the design, implementation, and use of libraries are underdeveloped arts. The goal of the Library-Centric Software Design workshop therefore is to place the various aspects of libraries on a sound technical and scientific footing. To that end, we welcome both research into fundamental issues and the documentation of best practices. The idea for a workshop on Library-Centric Software Design was born at the Dagstuhl meeting *Software Libraries: Design and Evaluation* in March 2005. Currently LCSD has a steering committee developing the workshop further, and coordinating the organization of future events. The committee is currently served by Josh Bloch, Jaakko Järvi, Sibylle Schupp, Dave Musser, Alex Stepanov, and Frank Tip. We aim to keep LCSD growing.

For the current workshop, we received 20 submissions, nine of which were accepted as technical papers, and additional four as position papers. The topics of the papers covered a wide area of the field of software libraries, including library evolution; abstractions for generic manipulation of complex mathematical structures; static analysis and type systems for software libraries; extensible languages; and libraries with run-time code generation capabilities. All papers were reviewed for soundness and relevance by three or more reviewers. The reviews were very thorough, for which we thank the members of the program committee. In addition to paper presentations, workshop activities included a keynote by Sean Parent, Adobe Inc. At the time of writing this foreword, we do not yet know the exact attendance of the workshop; the registrations received suggest close to 50 attendees.

We thank all authors, reviewers, and the organizing committee for their work in bringing about the LCSD workshop. We are very grateful to Sibylle Schupp, David Musser, and Jeremy Siek for their efforts in organizing the event, as well as to DongInn Kim and Andrew Lumsdaine for hosting the CyberChair system to manage the submissions. We also thank Tim Klinger and the OOPSLA workshop organizers for the help we received.

We hope you enjoy the papers, and that they generate new ideas leading to advances in this exciting field of research.

Jaakko Järvi
Joshua Bloch
(Program co-chairs)

Organization

Workshop Organizers

- **Josh Bloch**, Google Inc.
- **Jaakko Järvi**, Texas A&M University
- **David Musser**, Rensselaer Polytechnic Institute
- **Sibylle Schupp**, Chalmers University of Technology
- **Jeremy Siek**, Rice University

Program Committee

- **Dave Abrahams**, Boost Consulting
- **Olav Beckman**, Imperial College London
- **Hervé Brönnimann**, Polytechnic University
- **Cristina Gacek**, University of Newcastle upon Tyne
- **Douglas Gregor**, Indiana University
- **Paul Kelly**, Imperial College London
- **Doug Lea**, State University of New York at Oswego
- **Andrew Lumsdaine**, Indiana University
- **Erik Meijer**, Microsoft Research
- **Tim Peierls**, Prior Artisans LLC
- **Doug Schmidt**, Vanderbilt University
- **Anthony Simons**, University of Sheffield
- **Bjarne Stroustrup**, Texas A&M University and AT&T Labs
- **Todd Veldhuizen**, University of Waterloo

Contents

An Active Linear Algebra Library Using Delayed Evaluation and Runtime Code Generation

[Extended Abstract]

Francis P Russell, Michael R Mellor, Paul H J Kelly and Olav Beckmann

Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ, UK

ABSTRACT

Active libraries can be defined as libraries which play an active part in the compilation (in particular, the optimisation) of their client code. This paper explores the idea of delaying evaluation of expressions built using library calls, then generating code at runtime for the particular compositions that occur. We explore this idea with a dense linear algebra library for C++. The key optimisations in this context are loop fusion and array contraction.

Our library automatically fuses loops, identifies unnecessary intermediate temporaries, and contracts temporary arrays to scalars. Performance is evaluated using a benchmark suite of linear solvers from ITL (the Iterative Template Library), and is compared with MTL (the Matrix Template Library). Excluding runtime compilation overheads (caching means they occur only on the first iteration), for larger matrix sizes, performance matches or exceeds MTL – and in some cases is more than 60% faster.

1. INTRODUCTION

The idea of an "active library" is that, just as the library extends the language available to the programmer for problem solving, so the library should also extend the compiler. The term was coined by Czarnecki et al [5], who observed that active libraries break the abstractions common in conventional compilers. Active libraries are described in detail by Veldhuizen and Gannon [8].

This paper presents a prototype linear algebra library which we have developed in order to explore one interesting approach to building active libraries. The idea is to use a combination of delayed evaluation and runtime code generation to:

Delay library call execution Calls made to the library are used to build a "recipe" for the delayed computation. When execution is finally forced by the need for a result, the recipe will commonly represent a complex composition of primitive calls.

Generate optimised code at runtime Code is generated at runtime to perform the operations present in the delayed recipe. In order to obtain improved performance over a conventional library, it is important that the generated code should on average, execute faster than a statically generated counterpart in a conventional library. To achieve this, we apply optimisations that exploit the structure, semantics and context of each library call.

This approach has the advantages that:

- There is no need to analyse the client source code.
- The library user is not tied to a particular compiler.
- The interface of the library is not over complicated by the concerns of achieving high performance.
- We can perform optimisations across both statement and procedural bounds.
- The code generated for a recipe is isolated from client-side code - it is not interwoven with non-library code.

This last point is particularly important, as we shall see: because the structure of the code for a recipe is restricted in form, we can introduce compilation passes specially targeted to achieve particular effects.

The disadvantage of this approach is the overhead of runtime compilation and the infrastructure to delay evaluation. In order to minimise the first factor, we maintain a cache of previously generated code along with the recipe used to generate it. This enables us to reuse previously optimised and compiled code when the same recipe is encountered again.

There are also more subtle disadvantages. In contrast to a compile-time solution, we are forced to make online decisions about what to evaluate, and when. Living without static analysis of the client code means we don't know, for example, which variables involved in a recipe are actually live when the recipe is forced. We return to these issues later in the paper.

Our exploration covers the following ground:

1. We present an implementation of a C++ library for dense linear algebra which provides functionality sufficient to operate with the majority of methods available in the Iterative Template Library [6] (ITL), a set of templated linear iterative solvers for C++.
2. This implementation delays execution, generates code for delayed recipes at runtime, and then invokes a vendor C compiler at runtime - entirely transparently to the library user.
3. To avoid repeated compilation of recurring recipes, we cache compiled code fragments (see Section 4).
4. We implemented two optimisation passes which transform the code prior to compilation: loop fusion, and array contraction (see Section 5).
5. We introduce a scheme to predict, statistically, which intermediate variables are likely to be used after recipe execution; this is used to increase opportunities for array contraction (see Section 6).
6. We evaluate the effectiveness of the approach using a suite of iterative linear system solvers, taken from the Iterative Template Library (see Section 7).

Although the exploration of these techniques has used only dense linear algebra, we believe these techniques are more widely applicable. Dense linear algebra provides a simple domain in which to investigate, understand and demonstrate these ideas. Other domains we believe may benefit from these techniques include sparse linear algebra and image processing operations.

The contributions we make with this work are as follows:

- Compared to the widely used Matrix Template Library [7], we demonstrate performance improvements of up to 64% across our benchmark suite of dense linear iterative solvers from the Iterative Template Library. Performance depends on platform, but on a 3.2GHz Pentium 4 (with 2MB cache) using the Intel C Compiler, average improvement across the suite was 27%, once cached complied code was available.
- We present a cache architecture that finds applicable pre-compiled code quickly, and which supports annotations for adaptive re-optimisation.
- Using our experience with this library, we discuss some of the design issues involved in using the delayed-evaluation, runtime code generation technique.

We discuss related work in Section 8.

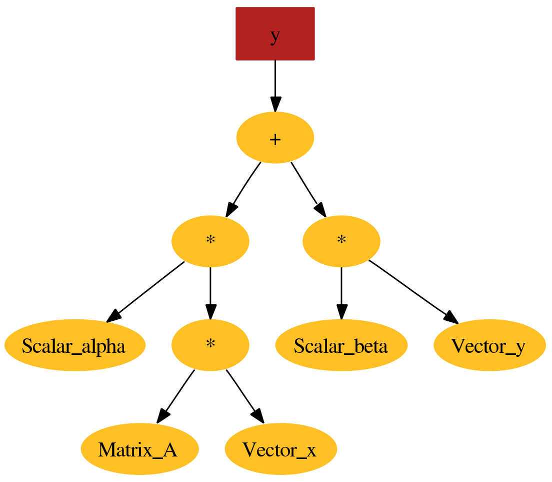


Figure 1: An example DAG. The rectangular node denotes a handle held by the library client. The expresssion represents the matrix-vector multiply function from Level 2 BLAS, $y = \alpha Ax + \beta y$.

2. DELAYING EVALUATION

Delayed evaluation provides the mechanism whereby we collect the sequences of operations we wish to optimise. We call the runtime information we obtain about these operations *runtime context information.*

This information may consist of values such as matrix or vector sizes, or the various relationships between successive library calls. Knowledge of dynamic values such as matrix and vector sizes allows us to improve the performance of the implementation of operations using these objects. For example, the runtime code generation system (see 3) can use this information to specialise the generated code. One specialisation we do is with loop bounds. We incorporate dynamically known sizes of vectors and matrices as constants in the runtime generated code.

Delayed evaluation in the library we developed works as follows:

- Delayed expressions built using library calls are represented as Directed Acyclic Graphs (DAGs).
- Nodes in the DAG represent either data values (literals) or operations to be performed on them.
- Arcs in the DAG point to the values required before a node can be evaluated.
- Handles held by the library client may also hold references to nodes in the expression DAG.
- Evaluation of the DAG involves replacing non-literal nodes with literals.
- When a node no longer has any nodes or handles depending on it, it deletes itself.

An example DAG is illustrated in Figure 1. The leaves of the DAG are literal values. The red node represents a handle held by the library client, and the other nodes represent delayed expressions. The three multiplication nodes do not have a handle referencing them. This makes them in effect, unnamed. When the expression DAG is evaluated, it is possible to optimise away these values entirely (their values are not required outside the runtime generated code). For expression DAGs involving matrix and vector operations, this enables us to reduce memory usage and improve cache utilisation.

Delayed evaluation also gives us the ability to optimise across successive library calls. This *Cross Component Optimisation* offers the possibility of greater performance than can be achieved by using separate hand-coded library functions.

Work by Ashby[1] has shown the effectiveness of cross component optimisation when applied to Level 1 Basic Linear Algebra Subprograms (BLAS) routines implemented in the language Aldor.

Unfortunately, with each successive level of BLAS, the improved performance available has been accompanied by an increase in complexity. BLAS level 3 functions typically take large a number of operands and perform a large number of more primitive operations simultaneously.

The burden then falls on the the library client programmer to structure their algorithms to make the most effective use of the BLAS interface. Code using this interface becomes more complex both to read and understand, than that using a simpler interface more oriented to the domain.

Delayed evaluation allows the library we developed to perform cross component optimisation at runtime, and also equip it with a simple interface, such as the one required by the ITL set of iterative solvers.

3. RUNTIME CODE GENERATION

Runtime code generation is performed using the TaskGraph[3] system. The TaskGraph library is a C++ library for dynamic code generation. A TaskGraph represents a fragment of code which can be constructed and manipulated at runtime, compiled, dynamically linked back into the host application and executed. TaskGraph enables optimisation with respect to:

Runtime Parameters This enables code to be specialised to its parameters and other runtime contextual information.

Platform SUIF-1, the Stanford University Intermediate Format is used as an internal representation in TaskGraph, making a large set of dependence analysis and restructuring passes available for code optimisation.

Characteristics of the TaskGraph approach include:

Simple Language Design TaskGraph is implemented in C++ enabling it to be compiled with a number of widely available compilers.

Explicit Specification of Dynamic Code TaskGraph requires the application programmer to construct the code explicitly as a data structure, as opposed to annotation of code or automated analysis.

Simplified C-like Sub-language Dynamic code is specified with the TaskGraph library via a sub-language similar to C. This language is implemented though extensive use of macros and C++ operator overloading. The language has first-class arrays, which facilitates dependence analysis.

An example function in C++ for generating a matrix multiply in the TaskGraph sub-language resembles a C implementation:

```
void TG_mm_ijk(unsigned int sz[2], TaskGraph &t)
{
  taskgraph(t) {
  tParameter(tArrayFromList(float, A, 2, sz));
  tParameter(tArrayFromList(float, B, 2, sz));
  tParameter(tArrayFromList(float, C, 2, sz));
  tVar(int, i); tVar(int, j); tVar(int, k);

  tFor(i, 0, sz[0]-1)
    tFor(j, 0, sz[1]-1)
      tFor(k, 0, sz[0] -1)
        C[i][j] += A[i][k] * B[k][j];
  }
}
```

The generated code is specialised to the matrix dimensions stored in the array sz. The matrix parameters A, B, and C are supplied when the code is executed.

Code generated by the library we developed is specialised in the same way. The constant loop bounds and array sizes make the code more amenable to the optimisations we apply later. These are described in Section 5.

4. CODE CACHING

As the cost of compiling the runtime generated code is extremely high (compiler execution time in the order of tenths of a second) it was important that this overhead be minimised.

Related work by Beckmann[4] on the efficient placement of data in a parallel linear algebra library cached execution plans in order to improve performance. We adopt a similar strategy in order to reuse previously compiled code. We maintain a cache of previously encountered recipes along with the compiled code required to execute them. As any caching system would be invoked at every force point within a program using the library, it was essential that checking for cache hits would be as computationally inexpensive as possible.

As previously described, delayed recipes are represented in the form of directed acyclic graphs. In order to allow the fast resolution of possible cache hits, all previously cached recipes are associated with a hash value. If recipes already exist in the cache with the same hash value, a full check is then be performed to see if the recipes match.

Time and space constraints were of paramount importance in the development of the caching strategy and certain concessions were made in order that it could be performed quickly. The primary concession was that both hash calculation and isomorphism checking occur on flattened forms of the delayed expression DAG ordered using a topological sort.

This causes two limitations:

- It is impossible to detect the situation where the presence of commutative operations allow two differently structured delayed expression DAGs to be used in place of each other.
- As there can be more than one valid topological sort of a DAG, it is possible for multiple identically structured expression DAGs to exist in the code cache.

As we will see later, neither of these limitations significantly affects the usefulness of the cache, but first we will briefly describe the hashing and isomorphism algorithms.

Hashing occurs as follows:

- Each DAG node in the sorted list is assigned a value corresponding to its position in the list.
- A hash value is calculated for each node corresponding to its type and the other nodes in the DAG it depends on. References to other nodes are hashed using the numerical values previously assigned to each node.
- The hash values of all the nodes in the list are combined together in list order using a non-commutative function.

Isomorphism checking works similarly:

- Nodes in the sorted lists for each graph are assigned a value corresponding to their location in their list.
- Both lists are checked to be the same size.
- The corresponding nodes from both lists are checked to be the same type, and any nodes they reference are checked to see if they have been assigned the same numerical value.

Isomorphism checking in this manner does not require that a mapping be found between nodes in the two DAGs involved (this is already implied by each node's location in the sorted list for each graph). It only requires determining whether the mapping is valid.

If the maximum number of nodes a node can refer to is bounded (maximum of two for a library with only unary and binary operators) then both hashing and isomorphism checking between delayed expression DAGs can be performed in linear time with respect to the number of nodes in the DAG.

We previously stated that the limitations imposed by using a flattened representation of an expression DAG does not significantly effect the usefulness of the code cache. We expect the code cache to be at its most useful when the same sequence of library calls are repeatedly encountered (as in a loop). In this case, the generated DAGs will have identical structures, and the ability to detect non-identical DAGs that compute the same operation provides no benefit.

The second limitation, the need for identical DAGs matched by the caching mechanism to also have the same topological sort is more important. To ensure this, we store the dependency information held at each DAG node using lists rather than sets. By using lists, we can guarantee that two DAGs constructed in an identical order, will also be traversed in the same order. Thus, when we come to perform our topological sort, the nodes from both DAGs will be sorted in the same order.

The code caching mechanism discussed, whilst it cannot recognise all opportunities for reuse, is well suited for detecting repeatedly generated recipes from client code. For the ITL set of iterative solvers, compilation time becomes a constant overhead, regardless of the number of iterations executed.

5. LOOP FUSION AND ARRAY CONTRACTION

We implemented two optimisations using the TaskGraph back-end, SUIF. A brief description of these transformations follow.

Loop fusion[2] can lead to an improvement in performance when the fused loops use the same data. As the data is only loaded into the cache once, the fused loops take less time to execute than the sequential loops. Alternatively, if the fused loops use different data, it can lead to poorer performance, as the data used by the fused loop displace each each other in the cache.

A brief example involving two vector additions. Before loop fusion:

```
for (int i=0; i<100; ++i)
  a[i] = b[i] + c[i];

for(int i=0; i<100; ++i)
  e[i] = a[i] + d[i];
```

After loop fusion:

```
for (int i=0; i<100; ++i) {
  a[i] = b[i] + c[i];
  e[i] = a[i] + d[i];
}
```

In this example, after fusion, the value stored in vector *a* can be reused for the calculation of *e*.

The loop fusion pass implemented in our library requires that the loop bounds be constant. We can afford this limitation because our runtime generated code has already been specialised with loop bound information. Our loop fuser does not possess a model of cache locality to determine which loop fusions are likely to lead to improved performance. Despite this, visual inspection of the code generated during execution of the iterative solvers indicates that the fused loops commonly use the same data. This is most likely due to the structure of the dependencies involved in the operations required for the iterative solvers.

Array contraction[2] is one of a number of memory access transformations designed to optimise the memory access of a program. It allows the dimensionality of arrays to be reduced, decreasing the memory taken up by compiler generated temporaries, and the number of cache lines referenced. It is often facilitated by loop fusion.

Another example. Before array contraction:

```
for (int i=0; i<100; ++i) {
  a[i] = b[i] + c[i];
  e[i] = a[i] + d[i];
}
```

After array contraction:

```
for (int i=0; i<100; ++i) {
  a = b[i] + c[i];
  e[i] = a + d[i];
}
```

Here, the array a can be reduced to a scalar value as long as it is not required by any code following the two fused loops.

We use this to technique to optimise away temporary matrices or vectors in the runtime generated code. This is important because the DAG representation of the delayed operations does not hold information on what memory can be reused. However, we can determine whether or not each node in the DAG is referenced by the client code, and if it is not, it can be allocated locally to the runtime generated code and possibly be optimised away. For details of other memory access transformations, consult Bacon et al.[2].

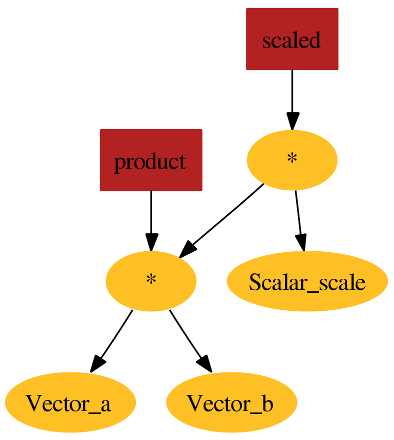
6. LIVENESS ANALYSIS

When analysing the runtime generated code produced by the iterative solvers, it became apparent that a large number of vectors were being passed in as parameters. We realised that by designing a system to recover runtime information, we had lost the ability to use static information.

Consider the following code that takes two vectors, finds their cross product, scales the result and prints it:

```
void printScaledCrossProduct(Vector<float> a,
                             Vector<float> b,
                             Scalar<float> scale)
{
  Vector<float> product = cross(a, b);
  Vector<float> scaled = mul(product, scale);
  print(scaled);
}
```

This operation can be represented with the following DAG:



The value pointed to by the handle *product* is never required by the library client. From the client's perspective the value is dead, but the library must assume that any value which has a handle may be required later on. Values required by the library client cannot be allocated locally to the runtime generated code, and therefore cannot be optimised away through techniques such as array contraction. Runtime liveness analysis permits the library to make estimates about the liveness of nodes in repeatedly executed DAGs, and allow them to be allocated locally to runtime generated code if it is believed they are dead, regardless of whether they have a handle.

Having already developed a system for recognising repeatedly executed delayed expression DAGs, we developed a similar mechanism for associating collected liveness information with expression DAGs.

Nodes in each generated expression DAG are instrumented and information collected on whether the values are live or dead. The next time the same DAG is encountered, the previously collected information is used to annotate each node in the DAG with an estimate with regards to whether it is live or dead. As the same DAG is repeatedly encountered, statistical information about the liveness of each node is built up.

If an expression DAG node is estimated to be dead, then it can be allocated locally to the runtime generated code and possibly optimised away. This could lead to a possible performance improvement. Alternatively, it is also possible that the expression DAG node is not dead, and its value is required by the library client at a later time. As the value was not saved the first time it was computed, the value

Option	Description
-O3	Enables the most aggressive level of optimisation including loop and memory access transformations, and prefetching.
-restrict	Enables the use of the *restrict* keyword for qualifying pointers. The compiler will assume that data pointed to by a *restrict* qualified pointer will only be accessed though that pointer in that scope. As the *restrict* keyword is not used anywhere in the runtime generated code, this should have no effect.
-ansi-alias	Allows icc to perform more aggressive optimisations if the program adheres to the ISO C aliasing rules.
-xW	Generate code specialised for Intel Pentium 4 and compatible processors.

Table 1: The options supplied to Intel C/C++ compilers and their meanings.

must be computed again. This could result in a performance decrease of the client application if such a situation occurs repeatedly.

7. PERFORMANCE EVALUATION

We evaluated the performance of the library we developed using solvers from the ITL set of templated iterative solvers running on dense matrices of different sizes. The ITL provides templated classes and methods for the iterative solution of linear systems, but not an implementation of the linear algebra operations themselves. ITL is capable of utilising a number of numerical libraries, requiring only the use of an appropriate header file to map the templated types and methods ITL uses to those specific to a particular library. ITL was modified to use our library through the addition of a header file and other minor modifications.

We compare the performance of our library against the Matrix Template Library[7]. ITL already provides support for using MTL as its numerical library. We used version 9.0 of the Intel C compiler for runtime code generation, and version 9.0 of the Intel C++ compiler for compiling the MTL benchmarks. The options passed to the Intel C and C++ compilers are described in Table 1.

We will discuss the observed effects of the different optimisation methods we implemented, and we conclude with a comparison against the same benchmarks using MTL.

We evaluated the performance of the solvers on two architectures, both running Mandrake Linux version 10.2:

1. Pentium IV processor running at 3.0GHz with Hyperthreading, 512 KB L2 cache and 1 GB RAM.
2. Pentium IV processor running at 3.2GHz with Hyperthreading, 2048 KB L2 cache and 1 GB RAM.

The first optimisation implemented was loop fusion. The majority of benchmarks did not show any noticeable improvement with this optimisation. Visual inspection of the runtime generated code showed multiple loop fusions had occurred between vector-vector operations but not between matrix-vector operations. As we were working with dense matrices, we believe the lack of improvement was due to the fact that the vector-vector operations were $O(n)$ and the matrix-vector multiplies present in each solver were $O(n^2)$.

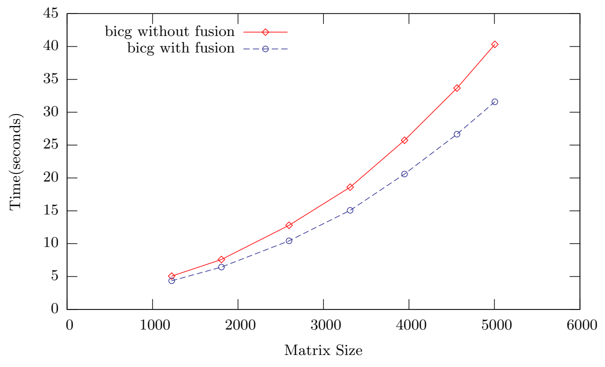


Figure 2: 256 iterations of the BiConjugate Gradient (BiCG) solver running on architecture 1 with and without loop fusion, including compilation overhead.

The exception to this occurred with the BiConjugate Gradient solver. In this case the loop fuser was able to fuse a matrix-vector multiply and a transpose matrix-vector multiply with the result that the matrix involved was only iterated over once for both operations. A graph of the speedup obtained across matrix sizes is shown in Figure 2.

The second optimisation implemented was array contraction. We only evaluated this in the presence of loop fusion as the former is often facilitated by the latter. The array contraction pass did not show any noticeable improvement on any of the benchmarks applications. On visual inspection of the runtime generated code we found that the array contractions had occurred on vectors, and these only affected the vector-vector operations. This is not surprising seeing that only one matrix was used during the execution of the linear solvers and as it was required for all iterations, could not be optimised away in any way. We believe that were we to extend the library to handle sparse matrices, we would be able to see greater benefits from both the loop fusion and array contraction passes.

The last technique we implemented was runtime liveness analysis. This was used to try to recognise which expression DAG nodes were dead to allow them to be allocated locally to runtime generated code.

The runtime liveness analysis mechanism was able to find vectors in three of the five iterative solvers that could be allocated locally to the runtime generated code. The three solvers had an average of two vectors that could be optimised away, located in repeatedly executed code. Unfortunately, usage of the liveness analysis mechanism resulted in an overall decrease in performance. We discovered this to be because the liveness mechanism resulted in extra constant

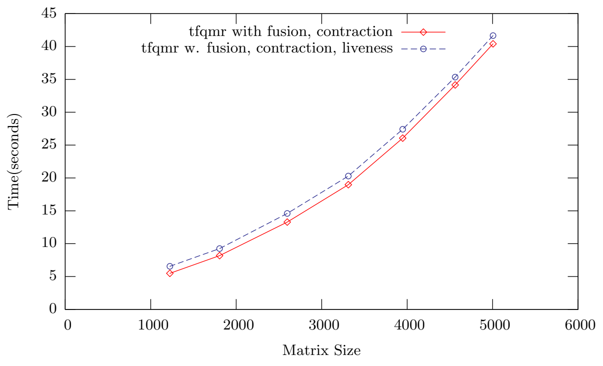


Figure 3: 256 iterations of the Transpose Free Quasi-Minimal Residual (TFQMR) solver running on architecture 1 with and without the liveness analysis enabled, including compilation overhead.

overhead due to more compiler invocations at the start of the iterative solver. This was due to the statistical nature of the liveness prediction, and the fact that as it changed its estimates with regard to whether a value was live or dead, a greater number of runtime generated code fragments had to be produced. Figure 3 shows the constant overhead of the runtime liveness mechanism running on the Transpose Free Quasi-Minimal Residual solver.

We also compared the library we developed against the Matrix Template Library, running the same benchmarks. We enabled the loop fusion and array contraction optimisations, but did not enable the runtime liveness analysis mechanism because of the overhead already discussed. We found the performance increase we obtained to be architecture specific.

On architecture 1 (excluding compilation overhead) we only obtained an average of 2% speedup across the solver and matrix sizes we tested. The best speedup we obtained on this architecture (excluding compilation) was on the BiConjugate Gradient solver, which had a 38% speedup on a 5005x5005 matrix. It should be noted that the BiConjugate Gradient solver was the one for which loop fusion provided a significant benefit.

On architecture 2 (excluding compilation overhead) we obtained an average 27% speedup across all iterative solvers and matrix sizes. The best speedup we obtained was again on the BiConjugate Gradient solver, which obtained a 64% speedup on a 5005x5005 matrix. A comparison of the BiConjugate Gradient solver against MTL running on architecture 2 is shown in Figure 4.

In the figures just quoted, we excluded the runtime compilation overhead, leaving just the performance increase in the numerical operations. As the iterative solvers use code caching, the runtime compilation overhead is independent of the number of iterations executed. Depending on the number of iterations executed, the performance results including compilation overhead would vary. Furthermore, mechanisms such as a persistent code cache could allow the compilation

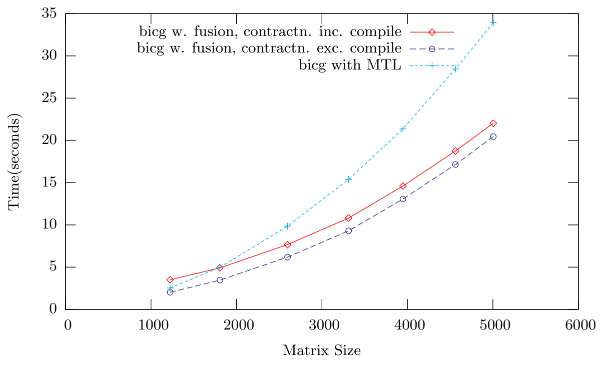


Figure 4: 256 iterations of the BiConjugate Gradient (BiCG) solver using our library and MTL, running on architecture 2. Execution time for our library is shown with and without runtime compilation overhead.

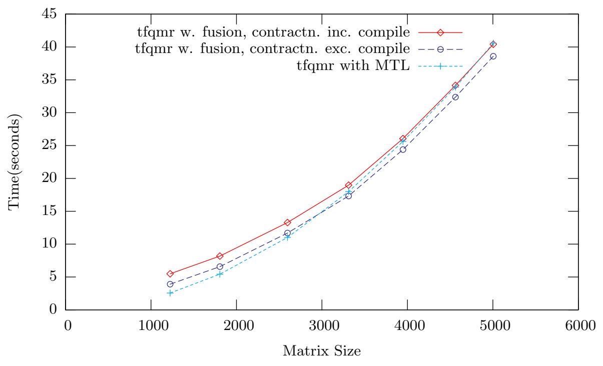


Figure 5: 256 iterations of the Transpose Free Quasi-Minimal Residual (TFQMR) solver using our library and MTL, running on architecture 1. Execution time for our library is shown with and without runtime compilation overhead.

overheads to be significantly reduced. These overheads will be discussed in Section 9.

Figure 5 shows the execution time of Transpose Free Quasi-Minimal Residual solver running on architecture 1 with MTL and the library we developed. Figure 6 shows the execution time of the same benchmark running on architecture 2. For our library, we show the execution time including and excluding the runtime compilation overhead.

Our results appear to show that cache size is extremely important with respect to the performance we can obtain from our runtime code generation technique. On our first architecture, we were unable to achieve any significant performance increase over MTL but on architecture 2, which had a 4x larger L2 cache, the increases were much greater. We believe this is due to the Intel C Compiler being better able to utilise the larger cache sizes, although we have not yet managed to determine what characteristics of the runtime

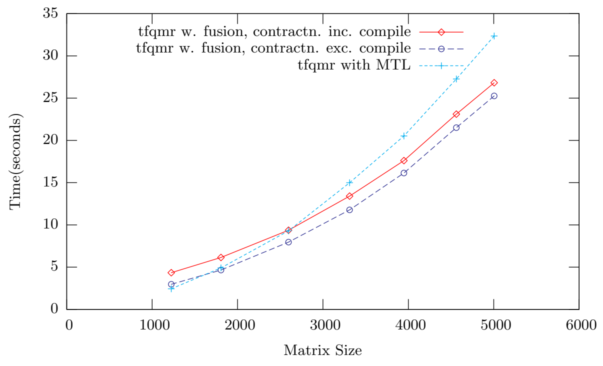


Figure 6: 256 iterations of the Transpose Free Quasi-Minimal Residual (TFQMR) solver using our library and MTL, running on architecture 2. Execution time for our library is shown with and without runtime compilation overhead.

generated code allowed it to be optimised more effectively than the same benchmark using MTL.

8. RELATED WORK

Delayed evaluation has been used previously to assist in improving the performance of numerical operations. Work done by Beckmann[4] has used delayed evaluation to optimise data placement in a numerical library for a distributed memory multicomputer. The developed library also has a mechanism for recognising repeated computation and reusing previously generated execution plans. Our library works similarly, except both our optimisations and searches for reusable execution plans target the runtime generated code.

Other work by Beckmann uses the TaskGraph library[3] to demonstrate the effectiveness of specialisation and runtime code generation as a mechanism for improving the performance of various applications. The TaskGraph library is used to generate specialised code for the application of a convolution filter to an image. As the size and the values of the convolution matrix are known at the runtime code generation stage, the two inner loops of the convolution can be unrolled and specialised with the values of the matrix elements. Another example shows how a runtime search can be performed to find an optimal tile size for a matrix multiply. TaskGraph is also used as the code generation mechanism for our library.

Work by Ashby[1] investigates the effectiveness of cross component optimisation when applied to Level 1 BLAS routines. BLAS routines written in Aldor are compiled to an intermediate representation called FOAM. During the linking stage, the compiler is able to perform extensive levels of cross component optimisation. It is these form of optimisations that we attempt to exploit to allow us to develop a technique for generating high performance code without sacrificing interface simplicity.

9. CONCLUSIONS AND FURTHER WORK

One conclusion that can be made from this work is the importance of cross component optimisation. Numerical libraries such as BLAS have had to adopt a complex interface to obtain the performance they provide. Libraries such as MTL have used unconventional techniques to work around the limitations of conventional libraries to provide both simplicity and performance. The library we developed also uses unconventional techniques, namely delayed evaluation and runtime code generation, to work around these limitations. The effectiveness of this approach provides more compelling evidence towards the benefits of Active Libraries[5].

We have shown how a framework based on delayed evaluation and runtime code generation can achieve high performance on certain sets of applications. We have also shown that this framework permits optimisations such as loop fusion and array contraction to be performed on numerical code where it would not be possible otherwise, due to either compiler limitations (we do not believe GCC or ICC will perform array contraction or loop fusion) or the difficulty of performing these optimisations across interprocedural bounds.

Whilst we have concentrated on the benefits such a framework can provide, we have paid less attention to the situations in which it can perform poorly. The overhead of the delayed evaluation framework, expression DAG caching and matching and runtime compiler invocation will be particularly significant for programs which have a large number of force points, and/or use small sized matrices and vectors. A number of these overheads can be minimised. Two techniques to reduce these overheads are:

Persistent code caching This would allow cached code fragments to persist across multiple executions of the same program and avoid compilation overheads on future runs.

Evaluation using BLAS or static code Evaluation of the delayed expression DAG using BLAS or statically compiled code would allow the overhead of runtime code generation to be avoided when it is believed that runtime code generation would provide no benefit.

Investigation of other applications using numerical linear algebra would be required before the effectiveness of these techniques can be evaluated.

Other future work for this research includes:

Sparse Matrices Linear iterative solvers using sparse matrices have many more applications than those using dense ones, and would allow the benefits of loop fusion and array contraction to be further investigated.

Client Level Algorithms Currently, all delayed operations correspond to nodes of specific types in the delayed expression DAG. Any library client needing to perform an operation not present in the library would either need to extend it (difficult), or implement it using element level access to the matrices or vectors involved (poor performance). The ability of the client to specify algorithms to be delayed would significantly improve the usefulness of this approach.

Improved Optimisations We implemented limited methods of loop fusion and array contraction. Other optimisations could improve the code's performance further, and/or reduce the effect the quality of the vendor compiler used to compile the runtime generated code has on the performance of the resulting runtime generated object code.

10. REFERENCES

[1] T. J. Ashby, A. D. Kennedy, and M. F. P. O'Boyle. Cross component optimisation in a high level category-based language. In *Euro-Par*, pages 654–661, 2004.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[3] O. Beckmann, A. Houghton, M. Mellor, and P. H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, pages 291–306, 2003.

[4] O. Beckmann and P. H. J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In *LCR98: Languages, Compilers and Run-time Systems for Scalable Computers*, number 1511 in LNCS, pages 123–138. Springer-Verlag, May 1998.

[5] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In *Generic Programming. Proceedings*, number 1766 in LNCS, pages 25–39, 2000.

[6] L.-Q. Lee, A. Lumsdaine, and J. Siek. Iterative Template Library. http://www.osl.iu.edu/download/research/itl/slides.ps.

[7] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE*, pages 59–70, 1998.

[8] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat

Lubomir Bourdev
Adobe Systems Inc.
lbourdev@adobe.com

Jaakko Järvi
Texas A&M University
jarvi@cs.tamu.edu

Abstract

Generic programming using C++ results in code that is efficient but inflexible. The inflexibility arises, because the exact types of inputs to generic functions must be known at compile time. We show how to achieve run-time polymorphism without compromising performance by instantiating the generic algorithm with a comprehensive set of possible parameter types, and choosing the appropriate instantiation at run time. The major drawback of this approach is excessive template bloat, generating a large number of instantiations, many of which are identical at the assembly level. We show practical examples in which this approach quickly reaches the limits of the compiler. Consequently, we combine the method of run-time polymorphism for generic programming with a strategy for reducing the amount of necessary template instantiations. We report on using our approach in GIL, Adobe's open source Generic Image Library. We observed notable reduction, up to 70% at times, in executable sizes of our test programs. Even with compilers that perform aggressive template hoisting at the compiler level, we achieve notable code size reduction, due to significantly smaller dispatching code. The framework draws from both the generic programming and generative programming paradigm, using static metaprogramming to fine tune the compilation of a generic library. Our test bed, GIL, is deployed in a real world industrial setting, where code size is often an important factor.

Categories and Subject Descriptors D.3.3 [*Programming Techniques*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Techniques*]: Language Constructs and Features—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Performance, Languages

Keywords generic programming, C++ templates, template bloat, template metaprogramming

1. Introduction

Generic programming, pioneered by Musser and Stepanov [19], and introduced to C++ with the STL [24], aims at expressing algorithms at an abstract level, such that the algorithms apply to as broad class of data types as possible. A key idea of generic programming is that this abstraction should incur no performance degradation: once a generic algorithm is specialized for some concrete data types, its performance should not differ from a similar algorithm written directly for those data types. This principle is often referred to as *zero abstraction penalty*. The paradigm of generic programming has been successfully applied in C++, evidenced, e.g., by the STL, the Boost Graph Library (BGL) [21], and many other generic libraries [3,5,11,20,22,23]. One factor contributing to this success is the compilation model of templates, where specialized code is generated for every different instance of a template. We refer to this compilation model as the *instantiation model*.

We note that the instantiation model is not the only mechanism for compiling generic definitions. For example, in Java [13] and Eiffel [10] a generic definition is compiled to a single piece of byte or native code, used by all instantiations of the generic definition. C# [9, 18] and the ECMA .NET framework delay the instantiation of generics until run time. Such alternative compilation models address the code bloat issue, but may be less efficient or may require run-time compilation. They are not discussed in this paper.

With the instantiation model, zero abstraction penalty is an attainable goal: later phases of the compilation process make no distinction between code generated from a template instantiation and non-template code written directly by the programmer. Thus, function calls can be resolved statically, which enables inlining and other optimizations for generic code. The instantiation model, however, has other less desirable characteristics, which we focus on in this paper.

In many applications the exact types of objects to be passed to generic algorithms are not known at compile time. In C++ all template instantiations and code generation that they trigger occur at compile time—*dynamic dispatching* to templated functions is not (directly) supported. For efficiency, however, it may be crucial to use an algorithm instantiated for particular concrete types.

In this paper, we describe how to instantiate a generic algorithm with all possible types it may be called with, and generate code that dispatches at run time to the right instantiation. With this approach, we can combine the flexibility of dynamic dispatching and performance typical for the instantiation model: the dispatching occurs only once per call to a generic algorithm, and has thus a negligible cost, whereas the individual instantiations of the algorithms are compiled and fully optimized knowing their concrete input types. This solution, however, leads easily to excessive number of template instantiations, a problem known as *code bloat* or *template bloat*. In the instantiation model, the combined size of the instantiations grows with the number of instantiations: there is typically no code sharing between instantiations of the same templates with different types, regardless of how similar the generated code is.[1]

LCSD '06 October 22nd, Portland, Oregon.
ACM [to be supplied].

[1] At least one compiler, Visual Studio 8, has advanced heuristics that can optimize for code bloat by reusing the body of assembly-level identical

This paper reports on experiences of using the generic programming paradigm in the development of the Generic Image Library (GIL) [5] in the Adobe Source Libraries [1]. GIL supports several image formats, each represented internally with a distinct type. The static type of an image manipulated by an application using GIL is often not known; the type assigned to an image may, e.g., depend on the format it was stored on the disk. Thus, the case described above manifests in GIL: an application using GIL must instantiate the relevant generic functions for all possible image types and arrange that the correct instantiations are selected based on the arguments' dynamic types when calling these functions. Following this strategy blindly may lead to unmanageable code bloat. In particular, the set of instantiations increases exponentially with the number of image type parameters that can be varied independently in an algorithm. Our experience shows that the number of template instantiations is an important design criterion in developing generic libraries.

We describe the techniques and the design we use in GIL to ensure that specialized code for all performance critical program parts is generated, but still keep the number of template instantiations low. Our solution is based on the realization that even though a generic function is instantiated with different type arguments, the generated code is in some cases identical. We describe mechanisms that allow the different instantiations to be replaced with a single common instantiation. The basic idea is to decompose a complex type into a set of orthogonal parameter dimensions (with image types, these include color space, channel depth, and constness) and identify which parameters are important for a given generic algorithm. Dimensions irrelevant for a given operation can be cast to a single "base" parameter value. Note that while this technique is presented as a solution to dealing with code bloat originating from the "dynamic dispatching" we use in GIL, the technique can be used in generic libraries without a dynamic dispatching mechanism as well.

In general, a developer of a software library and the technologies supporting library development are faced with many, possibly competing, challenges, originating from the vastly different context the libraries can be used. Considering GIL, for example, an application such as Adobe Photoshop requires a library flexible enough to handle the variation of image representations at run time, but also places strict constraints on performance. Small memory footprint, however, becomes essential when using GIL as part of a software running on a small device, such as a cellular phone or a PDA. Basic software engineering principles ask for easy extensibility, etc. The design and techniques presented in this paper help in building generic libraries that can combine efficiency, flexibility, extensibility, and compactness.

C++'s template system provides a programmable sub-language for encoding compile-time computations, the uses of which are known as *template metaprogramming* (see e.g. [25], [8, §.10]). This form of generative programming proved to be crucial in our solution: the process of pruning unnecessary instantiations is orchestrated with template metaprograms. In particular, for our metaprogramming needs, we use the Boost Metaprogramming Library (MPL) [2, 14] extensively. In the presentation, we assume some familiarity with the basic principles of template metaprogramming in C++.

The structure of the paper is as follows. Section 2 describes typical approaches to fighting code bloat. Section 3 gives a brief introduction to GIL, and the code bloat problems therein. Section 4 explains the mechanism we use to tackle code bloat, and Section 5 describes how to apply the mechanism with dynamic dispatching to generic algorithms. We report experimental results in Section 6, and conclude in Section 7.

functions. In the results section we demonstrate that our method can result in noticeable code size reduction even in the presence of such heuristics.

2. Background

One common strategy to reduce code bloat associated with the instantiation model is *template hoisting* (see e.g. [6]). In this approach, a class template is split into a non-generic base class and a generic derived class. Every member function that does not depend on any of the template parameters is moved, hoisted, into the base class; also non-member functions can be defined to operate directly on references or pointers to objects of the base-class type. As a result, the amount of code that must be generated for each different instantiation of the derived class decreases. For example, red-black trees are used in the implementation of *associative containers* map, multimap, set, and multiset in the C++ Standard Library [15]. Because the tree balancing code does not need to depend on the types of the elements contained in these containers, a high-quality implementation is expected to hoist this functionality to non-generic functions. The GNU Standard C++ Library v3 does exactly this: the tree balancing functions operate on pointers to a non-generic base class of the tree's node type.

In the case of associative containers, the tree node type is split into a generic and non-generic part. It is in principle possible to split a template class into several layers of base classes, such that each layer reduces the number of template parameters. Each layer then potentially has less type variability than its subclasses, and thus two different instantiations of the most derived class may coalesce to a common instantiation of a base class. Such designs seem to be rare.

Template hoisting within a class hierarchy is a useful technique, but it allows only a single way of splitting a data type into sub-parts. Different generic algorithms are generally concerned with different aspects of a data-type. Splitting a data type in a certain way may suit one algorithm, but will be of no help for reducing instantiations of other algorithms. In the framework discussed in this paper, the library developer, possibly also the client of a library, can define a partitioning of data-types, where a particular algorithm needs to be instantiated only with one representative of each equivalence class in the partition.

We define the partition such that differences between types that do not affect the operation of an algorithm are ignored. One common example is pointers - for some algorithms the pointed type is important, whereas for others it is ok to cast to void∗. A second example is differences due to constness (consider STL's iterator and const_iterator concept). The generated code for invoking a *non-modifying* algorithm (one which accepts immutable iterators) with mutable iterators will be identical to the code generated for an invocation with immutable iterator. Some algorithms need to operate bitwise on their data, whereas others depend on the type of data. For example, assignment between a pair of pixels is the same regardless of whether they are CMYK or RGBA pixels, whereas the type of pixel matters to an algorithm that sets the color to white, for example.

3. Generic Image Library

The Generic Image Library (GIL) is Adobe's open source image processing library [5]. GIL addresses a fundamental problem in image processing projects — operations applied to images (such as copying, comparing, or applying a convolution) are logically the same for all image types, but in practice image representations in memory can vary significantly, which often requires providing multiple variations of the same algorithm. GIL is used as the framework for several new features planned for inclusion in the next version of Adobe Photoshop. GIL is also being adopted in several other imaging projects inside Adobe. Our experience with these efforts show that GIL helps to reduce the size of the core image manipulation source code significantly, as much as 80% in a particular case.

Images are 2D (or more generally, n-dimensional) arrays of pixels. Each pixel encodes the color at the particular point in the image. The color is typically represented as the values of a set of *color channels*, whose interpretation is defined by a *color space*. For example, the color red can be represented as 100% red, 0% green, and 0% blue using the RGB color space. The same color in the CMYK color space can be approximated with 0% cyan, 96% magenta, 90% yellow, and 0% black. Typically all pixels in an image are represented with the same color space.

GIL must support significant variation within image representations. Besides color space, images may vary in the ordering of the channels in memory (RGB vs. BGR), and in the number of bits (depth) of each color channel and its representation (8 bit vs. 32 bit, unsigned char vs. float). Image data may be provided in *interleaved* form (RGBRGBRGB...) or in *planar* form where each color plane is separate in memory (RRR..., GGG... BBB...); some algorithms are more efficient in planar form whereas others perform better in interleaved form. In some image representations each row (or the color planes) may be aligned, in which case a gap of unused bytes may be present at the end of each row. There are representations where pixels are not consecutive in memory, such as a sub-sampled view of another image that only considers every other pixel. The image may represent a rectangular sub-image in another image or an upside-down view of another image, for example. The pixels of the image may require some arbitrary transformation (for example an 8-bit RGB view of 16-bit CMYK data). The image data may not be at all in memory (a virtual image, or an image inside a JPEG file). The image may be synthetic, defined by an arbitrary function (the Mandelbrot set), and so forth.

Note that GIL makes a distinction between *images* and *image views*. Images are containers that own their pixels, views do not. Images can return their associated views and GIL algorithms operate on views. For the purpose of this paper, these differences are not significant, and we use the terms image and image views (or just views) interchangeably.

The exact image representation is irrelevant to many image processing algorithms. To compare two images we need to loop over the pixels and compare them pairwise. To copy one image into another we need to copy every pixel pairwise. To compute the histogram of an image, we need to accumulate the histogram data over all pixels. To exploit these commonalities, GIL follows the generic programming approach, exemplified by the STL, and defines abstract representations of images as *concepts*. In the terminology of generic programming, a concept is the formalization of an abstraction as a set of requirements on a type (or types) [4, 16]. A type that implements the requirements of a concept is said to *model* the concept. Algorithms written in terms of image concepts work for images in any representation that model the necessary concepts. By this means, GIL avoids multiple definitions for the same algorithm that merely accommodate for inessential variation in the image representations.

GIL supports a multitude of image representations, for each of which a distinct typedef is provided. Examples of these types are:

- rgb8_view_t: 8-bit mutable interleaved RGB image
- bgr16c_view_t: 16-bit immutable interleaved BGR image
- cmyk32_planar_view_t: 32-bit mutable planar CMYK image
- lab8c_step_planar_view_t: 8-bit immutable LAB planar image in which the pixels are not consecutive in memory

The actual types associated with these typedefs are somewhat involved and not presented here.

GIL represents color spaces with distinct types. The naming of these types is as expected: rgb_t stands for the RGB color space, cmyk_t for the CMYK color space, and so forth. Channels can be represented in different permutations of the same set of color values. For each set of color values, GIL identifies a single color space as the *primary* color space — its permutations are *derived* color spaces. For example, rgb_t is a primary color space and bgr_t is its *derived* color space.

GIL defines two images to be *compatible* if they have the same set and type of channels. That also implies their color spaces must have the same primary color space. Compatible images may vary any other way - planar vs. interleaved organization, mutability, etc. For example, an 8-bit RGB planar image is compatible with an 8-bit BGR interleaved image. Compatible images may be copied from one another and compared for equality.

3.1 GIL Algorithms

We demonstrate the operation of GIL with a simple algorithm, copy_pixels(), that copies one image view to another. Here is one way to implement it:[2]

```
template <typename View1, typename View2>
void copy_pixels(const View1& src, const View2& dst) {
  std::copy(src.begin(), src.end(), dst.begin());
}
```

A requirement of copy_pixels is that the two image view types be compatible and have the same dimensions, and that the destination be mutable. An attempt to instantiate copy_pixels with incompatible images results in a compile-time error.

Each GIL image type supports the begin() and end() member functions as defined in the STL's Container concept. Thus the body of the algorithm just invokes the copy() algorithm from the C++ standard library. If we expand out the std::copy() function, copy_pixels becomes:

```
template <typename View1, typename View2>
void copy_pixels(const View1& src, const View2& dst) {
   typedef typename View1::iterator src_it = src.begin();
   typedef typename View2::iterator dst_it = dst.begin();
   while (src_it != dst.end()) {
      *dst_it++ = *src_it++;
   }
}
```

Each image type is required to have an associated iterator type that implements iteration over the image's pixels. Furthermore, each pixel type must support assignment. Note that the source and target images can be of different (albeit compatible) types, and thus the assignment may include a (lossless) conversion from one pixel type to another. These elementary operations are implemented differently by different image types. A built-in pointer type can serve as the iterator type of a simple interleaved image[3], whereas in a planar RGB image it may be a bundle of three pointers to the corresponding color planes. The iterator increment operator ++ for interleaved images may resolve to a pointer increment, for step images to advancing a pointer by a given number of bytes, and for a planar RGB iterator to incrementing three pointers. The dereferencing operator $*$ for simple interleaved images returns a reference type; for planar RGB images it returns a *planar reference proxy* object containing three references to the three channels. For a complex image type, such as one representing an RGB view over CMYK data, the dereferencing operator may perform color conversion.

[2] Note that GIL image views don't own the pixels and don't propagate their constness to the pixels, which explains why we take the destination as a const reference. Mutability is incorporated into the image view type.

[3] Assuming the image has no gap at the end of each row

Due to the instantiation model, the calls to the implementations of the elementary image operations in GIL algorithms can be resolved statically and usually inlined, resulting in an efficient algorithm specialized for the particular image types used. GIL algorithms are targeted to match the performance of code hand-written for a particular image type. Any difference in performance from that of hand-written code is usually due to abstraction penalty, for example, the compiler failing to inline a forwarding function, or failing to pass small objects of user-defined types in registers. Modern compilers exhibit zero abstraction penalty with GIL algorithms in many common uses of the library.

3.2 Dynamic dispatching in GIL

Sometimes the exact image type with which the algorithm is to be called is unknown at compile time. For this purpose, GIL implements the variant template, i.e. a discriminated union type. The implementation is very similar to that of the Boost Variant Library [12]. One difference is that the Boost variant template can be instantiated with an arbitrary number of template arguments, while GIL variant accepts exactly one argument [4]. This argument itself represents a collection of types and it must be a model of the Random Access Sequence concept, defined in MPL. For example, the vector template in MPL models this concept. A variant object instantiated with an MPL vector holds an object whose type can be any one of the types contained in the type vector.

Populating a variant with image types, and instantiating another template in GIL, any_image_view, with the variant, yields a GIL image type that can hold any of the image types in the variant. Note the difference to polymorphism via inheritance and dynamic dispatching: in polymorphism via virtual member functions, the set of virtual member functions, and thus the set of algorithms, is fixed but the set of data types implementing those algorithms is extensible; with variant types, the set of data types is fixed, but there is no limit to the number of algorithms that can be defined for those data types. The following code illustrates the use of the any_image_view type:[5]

```
typedef variant<mpl::vector<rgb8_view_t, bgr16c_view_t,
                    cmyk32_planar_view_t,
                    lab8_step_planar_view_t> > my_views_t;

any_image_view<my_views_t> v1, v2;
jpeg_read_view(file_name1, v1);
jpeg_read_view(file_name2, v2);
  ...
copy_pixels(v1, v2);
```

Compiling the call to copy_pixels involves examining the run time types of v1 and v2 and dispatching to the instantiation of copy_pixels generated for those types. Indeed, GIL overloads algorithms for any_image_view types, which do exactly this. Consequently, all run time dispatching occurs at a higher level, rather than at the inner loops of the algorithms; any_image_view containers are practically as efficient as if the exact image type was known at compile time. Obviously, the precondition to dispatching to a specific instantiation is that the instantiation has been generated. Unless we are careful, this may lead to significant template bloat, as illustrated in the next section.

3.3 Template bloat originating from GIL's dynamic dispatching

To ease the definition of lists of types for the any_image_view template, GIL implements type generators. One of these generators is cross_vector_image_view_types, which generates all image types that are combinations of given sets of color spaces and channels, and the interleaved/planar and step/no step policies, as the following example demonstrates:

```
typedef mpl::vector<rgb_t,bgr_t,lab_t,cmyk_t>::type ColorSpaceV;
typedef mpl::vector<bits8,bits16,bits32>::type ChannelV;

typedef any_image_view<cross_vector_image_view_types<
  ColorSpaceV, ChannelV,
  kInterleavedAndPlanar, kNonStepAndStep> > any_view_t;

any_view_t v1, v2;

v1 = rgb8_planar_view_t(..);
v2 = bgr8_view_t(..);

copy_pixels(v1, v2);
```

This code defines any_image_t to be one of $4 \times 3 \times 2 \times 2 = 48$ possible image types. It can have any of the four listed color spaces, any of the three listed channel depths, it can be interleaved or planar and its pixels can be adjacent or non-adjacent in memory. The above code generates $48 \times 48 = 2304$ instantiations. Without any special handling, the code bloat will be out of control.

In practice, the majority of these combinations are between incompatible images, which in the case of run-time instantiated images results in throwing an exception. Nevertheless, such exhaustive code generation is wasteful since many of the cases generate essentially identical code. For example, copying two 8-bit interleaved RGB images or two 8-bit interleaved LAB images (with the same channel types) results in the same assembly code — the interpretation of the channels is irrelevant for the copy operation. The following section describes how we can use metaprograms to avoid generating such identical instantiations.

4. Reducing the Number of Instantiations

Our strategy for reducing the number of instantiations is based on decomposing a complex type into a set of orthogonal parameter dimensions (such as color space, channel depth, constness) and identifying which dimensions are important for a given operation. Dimensions irrelevant for a given operation can be cast to a single "base" parameter value. For example, for the purpose of copying, all LAB and RGB images could be treated as RGB images. As mentioned in Section 2, for each algorithm we define a partition among the data types, select the equivalence class representatives, and only generate an instance of the algorithm for these representatives. We call this process *type reduction*.

Type reduction is implemented with metafunctions which map a given data type and a particular algorithm to the class representative of that data type for the given algorithm. By default, that reduction is identity:

```
template <typename Op, typename T>
struct reduce { typedef T type; };
```

By providing template specializations of the reduce template for specific types, the library author can define the partition of types for each algorithm. We return to this point later. Note that the algorithm is represented with the type Op here; we implement GIL algorithms internally as function objects instead of free-standing function templates. One advantage is that we can represent the algorithm with a template parameter.

We need a generic way of invoking an algorithm which will apply the reduce metafunction to perform type reduction on its arguments prior to entering the body of the algorithm. For this purpose, we define the apply_operation function[6]:

[4] The Boost Variant Library offers similar functionality with the make_variant_over metafunction.

[5] The mpl::vector instantiation is a compile-time data structure, a vector whose elements are types; in this case the four image view types.

[6] Note that reinterpret_cast is not portable. To cast between two arbitrary types GIL uses instead static_cast<T*>(static_cast<void*>(arg)). We omit this detail for readability.

```
struct invert_pixels_op {
  typedef void result_type;

  template <typename View>
  void operator()(const View& v) const {
    const int N = View::num_channels;
    typename View::iterator it = v.begin();
    while (it != v.end()) {
      typename View::reference pix=*it;
      for (int i=0; i<N; ++i)
        pix[i]=invert_channel(pix[i]);
      ++it;
    }
  }
};

template <typename View>
inline void invert_pixels(const View& v) {
  apply_operation(v, invert_pixels_op());
}
```

Figure 1. The invert_pixels algorithm.

```
template <typename Arg, typename Op>
inline typename Op::result_type
apply_operation(const Arg& arg, Op op) {
    typedef typename reduce<Op,Arg>::type base_t;
    return op(reinterpret_cast<const base_t&>(arg));
}
```

This function provides the glue between our technique and the algorithm. We have overloads for the one and two argument cases, and overloads for variant types. The apply_operation function serves two purposes — it applies reduction to the arguments and invokes the associated function. As the example above illustrates, for templated types the second step amounts to a simple function call. In Section 5 we will see that for variants this second step also resolves the static types of the objects stored in the variants, by going through a switch statement.

Let us consider an example algorithm, invert_pixels. It inverts each channel of each pixel in an image. Figure 1 shows a possible implementation (which ignores performance and focuses on simplicity) that can be invoked via apply_operation.

With the definitions this far, nothing has changed from the perspective of the library's client. The invert_pixels() function merely forwards its parameter to apply_operation(), which again forwards to invert_pixels_op(). Both apply_operation() and invert_pixels() are inlined, and the end result is the same as if the algorithm implementation was written directly in the body of invert_pixels(). With this arrangement, however, we can control instantiations with defining specializations for the reduce metafunction. For example, the following statement will cause 8-bit LAB images to be reduced to 8-bit RGB images when calling invert_pixels:

```
template<>
struct reduce<invert_pixels_op, lab8_view_t> {
  typedef rgb8_view_t type;
};
```

This approach extends to algorithms taking more than one argument — all arguments can be represented jointly as a tuple. The reduce metafunction for binary algorithms can have specializations for std::pair of any two image types the algorithm can be called with — Section 4.1 shows an example. Each possible pair of input types, however, can be a large space to consider. In particular, using variant types as arguments to binary algorithms (see Section 5) generates a large number of such pair types, which can take a toll on compile times. Fortunately, for many binary algorithms it is possible to apply unary reduction independently on each of the input arguments first and only consider pairs of the argument types after reduction – this is potentially a much smaller set of pairs. We call such preliminary unary reduction *pre-reduction*. Here is the apply_operation taking two arguments:

```
template <typename Arg1 typename Arg2, typename Op>
inline typename Op::result_type
apply_operation(const Arg1& arg1, const Arg2& arg2, Op op) {
   // unary pre−reduction
    typedef typename reduce<Op,Arg1>::type base1_t;
    typedef typename reduce<Op,Arg2>::type base2_t;

   // binary reduction
    typedef std::pair<const base1_t*, const base2_t*> pair_t;
    typedef typename reduce<Op,pair_t>::type base_pair_t;

    std::pair<const void*,const void*> p(&arg1,&arg2);
    return op(reinterpret_cast<const base_pair_t&>(p));
}
```

As a concrete example of a binary algorithm that can be invoked via apply_operation, the copy_pixels() function can be defined as follows:

```
struct copy_pixels_op {
  typedef void result_type;

  template <typename View1, typename View2>
  void operator()(const std::pair<const View1*,
                                  const View2*>& p) const {
    typedef typename View1::iterator src_it = p.first→ begin();
    typedef typename View2::iterator dst_it = p.second→ begin();
    while (src_it != dst.end()) {
      *dst_it++ = *src_it++;
    }
  }
};


template <typename View1, typename View2> inline void
copy_pixels(const View1& src, const View2& dst) {
   apply_operation(src, dst, copy_pixels_op());
}
```

We note that the type reduction mechanism relies on an unsafe cast operation, which relies on programmers assumptions not checked by the compiler or the run time system. The library author defining the reduce metafunction must thus know the implementation details of the types that are being mapped to the class representative, as well as the implementation details of the class representative. A client of the library defining new image types can specialize the reduce template to specify a partition within those types, without needing to understand the implementations of the existing image types in the library.

4.1 Defining reduction functions

In general, the reduce metafunction can be implemented by whatever means is most suitable, most straightforwardly by enumerating all cases separately. Commonly a more concise definition is possible. Also, we can identify "helper" metafunctions that can be reused in the type reduction for many algorithms. To demonstrate, we describe our implementation for the type reduction of the copy_pixels algorithm. Even though we use MPL in GIL extensively, following the definitions requires no knowledge of MPL; here we use a traditional static metaprogramming style of C++, where branching is expressed with partial specializations.

The copy_pixels algorithm operates on two images — we thus apply the two phase reduction strategy discussed in Section 4, first pre-reducing each image independently, followed by the pair-wise reduction.

To define the type reductions for GIL image types, reduce must be specialized for them:

```
template <typename Op, typename L>
struct reduce<Op, image_view<L> >
  : public reduce_view_basic<Op, image_view<L>,
        view_is_basic<image_view<L> >::value> {};

template <typename Op, typename L1, typename L2>
struct reduce<Op, std::pair<const image_view<L1>*,
                            const image_view<L2>*> >
  : public reduce_views_basic<
       Op, image_view<L1>, image_view<L2>,
       mpl::and_<view_is_basic<image_view<L1> >,
                 view_is_basic<image_view<L2> > >::value> {};
```

Note the use the use *metafunction forwarding* idiom from the MPL, where one metafunction is defined in terms of another metafunction by inheriting from it, here reduce is defined in terms of reduce_view_basic.

The first of the above specializations will match any GIL image_view type, the second any pair[7] of GIL image_view types. These specializations merely forward to reduce_view_basic and reduce_views_basic—two metafunctions specific to reducing GIL's image view types. view_is_basic template defines a compile time predicate that tests whether a given view type is one of GIL's built-in view types, rather than a view type defined by the client of the library. We can only define the reductions of view types known to the library, the ones satisfying the prediacte—for all other types GIL applies identity mappings using the following default definitions for reduce_view_basic and reduce_views_basic:

```
template <typename Op, typename View, bool IsBasic>
struct reduce_view_basic { typedef View type; };

template <typename Op, typename V1, typename V2,
          bool AreBasic>
struct reduce_views_basic {
  typedef std::pair<const V1*, const V2*> type;
};
```

The above metafunctions are not specific to a particular type reduction and are shared by reductions of all algorithms.

The following reductions that operate on the level of color spaces are also useful for many algorithms in GIL. Different color spaces with the same number of channels can all be reduced to one common type. We choose rgb_t and rgba_t as the class representatives for three and four channel color spaces, respectively. Note that we do not reduce different permutations of channels. For example, we cannot reduce bgr_t to rgb_t because that will violate the channel ordering.

```
template <typename Cs> struct reduce_color_space {
  typedef Cs type;
};

template <> struct reduce_color_space<lab_t> {
  typedef rgb_t type;
};

template <> struct reduce_color_space<hsb_t> {
  typedef rgb_t type;
};

template <> struct reduce_color_space<cmyk_t> {
  typedef rgba_t type;
};
```

We can similarly define a binary color space reduction — a metafunction that takes a pair of (compatible) color spaces and returns a pair of reduced color spaces. For brevity, we only show the interface of the metafunction:

[7] We represent the two types as a pair of constant pointers because it makes the implementation of reduction with a variant (described in Section 5) easier.

```
template <typename SrcCs, typename DstCs>
struct reduce_color_spaces {
    typedef ... first_t;
    typedef ... second_t;
};
```

The equivalence classes defined by this metafunction represent the color space pairs where the mapping of channels from first to second color space is preserved. We can represent such mappings with a tuple of integers. For example, the mapping of pair<rgb_t,bgr_t> is $\langle 2, 1, 0 \rangle$, as the first channel r maps from the position 0 to position 2, g from position 1 to 1, and b from 2 to 1. Mappings for pair<bgr_t,bgr_t> and pair<lab_t,lab_t> are represented with the tuple $\langle 0, 1, 2 \rangle$. We have identified eight mappings that can represent all pairs of color spaces that are used in practice. New mappings can be introduced when needed as specializations.

With the above helper metafunctions, we can now define the type reduction for copy_pixels. First we define the unary pre-reduction that is performed for each image view type independently. We perform reduction in two aspects of the image: the color space is reduced with the reduce_color_space helper metafunction, and both mutable and immutable views are unified. We use GIL's derived_view_type metafunction (we omit the definition for brevity) that takes a source image view type and returns a related image view in which some of the parameters are different. In this case we are changing the color space and mutability:

```
template <typename View>
struct reduce_view_basic<copy_pixels_fn,View,true> {
private:
  typedef typename
    reduce_color_space<typename View::color_space_t>::type Cs;
public:
  typedef typename derived_view_type<
    View, use_default, Cs, use_default, use_default, mpl::true_
  >::type type;
};
```

Note that this reduction introduces a slight problem — it would allow us to copy (incorrectly) between some incompatible images — for example from hsb8_view_t into lab8_view_t, as they both will be reduced to rgb8_view_t. However, such calls should never occur, as calling copy_pixels with incompatible images violates its precondition. Even though this pre-reduce significantly improves compile times, due to the above objection we did not use it in our measured experiments.

The first step of binary reduction is to check whether the two images are compatible; the views_are_compatible predicate provides this information. If the images are not compatible, we reduce to error_t — a special tag denoting type mismatch error. All algorithms throw an exception when given error_t:

```
template <typename V1, typename V2>
struct reduce_views_basic<copy_pixels_fn, V1, V2, true>
   : public reduce_copy_pixop_compat<V1,V2,
               mpl::and_<views_are_compatible<V1,V2>,
               view_is_mutable<V2> >::value > {};

template <typename V1, typename V2, bool IsCompatible>
struct reduce_copy_pixop_compat {
  typedef error_t type;
};
```

Finally, if the two image views are compatible, we reduce their color spaces pairwise, using the reduce_color_spaces metafunction discussed above. Figure 2 shows the code, where the metafunction derived_view_type again generates the reduced view types that change the color spaces, but keep other aspects of the image view types the same.

Note that we can easily reuse the type reduction policy for copy_pixels for other algorithms for which the same policy applies:

```
template <typename V1, typename V2>
struct reduce_copy_pixop_compat<V1, V2, true> {
private:
  typedef typename V1::color_space_t Cs1;
  typedef typename V2::color_space_t Cs2;
  typedef typename
    reduce_color_spaces<Cs1,Cs2>::first_t RCs1;
  typedef typename
    reduce_color_spaces<Cs1,Cs2>::second_t RCs2;

  typedef typename
    derived_view_type<V1, use_default, RCs1>::type RV1;
  typedef typename
    derived_view_type<V2, use_default, RCs2>::type RV2;

public:
  typedef std::pair<const RV1*, const RV2*> type;
};
```

Figure 2. Type reduction for copy_pixels of compatible images.

```
template <typename V, bool IsBasic>
struct reduce_view_basic<resample_view_fn, V, IsBasic>
  : public reduce_view_basic<copy_pixels_fn, V, IsBasic> {};

template <typename V1, typename V2, bool AreBasic>
struct reduce_views_basic<resample_view_fn, V1, V2, AreBasic>
  : public reduce_views_basic<copy_pixels_fn, V1, V2, AreBasic> {};
```

5. Minimizing Instantiations with Variants

Type reduction is most necessary, and most effective with variant types, such as GIL-s any_image_view, as a single invocation of a generic algorithm would normally require instantiations to be generated for all types in the variant, or even for all combinations of types drawn from several variant types. This section describes how we apply the type reduction machinery in the case of variant types.

Variants are comprised of three elements — a type vector of possible types the variant can store (Types), a run-time value (index) to this vector indicating the type of the object currently stored in the variant, and the memory block containing the instantiated object (bits). Invoking an algorithm, which we represent as a function object, amounts to a switch statement over the value of index, each case N of which casts bits to the N-th element of Types and passes the casted value to the function object. We capture this functionality in the apply_operation_base template:[8]

```
template <typename Types, typename Bits, typename Op>
typename Op::result_type
apply_operation_base(const Bits& bits, int index, Op op) {
    switch (index) {
    ...
    case N: return op(reinterpret_cast<const
                typename mpl::at_c<Types, N>::type&>(bits));
    ...
    }
}
```

As we discussed before, such code instantiates the algorithm with every possible type and can lead to code bloat. Instead of calling this function directly from the apply_operation function template overloaded for variants, we first subject the Types vector to reduction:

[8] The number of cases in the switch statement equals the size of the Types vector. We use the preprocessor to generate such functions with different number of case statements and we use specialization to select the correct one at compile time.

```
template <typename Types, typename Op>
struct unary_reduce {
    typedef ... reduced_t;
    typedef ... unique_t;
    typedef ... indices_t;

    static int map_index(int index) {
        return dynamic_at_c<indices_t>(index);
    }

    template <typename Bits>
    static typename Op::result_type
    apply(const Bits& bits, int index, Op op) {
        return apply_operation_base<unique_t>
            (bits,map_index(index),op);
    }
}
```

Figure 3. Unary reduction for variant types.

```
template <typename Types, typename Op>
inline typename Op::result_type
apply_operation(const variant<Types>& arg, OP op) {
  return unary_reduce<Types,Op>::
    template apply(arg._bits,arg._index,op);
}
```

The unary_reduce template performs type reduction, and its apply member function invokes apply_operation_base with the smaller, reduced, set of types. The definition of unary_reduce is shown in Figure 3. The definitions of the three typedefs are omitted, but they are computed as follows:

- reduced_t — a type vector that holds the reduced types corresponding to each element of Types. That is, reduced_t[i] == reduce<Op, Types[i]>::type
- unique_t — a type set containing the same elements as the type vector reduced_t, but without duplicates.
- indices_t — a type set containing the indices (represented as MPL integral types, which wrap integral constants into types) mapping the reduced_t vector onto the unique_t set, i.e., reduced_t[i] == unique_t[indices_t[i]]

The dynamic_at_c function is parameterized with a type vector of MPL integral types, which are wrappers that represent integral constants as types. The dynamic_at_c function takes an index to the type vector and returns the element in the type vector as a run-time value. That is, we are using a run-time index to get a run-time value out from a type vector. The definitions of dynamic_at_c function are generated with the preprocessor; the code looks similar to the following[9]:

```
template <typename Ints>
static int dynamic_at_c(int index) {
   static int table[] = {
       mpl::at_c<Ints,0>::value,
       mpl::at_c<Ints,1>::value,
       ...
   };
   return table[index];
}
```

Some algorithms, like copy_pixels, may have two arguments each of which may be a variant. Without any type reduction, applying a

[9] In reality the number of table entries must equal the size of the type vector. We use the Boost Preprocessor Library [17] to generate function objects specialized over the size of the type vector, whose application operators generate tables of appropriate sizes and perform the lookup. We dispatch to the right specialization at compile time, thereby assuring the most compact table is generated.

binary variant operation is implemented using a double-dispatch — we first invoke apply_operation_base with the first variant, passing it a function object, which, when invoked, will in turn call apply_operation_base on the second argument, passing it the original function. If N is the number of types in each input variant, this implementation will generate N^2 instantiations of the algorithm and $N + 1$ switch statements having N cases each.

We can, however, possibly achieve more reduction if we consider the argument types together, rather than each independently. Figure 4 shows the definition of the overload for the binary apply_operation function template. We leave several details without discussion, but the general strategy can be observed from the code:

1. Perform unary_reduce on each input argument to obtain the set of unique reduced types, unique1_t and unique2_t. A binary algorithm can define pre-reductions for its argument types, such as the color space reductions described in Section 4.1. Any pre-reductions at this step are beneficial, as they reduce the amount of compile-time computations preformed in the next step.
2. Compute bin_types, a type vector for the cross-product of the unique pre-reduced types. Its elements are all possible types of the form std::pair<const T1∗, const T2∗> with T1 and T2 drawn from unique1_t and unique2_t respectively.
3. Perform unary reduction on bin_types, to obtain unique_t — the set of unique pairs after reducing each pair under the binary operation.

Finally, to invoke the binary operation we use a switch statement over the unique pairs of types left over after reduction. We map the two indices to the corresponding single index over the unique set of pairs. This version is advantageous because it instantiates far fewer than N^2 number of types and uses a single switch statement instead of two nested ones.

```
template <typename Types1, typename Types2, typename Op>
struct binary_reduce {
  typedef unary_reduce<Types1,Op> unary1_t;
  typedef unary_reduce<Types2,Op> unary2_t;
  typedef typename unary1_t::unique_t unique1_t;
  typedef typename unary2_t::unique_t unique2_t;

  typedef cross_product_pairs<unique1_t, unique2_t> bin_types;
  typedef unary_reduce<bin_types,Op> binary_t;
  typedef typename binary_t::unique_t unique_t;

  static inline int map_indices(int index1, int index2) {
    int r1=unary1_t::map_index(index1);
    int r2=unary1_t::map_index(index2);
    return bin_reduced_t::map_index(
                    r2*mpl::size<unique1_t>::value + r1);
  }
public:
  template <typename Bits1, typename Bits2>
  static typename Op::result_type
  apply(const Bits1& bits1, int index1,
        const Bits2& bits2, int index2, Op op) {
    std::pair<const void*,const void*> pr(&bits1, &bits2);

    return apply_operation_base<unique_t>
              (pr, map_indices(index1,index2),op);
  }
};

template <typename T1, typename T2, typename BinOp>
inline typename BinOp::result_type apply_operation(
  const variant<T1>& arg1, const variant<T2>& arg2, BinOp op)
{
  return binary_reduce<T1,T2,Op>::
      template apply(arg1._bits,arg1._index,
                     arg2._bits,arg2._index, op);
}
```

Figure 4. Binary reduction for variant types.

6. Experimental Results

To assess the effectiveness of type reduction in practice, we measured the executable sizes, and compilation times, of programs that called GIL algorithms with objects of variant types when type reduction was applied, and when it was not applied.

6.1 Compiler Settings

For our experiments we used the C++ compilers of GCC 4.0 on OS X 10.4 and Visual Studio 8 on Windows XP. For GCC we used the optimization flag −O2, and removed the symbol information from the executables with the Unix *strip* command prior to measuring their size. Visual Studio 8 was set to compile in release mode, using all settings that can help reduce code size, in particular the "Minimize Size" optimization (/O1), link-time code generation (/Gl), and eliminating unreferenced data (/OPT:REF). With these the compiler can in some cases detect that two different instances of template functions generate the same code, and avoid the duplication of that code. This makes template bloat a lesser problem in the Visual Studio compiler, as type reduction possibly occurs directly in the compiler. We show, however, improvement even with the most aggressive code-size minimization settings.

6.2 Test Images

For testing type reduction with unary operations, we use an extensive variant of GIL image views, varying in color space (Grayscale, RGB, BGR, LAB, HSB, CMYK, RGBA, ABGR, BGRA, ARGB), in channel depth (8-bit, 16-bit and 32-bit) and in whether the pixels are consecutive in memory or offset by a run-time specified step. This amounts to $10 \times 3 \times 2 = 60$ combinations of interleaved images. In addition, we include planar versions for the primary color spaces (RGB, LAB, HSB, CMYK and RGBA) which adds another $5 \times 3 \times 2 = 30$ combinations for a total of 90 image types.[10]

Binary operations result in explosion in the number of combinations to consider for type reduction. The practical upper limit for direct reduction, with today's compilers and typical desktop computers, is about 20×20 combinations; much beyond that consumes notable amounts of compilation resources.[11] Thus, for binary operations we use two smaller test sets. Test B consists of ten images — Grayscale, BGR, RGB, step RGB, planar RGB, planar step RGB, LAB, step LAB, planar LAB, planar step LAB, all of which are in 8-bit. Test C consists of twelve 8-bit images — in RGB, LAB and HSB, each of which can be planar or interleaved, step or non-step.

To summarize: the test set A contains 90 image types, B contains 10 image types, and C contains 12 image types.

6.3 Test Algorithms

We tested with three algorithms — invert_pixels, copy_pixels and resample_view.

[10] We split the images in two sets because GIL does not allow planar versions of grayscale (as it is identical to interleaved) or derived color spaces (because they can be represented by the primary color spaces by rearranging the order of the pointers to the color planes in the image construction).

[11] GIL determines how complex a given binary type reduction will be and suppresses computing it directly when the number of combinations exceeds a limit. In such a case, the binary operation is represented via double-dispatch as two nested unary operations. This allows more complex binary functions to compile, but the type reduction may miss some possibilities for sharing instantiations.

	S_n	S_r	Decrease in %
Test 1.	201.6	107.5	47%
Test 2.	252.8	75.9	70%
Test 3.	259.8	144.0	45%
Test 4.	318.7	98.8	69%
Test 5.	62.2	31.2	50%

Table 1. Size, in kilobytes, of the generated executable in the five test programs compiled with GCC 4.0 C++ compiler, without (S_n) and with (S_r) type reduction. The fourth column shows the percent decrease in the size of the generated code that was achieved with type reduction.

The unary algorithm invert_pixels inverts each channel of each pixel in an image. Although less useful than other algorithms, invert_pixels is simple and allows us to measure the effect of our technique without introducing too much GIL-related code. As a channel-independent operation, invert_pixels does not depend on the color space or ordering of the channels. We tested invert_pixels with the test set A: type reduction maps the 90 image types in test set A down to 30 equivalence classes.

The copy_pixels algorithm, as discussed in Sections 3 and 4, is a binary algorithm performing channel-wise copy between compatible images and throws an exception when invoked with incompatible images. Applied to test images B, our reduction for copy_pixels reduces the image pair types from $10 \times 10 = 100$ down to 26 (25 plus one "incompatible image" case). Without this reduction there are 42 compatible combinations and 58 incompatible ones. The code for the invalid combinations is likely to be shared even without reduction. Thus our reduction transforms 43 cases into 26 cases, which is approximately a 40% reduction.

For test images C, our reduction for copy_pixels reduces the image pairs from $12 \times 12 = 144$ down to 17 (16 plus the "incompatible image" case). Without the reduction, there would be 48 valid and 96 invalid combinations. Thus our reduction transforms 49 into 17 cases, which is approximately a 65% reduction.

We also use another binary operation — resample_view. It resamples the destination image from the source under an arbitrary geometric transformation and interpolates the results using bicubic, bilinear or nearest-neighbor methods. It is a bit more involved than copy_pixels and is therefore less likely to be inlined. It shares the same reduction rules as copy_pixels (works for compatible images and throws an exception for incompatible ones). We test resample_pixels with test images B and C (again, A is too big for a binary algorithm to handle).

In summary we are running 5 tests: (1) copy_pixels on test images B, (2) copy_pixels on test images C, (3) resample_view on test images B, (4) resample_view on test images C, and (5) invert_pixels on test images A.

6.4 Test Results

Our results are obtained as follows: For each of the five tests in an otherwise empty program, we construct an instance of any_image with the corresponding image type set and invoke the corresponding algorithm. We measure the size of the resulting executable and subtract from it the size of the executable if the algorithm is not invoked (but the any_image_view instance is still constructed). The resulting difference in code sizes can thus be attributed to just the code generated from invoking the algorithm. We compute these differences for both platforms, with and without the reduction mechanism, and report the results on Tables 1 and 2.

The results show that we are, on the average, cutting the executable size by more than half under GCC and as much as 70% at times. Since Visual Studio can already avoid generating instantiations whose assembly code is identical, our gain with this compiler is less pronounced. However, we can still observe reduction in the executable size, as much as 32% at times. We believe this is due to two factors — first, Visual Studio's optimization cannot be applied when the code is inlined (which is the case for tests 1, 2 and 5). Indeed those tests show the largest gain. But even for non-inlined code in test 3 we observed a notable reduction. We believe this is due to the simplification of the switch statements. Test 3 without reduction generates 11 (nested) switch statements of 10 cases each, whereas we only generate one switch statement with 26 cases. We also tried inlining resample_view under Visual Studio and got roughly 30% code reduction for tests 3 and 4, (in addition to being about 20% faster to compile, and slightly faster to execute since we avoid two function calls and a double-dispatch).

	S_n	S_r	Decrease in %
Test 1.	42.0	34.5	18%
Test 2.	41.5	26.0	37%
Test 3.	46.0	42.0	8%
Test 4.	33.5	34.0	-1%
Test 5.	24.0	16.5	31%

Table 2. Size, in kilobytes, of the generated executable in the five test programs compiled with Visual Studio 8's C++ compiler, without (S_n) and with (S_r) type reduction. The fourth column shows the percent decrease in the size of the generated code that was achieved with type reduction.

	Visual Studio 8	GCC
Test 1.	106%	116%
Test 2.	78%	97%
Test 3.	87%	118%
Test 4.	75%	103%
Test 5.	194%	307%

Table 3. The effect of type reduction to compilation times in the five test programs. The percentages are computed as $100 \times T_r/T_n$, where T_n is the compilation time without type reduction and T_r the compilation time using type reduction.

We also measured the time to compile each of the five tests of both platforms when reduction is enabled and compared it to the time when no reduction is enabled. The results are reported in Table 3. We believe there are two main factors in play. On the one hand our reduction techniques involve some heavy-duty template meta-programming, which slows down compiling. On the other hand, the number of instantiated copies of the algorithm is greatly reduced, which reduces the amount of work for the later phases of compiling, in particular if the algorithm's implementation is of substantial size. In addition, a large portion of the types generated during the reduction step are not algorithm-dependent and might be reused when another related algorithm is compiled with the same image set. Finally, when compile times are a concern, our technique may be enabled only towards the end of the product cycle.

7. Conclusions

Combining run-time polymorphism and generic programming with the instantiation model of C++ is non-trivial. We show how variant types can be used for this purpose but, without caution, this easily leads to a severe code bloat. As its main contribution, the paper describes library mechanism for significantly reducing code bloat that results from invoking generic algorithms with variant types, and demonstrates their effectiveness in the context of a production quality generic library.

We discussed the problems of the traditional class-centric approach to addressing code bloat: template hoisting within class hierarchies. This approach requires third-party developers to abide by a specific hierarchy in a given module, and can be inflexible — one hierarchy may allow template hoisting for certain algorithms but not for others. Moreover, complex relationships involving two or more objects may not be representable with a single hierarchy.

We presented an alternative, algorithm-centric approach to addressing code bloat, which allows the definition of partitions among types, each specific to one or more generic algorithms. The algorithms need to be instantiated only for one representative of the equivalence class in each partition. Our technique does not enforce a particular hierarchical structure that extensions to the library must follow. The rules for type reduction are algorithm-dependent and implemented as metafunctions. The clients of the library can define their own equivalence classes by specializing a particular type reduction template defined in a generic library, and have the induced type reductions be applied when using the generic algorithms. Also, new algorithms can be introduced by third-party developers and all they need to do is define the reduction rules for their algorithms. Algorithm reduction rules may be inherited; we discussed the copy_pixels and resample_view algorithms which have identical reduction rules.

The primary disadvantage of our technique is that it relies on a cast operation, the correctness of which is not checked. The reduction specifications declare that a given type can be cast to another given type when used in a given algorithm. That requires intimate knowledge of the type and the algorithm. Nevertheless, we believe the generality and effectiveness of algorithm-centric type reduction justify the safety concerns. We demonstrated that this technique can result in reducing the size of the generated code in half for compilers that don't support template bloat reduction. Even for compilers that employ aggressive pruning of duplicate identical template instantiations, our technique can result in further noticeable decrease in code size.

The framework presented in this paper is essentially an *active library*, as defined by Czarnecki et al. [7]. It draws from both generic and generative programming, static metaprogramming with C++ templates in particular. We accomplish a high degree of reuse and good performance with the generic programming approach to library design. Static metaprogramming allows us to fine tune the library's internal implementation — for example, to decrease the amount of code to be generated.

Our future plans include experimenting with the framework in domains other than imaging. We have experience on generic libraries for linear algebra, which seems to be a promising domain, sharing similarities with imaging: a large number of variations in many aspects of the data types (matrix shapes, element types, storage orders, etc.).

Acknowledgments

We are grateful for Hailin Jin for his contributions to GIL and insights on early stages of this work. This work was in part supported by the NSF grant CCF-0541014.

References

[1] *Adobe Source Libraries*, 2006. `opensource.adobe.com`.

[2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[3] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer, August 2001.

[4] Matthew H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] Lubomir Bourdev and Hailin Jin. *Generic Image Library*, 2006. `opensource.adobe.com/gil`.

[6] Martin D. Carroll and Margaret A. Ellis. *Designing and Coding Reusable C++*. Addison-Wesley, 1995.

[7] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glck, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries (extended abstract). In M. Jazayeri, D. Musser, and R. Loos, editors, *Generic Programming. Proceedings*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000.

[8] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9] ECMA. *C# Language Specification*, June 2005. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

[10] ECMA International. *Standard ECMA-367: Eiffel analysis, design and programming Language*, June 2005.

[11] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software – Practice and Experience*, 30(11):1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.

[12] Eric Friedman and Itay Maman. The Boost.Variant library. `http://www.boost.org/libs/variant`, January 2004.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[14] Aleksei Gurtovoy and David Abrahams. The Boost C++ metaprogramming library. `www.boost.org/libs/mpl`, 2002.

[15] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland, 1998.

[16] D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180, July 1992.

[17] Vesa Karvonen and Paul Mensonides. The Boost.Preprocessor library. `http://www.boost.org/libs/preprocessor`, 2002.

[18] 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.

[19] David A. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Rome, Italy, 1988.

[20] W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The Bioinformatics Template Library–generic components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

[21] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[22] Jeremy Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[23] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. Generic programming for high performance numerical linear algebra. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[24] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, April 1994. `http://www.hpl.hp.com/techreports`.

[25] Todd L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

Generic Library Extension in a Heterogeneous Environment

Cosmin Oancea Stephen M. Watt

Department of Computer Science
The University of Western Ontario
London Ontario, Canada N6A 5B7
{coancea,watt}@csd.uwo.ca

Abstract

We examine what is necessary to allow generic libraries to be used naturally in a heterogeneous environment. Our approach is to treat a library as a software component and to view the problem as one of component extension. Language-neutral library interfaces usually do not support the full range of programming idioms that are available when a library is used natively. We address how language-neutral interfaces can be extended with import bindings to recover the desired programming idioms. We also address the question of how these extensions can be organized to minimize the performance overhead that arises from using objects in manners not anticipated by the original library designers. We use C++ as an example of a mature language, with libraries using a variety of patterns, and use the Standard Template Library as an example of a complex library for which efficiency is important. By viewing the library extension problem as one of component organization, we enhance software composibility, hierarchy maintenance and architecture independence.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-Oriented Programming; D.2.2 [*Software Engineering*]: Modules and Interfaces, Software Libraries

General Terms Languages, Design

Keywords Generalized algebraic data types, Generics, Parametric Polymorphism, Software Component Architecture, Templates

1. Introduction

Library extension is an important problem in software design. In its simplest form, the designer of a class library must consider how to organize its class hierarchy so that there are base classes that library clients may usefully specialize. More interesting questions arise when the designers of a library wish to provide support for extension of multiple, independent dimensions of the library's behavior. In this situation, there are questions of how the extended library's hierarchy relates to the original library's hierarchy, how objects from independent extensions may be used and how the extensions interact.

This paper examines the question of library extension in a heterogeneous environment. We consider the situation where software libraries are made available as components in a multi-language, potentially distributed environment. In this setting, the programmer finds it difficult and rather un-safe to compose libraries based on low level language-interoperability solutions. Therefore, components are usually constructed and accessed through some framework such as CORBA [14], DCOM [6] or the .NET framework [5]. In each case, the framework provides a language-neutral interface to a constructed component. These interfaces are typically simplified versions of the implementation language interface to the same modules because of restrictions imposed by the component framework. Restrictions are inevitable: Each framework supports some set of common features provided by the target languages at the time the framework was defined. However, programming languages and our understanding of software architecture evolves over time, so mature component frameworks will lack support for newer language features and programming styles that have become common-place in the interim. If a library's interface is significantly diminished by exporting it through some component architecture, then it may not be used in all of the usual ways that those experienced with the library would expect. Programmers will have to learn a new interface and, in effect, learn to program with a new library.

We have described previously the Generic Interface Definition Language framework, GIDL [8], a CORBA IDL extension with support for parametric polymorphism and (operator) overloading, which allows interoperability of generic libraries in a multi-language environment. GIDL is designed to be a *generic* component architecture *extension*. Here "generic" has two meanings: First GIDL encapsulates a common model for parametric polymorphism that accommodates a wide spectrum of requirements for specific semantics and binding times of the supported languages: C++, Java, and Aldor [16]. Second, the GIDL framework can be easily adapted to work on top of various IDL-based component-systems in use today such as CORBA, DCOM, JNI [15].

This paper explores the question of how to structure the GIDL C++ language bindings to achieve two high-level goals: The first goal is to design an extension framework as a component that can easily be plugged-in on top of different underlying architectures, and together with other extensions. The second goal is to enable the GIDL software components to reproduce as much of their original native language interfaces as possible, and to do so without introducing significant overhead. This allows programmers familiar with the library to use it as designed. In these contexts, we identify the language mechanisms and programming techniques that foster a better code structure in terms of interface clarity, type safety, ease of use, and performance.

While our earlier work [8] presented the high-level ideas employed in implementing the GIDL extension mechanism, this paper takes a different perspective, in some way similar to that of Odersky and Zenger. In [11], they argue that one reason for inadequate advancement in the area of component systems is the fact that mainstream languages lack the ability to abstract over the required services. They identify three language abstractions, namely *abstract type members*, *selftype annotations*, and *modular mixin composition* that enable the design of first-class value components (components that use neither static data nor hard references).

We look at the GIDL extension as a component that can be employed on top of other underlying architectures and which can be, at its turn, further extended. Consequently, we identify the following as desirable properties of the extension:

- The extension interface should be type-precise and it should allow type-safety reasoning with respect to the extension itself. The type-safety result for the whole framework would thus be derived from the ones of the extensions and of the underlying architecture.
- The extension should be split in first-class value components. In the GIDL case for example, one component should encapsulate the underlying architecture specifics and be statically generated. The other one should generically implement the extension mechanism. This would allow GIDL to be plugged in with various backend-architectures without modifying the compiler.
- The extension should preserve the look and feel of the underlying architecture, or at least not complicate its use.
- The extension overhead should be within reasonable limits, and there should be good indication that compiler techniques may be developed to eliminate it.

In the context of GIDL's C++ bindings, we identify the language concepts and programming strategies that enable a better code structure in the sense described above. We particularly recognize *the generalized algebraic data types* paradigm [17] to be essential in enforcing a clear and concise meta-interface of the extension. In agreement with [11], we also find that the use of (C++ simulated) *abstract type members*, and *traits* allows the extension to be split into first-class value components. This derives the obvious software maintenance benefits.

The second part of this paper reports on an experiment where we have used GIDL to export part of the C++ Standard Template Library (STL) functionality to a multi-language, potentially distributed use. We had two main objectives:

The first objective was to determine to what degree the interface translation could preserve the coding style "look and feel" of the original library. Ideally, the STL and its GIDL-exported programs should differ only in the types used. This allows the STL programmers to easily "learn" to use the GIDL interface to write for example distributed applications. More importantly, this opens the door to a richer composition between GIDL and STL objects, as enabled by the STL orthogonal design of its domains. For example GIDL iterators are themselves valid STL iterators and thus they can be manipulated by the STL containers and algorithms. In this context we investigate the issues that prevent the translation to conform with the library semantics, the techniques to amend them, and the trade-offs between translation ease-of-use and performance.

The second objective was to determine whether the interface translation could avoid introducing excessive overhead. We show how this can be achieved through the use of various helper classes that allow the usual STL idioms to be used, while avoiding unnecessary copying of aggregate objects.

The rest of the paper is organized as follows. Section 2 briefly recalls the GADT programming technique, and gives a high-level review of the GIDL framework. Section 3 presents the rationale for employing GADT-based techniques to extend existing frameworks, and outlines the issues to be addressed when translating the STL library to a heterogeneous environment. Section 4 describes the design of the GIDL bindings for the C++ language. Section 5 describes the "black-box" type translation of the STL library to a multi-language, distributed environment via GIDL and discusses certain usability/efficiency trade-offs. Finally Section 6 presents some concluding remarks.

```
data Exp t where
    Lit      :: Int      -> Exp Int
    Plus     :: Exp Int  -> Exp Int -> Exp Int
    Equals   :: Exp Int  -> Exp Int -> Exp Bool
    Fst      :: Exp(a,b) -> Exp a
eval :: Exp t -> t
eval e = case e of
    Lit    i     -> i
    Plus   e1 e2 -> eval e1 + eval e2
    Equals e1 e2 -> eval e1 == eval e2
    Fst    e     -> fst (eval e)
```

Figure 1. GADT-Haskell interpreter example.

```
public class Pair<A,B>                              { /* ... */ }
public abstract class Exp<T>                        { /* ... */ }

public class Lit : Exp<int>
{   public Lit(int val)                        { /* ... */ }   }
public class Plus : Exp<int>
{   public Plus(Exp<int> a, Exp<int> b)        { /* ... */ }   }
public class Equals : Exp<bool>
{   public Equals(Exp<int> e1, Exp<int> e2) { /* ... */ }   }
public class Fst<A,B> : Exp<A>
{   public Fst(Exp<Pair<A,B>> e)               { /* ... */ }   }
```

Figure 2. GADT-C# interpreter example.

2. Background

The first subsection of this chapter introduces at a high-level the *generalized algebraic data types* [17, 4] (GADT) concept and illustrates its use through a couple of examples. The second subsection briefly recounts the architectural design of the GIDL framework and the semantics of the parametric polymorphism model it introduces. A detailed account of this work is given elsewhere [8].

2.1 Generalized Algebraic Data Types

Functional languages such as Haskell and ML support generic programming through user-defined (type) parameterized algebraic datatypes (PADTs). A datatype declaration defines both a named type and a way of constructing values of that type. For example a binary tree datatype, parameterized under the types of the keys and values it stores, can be defined as below.

```
data BinTree k d =  Leaf k d |
                    Node k d (BinTree k d) (BinTree k d)
```

Both value constructors have the generic result type `BinTree k d`, and any value of type `BinTree k d` is either a leaf or a node, but it cannot be statically known which. `BinTree` is an example of a *regular* datatype since all its recursive uses in its definition are uniformly parameterized under the parametric types `k` and `d`.

Generalized algebraic data types (GADTs) enhance the functional programming language PADTs by allowing constructors whose results are instantiations of the datatype with other types than the formal type parameters. Figure 1 presents part of the definition of the types needed to implement a simple language interpreter. Note that all the type-constructors (`Lit`, `Plus`, `Equals`, and `Fst`) refine the type parameter of `Exp`, and use the `Exp` datatype at different instantiations in the parameters of each constructor. Also `Fst` uses the type variable `B` that does not appear in its result type. These are recognized as attributes of the GADT concept; its usefulness is illustrated by the fact that one can now write a well-typed evaluator function (`eval`). The example is inspired from [4] and is written in an extension of Haskell with GADTs.

Kennedy and Russo[4] show, among other things, that existing object oriented programming languages such as Java and C# can express a large class of GADT programs through the use of generics, subclassing and virtual dispatch. A C# implementation of the interpreter using GADTs is sketched in Figure 2.

```
/*********************** GIDL interface ***********************/
interface Comparable< K >
{ boolean operator">" (in K k); boolean operator"=="(in K k); };

interface BinTree< K:-Comparable<K>, D >
{ D getData();     K getKey();     D find(in K k);             };
interface Leaf< K:-Comparable<K>, D > : BinTree<K,D>
{ void init(in K k, in D d);                                   };
interface Node< K:-Comparable<K>, D > : BinTree<K,D>
{ BinTree<K,D> getLeftTree();    BinTree<K,D> getRightTree(); };

interface Integer : Comparable<Integer>    { long getValue(); };
interface TreeFactory<K:-Comparable<K>, D> {
  Integer       mkInt(in long val);
  BinTree<K,D>  mkLeaf(in K k, in D d);
  BinTree<K,D>  mkNode
  (in K k, in D d, in BinTree<K;D> right, in BinTree<K;D> left);
};
/*********************** C++ client code **********************/
TreeFactory<Integer, Integer> fact(...);  // get a factory object
Integer  i6=fact.mkInt(6),  i7=fact.mkInt(7), i8=fact.mkInt(8);
BinTree<Integer, Integer>     b6=fact.mkLeaf(i6,i6),
           b8=fact.mkLeaf(i8,i8), tree=fact.mkNode(i7,i7,b6,b8);
int      res = tree.find(i8).getValue();  // 8
```

Figure 3. GIDL specification and C++ client code for a binary tree

2.2 The GIDL Framework

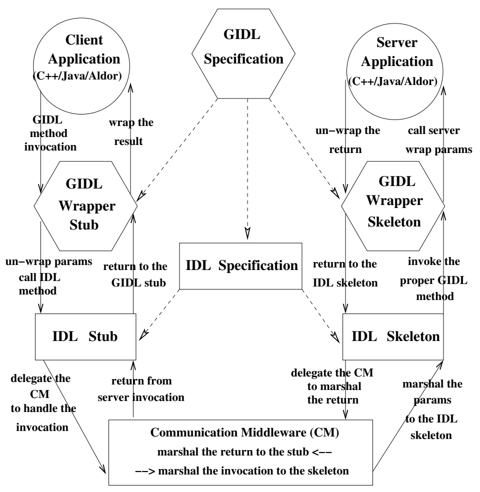
The Generic Interface Definition Language framework [8] (GIDL for short) is designed to be a *generic* component architecture extension that provides support for parameterized components and that can be easily adapted to work on top of various software component architectures in use today: CORBA, DCOM, JNI. (The current implementation is on top of CORBA). We summarize the GIDL model for parametric polymorphism in Section 2.2, and briefly describe the GIDL architecture in Section 2.2. An in depth presentation of these topics can be found in [8].

The GIDL language

GIDL extends CORBA–IDL [12] language with support for *F-bounded parametric polymorphism*. Figure 3 shows abstract data type (ADT)-like GIDL interfaces for a binary tree that is type-parameterized under the types of data and keys stored in the nodes. The type-parameter K in the definition of the BinTree interface is qualified to export the whole functionality of its qualifier Comparable<K>; that is, the comparison operations > and ==. GIDL also supports a stronger qualification denoted by : that enforces a subtyping relation between the instantiation of the type parameter and the qualifier. Figure 3 also presents C++ client code that builds a binary tree and finds in the tree the data of a node that is identified through its key. Note that the code is very natural for the most parts; the only place where CORBA specifics appear is in the creation of the factory object (fact).

The GIDL Extension Architecture

Figure 4 illustrates at a high level the design of the GIDL framework. The implementation employs a generic type erasure mechanism, based on the subtyping polymorphism supported by IDL. A GIDL specification compiled with the GIDL compiler generates an IDL file where all the generic types have been *erased*, together with GIDL wrapper stub and skeleton bindings, which recover the lost generic type information. Currently GIDL provides language bindings for C++, Java, and Aldor. Compiling the IDL file creates the underlying architecture (UA) stub and skeleton bindings. Every GIDL-stub (client) wrapper object references a UA-stub object. Every GIDL-skeleton (server) wrapper inherits from the corresponding UA-skeleton type. This technique is somewhat related with the "reified type" pattern of Ralph Johnson [3], where objects are used to carry type information.



Client Application (C++/Java/Aldor)
GIDL Specification
Server Application (C++/Java/Aldor)
GIDL method invocation
wrap the result
un−wrap the return
call server wrap params
GIDL Wrapper Stub
GIDL Wrapper Skeleton
un−wrap params call IDL method
return to the GIDL stub
IDL Specification
return to the IDL skeleton
invoke the proper GIDL method
IDL Stub
IDL Skeleton
delegate the CM to handle the invocation
return from server invocation
delegate the CM to marshal the return
marshal the params to the IDL skeleton
Communication Middleware (CM)
marshal the return to the stub <−−
−−> marshal the invocation to the skeleton

Figure 4. GIDL architecture
circle – user code; *hexagon* – GIDL *component;*
rectangle – underlying architecture component;
dashed arrow – is compiled to;
solid arrow – method invocation flow

The solid arrows in Figure 4 depict method invocation. When a method of a GIDL stub wrapper object is called, the implementation retrieves the parameters' UA-objects, invokes the UA method on these, and perform the reverse operation on the result. The wrapper skeleton functionality is the inverse of the client. The wrapper skeleton method creates GIDL stub wrapper objects encapsulating the UA objects, thus recovering the generic type erased information. It then invokes the user-implemented server method with these parameters, retrieves the UA IDL-object or value of the result and passes it to the IDL skeleton.

The extension introduces an extra level of indirection with respect to the method invocation mechanism of the underlying framework. This is the price to pay for the generality of the approach: this generic extension will work on top of any UA vendor implementation while maintaining backward compatibility. However, since the GIDL wrappers are mainly storing generic type information, one can anticipate that the introduced overhead can be eliminated by applying aggressive compiler optimizations.

3. Problems Statement and High-Level Solutions

This section states and motivates the main issues addressed by this paper, and presents at the high-level the methods employed to solve them: Section 3.1 summarizes the rationale and the techniques we have used to structure the GIDL language bindings. Section 3.2 outlines the main difficulties a heterogeneous translation of the STL library has to overcome, and points to a solution that preserves the library semantics and programming patterns.

3.1 Software Extensions via GADTs

Among the GADTs applications, the literature enumerates: strongly typed evaluators, generic pretty printing, generic traversal and queries and typed LR parsing. This paper finds another important application of the GADT concept: in the context of software architecture extensions. This section describes things at a high-level, while Section 4 presents in detail the C++ binding.

Section 2.2 has introduced GIDL as a *generic extension framework* that enhances CORBA with support for parametric polymorphism. The GIDL wrapper objects can be seen as an aggregation of

```
class Foo_CORBA {    /* ... */    }
class Foo_GIDL  {
    Foo_CORBA obj;   /* ... */
    Foo_CORBA  getOrigObj ()            { return obj; }
    void       setOrigObj (Foo_CORBA o)        { ... }
    static Foo_CORBA  _narrow    (Foo_GIDL o)  { ... }
    static Foo_GIDL   _lift      (Foo_CORBA o) { ... }
    static Foo_GIDL   _lift      (CORBA_Any a) { ... }
    static CORBA_Any  _any_narrow(Foo_GIDL a)  { ... }
}
```

Figure 5. Pseudocode for the casting functionality of the `Foo_GIDL` GIDL wrapper. `Foo_CORBA` is its corresponding CORBA class. `CORBA_Any`-type objects can store any CORBA-type values.

```
class Base_GIDL<T_GIDL, T_CORBA> {
    T_CORBA  getOrigObj ()            { return obj; }
    void     setOrigObj (T_CORBA o)          { ... }
    static T_CORBA   _narrow    (T_GIDL o)   { ... }
    static T_GIDL    _lift      (T_CORBA o)  { ... }
    static T_GIDL    _lift      (CORBA_Any a){ ... }
    static CORBA_Any _any_narrow(T_GIDL a)   { ... }  /* ... */
}
class Foo_GIDL : Base_GIDL<Foo_GIDL, Foo_CORBA> ...
```

Figure 6. GADT pseudocode for the casting functionality of the `Foo_GIDL` GIDL wrapper.

a reference to the corresponding CORBA object, the generic type information associated with them and the two-way casting functionality they define (CORBA-GIDL types). It follows that a GIDL wrapper is composed of two main components: the functionality described in the GIDL interface, and the *casting* functionality needed by the system for the two way communication with the underlying framework (CORBA).

In this way, we deal with two parallel type hierarchies: the original one (CORBA) and the one of the extension (GIDL). Figure 5 shows that each type of the extension encapsulates the functionality to transform back and forth between values of its type and values of its corresponding CORBA type, and also between values of its type and values of the CORBA type `Any`. Values of type `Any` can store any other CORBA type values, so GIDL uses type `Any` as the erasure of the non-qualified type-parameter.

This functionality can be expressed in an elegant way via GADTs, by writing a parameterized base class that contains the implementation for the casting functionality together with a precise interface, and by instantiating this base class with corresponding pairs of GIDL-CORBA types. Figure 6 demonstrates this approach. We see *three main advantages* for integrating the GIDL casting functionality via GADTs:

- This functionality is written now as a system component and not mangled inside the GIDL wrapper. It can be integrated either by inheritance (see the C++ mapping), or by aggregation (see the Java mapping).
- In addition it constitutes a clear meta-interface that characterizes all the pairs of types from the two parallel hierarchies, and makes it easier to reason about the type-safety of the GIDL extension.
- Finally, this approach is valuable from a code maintenance / post facto extension point of view. The casting functionality code is dependent on the underlying framework (CORBA, JNI, DCOM). Implementing it as a meta-program (see the C++ mappings), besides the obvious software maintenance advantages of being *static* and written only once (thus short), allows the GIDL compiler to generate *generic* code that is independent on the underlying architecture. Porting the framework on top of a new architecture will require rewriting this static code, reducing the modifications to be done at the compiler's code generator level.

```
1.  Vector< Long, RAI<Long>, RAI<Long> > vect  = ...;
2.  RAI<Long> it_beg=vect.begin(), it_end=vect.end(), it=it_beg;
3.  while(it!=it_end)
4.      *it++ = (vect.size() - i);
5.  sort(it_beg, it_end);     cout<<*it_beg<<endl;
```

Figure 7. C++ client code using a GIDL translation of STL. `RAI` and `Vector` are the GIDL types that model the STL random access iterator and vector types; `sort` is the native STL function.

The problem with this approach is that if the `Foo_GIDL` interface is a subtype of say `Foo0_GIDL` then it inherits the casting functionality of `Foo0_GIDL` – an undesired side-effect. The C++ binding addresses this problem by making the GIDL wrapper inherit from two components: one which respects the original inheritance hierarchy and which contains the functionality described in the GIDL specification, and one implementing the *system* functionality (`Base_GIDL<Foo_GIDL, Foo_CORBA>`).
This method breaks the subtyping hierarchy between the GIDL wrappers, and instead mimics subtyping by means of automatic conversion. This solution will be discussed in detail in Section 4. Since Java does not support automatic conversions, the Java mapping defines the casting component as an inner class of the GIDL wrapper, and uses a mechanism that resembles virtual types in order to retrieve and invoke the proper caster. The GIDL Java bindings are not however the subject of this paper.

3.2 Preserving the STL Semantics and Code Idioms

Figure 7 gives an example of GIDL client code that retrieves a vector's iterator (`it_beg`), updates it, sorts it and displays its first element. To allow such code, the translation needs to conform with both the native library semantics and its coding idioms.

First, to preserve the STL semantics, certain type properties must be enforced statically. For example, the parameters of the `sort` function need to belong to an iterator type that allows random access to its elements. As discussed in Section 5.1 these properties are expressed at the GIDL interface level by means of parametric polymorphism and operator overloading.

Second, for the (distributed) program to yield the expected result, `it` and `it_beg` have to reference different implementation-object instances sharing the same internal representation. Otherwise, after the execution of the while-loop (lines $3-4$), `it_beg` either points to its end, or it is left unchanged. Moreover, the instruction `*it++ = i` is supposed to update the value of the iterator's current element. Neither one of these requirements is achieved with the GIDL semantics. As detailed in Section 5.3, we can obtain the expected behavior with an extension mechanism applied to the GIDL stubs that overrides the default behavior in favor of one that satisfies the STL coding style.

4. Building a Natural C++ Interface from GIDL

This section presents the rationale behind the GIDL C++ bindings. We start by presenting the GADT approach used to implement the casting functionality of the GIDL wrapper objects. We then show how the GIDL inheritance hierarchies are implemented and comment on the language features that we found most useful in this context. Finally, we demonstrate the ease of use of the GIDL extension and reason about the soundness of the translation mechanism.

4.1 The Generic Base Class

Figure 8 presents a simplified version of the base class for the wrapper object whose GIDL type is `String`, `WString` or some interface. The type parameter `T` denotes the current GIDL class, `A` is its corresponding CORBA class, while `A_v` denotes the CORBA smart pointer helper type that assists with memory management and parameter passing. The `BaseObject` class inherits from the

```
1 class ErasedBase { protected: void* obj; };
2 template<class T,class A,class A_v> class BaseObject :
3   public ErasedBase, public GIDL_Type<T> {
4  protected:
5   static void fillObjFromAny(CORBA::Any& a, A*& v) {
6     CORBA::Object_ptr co = new CORBA::Object();
7     a>>=co; A* w = A::_narrow(co); v = w;
8   }
9   static void fillAnyFromObj(CORBA::Any& a, A* v) { a<<=v; }
10 public:
11  typedef A GIDL_A;   typedef A_v GIDL_A_v;   typedef Self T;
12
13  BaseObject(A* ob)          { this->obj = ob; }
14  BaseObject(const A_v& a_v) {this->obj=a_v._retn();}
15  BaseObject(const T& ob)    { this->obj = ob.obj; } //
16  BaseObject(const GIDL::Any_GIDL& ob)
17  {T::fillObjFromAny(*ob.getOrigObj(),getOrigObj());}
18  template<class GG> BaseObject(
19    const BaseObject<GG,GG::GIDL_A,GG::GIDL_A_v>& o
20  ) { this->obj = (A*)o.getOrigObj(); }
21  /*** SIMILAR CODE FOR THE ASSIGNMENT OPERATORS ***/
22
23  operator A*() const { return (A*)obj; }
24  template < class GG > operator GG() const{
25    GG g; // test GG superclass of the current class!
26    if(0) { A* ob; ob = g.getOrigObj(); }
27    void*& ref = (void*&)g.getOrigObj();
28    ref = GG::_narrow(this->getOrigObj()); return g;
29  }
30  A*& getOrigObj()  const { return (A*) obj; }
31  void setOrigObj(A* o)   { obj = o; }
32
33  static A*& _narrow(const T& ob){return ob.getOrigObj();}
34  static CORBA::Any* _any_narrow(const T& ob) { /* ... */ }
35  static T _lift(CORBA::Any& a, T& ob)
36  { T::fillObjFromAny(a,ob.getOrigObj()); return ob; }
37  static T _lift(CORBA::Object* o) { return T(A::_narrow(o));}
38  static T _lift(const A* ob)      { return T(ob); }
39  /*** SIMILAR: _lift(A_v) AND _lift(CORBA::Any& v) ***/
40 };
```

Figure 8. The base class for the GIDL wrapper objects whose types are GIDL interfaces. (We have omitted the `inline` keyword)

`ErasedBase` class that stores the type-erased representation under the form of a void pointer, and from the `GIDL_Type`, the supertype of all GIDL types. The `fillObjFromAny` and `fillAnyFromObj` functions abstract the CORBA functionality of creating an object from a CORBA `Any`-type value, and vice-versa. They are re-written for the `String`/`WString` types as the CORBA specific calls differ. The implementation provides overloaded constructors, assignment operators and accessor functions that work over various CORBA and GIDL types, allowing the user to manipulate in an easy and transparent way GIDL wrapper objects.

The generic constructor (lines 18-20) receives as a parameter a GIDL object whose type is in fact `GG`. The use of `BaseObject<GG, GG::GIDL_A,GG::GIDL_A_v>`, together with the cast to `A*` in line 20, statically checks that the instantiation of the type `GG` is a GIDL interface type that is a subtype of the instantiation of `T` (with respect to the original GIDL specification). This irregular use of the `BaseObject` type constructor is one of the GADT characteristics. Note also the use of the *abstract type members* `GG::GIDL_A` and `GG::GIDL_A_v`. The mapping also defines a type-unsafe cast operator (lines 24-29) that allows the user to transform an object to one of a more specialized type. The implementation, however, statically ensures that the result's type is a subtype of the current type.

4.2 Handling Multiple Inheritance

We now present the rationale behind the C++ mapping of the GIDL inheritance hierarchies. There are two main requirements that guided our design:

```
template<class K, class D> BinTree {
    protected:    ::BinTree* obj;
    public:   // system functionality
        void setOrigObj(::BinTree* o) { obj = o; }
              // GIDL specification functionality /* ... */
};
template<class K, class D> Node : public virtual BinTree<K, D> {
    protected:    ::Node* obj;
    public:   // system functionality
        void setOrigObj(::Node* o)   { obj = o; }
              // GIDL specification functionality
        BinTree<K,D> getLeftTree()   { /* ... */ }
};
```

Figure 9. Naive translation for the C++ mapping

- As far as the representation is concerned, each GIDL wrapper stores precisely one (corresponding) CORBA-type object: its erasure. This is a performance concern. It is important to keep the object layout of the GIDL stub wrapper small.
- In terms of functionality, the GIDL wrapper features only the casting functionality associated with its type; in other words the *system* functionality is not subject to inheritance. This is a type-soundness, as well as a performance concern.

Throughout this section we refer to the GIDL specification in Figure 3. We first examine the shortcomings of a naïve translation that would preserve the inheritance hierarchy among the generated GIDL wrappers. Figure 9 shows such an attempt. If each GIDL wrapper stores its own representation as an object of its corresponding CORBA-type, the wrapper object layout will grow exponentially. An alternative would be to store the representation under the form of a void pointer in a base class and to use virtual inheritance (see the `BaseObject` class in Figure 8). However, then the system is not type-safe, since the user may call, for example, the `setOrigObj` function of the `BinTree` class to set the `obj` field of a `Node` GIDL wrapper. Now calling the `Node::getLeftTree` method on the wrapper will result in a run-time error. This happens because the `Node` wrapper inherits the *casting functionality* of the `BinTree` wrapper.

Figure 10 shows our solution. The abstract class `Leaf_P` models the inheritance hierarchy in the GIDL specification: it inherits from `BinTree_P` and it provides the implementation for the methods defined in the `Leaf` GIDL interface (n.n. `init`). Our mechanism resembles Scala [9] traits [10]. `Leaf_P` does not encapsulate state and does not provide constructors, but inherits from the `BinTree_P` "trait". It *provides the services* promised by the corresponding GIDL interface, and *requires an accessor* for the CORBA object encapsulated in the wrapper (the `getErasedObj` function).

Finally, the `Leaf` wrapper class aggregates the casting functionality and the services promised by the GIDL specification by inheriting from `Leaf_P` and `BaseObject` respectively. It rewrites the functionality that is not subject to inheritance: the constructors and the assignment operators by calling the corresponding operations in `BaseObject`. Note that there is no subtyping relation between the wrappers even if the GIDL specification requires it. However, the templated constructor ensures a type-safe, user-transparent cast between say `Leaf<A,B>` and `BinTree<A,B>`.

To summarize, the C++ binding uses GADTs and *abstract type members* to enforce a precise meta-interface of the extension. The latter we simulate in C++ by using templates in conjunction with `typedef` definitions. Further on, the functionality described in the GIDL interface is implemented via *traits*. We represent traits in C++ as abstract classes and the require services as abstract virtual methods. The latter are provided by the GIDL wrapper that "mixins" the two-way GIDL-CORBA casting with the functionality published in the specification. Our extension experiment constitutes another

```
template<class K,class D> class Leaf_P : public BinTree_P<K,D>{
  protected:
    virtual void*   getErasedObj() = 0;
    ::Leaf* getObject_Leaf(){ return (::Leaf*)getErasedObj(); }
  public:
    void init(const K& a1, const D& a2) {
      CORBA::Object_ptr& a1_tmp = K::_narrow(a1);
      CORBA::Any& a2_tmp = *D::_any_narrow(a2);
      getObject_Leaf()->init(a1_tmp, a2_tmp);
    }
};
template<class K,class D> class Leaf :
     public virtual Leaf_P< K, D >,
     public BaseObject<Leaf<K,D>,::Leaf,::Leaf_var>
{
  protected:
    typedef Leaf<K,D> T;
    typedef BaseObject<T,GIDL_A,GIDL_A_v> BT;
    void*   getErasedObj()     { return obj; }
  public:
    Leaf()                         : BT()  { }
    Leaf(const GIDL_A_v a)         : BT(a) { }
    Leaf(const GIDL_A* a)          : BT(a) { }
    Leaf(const T & a)              : BT(a) { }
    Leaf(const Any_GIDL & a)       : BT(a) { }
    template <class GG> Leaf(
      const BaseObject<GG, GG::GIDL_A, GG::GIDL_A_v>& a
    )                              : BT(a) { }
    /*** SIMILAR CODE FOR THE ASSIGNMENT OPERATORS ***/
};
```

Figure 10. Part of the C++ generated wrapper for the GIDL::`Leaf` interface. `::Leaf` and `::Leaf_var` are CORBA-types

empirical argument to strengthen Odersky and Zenger's claim that *abstract type members*, and *modular mixin composition* are vital in achieving first-class value components. We would add the GADT technique to that.

4.3 Ease of Use

One additional feature of the GIDL framework, in our view, is that it is much simpler to be used than its underlying CORBA architecture. At a high-level, this is accomplished by making the GIDL wrappers to encapsulate a variety of constructors, cast and assignment operators.

Figures 11`A` and `B` illustrate the CORBA/GIDL code that inserts GIDL/CORBA `Octet` and `String` objects into `Any` objects, then performs the reverse operation and prints the results. Note that the use of CORBA specific functions, such as `CORBA::Any::from_string`, is hidden inside the GIDL wrappers; the GIDL code is uniform with respect to all the types, and mainly uses constructors and assignment operators. All GIDL wrappers provide a casting operator to their original CORBA-type object that is transparently used in the statement that prints the two objects. Figure 11C presents the implementation of the generic assignment operator of the `Any_GIDL` type. Since `GIDL_Type` is an abstract supertype for all GIDL types, its use in the parameter declaration statically ensures that the parameter is actually a GIDL object. By construction, the only class that inherits from `GIDL_Type<T>` is `T`, therefore the dynamic cast is safe. Finally the method calls the `T::_lift` operation (see Figure 8) that fills in the object encapsulated by the GIDL `Any` wrapper with the appropriate value stored in the `T`-type object.

Figure 11D presents one of the shortcomings of our mapping. The GIDL wrapper for arrays, as for all the other GIDL wrapper-types, has as representation its corresponding CORBA generic-type erased object. The representation for an `Array_T`-type object will be an array of the CORBA `Any` type objects, since the erasure of the non-qualified type-parameter `T` is the `Any` CORBA type. Although the user may expect that a statement like `arr[i] = i` inside the `for`-loop should do the job, this is not the case. The reason is that

```
// A. CORBA code
using namespace CORBA;
Octet oc = 1; Char* str = string_dup("hello"); Any a_oc, a_str;
a_str <<= CORBA::Any::from_string(str, 0);
a_oc  <<= CORBA::Any::from_octet (oc);
a_oc  >>= CORBA::Any::to_octet   (oc);
a_str >>= CORBA::Any::to_string  (str, 0);
cout<<"Octet (1): "<<oc<<" string (hello): "<<str<<endl;

// B. GIDL code:
using namespace GIDL;
Octet_GIDL oc(1); String_GIDL str("hello"); Any_GIDL a_oc, a_str;
a_oc  = sh;  a_str = str;  oc    = a_oc;  str   = a_str;
cout<<"Octet (1): "<<oc<<" string (hello): "<<str<<endl;

// C. The implementation of the Any_GIDL::operator=
template<class T> void Any_GIDL::operator=(GIDL_Type<T>& b){
  T& a = dynamic_cast<T&>(b);
  if(!this->obj) this->obj = new CORBA::Any();
  T::_lift(this->obj, a);
}

// D. GIDL Arrays
interface Foo<T> {    //GIDL specification
    typedef T Array_T[100];
    T sum_and_revert(inout Array_T arr);
};
// C++ code using the GIDL specification above
Foo<Long_GIDL> foo = ...; Foo<Long_GIDL>::Array_T arr;
for(int i=0; i<100; i++) {
    Long_GIDL elem(i);    arr[i] =  elem;
}
int sum=foo.sum_and_invert(arr); Long_GIDL arr_0=arr[0];
cout<<"sum (4950): "<<sum<<" arr[0] (99): <<arr_0<<endl;
```

Figure 11. GIDL/CORBA use of the `Any` type

Data Type	In	Inout	Out	Return
fixed struct	ct struct&	struct&	struct&	struct
var struct	ct struct&	struct&	struct&	struct*
fixed array	ct array	array	array	array sl*
var array	ct array	array	array sl*	array sl*
any	ct any&	any&	any*&	any*
...	...	...	...	...

Table 1. CORBA types for in, inout, out parameters and the result. `ct` = `const`, `sl` = `slice`, `var` = variable.

`Any_GIDL` does not provide an assignment operator or constructor that takes an `int` parameter.

Another simplification that GIDL brings refers to the types of the `in`, `inout` and `out` parameter, and the type of the result. Table 1 shows several of these types as specified in the CORBA standard. The GIDL parameter passing scheme is much simpler: the parameter type for `in` is `const T&`, for `inout` and `out` is `T&`, for the result is `T`, where `T` denotes an arbitrary GIDL type. The necessary type-conversions are hidden in the GIDL wrapper.

4.4 Type-Soundness Discussion

We restrict our attention to the wrapper-types corresponding to the GIDL interfaces. The same arguments apply to the rest of the wrapper-types. Let us examine the type-unsafe operations of the `BaseObject` class, presented in Figure 8. Note first that any function that receives a parameter of type `Any_GIDL` or `CORBA::Any` is unsafe, as the user may insert an object of a different type than the one expected. For example the `Leaf(const Any_GIDL& a)` constructor expects that an object of CORBA type `Leaf` was inserted in `a`: the user may decide otherwise, however, and the system cannot statically enforce it. It is debatable whether the introduction of generics to CORBA has rendered the existence of the `Any` type un-

```
// GIDL specification
interface Foo<T, I:-Test, E: Test> {
  Test foo(inout T t,inout I i,inout E e);
}
// Wrapper stub for foo
template<class T, class I, classE>
GIDL::Test Foo<T,I,E>::foo( T& t, I& i, E& e ) {
  CORBA::Any&     et  = T::_any_narrow(t);
  CORBA::Object*& ei  = I::_narrow(i);
  CORBA::Test*&   ee  = E::_narrow(e);
  CORBA::Test*    ret = getObjectFoo()->foo(et, ei, ee);
  return GIDL::Test::_lift(ret);
}
// Wrapper skeleton for foo
template<class T, class I, class E> ::Test Foo_Impl<T,I,E>::foo
( CORBA::Any& et, CORBA::Object*& ei, ::Test*& ee ) {
  T& t=T::_lift(et);   I& i=I::_lift(ei);   E& e=E::_lift(ee);
  GIDL::Test  ret = fooGIDL(t, i, e);
  return GIDL::Test::_narrow(ret);
}
```

Figure 12. GIDL interface and the corresponding stub/skeleton wrappers for function `foo`

necessary in GIDL at the user level. We decided to keep it in the language for backward compatibility reasons. The drawback is that the user may manipulate it in a type-unsafe way.

In addition to these, there are two more unsafe operations:

```
template < class GG > operator GG() const { ... }
static T _lift (const CORBA::Object* o) { ... }.
```

The templated cast operator is naturally unsafe, as it allows the user to cast to a more specialized type. The `_lift` method is used in the wrapper to lift an export-based qualified generic type object (`:-`), since its erasure is `CORBA::Object*`. Its use inside the wrapper is type-safe; however, if the user invokes it directly, it might result in type-errors.

Our intent is that the user access to the GIDL wrappers should be restricted to the constructors, the assignment and cast operators, and the functionality described in the GIDL specification, while the rest of the casting functionality should be invisible. However this is not possible since the `_narrow` and `_lift` methods are called in the wrapper method implementation to cast the parameters, and hence need to be declared public.

A *type-soundness* result is difficult to formalize as we are unaware of such results for (subsets of) the underlying CORBA architecture, and the C++ language is type-unsafe. In the following we shall give some informal soundness arguments for a subset of the GIDL bindings. We assume that the user can access only wrapper constructors and operators and only those that do not involve the `Any` type. The precise GADT interface guarantees that the creation of GIDL objects will not yield type-errors. It remains to examine method invocations. It is trivial to see from the implementation of the `_lift`, `_narrow`, and `_any_narrow` functions (Figure 8) that the following relations hold:

`G::_lift[A*]`∘`G::_narrow[G] (a)` ∼ `a`
`G::_lift[Object*]`∘`G::_narrow[G] (a)` ∼ `a`
`G::_lift[Any]`∘`G::_any_narrow[G] (a)` ∼ `a`

where `[]` is used for the method's signature, ∘ stands for function composition, while `g1`∼`g2` denotes that `g1` and `g2` are equivalent in the sense that they encapsulate the reference to the same CORBA object implementation. (The reverse also holds.)

Figure 12 presents the GIDL operation `Foo::foo()` and its C++ stub/skeleton mapping. The stub wrapper will translate the parameter to an object of the corresponding CORBA erased type via the `_narrow`/`_any_narrow` methods. The skeleton wrapper does the reverse: lifts a CORBA type object to a corresponding GIDL type object. Since the instantiations for the `T`, `I`, and `E` type parameters are the same on the client and server side, the above relations and the exact GADT casting interface guarantee that the GIDL object passed as parameter to the stub wrapper by the client will have the same type and will hold a reference to the same object-implementation as the one that is delivered to the `fooGIDL` server implementation method. The same argument applies to the result object.

5. Library Translation: Trappers

The immediate use of GIDL is to enable applications that combine parameterized, multi-language components. This section investigates another important application: what is required to use GIDL as a vehicle to access generic libraries beyond their original language boundaries, and what techniques can automate this process? For the purpose of this paper, we restrict the discussion to the simpler case when the implementation shares a single process space.

We find C++'s *Standard Template Library* (STL) to be an ideal candidate for experimentation due to the wealth of generic types, the variety of operators, and high-level properties such as the orthogonality between *the algorithm and container domains* it exposes. Furthermore, the fact that, for performance reasons, STL does not hide the representation of its objects poses new translation-related challenges. In what follows, we review the STL library at a high level, show the GIDL specification for a server encapsulating part of STL's functionality, identify and propose solutions to two issues that prevent the translation from implementing the library semantics, and discuss the performance-related trade-offs.

5.1 STL at a High Level

STL [2] is a general purpose generic library known for providing a high level of modularity, usability, and extensibility to its components, without impacting the code's efficiency. The STL components are designed to be *orthogonal*, in contrast to the traditional approach where, for example, *algorithms* are implemented as methods inside *container* classes. This keeps the source code and documentation small, and addresses the extensibility issue as it allows the user algorithms to work with the STL containers and *vice-versa*. The orthogonality of the algorithm and container domains is achieved, in part, through the use of iterators: the algorithms are specified in terms of iterators that are exported by the containers and are data structure independent. STL specifies for each container/algorithm the iterator category that it provides/requires, and also the valid operations exported by each iterator category. These are however defined as English annotations in the standard, as C++ lacks the formalism to express them at the interface level.

Figures 13 and 14 present excerpts of the GIDL iterators and vector interfaces respectively. We simulate *selftypes* [11] by the use of an additional generic type, `It`, bounded via a mutual recursive export based qualification (`:-`). This abstracts the iterators functionality: `InpIt<T>` exports `==(InpIt<T>)` method, while `RaiIt<T>` exports the `==(RaiIt<T>)` method. An *input iterator* has to support operations such as: incrementation (`it++`), dereferencing (`*it`), and testing for equality/non-equality between two *input iterators* (`it1==it2`, `it1!=it2`). A *forward iterator* allows reading, writing, and traversal in one direction. A *bidirectional iterator* allows all the operations defined for the *forward iterator*, and in addition it allows traversal in both directions. *Random access iterators* are supposed to support all the operations specified for *bidirectional iterator*, plus operations as: addition and subtraction of an integer (`it+n, it-n`), constant time access to a location `n` elements away (`it[n]`), bidirectional big jumps (`it+=n; it-=n;`), and comparisons (`it1>it2;` etc). The design of iterators and containers is non-intrusive as it does not assume an inheritance hierarchy; we use inheritance between iterators only to keep the code short. The `STLvector` container does not expect the iterators to be subject to an inheritance hierarchy, but only to implement the functionality described in the STL specification: `RI` is expected to share

```
interface BaseIter<T, It:-BaseIter<T; It> > {
  unsigned long getErasedSTL();   It  cloneIt();
  void operator"++@p"();  void operator"++@a"();
};
interface InputIter<T,It:-InputIter<T;It> >:BaseIter<T,It>{
  T       operator"*"  ();
  boolean operator"==" (in It it);
  boolean operator"!=" (in It it);
};
interface ForwardIter<T, It:-ForwardIter<T; It> >
    : OutputIter<T, It>, InputIter<T; It>
    { void assign(in T t1);                            };
interface BidirIter<T, It:-BidirIter<T; It> >
    : ForwardIter<T, It>
    { void operator"--@p"();    void operator"--@a"(); };
interface RandAccessIter<T,It:-RandAccessIter<T,It> >
    : BidirIter<T, It> {
  boolean operator">"  (in It it);
  /*   same for "<", ">=", "<="   */
  Iterator operator"+" (in long n);
  Iterator operator"-" (in long n);
  void operator"+="    (in long n);
  void operator"-="    (in long n);
  T    operator"[]"(in long n);
  void assign(in T obj, in long index);
};

interface InpIt<T>   : InputIter<T, InpIt<T> >   {};
interface ForwIt<T>  : ForwardIter<T, ForwIt<T> >{};
interface BidirIt<T> : BidirIter<T, BidirIt<T> > {};
interface RAI<T>     : RandAccessIter<T, RAI<T> >{};
```

Figure 13. GIDL specification for STL iterators; @p/@a disambiguate between prefix/postfix operators

```
interface STLvector
<T, RI:-RandAccessIter<T,RI>; II:-InputIter<T,II> > {
  unsigned long getErasedSTL();
  RI begin ();   RI end();   T operator"[]"(in long n);
  void    insert(in RI pos, in long n, in T x);
  void    insert(in RI pos, in II first, in II last);
  RI      erase (in RI first, in RI last);
  void    assignAtIndex(in T obj, in long index);
  T       getAtIndex   (in long index);
  void    assign       (in II first, in II end);
  void    swap         (in STLvector<T, Ite, II> v); //....
};
```

Figure 14. GIDL specification for STL vector

structural similarity [1] with its qualifier `RandAccessIter`. Note that, unlike its underlying architecture, GIDL supports operator and method overloading.

As observed in [8], the GIDL interface is expressive, self-describing, and enforces the STL specification requirements at a high-level. Another interesting aspect is that GIDL stub wrappers for iterators are themselves valid STL iterators: They encapsulate the functionality specified by STL. They can also encapsulate the necessary type aliasing definitions, either by specifying them directly in the GIDL specification, or by making the GIDL stub wrapper extend the STL base class of their corresponding iterator category. For example `InputIter` stub extends the STL class `input_iterator<T,int>`. The latter is achieved by enriching the GIDL specification with meta data.

5.2 Implementation Approaches

GIDL is designed to be a *generic* extension framework that can plug in various back-ends as underlying architectures. An orthogonal, but nevertheless important, direction is to employ GIDL as middleware for exporting generic libraries' functionality to different environments than those for which they were originally designed. Our approach is to use a *black-box* translation scheme that wraps the library objects into GIDL objects and to study what other constructs are required to enforce the library semantics.

```
template <class T,class It,class It_impl,class II>
class STLvector_Impl :
  virtual public ::POA_GIDL::STLvector<T, It, II>,
  virtual public ::PortableServer::RefCountServantBase
{
  private:    vector<T>* vect;
  public:
    STLvector_Impl() { vect = new vector<T>(10); }
    virtual GIDL::UnsignedLong_GIDL  getErasedSTL()
        { return (CORBA::ULong)(void*)vect; }
    virtual void assign(T& val, GIDL::Long_GIDL& ind)
        { (*vect)[ind] = val; }
    virtual T getAtIndex(GIDL::Long_GIDL& ind)
        { return (*vect)[ind]; }
    virtual T operator[](GIDL::Long_GIDL& a1_GIDL)
        { return (*vect)[a1_GIDL]; }
    virtual It erase( It& it1_GIDL, It& it2_GIDL ) {
      T* it1 = (T*)it1_GIDL.getErasedSTL();
      T* it2 = (T*)it2_GIDL.getErasedSTL();
      vector<T>::iterator it_r = vect->erase(it1, it2);
      It_impl* it_impl = new It_impl(it_r, vect->size());
      return (*it_impl->_thisGIDL());
    }  // ...
};

template<class T,class It,class It_impl>
class InputIter_Impl :
  virtual public POA_GIDL::InputIter<T, It>,
  virtual public BaseIter_Impl<T, It, It_impl>,
  virtual public ::PortableServer::RefCountServantBase
{
  // private: T* iter; field inherited from BaseIter_Impl
  public:
    virtual It cloneItGIDL()
        { return (new It_impl(iter))->_thisGIDL(); }
    virtual GIDL::UnsignedLong_GIDL getErasedSTL()
        { return (CORBA::ULong)(void*)iter; }
    virtual T operator*()   { return *iter; }
    virtual GIDL::Boolean_GIDL operator==(It& it1_GIDL) {
      CORBA::ULong d1 = this->iter;
      CORBA::ULong d2 = it1_GIDL.getErasedSTL();
      return (d1==d2);
    };
};
```

Figure 15. GIDL vector and input iterator server implementations.

Figure 15 exemplifies our approach. Each implementation of a GIDL type holds a reference to the corresponding STL object that can be accessed via the `getErasedSTL` function in the form of an `unsigned long` value. The implementation of the `erase` function retrieves the STL objects corresponding to the GIDL wrapper parameters, calls the STL `erase` function on the STL vector reference, and creates a new GIDL server corresponding to the iterator result. Note that the semantics of the `erase` function are irrelevant in what the translation mechanism is concerned.

The GIDL code in Figure 16 provides, in our opinion, the look and feel of regular STL code. The only thing that differs are the types for the vector and iterators (lines 1-4). A vector is obtained in line 6. The `rai_beg` and `rai_end` iterators point to the start and the end of the vector element sequence. Then the loop in lines 12-15 assigns new values to the vector's elements.

There are, however, *two problems* with the current implementation. The first appears in line 14 where *dereferencing is followed by an assignment* as in `*rai=val`. In C++ this assigns the value `val` to the iterator's current element. The GIDL code does not accomplish this: the result of the `*` operator is a `Long_GIDL` object whose value is set to `val`. The iterator's current element is not updated as no request is made to the server. The origin of this problem is that GIDL does not support reference-type results, since the implementation and client code are not assumed to share the same process space.

```
1.  typedef GIDL::Long_GIDL   Long;
2.  typedef GIDL::RAI<Long>   rai_Long;
3.  typedef GIDL::InpIt<Long> inp_Long
4.  typedef GIDL::STLvector<Long,rai_Long,rai_Long>
5.          Vect_Long;
6.  Vect_Long vect   = ...;
7.  rai_Long iter    = vect.begin();
8.  rai_Long rai_end = vect.end();
9.  rai_Long rai_beg = iter;          // problem 2
10.
11. int count = 0;
12. while( rai_beg!=rai_end ) {
13.     if(*rai_beg!=33)
14.         *rai_beg++ = count++;     // problem 1
15. }
16. cout<<*iter<<endl;
```

Figure 16. GIDL client code that uses the STL library.

The second problem surfaces in line 16, where the user intends to print the first element of the vector. The copy constructor of the GIDL wrapper *does not create* a new implementation object, but instead *aliases* it: After line 9 is executed, both rai_beg and iter share the same implementation. Consequently, at line 16 all three iterators point to the end of the vector. The easy fix is to replace line 9 with `rai_Long rai_beg = iter.clone()` or with `rai_Long rai_beg = iter+0`. We are aiming, however, for a higher degree of composition between GIDL and STL components, where for example GIDL iterators can be used as parameters to STL algorithms. Since the STL library code is out of our reach, the direct fix is not an option.

One way to address the first problem is to introduce a new GIDL parameterized type, say WrapType<T>, whose object-implementation stores a T value while its GIDL interface provides accessors for it: `interface WrapType<T> { T get(); void set(in T t) }`. WrapType is a special GIDL type: its constructors and assignment operators call the set function, while its cast operator calls the get function to return the encapsulated T-type object. Instantiating the iterator and vector over WrapType<T> instead of T fixes the first issue. The main drawback of this approach is that it adds an extra indirection. In order to get the T type object two server calls are performed instead of one. Furthermore, it is not user-transparent, as the iterators and vectors need to be instantiated over the WrapType type. The next section discusses the techniques we employed to deal with these issues.

5.3 Trappers and Wrappers

We preserve the STL's programming idioms under GIDL by extending the GIDL wrapper with yet another component that enforces the library semantics. Figure 17 illustrates our approach. RaiIt_Lib refines the behavior of its corresponding GIDL wrapper RAI to match the library semantics.

First, it provides two sets of constructors and assignment operators. The one that receives as parameter a library wrapper object *clones* the iterator implementation object, while the other one aliases it. The change in Figure 16 is to make rai_Long and Vect_Long alias RaiIt_Lib<Long> and STLvect_Lib<Long,rai_Long,rai_Long> types, respectively. Now iter/rai_end alias the implementation of the iterators returned by the begin/end vector operations, while rai_beg clones it (see lines 7, 8, 9). At line 16 iter points to the first element of the vector, as expected.

Second, the RaiIt_Lib class defines a new semantics for the * operator that now returns a Trapper object. At a high-level, the *trapper* can be seen as a proxy for performing read/write operations. It captures the container and the index and uses container-methods to perform the operation. The "trapper" in Figure 17 extends its type parameter, and thus inherits all the type parameter operations. In addition it refines the assignment operator of T to call an iterator method to update its elements. This technique solves the problem encountered at line 14 in Figure 16 and it can be applied in a more general context to extend GIDL with *reference-type results*. Note that the use of the *trapper* is transparent for the user. The type TrapperIterStar does not appear anywhere in the client code. Furthermore, objects belonging to this type can be stored and manipulated as T& objects. For example, `T& t = *it; if(t<0) t=-t;` will successfully update the iterator's current element. This requires however that the GIDL wrappers declare the =(T&) operator *virtual*.

```
template<class T,class Iter> class TrapperIterStar : public T {
  protected:
    Iter it;
  public:
    TrapperIterStar(const Iter& i)
      { it = i; obj = (*it).getOrigObj(); }
    TrapperIterStar(const TrapperIterStar<T,Iter>& tr)
      { it = tr.it; obj = (*it).getOrigObj(); }

    void operator=(const T& t)
      { it.assign(t); obj = t.getOrigObj(); }
    void operator=(const TrapperIterStar<T,Iter>& tr)
      { it.assign(tr.getOrigObj()); obj = tr.getOrigObj(); }
};

template<class T> class RaiIt_Lib : public GIDL::RAI<T::Self> {
  private:
    typedef GIDL::RAI<T>                           It;
    typedef TrapperIterStar<T,It>                  Trapper;
    typedef GIDL::BaseObject<It,::RAI,::RAI_var>   GIDL_BT;
  public:
    typedef T                                      Elem_Type;
    typedef Self                                   It;

    RaiIt_Lib()           : GIDL_BT()               {}
    RaiIt_Lib(const It& r): GIDL_BT(r.getOrigObj()) {}
    RaiIt_Lib(const RaiIt_Lib<T>& r)
                : GIDL_BT(r.cloneIt().getOrigObj()) {}

    operator It()                  { return *this; }
    Trapper  operator*()  { return Trapper( *this ); }

    void operator=(const It& iter)
       { setOrigObj(iter.getOrigObj()); }
    void operator=(const InpIt_Lib<T>& iter)
       { setOrigObj(iter.cloneIt().getOrigObj()); }
};

template<class T,class RI,class II> class Vect_Lib
: public GIDL::STLvector<T::Self,RI::Self,II::Self>{...}
```

Figure 17. Library Iterator Wrapper and its associated Trapper that targets ease of use.

We conclude this section with several remarks. It is easy to anticipate how GIDL metadata can drive the compiler to generate the library wrapper code that captures the library semantics. All that is needed is the name of a method-member: cloneIt for the iterator's copy constructor and assign for the type-reference result. When available, the library wrappers should replace the GIDL corresponding types. For example, when using an STL algorithm with GIDL iterators, the former should be parameterized by the library wrapper types. Finally, note that nesting library wrappers is safe: We have that `RaiIt_Lib<RaiIt_Lib<Long> > it; **it=5;` works correctly. Also, the use of the Self abstract type member in the extension clause of the iterator/vector library wrappers ensures that the their inherited operations return GIDL wrapper objects. Therefore no unnecessary cloning operation are performed:

```
Vect_Lib<Long,RaiIt_Lib<Long>,RaiIt_Lib<Long> > v;
RaiIt_Lib<Long> it = vect.begin();
```

```
template<class T,class Iter> class TrapperIterStar {
  protected:    Iter it;
  public:
    TrapperIterStar(const Iter& i)   { it = i; }
    TrapperIterStar(const TrapperIterStar<T,Iter>& tr)
        { it = tr.it; }
    operator T()                     { return *it; }

    TrapperIterStar<T::Elem_Type, T> operator*() const
        { return *(*it); }
    void operator=(const TrapperIterStar<T,Iter>& trap)
        { it.assign(trap.it.operator*()); }
    void operator=(const T& t)       { it.assign(t); }
};
```

Figure 18. Trapper model that targets performance

Trapper Type	200000	20000	2000	200
EOU trapper	13.4	11.7	5	3.4
Perf. trapper I	1	1.4	1.5	1.68
Perf. trapper II	1	1.05	1.16	1.17

Table 2. The table shows the time ratio between trapper-based and optimal STL code that tests the read/write operation on the iterator's elements. The size of the iterator is varied from 200 to 200000.
EOU trapper = the one in Figure 17 (ease of use).
Perf Trapper I = the one in Figure 18 (performance).
Perf Trapper II = improved version of the latter, which by-passes the extra indirection introduced by the GIDL wrappers.

5.4 Ease of use - Performance Trade-off

The *trapper's* design is a trade-off between performance and ease of use. The implementation above targets ease of use, since a trapper object can be disguised and manipulated under the form of a `T&` object. An alternative, targeting performance, can model the trapper as a read/write lazy evaluator as shown in Figure 18. Note that the mix-in relation is cut off, and instead the support for nested iterators is achieved by exporting the `*` operator. It follows that the trapper cannot be captured as a `T&` object and used at a later time. The intent is that a trapper is subject to exactly one read or write operation (but not both), as in: `T t = *it++; *it = t; t.method1();`. The trapper's purpose is to postpone the action until the code reveals the type of the operation to be performed (read or write). Consequently, the constructors and the `=` operators are lighter, while a write operation accesses the server only once (instead of twice). Furthermore, this approach does not require the `=` operator to be declared *virtual* in the GIDL wrapper.

Table 2 shows the trapper-related performance results. Notice that the code using the trapper targeting ease of use is from 3.4 to 13.4 times slower than the optimal STL code, while the one targeting performance incurs an overhead of at most 68%. As the iterator size increases, the cache lines are broken and the overhead approaches 0. The test programs were compiled with the *gcc* compiler version $3.4.2$ under the maximum optimization level (`-O3`), on a 2.4 GHz Pentium 4 machine.

We found the *trapper* concept quite useful and we employed it to implement the GIDL arrays. The previous design was awkward in the sense that, for example, the `Long_GIDL` class was storing two fields: an `int` and a pointer to an `int`. The latter pointed to the address of the former when the object was not an array element and to the location in the array otherwise. All the operations were effected on the pointer field. By contrast, the *trapper* technique allows a natural representation consisting of only one `int` field.

6. Conclusions

We have examined a number of issues in the extension of generic libraries in heterogeneous environments. We have found certain programming language concepts and techniques to be particularly useful in extending libraries in this context: GADT, *abstract type members* and *traits*. Generic libraries that are exported through a language-neutral interface may no longer support all of their usual programming patterns. We have shown how particular language bindings can be extended to allow efficient, natural use of complex generic libraries. We have chosen the STL library as an example because it is atypically complex, with several orthogonal aspects that a successful component architecture must deal with. The techniques we have used are not specific to the STL library, and therefore may be adapted to other generic libraries. This is a first step in automating the export of generic libraries to a multi-language setting.

References

[1] P. Canning, W. Cook, W. Hill, and W. Olthoff. F-Bounded Polymorphism for Object Oriented Programming. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 273–280, 1989.

[2] A. S. David R. Musser, Gillmer J. Derge. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley (ISBN 0-201-37923-6), 2001.

[3] R. E. Johnson. Type Object. In *EuroPLoP*, 1996.

[4] A. Kennedy and C. V. Russo. Generalized Algebraic Data Types and Object-Oriented Programming. In *Proceedings of the 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–40, 2005.

[5] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 conference*, 2000.

[6] Microsoft. DCOM Technical Overview. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp, 1996.

[7] Sun Microsystems. JavaBeans http://java.sun.com/products/javabeans/reference/api/, 2006.

[8] C. E. Oancea and S. M. Watt. Parametric Polimorphism for Software Component Architectures. In *Proceedings of the 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 147–166, 2005.

[9] M. Odersky and al. Technical Report IC 2004/64, an Overview of the Scala Programming Language. Technical report, EPFL Lausanne, Switzerland, 2004.

[10] M. Odersky, V. Cremet, C. Rockl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proceedings of ECOOP'03*.

[11] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of the 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 41–57, 2005.

[12] OMG. Common Object Request Broker Architecture — OMG IDL Syntax and Semantics. Revision2.4 (October 2000), OMG Specification, 2000.

[13] OMG. Common Object Request Broker: Architecture and Specification. Revision2.4 (October 2000), OMG Specification, 2000.

[14] J. Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley and Sons, 2000. "Wiley computer publishing.".

[15] Sun. Java Native Interface Homepage, http://java.sun.com/j2se/1.4.2/docs/guide/jni/.

[16] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, and R. S. Sutor. *AXIOM Library Compiler User Guide*. Numerical Algorithms Group (ISBN 1-85206-106-5), 1994.

[17] H. Xi, C. Chen, and G. Chen. Guarded Recursive Data Type Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 224–235, 2003.

Adding Syntax and Static Analysis to Libraries via Extensible Compilers and Language Extensions*

Eric Van Wyk
University of Minnesota
evw@cs.umn.edu

Derek Bodin
University of Minnesota
bodin@cs.umn.edu

Paul Huntington
University of Minnesota
johnspa@cs.umn.edu

ABSTRACT

We show how new syntactic forms and static analysis can be added to a programming language to support abstractions provided by libraries. Libraries have the important characteristic that programmers can use multiple libraries in a single program. Thus, any attempt to extend a language's syntax and analysis should be done in a composable manner so that similar extensions that support other libraries can be used by the programmer in the same program. To accomplish this we have developed an extensible attribute grammar specification of Java 1.4 written in the attribute grammar specification language Silver. Library writers can specify, as an attribute grammar, new syntax and analysis that extends the language and supports their library. The Silver tools automatically compose the grammars defining the language and the programmer-selected language extensions (for their chosen libraries) into a specification for a new custom language that has language-level support for the libraries. We demonstrate how syntax and analysis are added to a language by extending Java with syntax from the query language SQL and static analysis of these constructs so that syntax and type errors in SQL queries can be detected at compile-time.

1. INTRODUCTION

Libraries play a critical role in nearly all modern programming languages. The Java libraries, C# libraries, the C++ Standard Template Library, and the Haskell Prelude all provide important abstractions and functionality to programmers in those language; learning a programming language now involves learning the intricacies of its libraries as well. The libraries are as much a part of these languages as their type systems. Using libraries to define new abstractions for a language helps to keep the definition of the language simpler than if these features where implemented as first class constructs of the language.

*Different aspects of this work are partially funded by NSF CAREER Award #0347860 and the McKnight Foundation.

LCSD '06 Portland, Oregon USA

An important characteristic of libraries is their compositionality. A programmer can use multiple libraries, from different sources, in the same application. Thus, libraries that support specific domains can be used in applications with aspects that cross multiple domains. For example, a Java application that stores data in a relational database, processes the data and displays it using a graphical user interface may use both the JDBC and the Swing libraries. Furthermore, abstractions useful to much smaller communities, such as the computational geometry abstractions in the CGAL C++ library, can also be packaged as libraries.

Libraries have a number of drawbacks, however. As mechanisms for extending languages they provide no means for library writers to add new syntax that may provide a more readable means of using the abstraction in a program. Traditional libraries provide no effective means for library writers to specify any static semantic analysis that the compiler can use to ensure that the library abstractions (methods or functions) are used correctly by the programmer. When libraries embed domain specific languages into the "host" language, as the JDBC library embeds SQL into Java, there is no means for statically checking that expressions in the embedded language are free of syntax and type errors. This is a serious problem with the JDBC library since syntax and type errors are not discovered at compile time but at run time. Traditional libraries also provide no means for specifying optimizations of method and function calls.

These drawbacks, especially in libraries for database access, have led some to implement the abstractions not as libraries but as constructs and types in the language. There is an trend in database systems towards more tightly integrating the application program with the database queries. Jim Gray [10] calls this removing the "inside the database" and "outside the database" dichotomy. In many cases, this means more tightly integrating the Java application program with the SQL queries to be performed on a database server. SQLJ is an example of this. Part 0 of the SQLJ standard [7] specifies how static database queries can be written directly in a Java application program. An SQLJ compiler checks these queries for syntax and type errors. This provides a much more natural programming experience than that provided by a low level API such as JDBC (Java DataBase Connector) which require the programmer to treat database query commands as Java Strings that are passed, as strings, to a database server where they are not checked for syntactic or type correctness until run time. More recently, $C\omega$ [3] and the Microsoft LINQ project [15] have extended C# and the .Net framework to directly support the querying of relational data.

These extended languages have added relational data query constructs because the technologies have matured to a relatively stable point and because very many programs are written that can make use of these features. Thus, if one is working in this domain, one can benefit from a language that directly supports the task at hand. Programmers working in less popular domains, however, are left with the library approach as it is the only way in which their domain-specific abstractions can be used in their programs. In the approach of SQLJ, Cω, and LINQ, a new monolithic language with new features is created, but there is no way for other communities to further extend Java or C# with new syntax and semantic analysis to support their domains.

In this paper we present a different, more general, approach to integrating programming and database query languages based on extensible languages and illustrate how new syntax and static analysis can be added to library-based implementations of new abstractions. The key characteristic of this approach is that multiple language extensions can be composed to form a new extended language that supports all aspects of a programming task. We have developed several modular, composable, language extensions to Java. In this paper we describe the extension that embeds SQL into Java to provide syntax and type checking for SQL queries and thus supports the implementation of these features in the JDBC library. We have built other extensions with domain-specific language features; one specifies program transformations that simplify the writing of robust and efficient computational geometry programs. Another general purpose extension adds pattern matching constructs from Pizza [17] to Java. Java and the language extensions are all specified as attribute grammars written in the attribute grammar specification language Silver. The Silver tools can automatically compose the grammars defining the host language Java and a programmer selected set of extensions to create a specification of a custom extended version of Java that has the features relevant to different specific domains. The tools then translate the specification to an executable compiler for the language.

Section 2 introduces the extensible language framework and its supporting tools. Section 3 describes a modular SQL extension to Java that we have constructed in order to illustrate what is possible in the framework. Section 4 provides the specifications of a subset of Java (Section 4.1) and some of the extension constructs (Section 4.2) to illustrate how the full extension in Section 3 was implemented. Section 5 describes related work, future work, and concludes.

2. EXTENSIBLE LANGUAGE SPECIFICATIONS AND SUPPORTING TOOLS

An extensible compiler allows the programmer to import the unique combination of general-purpose and domain-specific language features that raise the level of abstraction to that of a particular problem domain. These features may be new language constructs, semantic analyses, or optimizing program transformations, and are packaged as *modular language extensions*. Language extensions can be as simple as a *for-each* loop that iterates over collections or the set of SQL language constructs described in this paper.

To understand the type of language extensibility that we seek, an important distinction is made between two activities: (*i*) *implementing a language extension*, which is performed by a domain-expert feature designer and (*ii*) *selecting the language extensions* that will be imported into an extensible language in order to create an extended language. This second activity is performed by a programmer. This is the same distinction seen between library writers and library users. This distinction and the way that extensible languages and language extensions are used in our framework is diagrammed in Figure 1.

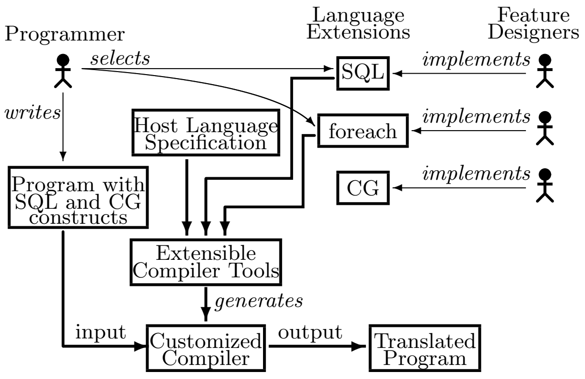


Figure 1: Using Extensible Languages and Language Extensions.

From the programmer's perspective, importing new language features should be as easy as importing a library in a traditional programming language. We want to maintain the compositional nature of libraries. They need only *select* the language extensions they want to use (perhaps the SQL and geometric (CG) extensions shown in Figure 1) and write their program to use the language constructs defined in the extensions and the "host" language. They need not know about the *implementation* of the host language or the extensions. The specifications for the selected language extensions and the host language are provided to the extensible compiler tools that generate a customized compiler. This compiler implements the unique combination of language features that the programmer needs to address the particular task at hand. Thus, there is an initial "compiler generation" step that the tools, at the direction of the programmer, must perform. Language extensions are not loaded into the compiler during compilation.

The feature designer's perspective is somewhat different; they are typically sophisticated domain experts with some knowledge of the implementation of the host language being extended. Critically, feature designers do not need to know about the implementations of other language extensions since they will not be aware of which language extensions a programmer will import. This paper shows how the functionality provided by a library can be enhanced by language extensions that provide new syntax to represent the abstractions provided by the library and new static analysis that can ensure that the library is used correctly.

2.1 Attribute grammars and Silver

Extensible languages and language extensions in our framework are based on *attribute grammars*. The extensible host language is specified as a complete attribute grammar and the language extensions are specified as attribute grammar fragments. These are written in Silver, an attribute grammar specification language developed to support this. The Silver extensible compiler tools combine the attribute grammar specifications of the host language and the programmer selected language extension to create an attribute grammar specification for the custom extended language desired by the programmer. An attribute grammar evaluator for this grammar implements the compiler for the extended language. We choose to define languages and extensions as attribute grammars, enhanced with with a mechanism called *forwarding* [19], because language specifications defined in the way can be automatically combined by the tools.

In Silver, attribute grammars are package as modules defining either a host language or a language extension. The module `edu:umn:cs:melt:java14` defines Java 1.4. It defines the concrete syntax of the language, the abstract syntax and the semantic analyses required to do most type checking analyses and to do package/type/expression name disambiguation. We continue to extend this with additional attribute definitions specifying additional static analyses. Our aim is to perform all static analyses performed by a traditional Java compiler. The grammar defines most aspects of a Java compiler but it does not specify byte-code generation. Language extensions add new constructs, like the SQL constructs described here, and their translation to pure Java 1.4 code, that a traditional Java compiler then converts to byte-codes for execution. The static analysis we perform is to support analysis of extensions and to ensure that any statically detectable errors (such as type errors and access violations) in the extended Java language are caught so that erroneous code is not generated. Programmers should not be expected to look at the generated Java code; errors should be reported on the code that they write.

In Section 4, we show Silver specifications for a simplified versions of our Java 1.4 grammar and the SQL grammar. We also show how the host language is composed with extensions to create an extended language specification.

3. A MODULAR LANGUAGE EXTENSION TO JAVA THAT EMBEDS SQL

In this section we describe a language extension that adds new constructs and semantic analyses that embed SQL into Java. The functionality is provided by the JDBC library — what is added is new syntax and analysis to statically detect syntax and type errors in SQL queries.

This extension specifies SQL productions for statically checking the syntax and type correctness of static SQL queries and queries which incorporate values from Java variables and expressions. It also allows for the dynamic creation of SQL queries in such a way that ensures that they are syntactically correct but it does not statically check the type correctness of dynamically generated queries. The integration component of the extension provides new constructs for registering database drivers, setting up connections, specifying and verifying the types of fields in tables on the database server that are used in the application, as well as executing queries and commands on the database server. The extended language will perform some semantic analysis on the SQL constructs to check for errors and then translate the constructs into pure Java code that uses the JDBC library. Figure 2 provides a code fragment written in the Java+SQL extended language. Figure 3 show the generated Java code fragment that the Java+SQL fragment translates to.

```
 1. register driver "com.mysql.jdbc.Driver" ;
 2. connection c="jdbc:mysql://db.domain.com/db..";
 3. import table person [ VARCHAR first_name ,
     VARCHAR last_name, INTEGER age ] ;
 4. ResultSet rs = using c query {
      SELECT last_name FROM person WHERE
         first_name LIKE "derek" };
 5. String s = "derek" ; int i = 4 ;
 6. rs = using c query { SELECT first_name
          FROM person WHERE first_name LIKE s };
 7. rs = using c query { SELECT first_name
         FROM person WHERE age > i } ;
 8. rs = using c query { SELECT first_name FROM person
         WHERE first_name LIKE $("der" + "ek") };
 9. sqlExpr e1=sql expr {person.first_name LIKE s};
10. sqlExpr e2=sql expr {e1 AND last_name LIKE "bodin"};
11. rs = using c query
      { SELECT first_name FROM person WHERE e2 };
```

Figure 2: SQL/Java example program fragment.

Static SQL queries: First consider lines 1-4 in Figure 2 that use constructs in the SQL extension to set up a connection to a database server and execute a static SQL query. Here three new statements added by the extension are used to register a database driver, to create a *connection* `c` to a specific database server, and to make the `person` table from that database available for use in the program. The `import table` construct specifies the fields and their type in the person database. At compile time, the declaration of the `person` table and the types of its fields are entered into an environment (symbol table) that is referenced during the type checking of the static queries.[1] Line 4 shows a statically specified SQL query.

In this example, since there are no syntax or type errors, the Java code in lines 1-4 of Figure 3 will be generated. The `register driver` and `connection` statement translate into the expected JDBC calls. The `import table` construct defines the type information used in the `using` and SQL constructs to perform static type checking, but it translates to the empty statement since it has no implementation in the generated Java code. The `using query` construct is translated to the expected JDBC `createStatement` and `executeQuery` methods calls on the JDBC Connection object `c`.

Accessing Java variables and expressions: The SQL

[1]Although not implemented in the current version of the extension, this construct could consult the database schema at compile time (as is done in SQLJ) to verify that the types in the database scheme match those given here.

```
 1. Class.forName ( "com.mysql.jdbc.Driver" ) ;
 2. Connection c = DriverManager.getConnection (
     "jdbc:mysql://db.domain.com:3306/db ...");
 3. // empty statement, was import
 4. ResultSet rs = c.createStatement().executeQuery(
     "SELECT " + "first_name" + " FROM " + "person" +
     "WHERE " + "first_name" + " LIKE " + "\"derek\" );
 5. String s = "derek" ;  int i = 4 ;
 6. rs = c.createStatement().executeQuery(
      "SELECT " + "first_name" + " FROM " + "person"
      + "WHERE " + "first_name" + " LIKE " + s);
 7. rs = c.createStatement().executeQuery(
      "SELECT " + "first_name" + " FROM " + "person"
      + "WHERE " + "age" + " > " + i);
 8. rs = c.createStatement().executeQuery(
      "SELECT " + "first_name" + " FROM " + "person"
      ++ "WHERE " + "first_name" + " LIKE " +
       ("der" + "ek"));
 9. String e1 = "person.first_name" + " LIKE " + s;
10. String e2 = e1 + " AND " + "last_name" +
                 " LIKE " + "\"bodin\"" ;
11. rs = c.createStatement().executeQuery(
      "SELECT " + "first_name" + " FROM " + "person"
       + "WHERE " + e2) ;
```

Figure 3: Translation to pure Java of SQL/Java example program fragment.

queries defined in the SQL extension can access values from Java variables and expressions and still statically check that they are syntactically and type correct. For example, consider the lines 4-7 of Figure 2. The Java+SQL compiler will type-check these queries and, since no type errors exist, generate the pure Java code seen in lines 4-7 of Figure 3 (modulo reformatting). The examples translate to Java code that creates a string that contains the query using the value of variable `s` or `i`. The third example, computes the value of the expression whose value is included in the query. The type checking of these queries is possible here because the productions in the language extension have access to the attributes of the host language attribute grammar that contain the names and types of the in-scope variables. Because `s` is not a field in the table `person` but is an in-scope local variable the reference to `s` in line 6 of Figure 2 is to the local variable. In line 8, the `$(...)` notation is used to embed Java expressions in SQL expressions.

Dynamic creation of SQL queries: The SQL extension also allows for the creation of dynamic SQL queries that can be statically verified to be syntactically correct. Consider lines 9–11 of Figure 2 that makes use of the new type `sqlExpr`. The variables `e1` and `e2` of type `sqlExpr` hold syntactically correct SQL expressions. SQL expressions are written according to the SQL grammar but are wrapped in a `sql expr` { ... } construct to avoid conflicts and ambiguities in the parser that may arise since the expression language of SQL and the expression language of Java are similar. The Java code generated for the SQL+Java code is show in lines 9–11 of Figure 3.

In the generated Java code of Figure 3, the queries are syntactically correct but the type information that is present in the extension constructs of Figure 2 is gone and queries are represented as strings. Thus, statically checking the syntactic correctness of queries constructed as strings requires a much more sophisticated analysis, like that found in the Java String Analyzer [5] and incorporated in the JDBC Checker tool [9]. The JDBC checker does check that dynamically created queries are also type correct. That is something that this simple extension does not do. Nothing prohibits that kind of sophisticated analysis being done by this tool however. In fact, extensible languages provide the right "hooks" for extracting information used by such analysis. The control flow information needed by JSA can be generated by appropriate attribute definitions.

Some may justifiably question some of the design decisions of this extension. For example, should we recognize `s` in line 6 of Figure 2 as a local variable, or should we require the use of the `$(...)` notation. It is not our aim to answer these sorts of questions. Our goal is to show how syntax and static analyses can be added to a programming language, not to argue the merits of specific SQL extension constructs.

4. EXTENSIBLE LANGUAGE IMPLEMENTATIONS VIA ATTRIBUTE GRAMMARS

In this section we describe some features of the host language attribute grammar specification to show how it is combined with the attribute grammar specification of a language extension, in this case the SQL extension, to create a specification for an extended language. Section 4.1 introduces attribute grammars and a portion of the specification of the host language. It also introduces an extension to attribute grammars, called *forwarding* that is useful in defining modular and composable language extensions. Section 4.2 shows a portion of the attribute grammar that defines the SQL language extension and discusses how it integrates SQL into Java. For presentation reasons, we have simplified and omitted some features of the specifications that do not aid in understanding how language features can be added as composable extensions.

4.1 Host language specification

Attribute grammars add a layer of semantics to the syntactic specifications provided by context free grammars by associating *attributes* with non-terminal symbols and attribute defining functions with productions. Attribute grammars as originally defined by Knuth [13] can be extended with a number of enhancements that makes them more practical to use for (modular) language specification. In our framework we incorporate higher-order attributes [21] which store syntax trees (without attribute values). These allow new language constructs (trees) to be generated and passed around the original syntax tree at compilation time. We also use a mechanism called *forwarding* [19] that makes the automatic combination of different language extensions possible.

A portion of a drastically simplified version of the Java 1.4 host-language attribute grammar specification is shown in Figure 4. It first specifies the name of this grammar module. These names, similar to Java package names, use Internet domain names to ensure uniqueness, and are used when grammars, representing languages and extensions, are composed. This process is described in Section 4.3. Next it defines a collection of non-terminal symbols; the non-terminal Stmt represents Java statements, Expr represents Java expressions, and Type for Java types. These nonterminals are used in the concrete productions to specify the parser for Java. The nonterminal TypeRep is used by abstract productions to represent types.

Next are specifications for the terminal symbol for identifiers (Id) and its defining regular expression and for semicolons. Next several synthesized attributes (syn attr) and inherited attributes (inh attr) are defined. Attributes label the nonterminal and terminal nodes in an object programs syntax tree. Synthesized attributes store information that propagates up the tree and thus productions define synthesized attributes that label their left-hand side non-terminal. Inherited attributes store information that propagates down the tree and thus productions define inherited attributes that label their right-hand side child non-terminals. The pp attribute of type String holds the pretty-print version of constructs it decorates. This attribute decorates (written occurs on) nonterminals Expr, Stmt, and Type. The inherited environment attribute env is a list of pairs mapping names to their type representations; it forms a symbol table that decorates Expr, Stmt, and Type. The typerep attribute decorates expressions to indicate their type and decorates type expressions (Type) to indicate their representation. An errors attribute is a list of strings.

Concrete productions (indicated by con prod) such in the local variable declaration production local_var_dcl are used by the parser generator. Abstract productions (indicated by abs prod) are not. Silver allows attributes to be defined in both concrete and abstract productions. The left and right-hand side nonterminals and some right-hand side terminal symbols are named. In local_var_dcl the nonterminal Stmt is named s and this name is used in the block of attribute definitions that follow the production signature. Some keyword and punctuation terminals symbols, defined in the manner of terminal SemiColon match just one lexeme and can be referenced in the production by that lexeme, as is done in local_var_dcl. This production defines pp as expected using the pretty print of the type expression t and the lexeme of the terminal id. Attributes values are referenced on nodes using the dot notation. It also defines its defs attribute to be the mapping from the name of the identifier to the representation of the type t. Definitions are collected according to the scope rules of the language to populate the env attribute. Details of this are elided.

The while production defines the Java while-loop. It defines the concrete syntax by specifying that a while loop is of nonterminal type Stmt and that it is composed of the keyword "while", a left paren, a condition of type Expr, a right paren, and a loop body of type Stmt. The pp attribute is defined as expected from the string literals and value of the pp attribute on the child non-terminals. In the specifications, square brackets denote lists and ++ denotes list, as well as string, concatenation. The pp attribute will label all nodes in the concrete syntax tree, but we will omit further definitions of pp on productions as their behavior is what is expected. The while loop also copies its env attribute to its children. When this is the expected behavior we will sometimes leave these copy rules out of the given specifications.

```
grammar edu:umn:cs:melt:java14;

nonterminal Expr, Stmt, Type, TypeRep ;
terminal Id / [a-zA-Z][a-zA-Z0-9_]* / ;
terminal SemiColon ';' ;

syn attr pp :: String occurs on Expr, Stmt, Type ;
syn attr name :: String occurs on TypeRep ;

inh attr env :: [ (String, TypeRep) ]
  occurs on Stmt, Expr, Type ;
syn attr defs :: [ (String, TypeRep) ] occurs on Stmt ;
syn attr typerep :: TypeRep occurs on Expr, Type ;
syn attr errors :: [ String ] ;

syn attr hostStmt :: Stmt occurs on Stmt ;
syn attr hostExpr :: Expr occurs on Expr ;
syn attr hostType :: Type occurs on Type ;

con prod local_var_dcl  s::Stmt ::= t::Type id::Id ';'
{ s.pp = t.pp ++ " " ++ id.lexeme ++ ";"
 s.defs = [ (id.lexeme, t.typerep) ] ;
 s.hostStmt = local_var_dcl(t.hostType,id); }

con prod while
s::Stmt ::= 'while' '(' cond::Expr ')' body::Stmt
{ s.pp = "while (" ++ cond.pp ++ ") \n" ++ body.pp ;
  cond.env = s.env ;
  body.env = s.env ;
  s.errors = (if cond.typerep.name != "boolean"
    then [ "Error: condition must be boolean"] else [ ])
    ++ cond.errors ++ body.errors ;
 s.hostStmt = while(cond.hostExpr,body.hostStmt);   }

con prod idRef  e::Expr ::= id::Id
{ e.typerep = lookup (e.env, id.lexeme) ;
  e.error = e.typerep.errors
  e.hostExpr = idRef(id); }

con prod booleanType  t::Type ::= 'boolean'
{ t.pp = "boolean" ;
  t.typerep = booleanTypeRep(); }

abs prod booleanTypeRep  tr::TypeRep ::=
{ tr.name = "boolean" ;
  tr.error = [ ] ; }

abs prod notFoundTypeRep  tr::TypeRep ::= n::String
{ tr.name = "NotFound" ;
  tr.error = ["Error " ++ n ++ "not found"]; }
```

Figure 4: Simplified Java host language Silver specification.

```
con prod driver
s::Stmt ::= 'register' 'driver' d::StringLiteral
{ s.pp = "register driver " ++ d.lexeme ++ ";"
  s.errors = ... check the d is valid ... ;
 forwards to ``Class.forName ( |s.lexeme| ) '' ;  }
```

Figure 5: SQL Driver construct specification.

The identifier reference production `idRef` passes the name `id.lexeme` to the `lookup` function to extract the `TypeRep` associated with that name from its environment `e.env`. If the name is not in the environment, `lookup` returns the `TypeRep` built by the `notFoundTypeRep` production. Type checking is done by name and implemented by examining the `name` attribute on a constructs `typerep` attribute. For example, in the `while` production, the name attribute on the type node of the condition expression is checked to see that it is "boolean". Type expressions (`Types`) are specified by concrete productions and the corresponding abstract productions construct their type representations which are used to label expressions with their type.

The attributes `hostStmt`, `hostExpr`, and `hostType` are used to generate the syntax tree in which all constructs defined in language extensions have been translated to their pure Java 1.4 representations. On each host language production, the `host` attribute for its left-hand side nonterminal is defined following the pattern shown in Figure 4. These attributes are not defined on language extension productions.

As stated, we provide only some of the simplified definitions of the Java host language that have been implemented in Silver. For example, not shown are the implementation of objects and classes which are a significant portion of the specification. But they are not critical here.

4.1.1 *Forwarding and language extensions*

Forwarding [19] is an extension to higher-order attribute grammars that allows new language constructs in modular language extensions to be defined in terms of existing host language constructs. But it also allows the explicit specification of semantic analyses (new ones specified by the extension or existing ones in the host language) by allowing productions to explicitly specify attribute definitions. We will use the SQL driver registration construct in line 1 of Figure 2 to illustrate this; its specification is shown Figure 5.

A valid but minimal language extension could simply rewrite this construct to the semantically equivalent statement shown in line 1 of Figure 3. While this would provide an implementation for the driver construct, it is inadequate because any errors made by the programmer would be reported as errors in the generated `Class.forName` construct that the programmer did not write. Forwarding solves this problem. It allows productions to define, in addition to attribute definitions, a semantically equivalent construct that they should, in essence, be translated to. This translation does not replace the existing construct however. This construct has the same non-terminal type as the left-hand side of the defining production. If, during compilation, a node in the syntax tree is queried for an attribute for which its production does not explicitly provide a definition then that query is forwarded to the semantically equivalent "forwards to" construct specified by the production which returns its value for that attribute. This construct also inherits, automatically, the same attribute values as the "forwarding" production.

In the case of the *driver* production in Figure 5, if a *driver Stmt* node is asked for its *errors* attributes, it returns the values computed by the *driver* production. The definition of this attribute it elided but will check if the string literal is a valid driver string — in Figure 2 it is the class name `com.mysql.jdbc.Driver`. This production also explicitly defines its `pp` attribute. Now consider the `host` attributes defined on all productions in the host language specification. These attributes contain the tree of the program in which the language extension constructs are translated to their representation in the host language. This is shown for the `while` production and others in Figure 4. If the `driver` construct is queried for the value of this attribute — as it would be during compilation — this query is passed to the forwarded-to construct and its *hostStmt* value is returned. This is how the extended language program is translated to one in the host language.

In Silver, the tree that a production forwards to is constructed using the names of the productions as tree-creation functions in which the parameters are the child trees. Throughout the paper we do not show these tree constructions but instead simply give a stylized string representing the concrete syntax of the constructed tree as this makes it easier to see what a construct forwards to. These strings use different quotes (`‘‘...’’` instead of `"..."`). Also, "holes" are specified by vertical bars (`| ... |` in which values from child trees can be included in the parameterized string. For example, in the specification of the `forwards to` construct of the `driver` construct, the lexeme of the terminal `s` is the parameter to the `Class.forName` construct. The vertical bars are in essence an unquote operator.

In the current Java implementation, we do not translate programs to Java byte code. If the *errors* attribute is empty, then the extended compiler outputs value of the *host* attribute on the root of the tree. This is pure Java code in which the language extensions have been compiled down to their representations in the host language.

4.2 Specification of the SQL extension

In Section 3 we described the SQL constructs in the SQL extension and showed what examples of the new constructs translate to. In this section we cover some of the attribute grammar specification of the SQL language extension. As in the Java specification, space limitations require us to present a simplified and reduced version of the actual specifications. We focus on how the SQL extensions work with the type system of the Java specification and how errors are collected to statically check type correctness of static queries.

The productions, nonterminals, and attributes defined in Figures 5 and 6 integrate SQL into Java. The table import production `sqlImport` and the `sqlQuery` production, for example, have Java defined non-terminals on their left-hand side, but, in most cases, SQL defined non-terminals on their right. The non-terminal *SqlQuery* (used by `sqlQuery`) and its productions are defined in Figure 7. The `sqlQuery` production reports errors from `sq::SqlQuery` and forwards to the JDBC code seen in the examples in Section 3.

Table Import Statement: In Figure 6, the `sqlImport` production adds to `defs` (and thus indirectly the the environment `env`) the table name and its type representation. This representation is just the environment (again as a list) consisting of column names and type representations specified in productions `sqlColType`, `sqlColTypesOne`, and

```
grammar edu:umn:cs:melt:java14:exts:sql ;

nonterminal SqlQuery, SqlColTypes, SqlColType ;
attribute defs, pp occurs on SqlQuery,
  SqlColTypeList, SqlColType ;

-- Table import and table type specifications --
con prod sqlImport
s::Stmt ::= 'import' 'table' t::Id
            '[' cols::SqlTypeCols ']' ';'
{ s.defs = [(t.lexeme, sqlTableTypeRep(cols.defs) ) ];
  s.pp = "import table [" ++ "] ;" ;
  forwards to skip();    }

con prod sqlColTypesOne
stfs::SqlColTypeList ::= stf::SqlColType
{ stfs.pp = stf.pp ;   stfs.defs = stf.defs ; }

con prod sqlColTypesCons
stfs::SqlColTypes ::= stf::SqlColType ','
                      stfs2::SqlColTypes
{ stfs.pp = stf.pp ++ stfs2.pp ;
  stfs.defs = stf.defs ++ stfs2.defs ; }

con prod sqlColType
stf::SqlColType ::= t::SqlType f::Id
{ stf.pp = t.pp ++ " " ++ f.lexeme ;
  stf.defs = [ ( f.lexeme, t.typerep ) ; }

-- SQL Column Types --
con prod sqlIntegerType st::SqlType ::= 'INTEGER' {..}
abs prod sqlIntegerTypeRep tr::TypeRep ::=        {..}
con prod sqlVarCharType st::SqlType ::= 'VARCHAR' {..}
abs prod sqlVarCharTypeRep tr::TypeRep ::=        {..}

abs prod sqlTableTypeRep
tr::TypeRep ::= cols::[(String, TypeRep)]
{ tr.name = "SqlTable" ;    tr.errors = [ ];
  tr.tableEnv = cols ;    }

syn attr tableEnv :: [(String, TypeRep)]
  occurs on TypeRep ;

-- Sql Query integration --
con prod sqlQuery
e::Expr ::= 'using' c::Id 'query' '{' sq::SqlQuery '}'
{ e.pp = "using " ++ c.lexme ++ " query {" ++
         sq.pp ++ "}" ;
  e.typerep = ... Java class ResultSet ... ;
  e.errors = sq.errors ;
  forwards to ``|id.lexeme|.createStatement().
               executeQuery(|sq.javaExpr|)'' ;   }

-- New Java Types for Sql Exprs for dynamic queries --
con prod sqlExprType  t::Type ::= 'sqlExpr'
{ t.pp = "sqlExpr" ;
  t.typerep = sqlExprTypeRep(); }

abs prod sqlExprTypeRep  tr::TypeRep ::= 'sqlExpr'
{ tr.name = "sqlExpr" ;
  tr.errors = [ ] ; }
```

Figure 6: Productions linking Java and SQL

sqlColTypesCons. These productions create this environment using the defs attribute. This is then packaged as a TypeRep by the production sqlTableTypeRep. This environment will be used to look up the type of column names used in SQL queries. The attribute definitions of the productions for SQL types (nonterminal SqlType) INTEGER and VARCHAR and type representations (nonterminal SqlType) are elided but define their pp, typerep, name, and errors attributes in just the same manner as the Java type in Figure 4 and the Java type slqExpr do. The production forwards to the Java skip statement since it has no Java representation. An alternative implementation would be to forward to Java code that verifies, at run time, that the specified type of the table matches the schema of the table in the database.

SQL Query: Also in Figure 6, the production sqlQuery defines the concrete syntax for the *using ... query* construct which is a Java Expr. It integrates the SQL queries with Java. It defines its pp as expected, its errors are those discovered in the SQL query sq, and its typerep is the representation of the Java class ResultSet. It forwards to the method calls (as illustrated in Figure 3) on the connection. The parameter to the executeQuery method is the Java string-valued expression generated by the SQL constructs shown in Figure 7 and stored in the attribute javaExpr.

In Figure 7 are the non-terminals and productions that define (a small subset of) SQL expressions. Syntax errors in SQL query expressions are detected by the parser since they are written directly in the object program (between "{" and "}") and not encapsulated in strings. Type errors are computed in much the same way as in the Java host language.

The production sqlSelect extracts the environment tableEnv from the type representation of the SQL table. This is passed down the SQL syntax tree in the colenv attribute where it is used by the sqlId production to look up the types of column names. This type information is then used to type-check the SQL queries. In production sqlId the identifier id could be a Java identifier or a column reference. If it is found in the environment containing column names, the attribute colenv, then it is a column reference and its javaExpr is the lexeme of that identifier with wrapped in quotes as a Java string literal. For example, in line 6 of Figure 2 the id first_name in the WHERE clause would be found in the column environment colenv. Thus, its translation to Java is the literal "first_name" seen in line 6 of Figure 3. If the identifier is found in the standard environment env then it is a Java variable and its javaExpr is the Java variable with that name. For example, in line 6 of Figure 2, s in the WHERE clause would be found in the environment env. Its translation to Java is a variable s seen in line 6 of Figure 3. The types of Java variables are then converted to SQL types. The converted local attribute in sqlId presents a simplified (from the actual implementation) mechanism for doing this. Java strings and integers are converted to their SQL versions. Java sqlExpr types used in dynamic queries are handled differently and discussed below.

Dynamic Queries: Dynamic creation of SQL queries creates a number of challenges for statically type checking SQL queries. The basis of the problem is that the Java identifiers that contain SQL expressions (e1 and e2 in Figure 2) have

```
grammar edu:umn:cs:melt:java14:exts:sql ;

nonterminal SqlQuery, SqlExpr ;
attribute pp occurs on SqlQuery, SqlExpr ;

inh attr colenv :: [(String,TypeRep)]
  occurs on SqlQuery, SqlExpr ;
syn attr javaExpr::Expr occurs on SqlQuery, SqlExpr;

con prod sqlSelect
sq::SqlQuery ::= 'SELECT' fields::SqlExpr 'FROM'
             table::Expr 'WHERE' where::SqlExpr
{ sq.pp = ... ;
  sq.javaExpr = `` "SELECT " + |fields.javaExpr| +
   "FROM" + |table.pp| + "WHERE" + |where.javaExpr|'';
  sq.errors = fields.errors ++ where.errors ++
     if table.typerep.name == "SqlTable" then [ ]
     else [ "Error: table must have type SqlTable" ]
  fields.env = sq.env ;
  fields.colenv = table.typerep.tableEnv ;
  where.env = sq.env ;
  where.colenv = table.typerep.tableEnv ;    }

con prod sqlId
se::SqlExpr ::= id::Id
{ se.pp = id.lexeme ;
  se.javaExpr = if sqltype.name != "NotFound"
                then `` "|id.lexeme|" ''
                else ``  |id.lexeme|  '' ;
  se.typerep = if sqltype.name != "NotFound"
               then sqltype else converted ;
  se.errors = se.typerep.errors ;
  local javatype :: TypeRep
        = lookup(se.env,id.lexeme);
  local sqltype  :: TypeRep
        = lookup(se.colenv,id.lexeme);
  local converted :: TypeRep
        = if javatype.name == "String"
          then sqlVarCharTypeRep()
     else if javatype.name == "int"
          then sqlIntegerTypeRep()
     else if javatype.name == "sqlExpr"
          then sqlExprTypeRep()
     else notFoundTypeRep(id.lexeme);
}
```

Figure 7: SQL query and expression specifications.

```
grammar edu:umn:cs:melt:java_sql ;

import edu:umn:cs:melt:java14 ;
syntax edu:umn:cs:melt:java14 ;
import edu:umn:cs:melt:java14:exts:sql ;
syntax edu:umn:cs:melt:java14:exts:sql ;
import edu:umn:cs:melt:java14:exts:rlp;
syntax edu:umn:cs:melt:java14:exts:rlp;

import core ;
abstract production main top::Main ::= args::String
{ forwards to java_main(args, parse) ;  }
```

Figure 8: Composed language Silver specification.

the Java type specified by the production `sqlExprTypeRep`. The information about the SQL type is not present in the particular type representation scheme used here. Thus, it cannot be determined if the SQL expression is, for example, an SQL `INTEGER` or `VARCHAR`. In the current implementation of the SQL language extension we do not attempt to statically type check dynamically generated queries. Instead, Java variables of type `sqlExpr` are given, in the production `sqlId` the converted type of `sqlExprTypeRep()`. In type checking, this type deemed to be compatible with all other types so that no errors are generated for such identifiers. Although checking such queries is possible and is done by other tools, such as the JDBC Checker [9], our goals here are not to create the most sophisticated embedding of SQL into Java. They are to show how syntax and static analysis can be added to to host language to support the functionality provided by a library.

4.3 Composition of Java and Language Extension Specifications

As outlined in Section 2, programmers compose host language and language extensions with no implementation level knowledge of the language or the extensions but need only select the desired extensions. We are currently developing an Eclipse plug-in for the extensible compiler framework that supports this selection process. The plug-in will automatically generate from the list of selected extensions the Silver specification that composes the host language and selected extensions. In Figure 8 is the Silver specification that would be generated if the programmer selected the SQL extension described above and the computational geometry extension that implements the randomized linear perturbation (`rlp`) scheme for handling data degeneracies in geometric algorithms. What the `rlp` extension does specifically is not of interest here. This composed extended language has features to support both the domains of relational database queries and computational geometry. The `import` statements import the grammar specifications in the named Silver module. The `syntax` statements import the concrete syntax specifications from the named modules to build the parser for the extended language. The final three lines import utility types like `Main`. The `main` production is similar in intent to the C `main` function; here it delegates to the main production `java_main` in the `java14` host specification. Although this shows that correctly-specified extensions can be easily composed, it is possible to write Silver specifications that when composed with the host language do not result in well-defined attribute grammars. Section 5.2 discusses issues of syntactic and semantic composability of extensions.

5. CONCLUSION

5.1 Related Work

There are other languages such as SQLJ [7], Cω [3] and .NET languages like C# that use the LINQ [15] .NET project that provide a more complete embedding of SQL constructs than we have specified in the SQL language extension above. Cω and LINQ also address the mismatch between the types in SQL and the host language. For example, SQL `INTEGER` types can have the value `NULL`, but Java `int` types cannot. Other mismatches between the object view and the relational table view of data are also addressed in these languages but not in the extension described in this paper.

These are new, well-crafted, useful languages, but they are *monolithic* solutions. They cannot be extended to provide the same sort of language and analysis support to other domains. The extensible language framework presented here illustrates how such language and analysis support can be provided in an extensible manner.

Similar work has been done in extending Java with XML language constructs. For example, JWig [6] is a Java based framework that allows programmers to write XML directly in Java. The framework analysis ensures that all statically and dynamically generated XML segments are syntactically correct XML. JWig does this by using a Java parser extended with new rules for, among other constructs, XML. This parser outputs pure Java which a standard Java compiler converts to byte-code, much like our framework. JWig then analyzes the byte-code using the Java Syntax Analyser [5]. It statically verifies that the XML segments generated and used in the original JWig program file are valid XML. One problem with this approach is that errors are reported by the Java compiler on the Java code generated by JWig. This can be confusing to programmers and is something that can be avoided in our framework since extensions can define their own error-checking analysis. This problem is shared by macro-based approaches to language extension. Some modern macro systems, such as Maya [1] and JTS [2], do however provide specific error checking facilities.

There is a significant amount of work in the language processing tools community for building extensions to languages. For example, the Polyglot extensible Java compiler [16] allows Java to be extended with powerful abstractions such as pattern matching [14]. However, this system requires one to write Java code to incorporate new extensions into Java. In the extensible language framework we propose, extensions selected by the programmer are naturally and automatically composed to form a new extended compiler. MetaBorg[4] is another system that allows one to extend a host language by adding concrete syntax for objects. This system is based on StrategoXT [20] and uses strategies and term-rewriting to process programs. Specifying semantic analyses, like the error checking, is less straight forward than it is using attributes and it is not clear that different extensions can be combined automatically.

The attribute grammar community has also addressed issues of modular language design. Of particular interest are the rewritable reference attribute grammars [8] in the JastAddII system. An extensible Java 1.4 compiler has also be specified in this system. Language extension constructs are translated to host language constructs by destructive rewrites on the syntax tree. Thus *all* analysis on an extension construct must be completed before any analysis on its translation to the host language. Although forwarding is similar to rewriting, it is non-destructive so that the original tree and the forwards-to tree exist simultaneously. This turns out to be important when multiple extensions introduce new semantic analyses because a construct will need both trees — the original for its analysis and the forwards-to tree for analyses from other extensions.

The Broadway compiler [12] allows library writers to specify, for the host language compiler, how uses of library abstractions can be optimized. This tool is based on abstract interpretation. However, it does not provide means for specifying new language constructs .

5.2 Future Work

Program comprehension: While new domain-specific syntax can be useful, if each member of a development group imported their own favorite set of extensions for use in the code for which they are responsible, group members may not understand each other's code. This problem is not unique to extensible languages, since libraries can be misused in a similar fashion, although at least the syntax if not the intent of library uses is clear. A solution is to use some discipline and restrict the set of libraries or language extensions to be used on a project. For extensible languages to have real-world impact, deployment issues such as these must be addressed.

Syntactic composability: We are also investigating techniques to ensure that the combination of syntactic specifications from several extensions will work together. Grammars used by Yacc-like tools are notoriously brittle and adding productions can easily move the grammar out of the class of grammars handled by the tools. One approach being investigated is the use of GLR parsers which can parse all context free grammars. Another approach would be the use of parsing expressions grammars, which are closed under composition [11]. Other systems, such as the Intentional Programming [18] system developed at Microsoft Research, avoid this problem by building a structure editor in which programmers manipulate the AST of the program directly.

Semantic composability: One requirement to ensure that the attribute grammar of the extended language is well-defined is that extensions need to honor the attribute dependencies of the non-terminals they extend. For example, the production `sqlQuery` in Figure 6 has a Java nonterminal (`Expr`) on its left-hand side and thus can be used anywhere that an expressions would be. Thus, it cannot define attributes such that a host language synthesized attribute depends on an inherited attribute on which it did not depend in the host language specification. Otherwise some host language production with an expression as a child may not define the inherited attribute required for computing the synthesized attribute. Although the standard definedness test [19] can perform this analysis on the extended language grammar a better approach would involve a modular analyses that can be performed by the feature designer that would ensure compatibility with other extensions that also pass the modular analysis.

Java Language Specification: As mentioned earlier, the specification of the static analysis in the Java 1.4 attribute grammar must be completed so that type errors are reported on the extended language program and not on the generated pure Java code. We intend to extend this specification to Java 1.5 as many features like the for-each loop and auto-boxing and unboxing can be specified as modular extensions to the Java 1.4 specification. Silver is currently available on the web at `http://www.melt.cs.umn.edu`.

5.3 Conclusions

We have shown how new syntax and new static analysis that supports the abstractions provided in a library can be specified so that they can be easily incorporated into an extensible language at the direction of the programmer. The new syntax and analysis address drawbacks of library-based approaches to specification of new abstractions by providing a more natural syntactic representation of the abstractions and, more importantly, providing analysis to statically check for their correct use. A key characteristic of the approach presented here is that multiple language extensions can be composed, and used by the programmer, in a manner similar to how multiple libraries can be used in the same program. This differs from the monolithic approach to language extension taken in SQLJ, Cω, and in the LINQ project.

Libraries have proved to be a very successful means for specifying, packaging, and distributing new abstractions that support the needs of different user (programmer) communities. Interested parties in small domains can design, implement, and distribute abstractions that support their work. The compositional and user-driven nature of libraries has been a key to their success, and thus we support both of these aspects in the framework for language extension presented in this paper.

Acknowledgements:
We thank Phil Russel for this assistance of the specification of the SQL extension. We thank Lijesh Krishnan for his help in the development of the Java 1.4 Silver attribute grammar.

6. REFERENCES

[1] J. Baker and W. Hsieh. Maya: Multiple-dispatch syntax extension in java. In *Proc. of ACM PLDI Conf.*, pages 270–281. ACM, 2002.

[2] D. Batory, D. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53. IEEE, 2–5 1998.

[3] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in cω. In *ECOOP 2005, Proceedings of 19th European Conference on Object-Oriented Programming*, pages 287–311, 2005.

[4] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. ACM Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 365–383, New York, NY, USA, 2004. ACM Press.

[5] A. S. Christensen, A. M. ller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. Static Analysis Symposium*, 2003.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending java for high-level web service construction. *ACM Trans. Prog. Lang. Syst.*, 25(6):814–875, 2003.

[7] A. Eisenberg and J. Melton. SQLJ part 0, now known as SQL/OLB (object-language bindings). *SIGMOD Rec.*, 27(4):94–100, 1998.

[8] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Proc. of ECOOP '04 Conf.*, pages 144–169, 2004.

[9] C. Gould, Z. Su, and P. Devanbu. Jdbc checker: A static analysis tool for sql/jdbc applications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 697–698, Washington, DC, USA, 2004. IEEE Computer Society.

[10] J. Gray. The next database revolution. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 1–4, New York, NY, USA, 2004. ACM Press.

[11] R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[12] S. Guyer and C. Lin. Broadway: A software architecture for scientific computing. In R. F. Boisvert and P. T. P. Tang, editors, *The Architecture of Scientific Software*, pages 175–192. Kluwer Academic Press, 2000.

[13] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971) pp. 95–96.

[14] J. Liu and A. C. Myers. Jmatch: Iterable abstract pattern matching for java. In *Proc. International Symposium on Practical Aspects of Declarative Languages*, January 2003.

[15] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM Press.

[16] N. Nystrom, M. R. Clarkson, and A. C. Myer. Polyglot: An extensible compiler framework for java. In *Proc. 12th International Conf. on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152. Springer-Verlag, 2003.

[17] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *Proc. of ACM POPL Conf.*, pages 146–159, 1997.

[18] C. Simonyi. The future is intentional. *IEEE Computer*, 32(5):56–57, May 1999.

[19] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.

[20] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[21] H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *ACM PLDI Conf.*, pages 131–145, 1990.

A Static Analysis for the Strong Exception-Safety Guarantee

Gustav Munkby and Sibylle Schupp
Dept. of Computing Science,
Chalmers University of Technology, Gothenburg
E-mail: {munkby,schupp}@cs.chalmers.se

ABSTRACT

Exception handling mechanisms provide a structured way to deal with exceptional circumstances, making it easier to read and reason about programs. Exception handling, however, cannot avoid the problem that the transfer of control might leave the program in an inconsistent state—resources might leak, invariants might be violated, the program state might be changed. Since client code often needs to know how a program behaves in the presence of exceptions, the *exception-safety classification* distinguishes three different classes of safety guarantees; this classification is used, for example, during the review process in the Boost organization for standardized libraries in C++. Classifying the safety level of a procedure requires understanding program invariants and tracking program state at any given point in the code, which is error-prone when done by hand. Yet, no tool support is available to date. In this paper we present the first automated analysis for exception guarantees. The analysis addresses two of the three safety guarantees, the *strong* and the *no-throw* guarantee. The analysis is implemented in the BangSafe tool set, which interfaces the Elsa parser for C++ and targets C++ programs. BangSafe itself is implemented in Ruby.

1. INTRODUCTION

In libraries and other modern software systems it is common to employ exception-handling mechanisms to deal with exceptional situations. Instead of intertwining normal code and error-handling code, guarding a possibly great number of single expressions with error checks, and introducing multiple exit points of a procedure, exception handling allows separating all error-handling code from the normal code and encapsulating it in exception handlers. This separation makes it easier not only to read the normal code, but also to reason about the error-handling code and to analyze it [6, 12, 15].

Proper use of exception handling consists of two separate responsibilities: error detection and error handling. In a stand-alone application, the code that detects an exceptional situation often also directly handles it in the way that is appropriate for the particular application. In a library designed for use by different clients in different contexts, however, the library designer cannot decide on one particular handling but must delegate this decision to the client. For any failing procedure that is visible outside the library, clients should be able to handle an exceptional situation on their own terms. For that, however, often more information is needed than just which kinds of exceptions might be active. If clients want to retry the failing operation, for example, or to ignore the exception, they need to know whether they can safely do so. If the exception has left the program in an inconsistent state—for example, with leaking resources, invalidated data type invariants, or a modified program state—it will be incorrect just to continue.

The so-called *exception-safety classification* supplies the required information by classifying procedures according to three different safety levels. The classification was introduced during the standardization of the Standard Template Library (STL) in C++ as part of the contract between library components and users. Although the classification itself is language-independent, it is applied most systematically in C++, for example, during the review process in the Boost organization for library standardization. Classifying the *exception-safety guarantee* of a procedure requires detecting the “worst” possible control flow of the procedure, i.e., the biggest possible change or inconsistency that could occur when an exception leaves the procedure. Since the control flow usually contains a great number of paths that are not directly visible at source-code level, it is in practice difficult to correctly classify a procedure. Passing a parameter, leaving a scope, or calling a sub-routine are all but examples where the compiler creates or includes code that complicates the control flow and must be traced even if it ultimately does not raise an exception. Thus, even if one is familiar with the semantics of the source language, it is not trivial to follow the flow of control by hand. Yet, no tool is available to automate the classification.

As an essential step towards the desired tool support, we have designed and implemented the first automated analysis for exception-safety guarantees. The analysis addresses the *strong* and the *no-throw* guarantee, the two strongest guarantees. The core idea of the algorithm is to partition the set of all possible control-flow graphs into five equivalence classes and to define a semiring structure from algebra on this set of equivalence classes. An appropriate initial annotation of the control-flow graph provided, the two semiring operators allow deriving the equivalence class of an entire control-flow graph from the equivalence classes of its subgraphs. The final exception-safety classification, thus, can be gained by combining the results of subclassifications, which characterizes the analysis as compositional. Furthermore, the analysis is an interprocedural and efficient 2-pass procedure over the annotated abstraction of the control-flow graph.

A prototype of the analysis is implemented in the BangSafe tool set, which interfaces the Elsa parser for C++ and handles a reasonable and relevant subset of C++ programs. BangSafe itself is implemented in Ruby. Compared to the ultimate goal of a fully-fledged tool of industrial strength, the prototype lacks support for a number of features of the C++ language. Yet, before such support is added, one should be confident that the analysis is sufficiently precise and its algorithmic logic correct. The main purpose of the prototype, therefore, is that it allows us to conduct the necessary experiments. Once the analysis is experimentally validated, we believe it to be extensible to handle all features used in C++ libraries, in particular since the underlying algorithm is both compositional and efficient.

The paper is organized as follows. In Sect. 2, we discuss the terminology and motivation of the exception-safety classification. The analysis is first outlined in Sect. 3, which describes the overall architecture, and then discussed in detail in Sect. 4, which presents the equivalence classes and the semiring formalization of the abstract representation, and Sect. 5, which contains the main algorithm. Sect. 6 puts together the various parts of the analysis and Sect. 7 provides examples, including an example of a false positive. A summary of related work (Sect. 8), an evaluation of the analysis combined with an outline of future work (Sect. 9), and a final summary (Sect. 10) conclude the paper.

2. EXCEPTION-SAFETY GUARANTEES

The exception-safety classification classifies procedures into one of three different safety levels, namely the basic, the strong, and the no-throw guarantee [1]. The basic guarantee is the weakest, and states that the procedure does not invalidate any of its invariants if an exception is thrown. It ensures that no resources leak and that all data-type invariants are valid. The strong guarantee, next, additionally specifies that if an exception is thrown from an operation, then the program state shall be the same as before the operation started. The strong guarantee ensures that any detectable changes, even operations such as creating or moving objects, are undone before the exception leaves the context. Finally, the no-throw guarantee means that the procedure may not throw any exceptions. Fig. 1 summarizes the three guarantees as they usually are spelled out.

basic: the invariants of the program are preserved.

strong: in addition to the basic guarantee, the operation provides rollback semantics in the event of an exception.

no-throw: the operation does not throw an exception.

Figure 1: Exception-safety guarantees

Classifying exception safety is no new idea. Already in the late 1970s, Cristian introduced the notion of procedures that are *weakly* and *strongly tolerant* towards exceptions [4]. In difference to the exception-safety guarantees, Cristian's tolerance levels are specified on an exception-by-exception basis. If a procedure fulfills the requirements of the strong exception-safety guarantee, Cristian's classification specifies it as weakly tolerant towards all exceptions occurring during its execution. A procedure that is strongly tolerant towards an exception can actually fix the problem that the exception signals. The corresponding exception-safety guarantee is the no-throw guarantee.

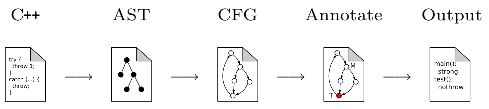
Exception-safety classification in C++ was introduced during the implementation of the C++ standard library [2], as part of the contractual specification between a procedure and its clients. Nowadays, the exception-safety classification is established and widely used. It is a standard topic in C++ courses [17], supported by a number of idioms [18], and used in the standardization process [5]. In our analysis, we therefore target C++. Libraries in other languages can, in principle, benefit from the exception-safety classification as well, but interaction with a virtual machine in languages such as Java complicates control flow considerably.

From the standpoint of automation, the strong and no-throw guarantees are similar to each other, while the basic guarantee requires an entirely different approach. For the strong and no-throw guarantees it suffices to detect the existence of state modifications and active exceptions, while the basic guarantee also requires an understanding of the invariants of the program and the ability to check them. Since the stronger guarantees include the requirements of the basic guarantee, it is somewhat surprising that the stronger guarantees are simpler to analyze. If the strong guarantee holds, however, no program state changes are allowed, which prevents breaking any invariants. We can therefore cover the basic guarantee requirements through the state property that governs the strong guarantee.

The analysis presented in this paper concerns the strong and the no-throw, but not the basic guarantee. It assumes that the full source code is available, so that all throw statements can be syntactically detected. It furthermore makes the simplifying assumption that all state modifications happen by way of object-level assignments, i.e., all side effects have been modeled correspondingly. The restriction to assignments allows for a syntactic approach, but incurs a loss of precision; we further discuss precision in Sect. 5.3.

3. ARCHITECTURE

Our exception-safety analysis is realized as a series of transformations. Fig. 2 gives an overview of the stages from the initial C++ input to the output containing the exception-safety classification. The first stage parses the input into an abstract syntax tree. In the next stage, a control-flow graph is constructed using the information from the abstract syntax tree. The third, and most important step, is the appli-



```
try {
  throw 1;
}
catch (...) {
  throw;
}
```

```
main():
  strong
test():
  nothrow
```

Figure 2: Transformations of the analysis

⟨*function*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩ '(' ⟨*arg*⟩* ')' ⟨*compound*⟩

⟨*compound*⟩ ::= '{' ⟨*stmt*⟩* '}'

⟨*stmt*⟩ ::= ⟨*compound*⟩ | ⟨*loop*⟩ | ⟨*branch*⟩ | ⟨*try*⟩ | ⟨*expr*⟩ ';'

⟨*loop*⟩ ::= 'for' '(' ⟨*expr*⟩ ';' ⟨*expr*⟩ ';' ⟨*expr*⟩ ')' ⟨*compound*⟩

⟨*branch*⟩ ::= 'if' '(' ⟨*expr*⟩ ')' ⟨*compound*⟩ 'else' ⟨*compound*⟩

⟨*try*⟩ ::= 'try' ⟨*compound*⟩ ⟨*handler*⟩+

⟨*handler*⟩ ::= 'catch' '(' ⟨*arg*⟩ ')' ⟨*compound*⟩

⟨*expr*⟩ ::= ⟨*literal*⟩ | ⟨*oper*⟩ | ⟨*call*⟩ | ⟨*throw*⟩ | ⟨*assignment*⟩

⟨*call*⟩ ::= ⟨*identifier*⟩ '(' ⟨*expr*⟩* ')'

⟨*throw*⟩ ::= 'throw' ⟨*expr*⟩

⟨*assignment*⟩ ::= ⟨*expr*⟩ '=' ⟨*expr*⟩

Figure 3: Supported input grammar

cation of our analysis-algorithm to add annotations to the control-flow graph; this part will be discussed in detail in Sect. 4 and Sect. 5. The last step, finally, interprets the top-level annotation and emits the classification.

Fig. 3 shows a segment of the grammar of the supported target language. The described language is characterized in two ways:

- It centers around the features of exception handling.
- It contains a subset of executable C++, to ensure that standard parsers are applicable.

Additionally, the target language includes all features that have a simple and straightforward mapping from C++ to the terminals and non-terminals used in Fig. 3. In the interest of readability, we have omitted those features.

The transformations are implemented as a set of command-line applications in Ruby, where each of them allows presenting and visualizing the various stages of the analysis in different ways. The most important tools include BangGraph, which produces an annotated control-flow graph for each procedure, and BangSafe, which runs the main algorithm and produces a trace with the exception-safety classification and invalidating paths.

4. ABSTRACT REPRESENTATION

As always in static analysis, the key is to find an abstract representation that is simple enough to solve the problem efficiently, but still ensures that the results can be used to interpret the original problem. To check the strong guarantee we must examine whether the program state at procedure exit is the same as at procedure entry. Therefore, we keep track of state modifications, throw-expressions, and their ordering. By varying only these parameters, and using the grammar specified in Fig. 3, we can reduce the set of all possible control-flow graphs to five equivalence classes. The abstract representation can then be interpreted as a mapping of a control-flow graph to one of the equivalence classes. Table 1 lists the five equivalence classes, including the smallest representative graph each and a description of its characteristics.

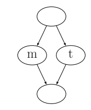
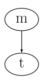
ϵ		The graph contains neither state-changes nor throw-expressions.
m		The graph contains a state-modification, but no throw-expression that exits the procedure.
t		The graph contains a throw that exits the procedure, but no state-modification occurs.
u		The graph contains a state-modification and a throw that exists the procedure, but only one of them can happen since they are in different execution branches.
s		The graph contains a state-modification that is followed by a throw that exits the procedure.

Table 1: Abstract representation

As the table shows, both checked guarantees can be represented by two different equivalence classes, one with state modification in the normal execution and one without. The equivalence classes ϵ and m map to the no-throw guarantee, and the equivalence classes t and u map to the strong guarantee. The strong guarantee is violated exactly when a graph falls in the equivalence class s, the no-throw guarantee is violated when the graph is labeled t, u, or s.

The idea of our analysis is to compute the classification of a control-flow graph by appropriately combining the annotations of its subgraphs. We therefore introduce the following two binary operators:

sequence Corresponds to the sequential execution of two graphs. This scenario occurs for most of the grammar rules (see Fig. 3).

Sequencing captures the case where an m is followed by a t. We note that a series of consecutive m can be abstracted to a single m. One might wonder why a t annotation, which represents an exiting exception, can be sequenced with another graph. If the analysis was to support automatic destructors, which is planned but not yet the case, an exiting exception is no longer necessarily the last expression.

branch Represents a choice between the execution of two alternative graphs. This operator, as expected, origi-

	ϵ	t	m	u	s
ϵ	ϵ	t	m	u	s
t	t	t	s	s	s
m	m	s	m	s	s
u	u	s	s	s	s
s	s	s	s	s	s

(a) *sequence*

	ϵ	t	m	u	s
ϵ	ϵ	t	m	u	s
t	t	t	u	u	s
m	m	u	m	u	s
u	u	u	u	u	s
s	s	s	s	s	s

(b) *branch*

Figure 4: Semiring operations

nates from the branch statement in the grammar (see Fig. 3).

Since the exception-safety guarantee is a worst-case guarantee, our analysis is interested in possible violations of the guarantee. In most cases we can therefore discard one of the branches and just keep the worse. There is one exception, though, namely when one branch leads to m and the other to t. Since both branches are equally 'bad', we must introduce a new symbol u to denote this type of structure.

Fig. 4 defines formally the binary operations *sequence* and *branch* on the set of equivalence classes $S = \{\epsilon, m, t, u, s\}$, listed in Table 1. The definition follows the intuitive understanding of the two operators; for example, both *branch* and *sequence* are intuitively expected to map an ϵ to itself. The *sequence* operation for u is defined as:

$$\forall x \in S : \text{sequence}(u, x) \equiv \text{branch}(\text{sequence}(m, x), \text{sequence}(t, x)),$$

which is natural, since we want the worse alternative of two branches.

Given these definitions, we can define a semiring for our abstract representation, $(S, \mathit{branch}, \mathit{sequence})$ [8]. It is easy to see that the *branch* operation is idempotent, thus allows defining a strict partial order:

$$\forall a, b \in S : a < b :\Leftrightarrow a \neq b \wedge \text{branch}(a, b) = b, \qquad (1)$$

which results in the following finite partial order of the classes:

$$\epsilon < m, t < u < s.$$

Intuitively speaking, these inequalities mean that larger elements constitute bigger threats to the invalidation of the strong guarantee. As Fig. 4 shows, the *sequence* operator respects this order, i.e.,

$$a \leq \text{sequence}(a, b) \ \wedge \ b \leq \text{sequence}(a, b). \qquad (2)$$

We use this ordering later, in the discussion of the analysis algorithm, when reasoning about the termination property.

As the reader might have noticed, using above classification, our abstract representation contains neither any terminology to describe loops nor any loop operations. Yet, it will become clear that the operations above suffice.

5. ALGORITHM

The analysis operates on the control-flow graph of a procedure in two stages. In the first stage, all nodes are identified that correspond to non-local modifications or exceptions that exit the procedure, and annotated with m and t. This annotation is based on a syntactic analysis of the source-code expressions. For correctly annotated call-sites, called procedures must be checked before their callers, thus the algorithm in its current form cannot deal with recursion. The second stage considers these nodes as single-element graphs, classified according to the equivalence relation in Table 1, and uses them as the base to incrementally combine more and more annotations, corresponding to larger and larger subgraphs, until the whole procedure is covered. Crucial for the incremental combination of annotations is the *local update rule*, which is applied locally to each node. In the following subsections, we describe the local update rule and reason about the correctness, complexity, and precision of the algorithm. We assume that the initial abstract annotation, from the first transformation of the control-flow graph, is available.

5.1 Local update rule

For the presentation of the local update rule, the notion of a maximal reachable subgraph is helpful. Let $G = (V, A)$ be a digraph and v be a node in G. The maximal reachable subgraph $G_v = (V_v, A_v)$ is defined as the maximal induced subgraph such that V_v contains v and all nodes x for which there exists a directed path from v to x. Since every path starting at v goes through one of v's successors, we can introduce the following set notation:

$$\begin{aligned} G_v &= (V_v \subseteq V, A_v \subseteq A) \\ V_v &= \{v\} \cup \bigcup_{\forall y:\ (v,y) \in A} V_y \\ A_v &= \{(x, y) \in A : x \in V_v \wedge y \in V_v\}, \end{aligned}$$

where V_v is defined as the union of the singleton set v and the set of nodes in a maximal reachable subgraphs rooted at one of v's neighbors y. Because of this representation as a union, one can determine the exception-safety classification of a procedure incrementally, by combining the local annotation of the current node and the classification of the maximal reachable subgraphs of its adjacent nodes. We can therefore define a local update procedure that will be executed for every node. To properly handle loops, as we will show, some nodes must be visited twice.

Fig. 5 describes in pseudo code how the local update works. The branching operator is applied to the previously recorded annotations, *states[x]*, for all adjacent nodes. Sequencing this result according to Fig. 4 with the annotation for the current node, yields the annotation representing the maximal subgraph reachable from the current node. This annotation is recorded so that it can be accessed without recalculation.

We apply the local update rule in a postorder traversal. This ordering is chosen to ensure that successors already encapsulate the annotation of the subgraph reachable from these nodes. Assuming the calculated annotation holds for

```
def local_update (node, cfg, states)
    succ_state = $\epsilon$
    for successor in adjacent_nodes(node, cfg)
        succ_state = branch(succ_state, states[successor])
    end
    states[node] = sequence(annotate(node), succ_state))
end
```

Figure 5: Local update rule

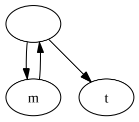


Figure 6: Control flow requiring two passes

all reachable subgraphs, the annotation for the procedure entry contains the final annotation of the procedure. When later inspecting another procedure that calls the analyzed procedure, we can attach this annotation to the call-site, without further computation. The single annotation for a whole procedure means that the algorithm scales well to the interprocedural case and that it is compositional.

5.2 Partial and total correctness

For partial correctness, we must prove that the annotation of the root node of the annotated control-flow graph contains the correct classification of the entire procedure. We divide the argumentation into two cases. First we assume there are no loops, and show that a single iteration using the update rule is sufficient. We then introduce loops and show that two traversals are sufficient.

Assuming no loops, a topological ordering can be established, which ensures that the annotation for the adjacent subgraphs will be computed before visiting the current node. Every new subgraph is a starting node sequenced with the branches to its adjacent subgraphs. If all adjacent subgraphs are correctly annotated, the correctness of the current annotation follows directly from the definition of a subgraph and the operations in Fig. 4.

By introducing loops, it will no longer be possible to define a topological ordering of the control-flow graph nodes. Fig. 6 contains the smallest possible example graph that will not be correctly classified after a single iteration of the analysis. The correct annotation for the unlabeled node is s. After one iteration, however, the analysis produces only u, because when the node labeled m is analyzed, the annotation of the subgraph reachable from the unlabeled node has not yet been computed.

For any loop-entry, we note that the reachability of other nodes is not affected by the existence of back-edges. As it is easy to see from Table 1, the only classification not solely determined by reachability is s. Since both a state modification and a thrown exception must be reachable for a procedure to be annotated with s (see Fig. 4), the annotation for the first iteration must be at least u. Applying another iteration, the loop-entry will be annotated s if and only if the loop contains an m; from there, its annotation cannot change any further. We therefore conclude that two iterations of the above traversal of the graph ensure that the procedure-entry is annotated with the correct classification of the whole procedure, with respect to Table 1.

To show total correctness, we also need to prove that the algorithm terminates. In the simplest implementation we could add a loop around the update procedure just described and loop until we reach a fixed point. Since we have already seen in Sect. 4 that our abstract representation can be represented as a finite partial order (Eq. 1) and our operations respect this ordering (Eq. 2), we know that a fixed point exists and that the loop terminates. From the discussion of partial correctness, however, we can more precisely claim that two iterations will suffice.

5.3 Complexity and precision

The complexity of the algorithm is dominated by the costs for graph traversal. Given that we can perform $O(1)$ table lookup, the complexity of the whole analysis is:

$$O(|V| + |A|),$$

where $|V|$ denotes the number of nodes and $|A|$ the number of edges.

The precision of the algorithm is lessened mainly by the following abstractions:

- Treating every state modification as irreversible. This essentially prevents the algorithm from detecting cases where the state is first invalidated, and then by a later state modification revalidated—a pattern very common in real code.
- Detecting whether a variable is only local in scope, is only done syntactically. Modifications to references and dereferencing pointer variables will therefore always lead to m annotations. This loss of precision happens early, in the initial annotation of the control-flow graph, and is propagated further by the local update rule. In the interprocedural case we lose further precision since we do not map modifications against passed arguments.
- Pessimizing on constructs with nondeterministic execution order. To ensure that all possible orders of execution are included without making the graph too big, procedure call arguments and other indeterministic ordering constructs are converted to loops. Unfortunately, this introduces infeasible execution paths.
- Merging several branches and always taking the worst. This will ensure that no possible problems are missed, but will in some cases result in a pessimistic judgment of a procedure. This is not a property of the algorithm, but a property of exception-safety classification in general.
- Assuming that all syntactically possible branches are executed and that all loops terminate. This means that infeasible paths are considered feasible.

6. PUTTING IT TOGETHER

We can now return to Fig. 2 in Sect. 3 and explain the four major transformations in more detail.

The first step is to parse C++, which is performed by the third-party program Elsa, developed by McPeak et al. [9, 10]. Elsa constructs an abstract syntax tree, annotated with type-checking information. It claims to support most of standard C++, but we have not verified that claim. Elsa emits the abstract syntax tree as an Xml-hierarchy. From there, the BangSafe toolset picks up the program representation and maps it to its own internal representation in Ruby.

The second step lowers the abstract syntax tree to an expression-level control-flow graph with support for interprocedural exception-flow. The control-flow graph is at expression level because the C++ language constructs for exceptions and state modifications are all expressions. To be able to construct the interprocedural exception-flow correctly we must ensure that callees are always processed before callers. Recursion would make this impossible, thus is not currently supported.

The actual analysis happens during the third transformation where the control-flow graph is annotated according to the exception-safety classification. As we explained in Sect. 5, the main algorithm of the analysis, the local update rule, is applied incrementally to all subgraphs of the control-flow graph of the procedure under consideration. As we also explained there, the local update rule assumes that the initial graph has been annotated with labels indicating state modifications and exiting exceptions. By the end of the third transformation, the root node of the control-flow graph contains the final classification of the procedure: it violates the strong guarantee exactly when the root node is labeled with s, and the no-throw guarantee when it is labeled t, u, or s.

The last step organizes the command-line output of the analysis. If the previous step has determined that the strong guarantee is invalidated, the tool emits not just the diagnostics but also the paths possibly leading to invalidation. This is done in the simplest of ways by just enumerating all possible paths through a procedure; to ensure termination, loops are contracted in a preprocessing step. Because of the enumeration, the complexity of the last step is exponential in the number of sequential branches if the strong guarantee has been violated.

7. EXAMPLES

In this section, we walk the reader through three different examples, to illustrate how the different transformations of our algorithm work. The first and the second example demonstrate correctly detected guarantees, and the last example demonstrates a false positive. All examples come from Stroustrup [17], but are slightly simplified. Most notably, we replaced class-based allocation by two functions. We also replaced the templates in Stroustrup's examples by instantiated templates, since our tool does not support uninstantiated templates.

All three examples use implementations of a constructor for a *vector* class in the style of the C++ standard library, which is implemented using a dynamically allocated buffer and three pointers. The constructor we are looking at allocates space for n elements and constructs as many copies of an initial value in the allocated space.

```
1 template<class T>
2 vector<T>::vector(size_type n, const T & val) {
3   v = cpp_malloc<T>(n);
4   space = last = v +  n;
5   for (T* p=v; p != last; ++p) {
6     new (p) T(val);
7   }
8 }
```

Figure 7: Source code for naive approach

The problems discussed in Stroustrup's examples come from the fact that copy construction of objects can fail. To enforce failure, for demonstration purposes, we devise a *bomb* class, which is specifically designed to throw exceptions when being copy-constructed.

7.1 The naive approach

The first example is listed in Fig. 7. Stroustrup calls it the naive approach. There are two sources of exceptions within the procedure:

- *cpp_malloc* throws if no memory is available (line 3).
- Placement *new* uses the copy constructor of the element type T to copy the initial value *val* (line 6). This copy constructor might throw.

By careful manual analysis, it is possible to figure out that if the latter throws, one has already allocated memory, which would be necessary to free. Yet, memory leakage is not the only problem. One might have actually successfully constructed a few copies before a copy construction fails. A correct program also needs to destruct these objects.

Our analysis will produce a trace describing that the strong guarantee is invalidated because it is possible to perform a *cpp_malloc* that is followed by at least one copy construction before throwing an exiting exception.

The first annotation pass generates the control-flow graph in Fig. 8, which has abstracted from the original control-flow graph all irrelevant nodes and kept only the ones with information about exception raises and state modifications; for technical reasons BangGraph introduces an exit node for each original node in the control-flow graph and keeps all annotations in these exit nodes. As we can see from the graph, at this point all inner nodes are labeled with m or t (see Table 1) and the root node, i.e., the procedure entry, with e (representing the ϵ in Table 1). After the next step, where the local-update rule is applied, some nodes will change to u or s, and the root node will contain the final classification of the procedure.

A closer look at the BangGraph output in Fig. 8 reveals that the analysis is overly conservative: we have five different m annotations, but only the one for the *cpp_malloc* invocation (line 3) and the one for the copy construction in

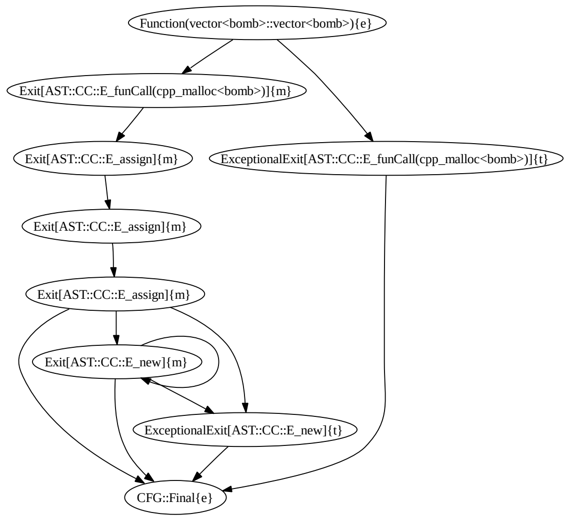

Function(vector<bomb>::vector<bomb>){e}
Exit[AST::CC::E_funCall(cpp_malloc<bomb>)]{m}
Exit[AST::CC::E_assign]{m}
ExceptionalExit[AST::CC::E_funCall(cpp_malloc<bomb>)]{t}
Exit[AST::CC::E_assign]{m}
Exit[AST::CC::E_assign]{m}
Exit[AST::CC::E_new]{m}
ExceptionalExit[AST::CC::E_new]{t}
CFG::Final{e}

Figure 8: BangGraph **for naive approach**

the new-expression (line 6) are needed. The other assignments are updates to instance variables of the constructor, which ideally should not produce m annotations. However, support for the special semantics of constructors has not yet been implemented.

If we ignore the back-edge in Fig. 8 and consider all successful copy constructions as a single operation, we can identify five distinct execution paths through the procedure:

- Failing early with *cpp_malloc*.
- Succeeding by skipping the loop, avoiding both normal and exceptional exits of the new-expression.
- Succeeding by completing the copy construction of all entries, skipping the exceptional exit of the new-expression.
- Failing at the first copy construction, going directly to the exceptional exit of the new-expression.
- Failing at a subsequent copy construction, entering first the normal exit of the new-expression, continuing to the exceptional exit.

Running BangSafe on the example, produces the trace in Fig. 9. This trace has not been pruned, to show the volume of information within the logfile. The interesting portion of the output starts at the line containing the constructor *vector<bomb>::vector<bomb>*, which shows that the procedure is correctly annotated with s and lists an invalidating path. This path describes the worst case where both *cpp_malloc* and one successful copy construction have been applied before an exception exits the constructor. The copy construction is, as explained earlier, embedded in the new-expression.

```
> bangsafe vector-naive.dump

Annotation for 'bad_alloc' is e
Annotation for 'operator new' is u
Annotation for 'cpp_malloc<>' is u
Annotation for 'vector<T>::vector<>' is u
Annotation for 'bomb' is u
Annotation for 'cpp_malloc<bomb>' is u
Annotation for 'vector<bomb>::vector<bomb>' is s
  Detected invalidating path:
  > m:4:AST::CC::E_funCall(cpp_malloc<bomb>)
  > m:4:AST::CC::E_assign
  > m:5:AST::CC::E_assign
  > m:5:AST::CC::E_assign
  > m:7:AST::CC::E_new
  > t:7:AST::CC::E_new(AST::CC::E_throw)
Annotation for 'bomb' is e
Annotation for 'bad_alloc' is e
Annotation for 'operator new' is e
Annotation for '~vector' is e
Annotation for 'operator=' is e
Annotation for 'operator delete' is e
Annotation for 'cpp_free<>' is e
Annotation for 'vector' is e
Annotation for 'cpp_malloc<int>' is u
Annotation for 'operator=' is e
Annotation for '~bomb' is e
Annotation for 'operator=' is e
Annotation for '~vector' is e
Annotation for 'operator=' is e
Annotation for 'vector<int>::vector<int>' is u
Annotation for 'vector' is e
Annotation for '~bad_alloc' is e
ERROR: instance of IO needed
```

Figure 9: BangSafe **for naive approach**

7.2 The naive approach, take two

Now suppose that we instantiate the same constructor with *int* instead of *bomb*. Then the example provides the strong guarantee because the copy construction of *int* cannot throw. The analysis can confirm that. We do not list the trace separately, but the result is part of Fig. 9, where the entry for *vector<int>::vector<int>* is u, which indicates that no state modification took place before the exception was thrown; thus the strong guarantee holds. This result is correct, since only allocation can fail, but all allocation takes place before any state modification.

The output of BangGraph for the instantiation with *int* looks identical to Fig. 8, except that the node labeled as an exceptional exit from the new-expression together with its attached edges no longer exist.

7.3 The revised example

As a final example, we discuss a case where our analysis produces a false positive. In this example, Stroustrup solves the exception-safety violation from the naive approach in a straightforward way, by using the standard routine *uninitialized_fill* to construct the objects and wrapping the whole block of code in a try-catch block to ensure that any allocated memory is freed on failure.

```
1  template <class T>
2  vector<T>::vector(size_type n, const T& val)
3  {
4          v = cpp_malloc<T>(n);
5          try {
6                  uninitialized_fill(v, v+n, val);
7                  space = last = v+n;
8          }
9          catch (...) {
10                 cpp_free(v);
11                 throw;
12         }
13 }
```

Figure 10: Revised source code

```
> bangsafe vector-revised.dump
...
Annotation for 'vector<bomb>::vector<bomb>' is s
  Detected invalidating path:
  > m:4:E_funCall(cpp_malloc<bomb>)
  > m:4:E_assign
  > m:6:E_funCall(uninitialized_fill<bomb*,bomb>)
  > m:10:E_funCall(cpp_free<bomb>)
  > t:11:E_throw()
...
```

Figure 11: BangSafe for revised example

As the BangSafe trace in Fig. 11 shows, the analysis has identified a path through the procedure that invalidates the strong guarantee. The path represents successful allocation, *cpp_malloc*, followed by failing *uninitialized_fill*. Since the result of the allocation is assigned to a variable (line 4) and *uninitialized_fill* causes an exception that exits the procedure, the annotated control-flow graph contains a sequence of m followed by t for this path. Therefore, the analysis produces an s annotation for the whole procedure. In reality this path constitutes no problem—it represents a false positive: the state is restored because the memory allocated is freed (line 9), *uninitialized_fill* actually provides the strong guarantee, and the assignment is to an instance variable in the constructor, thus could never have modified the program state.

The latter source of a false positive, conservative treatment of assignments, can be dealt with relatively easily, by adding special support for constructor semantics. A more fundamental problem is that the algorithm does not realize that the state changed by *cpp_malloc* is reverted by *cpp_free*. Using our syntactic abstraction, this restoration cannot be detected. Not being able to detect reversals is also the reason why *uninitialized_fill* is classified incorrectly with s instead of u.

8. RELATED WORK

Despite of the practical relevance of exception safety, only little theoretical work or practical support exist. The only automation available until recently was based on a test suite for the C++ standard library, which Abrahams developed as part of the implementation of STLport [1]. The testing technique has since been generalized and incorporated into the Boost *Test* library [14]. Testing requires tailoring the test-suite to the library or program being tested. It is on the one hand more powerful than our approach insofar it enables checking even the basic guarantee. On the other hand, it must be adapted for every target.

Alexandrescu et al. describe a manual analysis for exception-safety classification [3]. The abstraction in their approach is similar to ours, but based on a classification of procedures, not of control-flow graphs: they distinguish between *pure* procedures, which neither modify program state nor cause any side effects, and *impure* procedures, which are not pure. We can map our five equivalence classes in a straightforward manner to their terminology: s, u, and m correspond to an impure procedure, t and ϵ to a pure procedure. While we capture the final level of exception safety directly in one of the equivalence classes, Alexandrescu et al. need to further distinguish which kind of guarantee an impure procedure provides; pure procedures give the strong guarantee by definition. Like our analysis, theirs can check the strong and no-throw guarantees, but not the basic guarantee. The most important difference, yet, is that their algorithm is designed for manual application, thus pays less attention to the details necessary for automation.

In our analysis we rely on interprocedural exception-flow. This part of our analysis has been inspired by similar analyses by Robillard et al. for Java [13], and Schaefer et al. for Ada [16].

9. EVALUATION AND FUTURE WORK

The current prototype implementation can deal with programs in a subset of C++ that already allows us to test the design of the analysis-algorithm and to put on an experimentally validated basis the ideas that underlie our static analysis of the strong and no-throw exception-safety guarantee. The examples presented in Sect. 7 mark a good start for this kind of test, but we need to identify more test cases for the logic of the algorithm and run the prototype on them.

Ultimately, we want to be able to provide a fully-fledged tool for exception-safety classification. To that end, we have to evaluate the abstractions of the analysis and its precision by performing real-world benchmarking. Testing with real-world libraries presupposes that the analysis can deal with more features of the C++ language than it currently does. Most of these features we expect to impose no difficulties to the algorithmic logic of the analysis, although they might be technically non-trivial to implement. The thesis of the first author goes into detail about several possible extensions [11]. Of all possible extensions, the most interesting, and also the most complicated one, is the support for uninstantiated templates, which will require modifications of the analysis.

Enabling the analysis to support uninstantiated templates is not easy, mainly because one must identify all implicit subroutine calls that affect the final exception-safety classification. In C++, many expressions can implicitly lead to procedure calls, including automatic type conversions, overloaded operators, and return statements. If a template is instantiated, we can rely on the parser to lower the abstract syntax tree to explicitly include any procedure calls as part of its type checking. For uninstantiated templates, however, only limited type checking can take place.

Given the relevance of uninstantiated templates, one might wonder how useful a prototype is that provides support for instantiated templates only. Yet, we claim that a focus on instantiated templates is a proper simplification for a first prototype, particularly since we believe that the current analysis can be embedded in the more developed one that accomplishes full template support.

One might also wonder whether the upcoming introduction of *concepts* in the next major version of C++ will make support for templates easier [7]. In short, concepts allow expressing constraints on template parameters and therefore enable the compiler, without specializing a template, to determine the set of candidate functions that could be called from a specific call-site. Concepts can thus restrict the behavior of all admissible templates, but they cannot ensure that all candidates behave identically in terms of our analysis: our analysis needs to know what types of exceptions are thrown and whether state-modifications can take place inside a candidate, but those properties are not considered during the concept check. It is conceivable, however, that one can specify these properties of interest in terms of concepts and then explicitly state the mapping from a procedure to the provided exception-safety guarantees. Our analysis could then be adopted to produce such mappings automatically.

Benchmarking with real-world libraries is also necessary to assess the abstractions of the analysis and the resulting precision of the final exception-safety classification. Almost all static analyses are imprecise insofar they might produce false positives. Very practically, thus, their usefulness depends on the actual number of false positives. We expect that the most prominent reason for loss of precision is due to our assumption to detect the invalidation of the strong exception-safety guarantee by identifying explicit state modification. Given that the full source code is available, state modification is a necessary condition for breaking the strong guarantee. Yet, it is no sufficient condition—not every state modification violates the strong guarantee. To reduce the number of false positives, one could exploit the semantics of C++ to determine that some state modifications result in effects that are just local and will not escape the procedure boundaries. Currently, we can identify an assignment as local only if the assignment is directly to a local non-reference variable, but we must be conservative if the assignment is to a dereferenced pointer or reference. To improve the precision of the analysis, we want to investigate the benefits of adding support for a points-to analysis. We think that the points-to analysis would be most effective in combination with an improved interprocedural analysis. The analysis could then bind modifications of arguments to local variables in the caller instead of treating them as global state modifications. Again, experiments are necessary to decide whether those efforts are worthwhile.

Another practical question is the scalability of the analysis. The complete examples, including template instantiations, the vector class, and needed supporting procedures, consist of less than 100 lines of C++ source code each. By design, however, the analysis is prepared to handle larger programs. Its costs are linear in the size of the control-flow graph, it is an interprocedural analysis and, due to the semiring formalization, it is compositional: since all required information is contained within the equivalence-class annotations and the final result is kept in the root node of the control-flow graph of a program, an already analyzed program that becomes a subprogram of a larger program can be represented just by its root node, annotated with the classification; the semiring operations on the larger control-flow graph and the *local update rule* then propagate its classification correctly. In its current incarnation, the analysis is a whole-program source-code analysis, but refining it to a fragment analysis seems possible.

Finally, it is worthwhile pointing out that our analysis, like related analyses, assumes that side effects have been made explicit, for example, by a previous analysis.

10. SUMMARY

An important part of the contract between a library component and its client is the exception-safety guarantee that the library component assures; the actual safety level defines the options the client has when handling an exception. In particular in C++, exception-safety guarantees are part of the review process of libraries and, conversely, many library components are designed with the strong exception-safety guarantee in mind. Until now, however, exception-safety guarantees have to be determined by hand. We have designed and implemented a static analysis of the strong and the no-throw exception-safety guarantees, which automatically determines for a given library procedure whether or not it gives the strong or no-throw exception-safety guarantee. The analysis also allows a library developer to test whether the exceptional behavior of the library procedures is as intended.

It is not always necessary to demand the strong guarantee: depending on the application and its fault tolerance level, but also on the design of a component, its degree of mutability and its number of valid states, it might suffice to ask for the basic guarantee. Even though the analysis is designed to support only the two stronger guarantees, we believe that it is helpful even for the basic guarantee. The analysis explicitly lists the cases where the strong guarantee is invalidated, thereby highlighting the locations where the basic guarantee could be invalidated.

The core idea of the underlying algorithm, to recapitulate, is to introduce five equivalence classes of control-flow graphs and define a semiring structure over them. Based on an initial (syntactic) annotation of the control-flow graph with state modifications and exiting exceptions, the semiring operators propagate this annotation in an efficient and compositional manner.

Acknowledgements

This work was in part supported by the project CEDES, which is funded within the Intelligent Vehicle Safety Systems (IVSS) program, a joint research program by the Swedish industry and government. We thank the reviewers of LCSD for their insightful comments and suggestions.

11. REFERENCES

[1] D. Abrahams. Exception-safety in generic components. In M. Jazayeri, R. Loos, and D. R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 69–79. Springer, 1998.

[2] D. Abrahams and G. Colvin. Making the C++ standard library exception safe. Technical Report N1086 = 97-0048R1, C++ Standards Committee, 1997.

[3] A. Alexandrescu and D. B. Held. Smart pointers reloaded (ii): Exception safety analysis. *C/C++ Users Journal*, 21(12):40–44, December 2003.

[4] F. Cristian. A recovery mechanism for modular software. In *ICSE '79: Proceedings of the 4th International Conference on Software Engineering*, pages 42–50.A, Piscataway, NJ, USA, 1979. IEEE Press.

[5] B. Dawes. Boost library requirements and guidelines. `http://www.boost.org/more/lib_guide.htm`, November 2003.

[6] J. B. Goodenough. Structured exception handling. In *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 204–224, New York, NY, USA, 1975. ACM Press.

[7] D. Gregor and B. Stroustrup. Concepts. Technical Report N2081=06-0151, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2006.

[8] U. Hebisch and H. J. Weinert. *Semirings: Algebraic Theory & Applications in Computer Science*. World Scientific, January 1999.

[9] S. McPeak. Elkhound and Elsa. `http://www.cs.berkeley.edu/~smcpeak/elkhound/`, August 2005.

[10] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In E. Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2004.

[11] G. Munkby. Design and implementation of an algorithm for the strong exception-safety guarantee in C++. Master's thesis, Chalmers University of Technology, May 2006.

[12] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, 1975.

[13] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering Methodology*, 12(2):191–221, 2003.

[14] G. Rozental. Boost Test Library homepage. `http://www.boost.org/libs/test`, May 2005.

[15] B. G. Ryder and M. L. Soffa. Influences on the design of exception handling ACM SIGSOFT project on the impact of software engineering research on programming language design. *SIGSOFT Software Engineering Notes*, 28(4):29–35, 2003.

[16] C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in Ada. *Software—Practice & Experience*, 23(10):1157–1174, 1993.

[17] B. Stroustrup. *The C++ Programming Language*, chapter Appendix E. Addison-Wesley Professional, special edition, February 2000.

[18] H. Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems and Solutions*. AW C++ in Depth Series. Addison Wesley, August 2004.

Extending Type Systems in a Library

Type-safe XML-processing in C++

Yuriy Solodkyy
Texas A&M University
yuriys@cs.tamu.edu

Jaakko Järvi
Texas A&M University
jarvi@cs.tamu.edu

Esam Mlaih
Texas A&M University
mlaih@tamu.edu

Abstract

Type systems built directly into the compiler or interpreter of a programming language cannot be easily extended to keep track of run-time invariants of new abstractions. Yet, programming with domain specific abstractions could benefit from additional static checking. This paper presents library techniques for extending the type system of C++ to support domain specific abstractions. The main contribution is a programmable "subtype" relation. As a demonstration of the techniques, we implement a type system for defining type qualifiers in C++ as well as type system for the XML processing language, capable of, e.g., guaranteeing that a program only produces valid XML documents according to a given XML Schema.

Keywords type systems, XML, type qualifiers, C++, template metaprogramming, software libraries

1. Introduction

It is in general not possible to decide statically the set of all safe programs. Type systems of practical programming languages can only approximate this set, rejecting some safe programs, and accepting some unsafe ones. For example, the **if** statement below is rejected by a C++ compiler, even though the type-incorrect execution path would never be taken, and the initialization of j is accepted, even though i will always lead to a "division by zero" error:

```
int i = 1;
if (i == 1) i = 0; else i = "error";
int j = 1/i;
```

Replace i == 1 in the condition with an arbitrarily complex computation, and it is evident why practical type systems have this behavior: it is too inefficient, or impossible, to statically keep track of computations with certain abstractions to guarantee safety. There are, however, many abstractions for which ensuring safety with a type system would be neither inefficient nor impossible. Consider the following piece of code, accepted by a C++ compiler:

```
double U; // voltage
double I; // current
  ...
double P = U/I; // power, oops!
```

LCSD '06 October 22nd, Portland, Oregon.
ACM [to be supplied].

The variables obviously correspond to physical quantities, but the *units* of those are outside of the type system, and the easy error goes undetected.

There are numerous domain-specific abstractions for which type systems could in principle guarantee important run-time invariants — but the abstractions are not modeled as part of the type system of the programming language used. Of course, many type systems for domain-specific abstractions have been developed. For example, type systems for rejecting incorrect computations with physical quantities such as the one in our example above, can be found [22]. As other examples, there are type systems for tracking memory usage errors with a *non-null* annotation [9, 12, 13], automatically detecting format-string security vulnerabilities [26], keeping track of positive and negative values [7], ensuring that user pointers are never dereferenced in kernel space [21], preventing data races and deadlocks [3], and so forth. All of the above type systems can be based on annotating types with different kinds of *type qualifiers*, and tracking their use in expressions.

We note that none of the above type systems has found their way to mainstream languages. Whether programmers can benefit from such type systems becomes a question of whether the abstractions involved are common enough and safety properties important enough to warrant complicating the specification of a general-purpose programming language, and its compilers and interpreters. It is clear that programming languages cannot be extended to support typing disciplines for every possible domain. Ideally, it would be possible to *extend* type systems to guarantee run-time invariants of new domain specific abstractions.

Work towards the above goals exists: Chin, Markstrum and Millstein [7] present a framework for creating *type refinements*, capable of extending type systems with domain specific typing rules. They share our view that language designers cannot anticipate all of the practical ways in which types may be refined in a particular type system in order to enforce a particular invariant. The proposed solution is a framework for user-defined type refinements, allowing programmers to augment a language's type system with new type annotations to ensure invariants of interest. The framework allows the generation of a type-checker based on declarative rules. Other work with similar goals include that of optional, "pluggable" type systems [4]. While clearly beneficial, the above kind of frameworks have not yet found widespread use.

In this paper, instead of a special purpose framework, we advocate a more lightweight mechanism for refining type systems with domain-specific abstractions: as software libraries. We show that most of type refinements presented, for example, in [7, 13, 23], and available through dedicated frameworks can also be implemented as a library in a general-purpose programming language, namely C++. Our approach is therefore to refine the C++ type system with domain specific abstractions via libraries. The underlying C++ type system cannot obviously be altered — by *refining* the type system, we primarily mean defining the convertibility relations between data types of particular domains, and how these user-defined data types behave with respect to the built-in types of C++. Being constrained with the limits of expressiveness of a general-purpose programming language, not everything that special purpose frameworks would allow is possible. For example, the framework described in [7] ensures the soundness of the generated type-systems, which we cannot do automatically.

Prior work in this direction exists. E.g., C++ libraries for tracking physical units are presented in [2,5]. The introduction of several recent programming techniques and foundational C++ libraries, however, enable a more disciplined approach to defining such type system refinements. In this paper, we collect these techniques together, and show how to apply them for refining the C++ type system. This paper presents work in progress, providing the following contributions:

- We identify the necessary library tools that are needed for extending the C++ type systems for domain-specific types.
- We identify necessary language features in C++ that enable the definition of an arbitrary "subtyping" relation.
- We provide a library of primitives that allows easy extension of the C++ type system with user-defined typing rules.
- We demonstrate with two extensive examples: a type system for building type-qualifiers and a type system for XML documents, which can, e.g., guarantee statically that a program only produces XML documents that are valid according to a given XML Schema.

To wet an appetite we present here a small example of what can be done with our approach:

```
alt<seq<Name,Email>, seq<Name,Tel> > old_contact;
seq<Name, alt<Email,Tel> > new_contact = old_contact;
```

The alt and seq types represent, respectively, alternation and sequencing of XML elements while Name. The Email and Tel types represent particular XML elements. Thus, old_contact and new_contact are objects that represent fragments of XML data. We discuss these types in detail in Section 4.2. Our library statically assures that the two XML types are compatible and the initialization of new_contact with old_contact is safe, and generates the necessary code to conduct such a transformation. With our library, arbitrarily complex XML schemas can be represented as C++ types. These types provide static guarantees about dynamic content of their values, can aid in parsing appropriate XML documents, and provide safe conversion operations between fragments of XML data.

2. C++ building blocks

The toolbox of a C++ programmer has grown with some notable additions during the recent years. We note the following techniques that are relevant for defining type refinements:

- The ability to express interesting typing rules obviously necessitates that one can encode computations in a library. C++ templates are a Turing-complete language [30], and thus they allow arbitrary computations on types and constants to be performed at compile time. Such *template metaprograms* [29] have been frequently used in various C++ libraries. Template metaprogramming, however, remained a relatively ad-hoc activity until the introduction of the Boost Metaprogramming Library (MPL) [1, 15]. MPL provides a solid foundation for metaprogramming, defining essentially a little programming language and a supporting library for defining *metafunctions*, functions from types to types. MPL provides the ability to define higher-order metafunctions, lambda functions etc., and a host of data structures and algorithms for storing and manipulating types. For more complex typing rules, a framework like MPL is essential; we use MPL extensively to define relations between types, e.g., to provide a user-defined "subtyping" relation. For example the following application of the is_subtype metafunction defines whether the two types we used earlier are in "subtyping" relation:

```
typedef alt<seq<Name, Email>,
            seq<Name, Tel> > old_contact;
typedef seq<Name, alt<Email, Tel> > new_contact;
is_subtype<old_contact, new_contact>::type;
```

  Following the conventions of MPL, the result of the is_subtype metafunction is not a Boolean constant, but either the type mpl::true_ or mpl::false_.
- Type systems typically define in which context the use of objects of certain types is allowed, what operators between objects of different types are allowed, and so forth. With metafunctions, it is possible to define arbitrary sets of types and relations between types. The ability to *enable* or *disable* functions based on conditions defined by arbitrary metafunctions then allows one to define the contexts where the specified sets of types are valid. This ability is offered with the enable_if templates [18, 19]. We use these templates to enable certain operations, such as assignment, only when its parameters are in a subtyping relation. For example, the following assignment operation is only defined, i.e., only matches during the overload resolution, if the right-hand side of the assignment is an *arithmetic* type:

```
class A {
  ...
  template <class T>
  typename enable_if<is_arithmetic<T>, A&>::type
  operator=(const T&);
};
```

  The first argument to enable_if is the condition, the is_arithmetic metafunction is defined in the current draft specification of the C++ Standard Library, and the second argument is the type of the entire enable_if<...>::type type expression in the case where the condition is true. Thus, in the definition above, the return type of the assignment operation is A&.
- To be able to define typing rules and conversion operators based on the types' structural properties, the representation of the types must be accessible to template metaprograms. The tuple types in the Boost Fusion Library [20] provide us with such a representation: complicated types can be represented as Fusion's tuples, and manipulated with its algorithms at compile time. Fusion works quite similarly to MPL, but where, say, an MPL vector only contains types, a Fusion tuple contains values as well. Similar to MPL metafunctions, we can define metafunctions in Fusion too, but those metafunctions also have a run-time component. As a simple demonstration of the functionality offered by the Fusion library, below we first create a tuple type, populate its elements with values, define a function object that prints out its argument, filter out all non-arithmetic types from the tuple, and then print out the values that remain:

```
typedef fusion::tuple<std::string, int, char> grading_record;
grading_record rec = fusion::make_tuple("Joe Smith", 89, 'B');
struct print {
  template <class T>
  void operator()(const T& x) const { std::cout << x; }
}
for_each(filter_if<is_arithmetic<Sequence> >(rec), print());
```

  Note that MPL metafunctions (such as is_arithmetic) can be given as inputs to Fusion algorithms, as well as normal function objects (print()). Some Fusion algorithms, for example transform, requires a *hybrid* of a metafunction class and a function object. This algorithm transforms a tuple to another tuple, potentially transforming both the types and the values of the elements. Fusion tuples are used to represent the XML types and Fusion algorithms are the foundation when defining implicit conversions between XML types.

- During the past few years, several new C++ Libraries that implement new "language constructs" have been introduced. Though not strictly essential for our approach in general, in the XML typing library we find good use for the Boost Variant [8] and the Boost Optional [6] libraries, both of which provide notable new functionality to C++. For example, in the following, Contact is a discriminated union type that can hold an object of any of the three types Email, Tel, or ICQ, and an object of type MiddleName possibly contains a string:

  ```
  typedef boost::variant<Email, Tel, ICQ> Contact;
  typedef boost::optional<std::string> MiddleName;
  ```

  Both libraries are equipped with an API for convenient manipulation of the types.

We point out that MPL, Boost Fusion, Variant, and Optional, even enable_if, have all been developed using the *generic programming* methodology, building their interfaces to a large extent against common *concepts* (in the technical sense). As a result, the above libraries are highly interoperable. For example, the list of the element types of a variant type can be viewed as an MPL sequence, and thus manipulated either with the MPL or Fusion algorithms; enable_if expects MPL metafunctions as its condition argument, and so forth. Though mere libraries, the above set extends the C++ language in a significant way.

3. XTL - an eXtensible Typing Library

In this section we demonstrate how the library techniques from the previous section enable extending the C++ type system. We present the rationale and design of library components that support this task. We refer to these components and the accompanying conventions collectively as the *eXtensible Typing Library* (XTL).

3.1 Simple example

We start with a simple example, presented in [7], that extends the integer type with qualifiers pos and neg, which can be used to statically track when a value in a variable is positive or negative. Using techniques used in C++ for decades [28], we can express a simple but incomplete solution, shown in Figure 1. Constructors, assignment and conversion operator are supposed to capture relationship of the new type pos<T> and original type T. Objects of the underlying numeric type (T) can be used to initialize objects of pos<T> and neg<T> types. As this is an unsafe operation, it is equipped with a run-time assertion. Conversions back to the underlying numeric type are safe, and provided with the user-defined conversion operators to T. How the qualified types behave with various operators is encoded by overloading those operators, as demonstrated with the overloads of the arithmetic operators.

This straightforward solution, however, is fairly limited. When using solely the types pos<T> and neg<T>, the behavior is well defined, but the interaction of these types with other types, either built-in or user-defined, or with other possible qualifiers, is not. For example, the unsafe code in Figure 2 compiles without errors. We can identify several questions, the answers to which are not clear in this simple approach: What is the relationship between the element type T and type pos<T>? The provided constructor and conversion operator make them convertible to one another, but does this conversion loose any semantic information? Can values of one type always be implicitly converted to and used in place of the other? Are these types in a subtyping relation? How about the relationship of instantiations of pos and neg with different element types? What should, e.g., be the relationship between pos<**int**> and pos<**double**>? We note that the straightforward approach is lacking in many respects.

```
template <class T>
struct pos {
  explicit pos(const T& t) : m_t(t) { assert(t > 0); }
  operator T() const { return m_t; }
  pos& operator=(const pos& p) {
    m_t = p.m_t;
    return *this;
  }
  T m_t;
};

template <class T> struct neg;

template <class T>
pos<T> operator+(const pos<T>& a, const pos<T>& b);

template <class T>
T operator+(const pos<T>& a, const neg<T>& b);

template <class T>
T operator+(const neg<T>& a, const pos<T>& b);

template <class T>
neg<T> operator+(const neg<T>& a, const neg<U>& b);
// ... other operations
```

Figure 1. Straightforward implementation of type qualifiers pos and neg. The definition of the class neg is not shown; it is analogous to the definition of pos.

```
neg<int> ni(−20);
unsigned un = ni; // oops: un = (unsigned)(int)ni
```

Figure 2. Impact of standard conversions on pos.

3.2 XTL Subtyping

The central notion in XTL is a user-definable "subtyping" relation (not based on inheritance). XTL sets the policies of how the subtype relation is extended for new user-defined types, and provides the general building blocks to make the task effortless. In particular, when a user defines a type to be a subtype of another type, the rest of the framework assures that objects of the first type can be used in contexts where objects of the second type are expected. Note that even though we use the term *subtyping*, a conversion may in some cases be involved, e.g., in the case of XML types described in Section 4.2. In practice, when an object of a subtype is used in the context where supertype is expected, a user-defined conversion is implicitly performed. As part of defining the XTL subtyping relation for data types of a domain, the programmer defines the necessary conversion operators as well.

XTL defines a generic metafunction is_subtype<S,T> that evaluates to mpl::false_ by default and should be specialized to evaluate to mpl::true_ whenever type S is a subtype of type T. A type system built using XTL is responsible for specializing the is_subtype metafunction for primitives of a particular problem domain. Once these basic relations have been established, XTL provides an elaborate set of ready to use subtyping algorithms for compound types. Among such we support subtyping of function types, array types, type sequences, discriminated unions, and types refined with type qualifiers similar to those described in [12].

The XTL subtyping relation is fully under the control of the programmer. For example, if deemed useful, the C++ type **char** can be defined to be a subtype of **int**, **int** a subtype of **double**, **double** a subtype of complex<**double**>, and so forth. In fact, such safe conversions are commonly useful, so they are available through inclusion of a dedicated XTL header file. New types can be added to the XTL's subtyping relation by partially or explicitly specializing the is_subtype template. All specializations collectively constitute the metafunction, and thus the subtyping relation. The classes and types involved do not have to be altered when extending the is_subtype metafunction.

3.3 Subtype casting

Since physical representations of values in different types may vary, our definition of subtyping relation implies existence of a unified conversion mechanism capable of transforming physical representation of a subtype into a physical representation of a supertype. Such a conversion in XTL is accomplished with the subtype_cast function template, invoked as subtype_cast<T>(val), where T represents a supertype of val's type. This function deduces the type of val and if it is a subtype of T, converts val to an object of type T. Otherwise the subtype_cast function template is disabled with the enable_if mechanism, and a call to it results in a compile-time error.

We demonstrate the use of the XTL subtyping relation and subtype_cast with subtyping of function types. Consider the following two function types and corresponding values:

```
class A {}; class B : public A {};

typedef A (B_to_A)(B);
typedef B (A_to_B)(A);
A f(B);
B g(A);
```

According to the usual subtyping rules between function types (covariant on return types, contravariant on parameter types) such a function type A_to_B would be a subtype of B_to_A in some languages—these rules, however, are not part of the C++'s typing rules, as demonstrated with the following:

```
B_to_A* pf = &f; // ok
A_to_B* pg = &g; // ok
pf = &g; // error
```

XTL allows the definition of such a relation, after which an explicit subtype_cast succeeds:

```
pf = subtype_cast<B_to_A>(&g); // OK
```

At this point XTL checks the type-safety of the conversion and generates a proxy function that performs necessary conversions of the arguments and the result when the function is invoked. It is worth noting that these conversions are also done with the help of subtype_cast—types recognized by the XTL compose. For example, instead of types A and B, the parameter and return types of the above functions could be other function types, or pos and neg, or any types recognized by the XTL, and the framework would check for the appropriate subtype relation of those types. Thus, the framework allows extension with many domain-specific types independently, resulting in the expected behavior when the types interact. Crucial for this is that the subtyping relation of all abstractions is defined by extending the same is_subtype metafunction, and that the accompanying conversions occur via the subtype_cast function.

Before we proceed to revising our example, we note that explicit casting is not a very elegant feature. In many practical cases, we can avoid its use by providing implicit conversions as either constructor on a supertype or conversion operator on a subtype, which redirect their call to appropriate subtype_cast. Such implicit conversion is not, however, possible in the following cases:

- Both types are pre-existing types (e.g. built-in or standard types) that the developer of a type system cannot alter—neither to define a constructor in the super type nor a conversion operator in the subtype.
- The constructor and/or assignment operator that implement implicit conversions in a subtype is a member template, in which case a conversion operator in the supertype is not applied. See example in Figure 5 for details.

Taking this into account, it is advisable not to rely on implicit conversions between types encoded as part of the XTL framework in generic code, but rather use subtype_cast explicitly if conversions are necessary. This will assure that the code works with all applicable types.

3.4 pos and neg example revisited

To demonstrate the use of the XTL's subtyping relation, we rewrite our naive implementation of the pos and neg class templates, extending the subtyping relation appropriately, and defining the subtype casts. The pos class template is shown in Figure 3. The definition of neg is similar.

```
template <class T>
struct pos {
  explicit pos(const T& t) : m_t(t) { assert(t > 0); }

  template <class U>
  pos(const U& u,
      typename enable_if<
        is_subtype<U, pos<T> >, void>::type* = 0) :
          m_t(subtype_cast<T>(u)) {}

  template <class U>
  typename
    enable_if<is_subtype<U, pos<T> >, pos&>::type
  operator=(const U& u) {
    m_t = subtype_cast<T>(u);
    return *this;
  }

  template <class U>
  operator U() const { return subtype_cast<U>(m_t); }

  // non−template operator= and the data member as before
};
```

Figure 3. Definition of the pos class template within XTL

We can observe a new constructor in the pos class. Though taking two arguments, the second one has a default value, and thus the constructor implements an implicit conversion. The first argument seemingly matches any type, but in reality only types that are subtypes of pos<T> will be considered. This is made possible by the second parameter that acts as a guard: the constructor is only enabled if U is defined to be a subtype of pos<T> with the is_subtype metafunction. The rather complex type expression boils down to the type **void*** when the function is enabled, thus the parameter can accept the default value 0. This is an idiomatic use of the enable_if template to place a constraint to a constructor. The body of the constructor performs a conversion between the representations using the subtype_cast function.

The assignment operation has the same guard as the converting constructor described above. The condition is now, however, expressed as part of the return type of the operator. Again, this is idiomatic use of enable_if. The effect is that an object of type U can be assigned to a variable of type pos<T> exactly when U is a subtype of pos<T>.

The non-parametrized conversion operator was replaced by a parametrized one to rule out introduction of standard conversions into the chain of conversions, which was letting the counter example in Figure 2 work. Note that the enable_if template cannot be applied to a conversion operator, because it has neither explicit return type nor arguments to which we can bind the enable_if's condition [18, 19].

The subtyping rules for pos are defined outside of the pos class, by specializing the is_subtype metafunction:

```
namespace xtl {
template <class S>
struct is_subtype<pos<S>, S> : mpl::true_ {};
template <class S, class T>
struct is_subtype<pos<S>, pos<T> > : is_subtype<S, T> {};
}
```

Here, the first specialization states that a pos type is a subtype of its element type, and two pos types are in a subtyping relation if their element types are.

The subtype_cast function has some subtle behavior with C++ overloading rules, since call to it requires explicitly specifying a template argument. To evade the subtleties, this function directly forwards to the subtype_cast_impl function, wrapping the target type of the conversion inside a target template. Hence, subtype_cast_impl requires no explicit specialization, and is the function overloaded when extending XTL with new types. According to the above subtyping rules, we overload a function subtype_cast_impl function to tell the compiler how to convert between a subtype and a supertype in this case of pos-qualified types:

```
template <class T>
T subtype_cast_impl(target<T>, const pos<T>& p) {
  return p.m_t;
}
template <class T, class S>
pos<T>
subtype_cast_impl(target<pos<T> >, const pos<S>& p) {
  return pos<T>(subtype_cast<T>(p.m_t));
}
```

Definitions of operators now also change slightly to take subtyping into account:

```
template <class T, class U>
pos<typename join<T, U>::type>
operator+(const pos<T>& a, const pos<U>& b)
{
    typedef typename join<T, U>::type join_type;
    return pos<join_type>(
                subtype_cast<join_type>(a.m_t) +
                subtype_cast<join_type>(b.m_t));
}
```

The metafunction join used here is provided by XTL and returns the "join" of two types, i.e., the least supertype of its arguments. For two types of which one is a subtype of the other, this metafunction simply returns the supertype. For types that are not in subtyping relation, join relies on appropriate specializations that have to be provided by the designer of a type system. For example, assuming the relations **float** <: **double** and T <: complex<T>, **double** and complex<**float**> are not in a subtyping relation. They can be given a join type complex<**double**> by specializing the join metafunction.

3.5 Type qualifiers

The pos and neg qualifiers presented above are a simple example of an important direction for enriching type systems: refining a type with qualifiers. Type qualifiers modify existing types to capture additional semantic properties of the values flowing through the program. Probably the best-known example of a type qualifier is the **const** qualifier of C++ that is used to track immutability of values at different program points. Other examples include type qualifiers for distinguishing between user and kernel level pointers [21], safe handling of format strings [26], and tracking of values with certain mathematical properties [7].

Instead of implementing different type qualifiers to type systems in an ad-hoc manner, several systems, based on a general theory of type qualifiers, have been described [11–13]. These systems allow an economical definition of different domain-specific qualifiers.

In this section, we review common properties of type qualifiers, and show how to implement type qualifiers as a C++ template library using the XTL framework. To give a brief example we use *taintedness* analysis [26] that uses qualifiers untainted and tainted to tag data coming from trustworthy and potentially untrustworthy sources, respectively. The requirement is that tainted data may never flow where untainted data is expected. We may want to ensure that, say, data originating from measurements, considered as trustworthy (untainted) data, do not mix it with tainted data from untrustworthy sources (e.g. assumptions, values obtained from modeling etc.) to produce untrustworthy results. Note that type qualifiers can be composed—besides trustworthiness, values may have other properties we want to track: positiveness, constness, measurement units etc. The following pseudo-code involves multiple type qualifiers applied to the same type:

```
extern double untainted kg get_weight();
double const kg a = get_weight(); // OK, untainted dropped
double kg untainted b = a; // Error, no untainted in the RHS
b = get_weight(); // OK, qualifiers are preserved
```

We discuss later this section how to verify type safety of an assignment that involves multiple type qualifiers; here we just note that the order of application of type qualifiers to a type should not matter—it does not in our framework—and types that differ only in the order of qualifiers should be semantically equivalent. In what follows, by *qualified type* we mean a type that is obtained through applying one or more type qualifiers to an *unqualified type*.

As with pos and neg, we represent a type qualifier as a template class with a single parameter, representing the type being qualified. By taking advantage of the common properties of all type qualifiers, we can reduce the work that is necessary for defining a new qualifier. The developer of a type qualifier explicitly marks his template class as a type qualifier through specialization of a traits-like class is_qualifier, and defines the wrapper class. It is not necessary to alter the is_subtype metafunction or the subtype_cast functions. These rules follow according to whether the qualifier is *positive* or *negative* [12], which the programmer states in the definition of the qualifier class. An example definition is shown in Figure 4.

DEFINITION 1. *A type qualifier* q *is* positive *if* $T <: q\ T$ *for all types* T*. A type qualifier* q *is* negative *if* $q\ T <: T$ *for all types* T*.*

The C++ qualifier **const**, type qualifier tainted [26] and optional [6] are examples of positive type qualifiers because T <: **const** T, T <: tainted<T> and T <: optional<T>. Qualifiers pos, neg, nonnull, and untainted mentioned above are examples of negative type qualifiers because pos<T> <: T, nonnull<T> <: T, and untainted<T> <: T.

The overload resolution mechanism of C++ is based on comparing the structure of types of the formal and actual arguments and finding the best match. Thus, overloading various operators to implement the typing behavior of qualifiers in a straightforward manner of Figure 1, would "favor" the topmost qualifiers, and be dependent on the order of qualifiers.

To account for this, definitions of operations on type qualifiers have to be made independent of a particular order of qualifiers application. To do this, we note that type qualifiers do not change

```
template <class T>
struct untainted
  : negative_qualifier<typename unqualified_type<T>::type>
{
  typedef negative_qualifier<
            typename unqualified_type<T>::type
          > base;
  using base::unqualified_type;

  untainted() : base() {}
  untainted(const untainted& p) : base(p) {}
  explicit untainted(const unqualified_type& t) : base(t) {}

  template <class U>
  explicit untainted(
            const U& u,
            typename enable_if<
              is_subtype<U, untainted<T> >, void
            >::type* = 0
          ) : base(subtype_cast<T>(u)) {}

    template <class U>
    operator U() const
    { return subtype_cast<U>(*this); }

    untainted& operator=(const untainted& p) {
        unqualified_value() = p.unqualified_value();
        return *this;
    }

    template <class U>
    typename enable_if<
        is_subtype<U, untainted<T> >,
        untainted&
    >::type
    operator=(const U& u) {
        base::operator=(subtype_cast<T>(u));
        return *this;
    }
};
```

Figure 4. The definition of the untainted type qualifier class using the XTL framework.

the underlying operation, only the type of the result. For example, when we apply qualifier pos to type **int** we still use the addition operation defined on **int**s, but ask pos to be applied to the result type whenever both argument types are qualified with it. Taking this into account, XTL defines generic operations on qualified types and lets the user customize how a particular operation changes the type. For example, the following code shows how one would define the typing rules of the untainted and tainted qualifiers for the addition operator:

```
template <template<class> class A, template<class> class B>
struct plus {
  typedef mpl::identity<mpl::_1> type;
};

template <> struct plus<untainted, untainted>
{ typedef qual<untainted> type; };

template <> struct plus<untainted, tainted>
{ typedef qual<tainted> type; };
// ...
```

This traits-like template class takes two qualifier templates (note it expects template template parameters), and defines a metafunction that will be applied to the result type. It defaults to mpl::identity, which means that qualifiers neither have to be added to nor removed from the result type. The class template qual we use above defines a metafunction that applies a given qualifier template to the result type. XTL's generic implementation of a particular operation will loop through all possible combinations of qualifiers in argument type(s) and apply corresponding meta-functions to compute the qualifiers that should be applied to the result type.

To arrange that a particular operator is not dependent of the order of qualifiers in its argument types (for example that the overloading behavior of untainted<nonnull<optional<**int**>>> is the same as that of optional<untainted<nonnull<**int**>>>) the XTL uses enable_if to overload operators and functions for qualified types. The is_subtype metafunction can inspect the set of type qualifiers, and base the subtyping relation on the negativeness or positiveness of the qualifiers. Omitting some details, the rule of thumb is that to preserve a subtyping relation, a positive type qualifier can only be added to the right-hand side, and a negative type qualifier can only be removed from the left-hand side. For example: nonnull<optional<untainted<T>>> is a subtype of tainted<optional<nonnull<U>>> whenever T <: U. Here the negative type qualifier untainted was dropped from the left-hand side while positive type qualifier tainted was added to the right hand side. Other qualifiers were preserved. Dropping the optional qualifier in the right hand side would have made subtyping to fail. The XTL's definition of is_subtype has this behavior.

```
nonnull<neg<int> > a(−44);
pos<untainted<nonnull<int> > > b(2);

neg<nonnull<long> > m = a * b; // OK
nonnull<pos<double> > e = b − a; // Error: nonnull
pos<double> d = b − a; // OK: no nonnull

nonnull<tainted<double> > bc =
  subtype_cast<nonnull<tainted<double> > >(b);
nonnull<tainted<double> > bi = b; // same as above
bi = b;

pos<nonnull<int> > c(3);
string cc =
  subtype_cast<nonnull<tainted<string> > >(c);
string ci = c;
ci = c; // Oops
```

Figure 5. Example of working with XTL's qualifiers.

To give a feel of working with type qualifiers built with XTL, Figure 5 shows code using some of the type qualifiers mentioned above. The last assignment in the example fails to compile. This is because string defines its assignment operator as a template, and as a result XTL's conversion operator in the supertype is not tried. As discussed in Section 3.3, we may thus sometimes need to resort to an explicit conversion using subtype_cast; in this example, we could write ci = subtype_cast<string>(c);.

Even with the subtyping and casting functionality provided by the XTL, the definition of an individual type qualifier class is still fairly elaborate, but mostly boilerplate code. For cases where no special run-time checks are needed, the XTL provides two macros for taking care of this boilerplate:

```
DECLARE_POSITIVE_QUALIFIER(tainted);
DECLARE_NEGATIVE_QUALIFIER(untainted);
```

The metafunctions that describe how different operations carry the qualifiers must obviously still be defined.

We are experimenting with an alternative design, where all qualified types are represented as instances of a single qualified template, taking two type parameters: the element type and an MPL type list. In this design, an individual qualifier type becomes an empty *tag* class, simplifying the definition of new qualifiers even more. Another benefit is that the list of qualifiers is in a readily accessible form for querying and manipulating with metaprograms. The (subjective) drawback is a less natural syntax for the user.

4. Typing XML in C++

In this section, we describe how the XTL, with the help from several C++ template libraries, allow an elaborate extension to the C++'s type system: static typing of XML.

4.1 Background: regular expression types

Type systems that understand XML data have gained considerable interest. The central idea is to harness the type system to guarantee statically that a particular program cannot manipulate, or produce XML documents that do not conform to a particular DTD [32] or Schema [25]. The insight is that XML data corresponds directly to *regular expression types*, which then can be given a representation in the type systems of various languages. Some of the recent efforts in this direction include the XDuce language [16] specifically designed for XML processing, that has a direct representation for regular expression types; the C$_\omega$ [24] and Xtatic [14] languages that extend C# with regular expression types; and the HaXml [31] toolset, that uses Haskell's algebraic data types to represents XML data.

Regular expression types, e.g. as defined in XDuce, are sets of sequences over certain domains. Values from those domains denote singleton and composite sequences. Composite sequences are formed with the regular expression operators , (concatenation), | (alternation), $*$ (repetition) and ? (optionality) together with type constructors of the form $l[\cdots]$. If S and T are types, then S, T denotes all the sequences formed by concatenating a sequence from S and a sequence from T. $S|T$ denotes a type that is a union of sequences from S and sequences from T. Type $l[T]$, where T is a type and l ranges over a set of labels, defines a set of labeled sequences, where each sequence from T becomes classified with the label l. Type $T*$ denotes a set of sequences obtained by concatenating any finite number of times sequences from T. The empty sequence is denoted with () and $T?$ denotes any sequence T or an empty sequence.

Consider for example the following XML snippet describing a path to a file:

```
<path>
  <dir-name>C</dir-name>
  <dir-name>Media</dir-name>
  <dir-name>Video</dir-name>
  <file-name>Experience.mpg</file-name>
</path>
```

This snippet conforms to the following XML Schema:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsd:schema xmlns:xsd
  ="http://www.w3.org/2001/XMLSchema">
<xsd:element name="dir-name"  type="xsd:string"/>
<xsd:element name="file-name" type="xsd:string"/>
<xsd:element name="path">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="dir-name"
                   maxOccurs="unbounded"/>
      <xsd:element ref="file-name"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

The labels of XDuce mentioned above classify types similarly to how XML tags classify their content; For example, dir−name[...] corresponds to <dir−name>...</dir−name> in XML parlance. Still using XDuce syntax, the regular expression type

```
path[dir−name[string]∗, file−name[string]]
```

defines a set of XML snippets that match the above schema, that is, the set of XML snippets with "path" as a root element that contains zero or more "dir-name" elements followed by a "file-name" element.

An interesting feature of the XDuce language is the subtyping relation between regular expression types. This subtyping relation is defined as *semantic subtyping*, as the subset relation between languages generated by two tree automata [16]. Defining subtyping between regular expression types corresponds to defining which XML fragments are safely convertible to which other XML fragments. For example, the following subtyping relationships hold:

```
DirName∗, FileName <: (DirName | FileName)∗
(DirName, DirName)∗, DirName <: DirName, DirName∗
```

An application of subtyping between XML fragments is, for example, to provide backward compatibility of documents that correspond to older schema: code written against a newer schema should work for older schemas, as long as the type defined by the newer schema is a supertype of the type defined by the older one.

The decision problem for such subtyping relation is EXPTIME-hard [17] in the worst case, but has reported to be efficient in cases of practical interest.

4.2 Regular expression types in C++

We define an encoding of regular expression types in C++. Regular expression types are represented as nested template instantiations, consisting of sequence types, variants, and lists. We represent XML elements in our system as a pair of two types, the first of which represents the element's tag and second the element's data:

```
template <class Tag, class T = detail::empty>
class element {
  T data;
};
```

The Tag type denotes the name of the XML element, or the label in XDuce's regular expression types. Empty XML elements can be represented by an element instantiated with nothing but a tag type. Complex XML elements may have several levels of instantiation of element passed as their data type. Consider for example the following XML snippet:

```
<path>
  <file-name>Experience.mpg</file-name>
</path>
```

It could be given the following type in our library:

```
struct path { /*...*/ };
struct file_name { /*...*/ };
typedef xml::element<
            path,
            xml::element<file_name, std::string>
        > MyXMLDoc;
```

Tag-classes are also used to keep additional information about the tag: the name as a character array, XML node type, additional restrictions etc.

```
struct file_name {
  static const char* tag_name() { return "file_name"; }
  typedef xml::attribute node_type;
};
```

Sequencing of XML elements is represented with xml::seq class that is just a simple wrapper around Fusion's tuple class [20]. The reason we wrap Fusion tuples is that having a wrapper class abstracts us from a particular implementation, and allows us to change behavior of certain operations, e.g., by performing some preprocessing on the tuple types. For example, we wrap the I/O operators of Fusion tuples to do a *flattening* of sequences: e.g., the sequence (A, B, (C, D), E) is flattened to (A, B, C, D, E). Fusion tuples are MPL-compliant sequences [1, 15]; we operate on the fusion tuples with MPL algorithms in our subtyping algorithm. Here is an example of using sequencing of elements:

```
typedef xml::element<file_name, string> XMLFileName;
typedef xml::element<dir_name, string> XMLDirName;
typedef xml::element<
            path,
            xml::seq<XMLDirName, XMLDirName, XMLFileName>
        > XMLPathOfDepth3;
```

The empty sequence () is represented by xml::seq<>.

Alternation of XML elements is represented with the xml::alt class template that is again just a simple wrapper, now around Boost's variant template [8]. In case of alternation, the wrapping is to be able to redefine the I/O routines. Here is a small example of using alternation:

```
typedef xml::element<file_name, string> XMLFileName;
typedef xml::element<dir_name, string> XMLDirName;
typedef xml::alt<XMLFileName, XMLDirName> XMLPathChunk;
```

The empty union is represented by xml::alt<>.

A repetition of XML elements is represented with the xml::rep class, which is a wrapper over an std::vector of Fusion tuples. Using repetition, the path definition from Section 4.1 can be expressed as follows:

```
typedef xml::element<file_name, string> XMLFileName;
typedef xml::element<dir_name, string> XMLDirName;
typedef xml::element<
           path,
           xml::seq<xml::rep<XMLDirName>, XMLFileName>
        > XMLPath;
```

We chose to implement repetition rather than the more general *recursion* to account for restrictions imposed on us by the C++ language. We omit details here, but point out that our current implementation of the subtyping algorithm cannot handle all the cases that involve repetition: we are working on extending the algorithm.

4.2.1 Subtyping relation

We utilize the static metaprogramming capabilities of C++ to establish a subtyping relation between two regular expression types. Again, the is_subtype metafunction is harnessed for this purpose. We do not currently support the full semantic subtyping relation of XDuce; as we already mentioned an implementation of *general recursion* or even *right/tail recursion* as used in XDuce seems problematic as a pure C++ library. Nevertheless, implementation of *repetition*, which is functionally equivalent to right/tail recursion seems to be feasible, which is what we are currently working on.

Also, unlike XDuce, we currently do not allow subtagging — subtyping on tags. We assume that only elements with the same tags can be in subtyping relationship and those with different tags are automatically assumed not to be in such a relationship. We do not anticipate difficulties in implementing subtagging within our framework.

The implementation of is_subtype metafunction for XML types is fairly intricate and we do not show it here. It amounts to implementing the subtyping rules of XDuce (really a limited form of them per the restrictions mentioned above) using MPL. Once the is_subtype metafunction has been defined to recognize our XML types, we can exploit it to implement guards similar to those in the constructors of the qualified types—Figure 6 demonstrates with the converting constructor of the element class template.

```
template <class Tag, class T = detail::empty>
class element {
  // A converting constructor, matches if the
  // argument is a "subtype" in the library's
  // type system
  template <class UTag, class U>
  element(const element<UTag, U>&,
          typename enable_if<
                     is_subtype<
                       element<UTag, U>,
                       element< Tag, T>
                     >,
                     void
                   >::type* = 0
         )
  { ... }
  // rest of the class definition ...
};
```

Figure 6. The converting constructor of the element class template.

We do not overload subtype_cast_impl to define conversions on element types because we define such conversions on the element class itself. Calling subtype_cast on the element type then calls the most general implementation of subtype_cast_impl, which simply tries to apply either a standard or a user defined conversion on the type, which is exactly what we need.

4.3 Supporting functionality

In addition to the core "type system", we have implemented some supporting functionality as part of our XML processing library. This includes I/O, and automatic generation of the C++ types from an XML Schema. For I/O, we provide both direct streaming operations, and a module for interacting with the *Expat* parser [10]. For generating the C++ types corresponding to a particular XML Schema, we use XSL transformations.

Figure 7 presents a complete example of working with our XML framework. The code marked by comments are taken verbatim from the header file generated from a corresponding `.xsd` file (the XML Schema description).

4.4 Impact on compile times

Heavy use of template metaprogramming is known to increase compile times of C++ programs, often significantly. We conducted experiments to estimate the impact that library-defined type systems written using the XTL have on compile times. We tested both the uses of type qualifiers and the use of the XML framework. All tests were performed under GCC 3.4.4 on Intel Pentium M processor running at 2 GHz with 512 MB of RAM.

Our test-suit for type qualifiers consisted of 11 test programs, all semantically equivalent, but having a different number of qualifiers attached to the types of the values with which the program performed computations: program with index n used n qualifiers. Each program consisted of 20 functions, each of which instantiated values of two different types and then performed an operation on them. Each applied qualifier had up to 4 rules defined on it. Compilation times (in seconds) for these tests are given in Table 1. The compile time should be compared against the value in row 0, which defines the baseline: the equivalent program without any qualifiers.

The subtyping relation of the XML types is computationally more expensive than that of type qualifiers. As mentioned earlier, in the general case deciding subtyping of regular expression types is EXPTIME-hard. The computationally expensive cases are subtyping relations of the following form:

$$l(A_1, \cdots, A_n) <: l(B_{11}, \cdots, B_{1n})| \cdots |l(B_{k1}, \cdots, B_{kn})$$

```
using namespace xml;
using namespace std;

// >>>> generated definitions >>>> //
struct name { /*...*/ }; struct email { /*...*/ };
struct tel { /*...*/ }; struct icq { /*...*/ };
struct contact { /*...*/ };

typedef element<name, string> XMLName;
typedef element<email, string> XMLEmail;
typedef element<tel, alt<string,int> > XMLTel;
typedef element<icq, int> XMLICQ;
typedef alt<XMLEmail,XMLTel,XMLICQ> AnyField;
typedef element<contact,
        seq<XMLName,XMLEmail,XMLTel> > XMLOldContact;
typedef element<contact,
        seq<XMLName,AnyField,AnyField> > XMLNewContact;
// <<<< generated definitions <<<< //

int main() {
        try {
         ifstream xml("old−contact.xml"); // must exist
         XMLOldContact old_contact;
         xml >> old_contact; // parse file
         XMLNewContact new_contact = old_contact; // OK
         cout << new_contact; // output XML source
         // old_contact = new_contact; // ERROR
        }
        catch(invalid_input& x) {
            cerr << "Error parsing " << x.what();
            return −1;
    }
    return 0;
}
```

Figure 7. Complete example of working with XML. Type XMLOldContact is a subtype of XMLNewContact, which is why assignment new_contact = old_contact is allowed while old_contact = new_contact is not.

N	Time
0	1.12
1	1.72
2	2.61
3	2.81
4	3.18
5	3.94
6	4.97
7	5.55
8	6.28
9	15.40
10	19.22

Table 1. Compilation times, in seconds, of the type qualifiers test programs.

Verification of such a relation in the general case involves a number of steps that is proportional to the number of ways a k-element set can be split into n disjoint sets. Detailed discussion can be found in [17].

We wanted to test the effect of this worst-case scenario on compile times. Our test suite for the XML type system consisted of 81 tests—one for each combination of n and k (ranging from 1 to 9) from the above relation. In each of the tests we invoked is_subtype metafunction using XML types that trigger the exponential case with essentially the following code:

```
BOOST_STATIC_ASSERT((
    is_subtype<
        element<a, seq<AK0, ... , AKN> >,
        alt<element<a, seq<A00, ... , A0N> >,
            element<a, seq<A10, ... , A1N> >,
            ... ,
            element<a, seq<AK0, ... , AKN> >
           >
    >::type::value
));
```

Table 2 represents compilation times in seconds for different values of n and k. Empty entries correspond to tests that did not finish within 10 minutes.

n/k	1	2	3	4	5	6
1	1.42	1.60	1.62	1.73	1.93	2.08
2	1.42	1.72	2.25	3.56	7.98	24.91
3	1.38	1.67	2.47	5.75	29.85	
4	1.52	1.85	3.50	24.27	155.25	
5	1.44	2.03	6.74	130.72		
6	1.52	2.28	17.89	179.72		
7	1.46	2.64	54.94			
8	1.66	3.48	155.41			
9	1.58	4.91				

Table 2. Compilation times, in seconds, of the test programs for the XML type system.

We note that even though these times grow exponentially, other implementations of the subtyping algorithm have been reported to behave satisfactorily on practical examples [17], suggesting that the exponential case with large n and k does not manifest often in practice.

Summarizing the test results, using our approach to refining type systems can have a notable negative impact on compile times. For example in the case of the type qualifiers, the slowdown is quite reasonable, and typical for libraries relying on template metaprogramming. In the XML case, the pathological cases that lead to exponential growth in the cost of deciding subtyping also obviously increase the compilation times exponentially. The tests we performed are only suggestive.

5. Conclusions and future work

We presented a library solution for extending the type system of a general-purpose programming language with typing of domain-specific abstractions. The presented solution does not require any compiler support and can be fully implemented in standard C++ [27]. We have demonstrated that it is feasible to implement elaborate typing behavior purely as a library.

In the future, we plan to explore the limits of the approach, implementing different kinds of type system extensions in terms of the XTL tools we described, as well as polish implementations of the ones we presented, and make them publicly accessible. Our XML framework currently lacks full support of subtyping for repetition, which is part of our future work. Also, going more into XML domain, currently we only support a basic subset of primitive XML data types. We plan to extend this support to other built-in types as well as possibly provide a compile- and run-time support of facets. It is important to stress that the presented XML processing library is not an XML parser—its purpose is to provide compile-time guarantees that the XML code produced by a particular application corresponds to the appropriate XML Schema. It also provides means for automatic conversion between conforming XML representations, as well as provides a machinery to map XML Schema definitions to corresponding library abstractions.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[2] J. Barton and L. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[4] G. Bracha. Pluggable type systems. October 2004.

[5] W. E. Brown. Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation. In *In Second Workshop on C++ Template Programming*, Oct. 2001. in conjunction with OOPSLA'01, `http://www.oonumerics.org/tmpw01/brown.pdf`.

[6] F. Cacciola. Boost Optional. `http://www.boost.org/libs/optional/`.

[7] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 85–95, New York, NY, USA, 2005. ACM Press.

[8] I. M. Eric Friedman. Boost Variant. `http://www.boost.org/doc/html/variant.html`, 2002.

[9] D. Evans. Static detection of dynamic memory errors. In *PLDI '96: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, USA, May 1996. ACM Press.

[10] The Expat XML Parser. `http://expat.sourceforge.net/`, 2006.

[11] J. S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. University of California, Berkeley, 2002.

[12] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[13] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[14] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[15] A. Gurtovoy. The Boost MPL library. `http://www.boost.org/libs/mpl/doc/index.html`, July 2002.

[16] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In D. Suciu and G. Vossen, editors, *International Workshop on the Web and Databases (WebDB)*, May 2000. Reprinted in *The Web and Databases, Selected Papers*, Springer LNCS volume 1997, 2001.

[17] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[18] J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, June 2003.

[19] J. Järvi, J. Willcock, and A. Lumsdaine. *Boost enable_if Library*. Boost, 2003. `http://www.boost.org/libs/utility/enable_if.html`.

[20] D. M. Joel de Guzman. Boost Fusion. `http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion/doc/html/index.html`, April 2006.

[21] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004.

[22] A. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming (ESOP)*, volume 788 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[23] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 213–225, New York, NY, USA, 2003. ACM Press.

[24] M. Research. Cw. `http://research.microsoft.com/Comega/`, 2005.

[25] XML Schema. `http://www.w3.org/XML/Schema`, 2005.

[26] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers, 2001.

[27] I. . I. Standard. *Programming languages - C++*. American National Standards Institute, September 1998.

[28] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 1986.

[29] T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

[30] T. L. Veldhuizen. C++ templates are Turing complete. `www.osl.iu.edu/~tveldhui/papers/2003/turing.pdf`, 2003.

[31] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.

[32] Extensible markup language (XML^{TM}). `http://www.w3.org/XML`, 2005.

Anti-Deprecation: Towards Complete Static Checking for API Evolution

S. Alexander Spoon
LAMP, Station 14
Swiss Federal Institute of Technology in Lausanne (EPFL)
CH-1015 Lausanne
lex@lexspoon.org

ABSTRACT

API evolution is the process of migrating an inter-library interface from one version to another. Such a migration requires checking that all libraries which interact through the interface be updated. Libraries can be updated one by one if there is a transition period during which both updated and non-updated libraries can communicate through some transitional version of the interface. Static type checking can verify that all libraries have been updated, and thus that a transition period may end and the interface be moved forward safely. Anti-deprecation is a novel type-checking feature that allows static checking for more interface evolutions periods. Anti-deprecation, along with the more familiar deprecation, is formally studied as an extension to Featherweight Java. This formal study unearths weaknesses in two widely used deprecation checkers.

"In Java when you add a new method to an interface, you break all your clients.... Since changing interfaces breaks clients you should consider them as immutable once you've published them." –Erich Gamma [21]

"`NoSuchMethodError`" –Java VM, all too frequently

1. OVERVIEW

Libraries communicate with each other via application programming interfaces (API's), or *interfaces* for short. The key idea with interfaces is that so long as a set of libraries conform to their interfaces, those libraries will tend to function together when they are combined. This approach is a key part of standard discussions of software modularity [2].

This interfaces idea supports independent evolution of libraries, in that libraries can be updated so long as they continue to conform to their interfaces. However, this strategy does not address evolution of the interfaces themselves. Since in practice the first definition of an interface is often insufficient, practitioners need some approach for improving interfaces. This is the problem of *interface evolution.*

LCSD 2006 Portland, Oregon, USA

Interface evolution arises in practice for large-scale projects with multiple independent development groups. The Eclipse project, for example, includes plugin code written by development groups all over the world. For such projects, substantial attention is put onto the problem of safely upgrading interfaces [5].

Transition periods provide a general mechanism for evolving the interfaces between independently maintained libraries. A transition period is a period of time during which both updated and non-updated libraries can successfully communicate through an evolving interface.

During a transition period, all libraries that conform to the original version of an interface must be allowed to continue to function. As the transition period progresses, more and more libraries should be updated for the forthcoming version of the interface, while continuing to work with the transitional version of the interface. A transition period can successfully terminate when all libraries communicating through the interface have been either updated or abandoned. At that time, the interface itself can be upgraded.

Static type checking can be used to verify that a transition period may be safely entered or left. At the beginning of a transition period, static checking can ensure that all libraries conforming to the current interface will continue to conform to the new, transitional interface. At the end of a transition period, static checking can ensure that all checked libraries are ready to progress to the next version of the interface. The same checker can be used for both purposes if the checker has two strictness levels. The *strict* level is used to check the exit from transition periods, while the looser *transitional* level is used for all other type-checking purposes.

This article studies static type checking for *deprecation* and *anti-deprecation* of methods. Deprecation is widely used, while anti-deprecation appears to be novel for programming languages. After describing the features in general, the article defines them formally as an extension to Featherweight

```
public interface ConnectionListener {
  public void connectionClosed();
  public void connectionClosedOnError(Exception e);
}

public interface ConnectionListener2
extends ConnectionListener {
  public void connectionAuthenticated();
}
```

Figure 1: Two interfaces from Eclipse. The second interface is the same as the first except that it requires one new method.

```
public interface ConnectionListener {
  public void connectionClosed();
  public void connectionClosedOnError(Exception e);

  encouraged public void connectionAuthenticated();
}
```

Figure 2: With encouraged methods, the new method could have been gradually phased into the original interface.

Java [11], and proves several core properties about the formalism. This systematic study not only defines the new feature, but unearths two places where current deprecation checkers could be improved.

2. STATIC TRANSITION CHECKING

Static checking can help both entering and leaving transition periods. When entering a transition period, the checker can verify that clients will continue to compile and run, even if not all libraries using the interface are available. As the transition period moves forward, each library's developers can use the checker as they update their library to verify that their updates are sufficient for the next version of the interface. Once all libraries have been updated and checked, it is safe to move the interface forward.

Put another way, the entries and exits of transition periods are refactorings [14]. If the static checker is satisfied, then crossing these end points causes a change in program syntax but not in program behavior.

Not all libraries need to be available to those maintaining the interface. The conditions for entering a transition period are typically weak, thus giving interface maintainers broad liberty to start an interface transition. Leaving the transition period requires more work, but it does not need to be finished immediately. Every library whose components use the interface must be checked with the strict checker, but those checks can occur throughout the transition period. Once the (loose) organization of library maintainers have decided that sufficient checking has occurred, and if no errors are known to be present, the transition period can be left.

Organization processes for deciding that enough library assemblies have been checked that a transition period may be left are beyond the scope of this article. Presumably, however, some such agreement has been reached among the library developers. As one example arrangement, the maintainers of the interface might commit to a minimum length of evolution period. That length might be *e.g.* six months, a year, or five years. Anyone building assemblies that use that interface must periodically check their library, with a period no longer than the agreed length of evolution periods.

A static transition checker can be described as having two modes: *transitional* and *strict*. If a library passes the transitional checker, then the library can communicate with other libraries through the interface. If a library additionally passes the strict checker, then the library will also continue to work if the interface is updated. The strict checker takes into consideration extra annotations describing the desired interface changes, while the transitional checker mostly ignores such annotations.

Implementations can combine the two checking modes. All code must pass the transitional checker, while failure to additionally pass the strict checker causes interface-evolution warnings.

3. ANTI-DEPRECATION

Deprecation allows a static checker to emit warnings whenever a caller tries to use a method that is expected to disappear in a future version of an interface. A complementary scenario is also important: sometimes a future version of an interface will require an additional method. An annotation for such future required methods could be called *anti-deprecation.*

The typical usage for anti-deprecation is shown in Figures 1 and 2. Figure 1 shows one of Eclipse's "I*2" interfaces, an interface that is an extension of an earlier interface. Experience with the framework showed that the earlier interface was too thin, but given the nature of Java interfaces, new methods could not be added to the existing, published interface. Thus, the Eclipse developers added a second interface which merely extends the first interface and adds one new method. With encouraged methods, the designers would have had the option to phase in the method to the existing interface, as shown in Figure 2.

A simple way to annotate anti-deprecation is to add an `encouraged` keyword to the language. Unlike other methods, a method marked as `encouraged` cannot be called. Its presence only serves to mark that a future version of the interface will include that same method as `abstract`.

During transitional checking, `encouraged` methods are, for the most part, treated as if they were not present at all. The only restriction is that encouraged methods cannot override other non-encouraged methods. Allowing such would be complicated and unhelpful—after all, if a method is already present due to inheritance, what use is it to encourage it further? The one exception, that encouraged methods can nonetheless override other encouraged methods, is necessary so that encouraged methods can be added *over* other encouraged methods. In strict checking, even this case is not allowed, and the encouraged method deeper in the hierarchy needs to be removed.

$$
\begin{array}{rcl}
L & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \{\ \bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\ \} \\
X & ::= & \texttt{deprecated}\ m; \\
K & ::= & C(\bar{C}\ \bar{f})\ \{\ \texttt{super}(\bar{f});\ \texttt{this}.\bar{f} = \bar{f};\ \} \\
M & ::= & C\ m(\bar{C}\ \bar{f})\ MB \\
MB & ::= & \{\ \texttt{return}\ e\ ;\ \}\ \mid \texttt{abstract} \mid \texttt{encouraged} \\
e & ::= & x \mid e.f \mid e.m(\bar{e}) \mid \texttt{new}\ C(\bar{e}) \mid (C)e
\end{array}
$$

Figure 3: Syntax of FJ-ADE

During strict checking, `encouraged` methods add several requirements for programs to pass the checker. First, any method that overrides an `encouraged` method must have the required parameter types and return type. This requirement is present so that when an encouraged method is later promoted to a required method, all methods overriding it will have conforming types. Second, every subclass of a class with an encouraged method must either implement the method or be considered abstract and uninstantiable.

The combination of deprecation and anti-deprecation allows for an additional class of changes that neither mechanism supports alone: arbitrary changes to a method's signature. For example, one might wish to change the set of exceptions thrown by a method, or change a method's return type, or change its public or private visibility.

Such changes can always be accomplished using four transition periods. The first period introduces a new version of the method with a different name than the original method. Since the method is new, it can be given any type signature at all. The second period deprecates the original method, thus inducing callers to use the new version of the method. The third period replaces the deprecated original method with an encouraged method of the desired signature. The fourth period deprecates the temporary method name, thus inducing clients to change back to using the original method.

Alternatively, developers can choose a shorter two-phase sequence if they are content for the new method to have a different name from the original. They can simply stop after the first two transition phases.

These rules for `encouraged` and `deprecated` might seem pessimistic. These rules are formed under the assumption that developers in other groups might both implement any interface and invoke the methods it advertises. If this assumption were changed to restrict what other developers can do, then some interface changes could be safely performed with fewer or even no transition periods.

For example, suppose that one party controls an interface along with all of its implementors. In that case, that party can add methods to the interface without needing a transition period. They can simply make a simultaneous release of the updated interface and the updated implementors of that interface. Likewise, if one party controls all callers to an interface, *e.g.* as with call backs, that party can remove methods from the interface without needing a transition period.

The present work addresses the less constrained scenario where outside developers can both implement an interface and call through it. The main reason for this choice is that it is the more general and difficult case. However, notice that even when outside developers are expected to be more constrained in their work, it is desirable to allow them the greater flexibility. At the least, it is useful for testing if programmers can implement their own mock objects to stand in place of the usual ones [12, 9].

4. EXTENDING FEATHERWEIGHT JAVA

While `deprecated` and `encouraged` are simple to describe, it proves tricky to develop the precise rules for checking them so that transition periods can be safely entered and left. In order to determine the precise checking rules, the bulk of this article focuses on a formal study of a small language including these keywords.

The keywords are added to Featherweight Java (FJ) [11], a language that has several appealing characteristics: it is tiny, making it amenable to formal study; it uses familiar syntax, so that the work is more approachable; and it captures two features at the heart of object-oriented languages, message sending and inheritance.

In one way, though, the FJ language is a little too small for the present purpose: it does not include a notion of interfaces. Instead of adding a full interface concept, it suffices to add abstract methods. Abstract methods allow abstract classes, which for the present purpose serve as perfectly fine interfaces. The full extended language is called FJ-ADE because it is Featherweight Java with three new keywords: `abstract`, `deprecated`, and `encouraged`.

The notation is generally that of FJ. When a line of code is written down by itself as an assumption, the meaning is that that line of code appears somewhere in the program. A sequence is written $\bar{x}$, denoting the sequence $x_1, \ldots, x_n$, where $\#(\bar{x}) = n$. The empty sequence is $\bullet$ by itself, while a comma between two sequences denotes concatenation. Pairs of sequences are a shorthand for a sequence of pairs; for example, $\bar{C}\ \bar{x}$ means $C_1\ x_1\ \ldots\ C_n\ x_n$. The notation $x \in \bar{y}$ means that $x = y_i$ for some i. Negation, written $\neg P$, is not boolean negation, but instead means that P cannot be proven with the available inference rules.

The syntax of FJ-ADE is given in Figure 3. There are a few differences from FJ:

- Methods can be abstract. Any class that defines or inherits an abstract method is considered abstract and cannot be instantiated with `new`.
- Methods can be encouraged. An encouraged method will be added to a future version of the class with the specified type signature.
- Each class has a list of deprecated methods. Deprecated methods are going to be removed in a future version of the class.

Subtyping for FJ-ADE is shown in Figure 7. As in FJ, it exactly follows the class hierarchy.

$$\text{T-Var}\;\frac{x : C \in \Gamma}{\Gamma \vdash x : C}$$

$$\text{T-Field}\;\frac{\Gamma \vdash e_0 : C_0 \qquad \mathit{fields}(C_0) = \bar{C}\ \bar{f}}{\Gamma \vdash e_0.f_i : C_i}$$

$$\text{T-New}\;\frac{\begin{array}{c}\mathit{fields}(C) = \bar{D}\ \bar{f} \qquad \mathit{str}; \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C} <: \bar{D} \\ \neg \mathit{abstract}(C) \qquad (\mathit{str} = \texttt{trans}) \vee (\neg \mathit{postabs}(C))\end{array}}{\mathit{str}; \Gamma \vdash \texttt{new}\ C(\bar{e}) : C}$$

$$\text{T-Invk}\;\frac{\begin{array}{c}\mathit{str}; \Gamma \vdash e_0 : C_0 \\ \mathit{mtype}(m, C_0, \texttt{false}, \mathit{str} = \texttt{trans}) = \bar{D} \rightarrow C \\ \mathit{str}; \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C} <: \bar{D}\end{array}}{\mathit{str}; \Gamma \vdash e_0.m(\bar{e}) : C}$$

$$\text{T-UCast}\;\frac{\Gamma \vdash e_0 : D \qquad D <: C}{\Gamma \vdash (C)e_0 : C}$$

$$\text{T-DCast}\;\frac{\Gamma \vdash e_0 : D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)e_0 : C}$$

$$\text{T-SCast}\;\frac{\Gamma \vdash e_0 : D \qquad D \not<: C \qquad D \not<: C \qquad \textit{stupid warning}}{\Gamma \vdash (C)e_0 : C}$$

Figure 4: Typing of expressions

$$\text{T-Method-Fresh}\;\frac{\begin{array}{c}\mathit{str}; \bar{x} : \bar{C}, \texttt{this} : C \vdash e_0 : E_0 \qquad E_0 <: C_0 \\ \texttt{class}\ C\ \mathit{extends}\ D\ \{\ldots\} \\ \neg \mathit{mavail}(m, D, (\mathit{str} = \texttt{strict}), \texttt{true})\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \{\ \texttt{return}\ e_0;\ \}\ \ \mathit{str}\text{−OK IN } C}$$

$$\text{T-Method-Over}\;\frac{\begin{array}{c}\mathit{str}; \bar{x} : \bar{C}, \texttt{this} : C \vdash e_0 : E_0 \qquad E_0 <: C_0 \\ \texttt{class}\ C\ \mathit{extends}\ D\ \{\ldots\} \\ \mathit{mtype}(m, D, (\mathit{str} = \texttt{strict}), \texttt{true}) = \bar{D} \rightarrow D_0 \\ \bar{C} = \bar{D} \qquad C_0 = D_0\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \{\ \texttt{return}\ e_0;\ \}\ \ \mathit{str}\text{−OK IN } C}$$

$$\text{T-Method-Abs}\;\frac{\begin{array}{c}\texttt{class}\ C\ \mathit{extends}\ D\ \{\ldots\} \\ \neg \mathit{mavail}(m, D, (\mathit{str} = \texttt{strict}), \texttt{true})\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \ \texttt{abstract}\ \ \mathit{str}\text{−OK IN } C}$$

$$\text{T-Method-Enc}\;\frac{\begin{array}{c}\texttt{class}\ C\ \mathit{extends}\ D\ \{\ldots\} \\ \neg \mathit{mavail}(m, D, (\mathit{str} = \texttt{strict}), \texttt{true})\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \ \texttt{encouraged}\ \ \mathit{str}\text{−OK IN } C}$$

Figure 5: Typing of methods

$$\text{T-Class}\;\frac{\begin{array}{c}K = C(\bar{D}\ \bar{g},\ \bar{C}\ \bar{f})\ \{\ \texttt{super}(\bar{g});\ \texttt{this}.\bar{f} = \bar{f};\ \} \\ \mathit{fields}(D) = \bar{D}\ \bar{g} \qquad \bar{M}\ \mathit{str}\text{−OK IN } C \\ \forall m \in \bar{X} : \mathit{candep}(C, m)\end{array}}{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\ \}\ \ \mathit{str}\text{−OK}}$$

Figure 6: Typing of classes

$$C <: C$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ E\ \{\ldots\} \qquad E <: D}{C <: D}$$

Figure 7: Subtyping

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\} \\ \mathit{fields}(D) = \bar{D}\ \bar{g}\end{array}}{\mathit{fields}(C) = \bar{D}\ \bar{g}, \bar{C}\ \bar{f}}$$

Figure 8: Field lookup

An entire program is denoted CT or CT'. Notationally, CT is a table, and $CT(C)$ is the class named C in program CT. Valid programs have several syntactic restrictions: the inheritance hierarchy is non-cyclical, all field names and parameter names are distinct, $\texttt{Object} \notin \mathit{dom}(CT)$, and every class name appearing in the program is in the domain of CT.

The *fields* function, defined in Figure 8, computes the complete list of fields in a class.

The *mtype* function, defined in Figure 9, looks up the type of a method assuming it is invoked on a particular class. As compared to FJ, FJ-ADE's *mtype* function has two new flag parameters: one determining whether to include methods that are merely encouraged, and one determining whether to include methods that have been deprecated. While FJ's *mtype* considers all methods equally, FJ-ADE's *mtype* optionally declines to consider deprecated or encouraged methods or both, according to the two flags. Such methods are significant or not in different contexts in the type checker, and thus *mtype* must have extra parameters.

One particular complication is the treatment of deprecated methods when the fourth flag is `false`. In that case, *mtype* is still defined for that method if the chain of methods it overrides includes a non-deprecated method, and if all methods in that chain up to the non-deprecated method have the same type signature. The addition of this case means that core properties about *mtype* remain simple. See Lemma 1 and Lemma 2.

The *mavail* relation, also shown in Figure 9, claims that a method is available in a class without being specific about the method's type. Its arguments are the same as for *mtype*.

The *mbody* function, defined in Figure 10, is used during evaluation to find the method responding to a message-send expression. It is the same as in FJ except that there are two new clauses to support abstract and encouraged methods.

The *abstract* function, also defined in Figure 10, checks whether a class defines or inherits an abstract method. Note that this definition ignores `encouraged` methods, because those methods are not yet available.

$$\textsc{MT-Here}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\} \\ B\ m(\bar{B}\ \bar{x})\ MB \in \bar{M} \\ \mathit{enc} \vee (MB \neq \texttt{encouraged}) \\ \mathit{dep} \vee (m \notin \bar{X})\end{array}}{\mathit{mtype}(m, C, \mathit{enc}, \mathit{dep}) = \bar{B} \to B}$$

$$\textsc{MT-Inher}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\} \\ m \notin \bar{M} \\ \mathit{mtype}(m, D, \mathit{enc}, \mathit{dep}) = \bar{B} \to B\end{array}}{\mathit{mtype}(m, C, \mathit{enc}, \mathit{dep}) = \bar{B} \to B}$$

$$\textsc{MT-Depover}\ \frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\bar{C}\ \bar{f};\ K\ \bar{X}\ \bar{M}\} \\ B\ m(\bar{B}\ \bar{x})\ MB \in \bar{M} \qquad m \in \bar{X} \\ \mathit{mtype}(m, D, \mathit{enc}, \texttt{false}) = \bar{B} \to B\end{array}}{\mathit{mtype}(m, C, \mathit{enc}, \texttt{false}) = \bar{B} \to B}$$

$$\frac{\mathit{mtype}(m, C, \mathit{enc}, \mathit{dep}) = \bar{D} \to D}{\mathit{mavail}(m, C, \mathit{enc}, \mathit{dep})}$$

Figure 9: Method type lookup

Post-abstract classes are those that might become abstract after the program evolves forward. The *postabs* function, defined in Figure 11, gives a conservative notion of post-abstract classes. It is defined in terms of a *postneeds* function which claims, more specifically, that the class might lack a particular method following either the removal of `deprecated` methods or the upgrading of `encouraged` methods to `abstract` or both. A class that *postneeds* any method at all is considered post-abstract.

The type checker of FJ needs to be updated in two ways for FJ-ADE. First, it needs to address the three new keywords. Second, it needs to have both a strict and transitional mode. An FJ-ADE typing judgement is written $\mathit{str}; \Gamma \vdash e : C$. As usual, Γ is a static typing environment, e is an expression, and C is a type (*i.e.*, a class). The *str* flag specifies whether to use strict type checking ($\mathit{str} = \texttt{strict}$) or transitional type checking ($\mathit{str} = \texttt{trans}$).

The typing rules for expressions are shown in Figure 4. Only two rules differ from FJ. First, the T-Invk judgement must specify the two extra parameters of *mtype*. The first argument is always `false`, because methods that are present merely for encouragement are not allowed to be invoked, not even in transitional mode. In transitional mode, encouraged methods might not be implemented yet. In strict mode they must be available, but they are left unavailable so that the strict checker does not admit any programs the transitional checker rejects. The second argument is `true` in transitional mode and `false` otherwise, because deprecated methods can be used only during transitional checking.

The other changed rule is T-New, which now disallows instantiating abstract classes. This rule means that an invariant during evaluation is that all instantiated objects are concrete, thus making it safe for T-Invk to consider abstract

$$\textsc{MB-Conc}\;\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\}\\ B\ m(\bar{B}\ \bar{x})\ \{\ \texttt{return}\ e\ \}\in\bar{M}\end{array}}{\mathit{mbody}(m,C)=\bar{x}.e}$$

$$\textsc{MB-Abs}\;\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\}\\ B\ m(\bar{B}\ \bar{x})\ \texttt{abstract}\in\bar{M}\end{array}}{\mathit{mbody}(m,C)=\texttt{abstract}}$$

$$\textsc{MB-Enc}\;\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\}\\ B\ m(\bar{B}\ \bar{x})\ \texttt{encouraged}\in\bar{M}\end{array}}{\mathit{mbody}(m,C)=\texttt{encouraged}}$$

$$\textsc{MB-Inher}\;\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\}\\ m\notin\bar{M}\qquad \mathit{mbody}(m,D)=\mathit{MB}\end{array}}{\mathit{mbody}(m,C)=\mathit{MB}}$$

$$\frac{\mathit{mbody}(m,C)=\texttt{abstract}}{\mathit{abstract}(C)}$$

Figure 10: Method lookup

$$\frac{\mathit{postneeds}(m,C)}{\mathit{postabs}(C)}$$

$$\textsc{PN-Abs}\;\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\}\\ B\ m(\bar{B}\ \bar{x})\ \texttt{abstract}\in\bar{M}\end{array}}{\mathit{postneeds}(m,C)}$$

$$\textsc{PN-Enc}\;\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\}\\ B\ m(\bar{B}\ \bar{x})\ \texttt{encouraged}\in\bar{M}\end{array}}{\mathit{postneeds}(m,C)}$$

$$\textsc{PN-Deprec}\;\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\}\\ \mathit{postneeds}(m,D)\qquad m\in X\end{array}}{\mathit{postneeds}(m,C)}$$

$$\textsc{PN-Inher}\;\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{K\ \bar{X}\ \bar{M}\}\\ \mathit{postneeds}(m,D)\qquad m\notin M\end{array}}{\mathit{postneeds}(m,C)}$$

Figure 11: Post-abstract classes

$$\frac{\begin{array}{c}\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\}\\ \neg\mathit{mavail}(m,D,\texttt{false},\texttt{false})\end{array}}{\mathit{candep}(m,C)}$$

$$\frac{\mathit{postneeds}(m,C)}{\mathit{candep}(m,C)}$$

Figure 12: Deprecated methods can only override other deprecated methods and potentially abstract methods.

methods as potential callees. In strict mode, T-New also disallows instantiating post-abstract classes. Post-abstract classes are not abstract now, but might become so after forthcoming interface changes.

The rules for typing methods are given in Figure 5. The main change from FJ is that, under strict typing, any method overriding an encouraged method must have the same signature that was encouraged. An additional change is that abstract methods and encouraged methods may only override encouraged methods. In principle abstract methods could be allowed in more places, but the complication provides no insight for the present purposes.

Finally, the rule for typing a class is given in Figure 6. The only difference from FJ is that the list of deprecated methods must be checked. The precise rule is given by the *candep* relation shown in Figure 12. Deprecated methods may not override concrete methods; they may only override deprecated, abstract, and encouraged methods.

It is not useful to have a deprecated method to override a concrete, non-deprecated method. Code can type check against the superclass with no deprecation warning, because the superclass's implementation is not deprecated. At run time, such code might actually invoke the deprecated method. In such a case, removing the deprecated method will mean the program's behavior changes.

If this behavior change is acceptable, and the overriding method does not need to be called, then that method should simply be removed outright. If the change is not acceptable, then either the method should be kept indefinitely, or the superclass's method should be deprecated so that no clients can call it.

That concludes the typing rules. The semantics of FJ-ADE, which are exactly the same as those of FJ, are shown in Figure 13.

5. PROPERTIES

Given the careful definition of FJ-ADE, we can now study some properties that it enjoys. The properties are divided into two parts: typical type-soundness properties, and properties to support statically checked interface evolution.

The proofs contain no surprises, so they and some lemmas are omitted. The full proofs appear in an extended technical report [18].

5.1 Type soundness

There are three type-soundness properties worth dwelling on. The first two show that FJ-ADE is type sound in the usual sense: it enjoys both subject reduction and progress theorems. The last property is that strict checking implies transitional checking.

Theorem 1. *(Subject Reduction). Suppose CT is str-OK. If* $\mathit{str};\Gamma\vdash e:C$ *and* $e\longrightarrow e'$*, then* $\mathit{str};\Gamma\vdash e':C'$ *for some* $C'<:C$.

The proof structure is very close to that for subject reduction for FJ. The main differences are in the supporting lem-

$$\text{R-Field}\ \frac{\mathit{fields}(C) = \bar{C}\ \bar{f}}{(\texttt{new}\ C(\bar{e})).f_i \longrightarrow e_i}$$

$$\text{R-Invk}\ \frac{\mathit{mbody}(m, C) = \bar{x}.e_0}{(\texttt{new}\ C(\bar{e})).m(\bar{d}) \longrightarrow [\bar{d}/\bar{x}, \texttt{new}\ C(\bar{e})/\texttt{this}]e_0}$$

$$\text{R-Cast}\ \frac{C <: D}{(D)(\texttt{new}\ C(\bar{e})) \longrightarrow \texttt{new}\ C(\bar{e})}$$

$$\text{RC-Field}\ \frac{e \longrightarrow e'}{e.f \longrightarrow e'.f}$$

$$\text{RC-Invk-Recv}\ \frac{e \longrightarrow e'}{e.m(\bar{e}) \longrightarrow e'.m(\bar{e})}$$

$$\text{RC-Invk-Arg}\ \frac{e \longrightarrow e'}{e_0.m(\bar{d}, e, \bar{f}) \longrightarrow e_0.m(\bar{d}, e', \bar{f})}$$

$$\text{RC-New-Arg}\ \frac{e \longrightarrow e'}{\texttt{new}\ C(\bar{d}, e, \bar{f}) \longrightarrow \texttt{new}\ C(\bar{d}, e', \bar{f})}$$

$$\text{RC-Cast}\ \frac{e \longrightarrow e'}{(C)e \longrightarrow (C)e'}$$

Figure 13: Evaluation

mas.

The first lemma is that *mtype*'s last two arguments do not affect the type the function calculates, but only whether the function is defined or not. Further, changing one argument or both from `false` to `true` can only cause the function to change from undefined to defined, never from defined to undefined. That is, the truer the third and fourth arguments, the more often *mtype* is defined.

Lemma 1. *(Internal Consistency of mtype). Suppose dep, dep′, enc, and enc′ are four booleans such that enc ⇒ enc′ and dep ⇒ dep′. If* $\mathit{mtype}(C, m, \mathit{enc}, \mathit{dep}) = \bar{C} \to C_0$*, then* $\mathit{mtype}(C, m, \mathit{enc}', \mathit{dep}') = \bar{C} \to C_0$*.*

The following lemma shows that, roughly, once *mtype* returns a result at one point in the class hierarchy, it returns the same result deeper in the hierarchy under that point. Note, though, that this is only true so long as `encouraged` methods are ignored; during transitional checking, methods are allowed to change the type signature when they override a method that is merely `encouraged`.

Lemma 2. *(Subclasses and mtype). Suppose CT is str-OK and that* $\mathit{mtype}(m, D, \texttt{false}, \mathit{dep}) = \bar{C} \to C_0$*. For all* $C <: D$*, also* $\mathit{mtype}(m, C, \texttt{false}, \mathit{dep}) = \bar{C} \to C_0$*.*

Lemma 3. *(Term Substitution Preserves Typing). Suppose CT is str-OK. If* $\mathit{str}; \Gamma, \bar{x} : \bar{B} \vdash e : D$*, and* $\mathit{str}; \Gamma \vdash \bar{d} : \bar{A}$ *where* $\bar{A} <: \bar{B}$*, then* $\mathit{str}; \Gamma \vdash [\bar{d}/\bar{x}]e : C$*, for some* $C <: D$*.*

Lemma 4. *(Weakening). If* $\mathit{str}; \Gamma \vdash e : C$*, then* $\mathit{str}; \Gamma, x : B \vdash e : C$*.*

The next lemma is modified from that for FJ by adding two arguments to the use of *mtype*. The choice of parameters—`false` and $(\mathit{str} = \texttt{trans})$—are those used by T-Invk.

Lemma 5. *Suppose that CT is str-OK,* $\mathit{mbody}(m, C_0) = \bar{x}.e$*, and* $\mathit{mtype}(m, C_0, \texttt{false}, (\mathit{str} = \texttt{trans})) = \bar{D} \to D$*. Then, there is a* D_0 *with* $C_0 <: D_0$*, and a C with* $C <: D$*, such that* $\mathit{str}; \bar{x} : \bar{D}, \texttt{this} : D_0 \vdash e : C$*.*

Theorem 2. *(Progress). Suppose CT is trans-OK, and e is any well-typed expression.*

1. *If e includes* $(\texttt{new}\ C_0(\bar{e})).f$ *as a subexpression, then* $\mathit{fields}(C_0) = \bar{C}\ \bar{f}$ *and* $f \in \bar{f}$ *for some* $\bar{C}$ *and* $\bar{f}$*.*
2. *If e includes* $(\texttt{new}\ C_0(\bar{e})).m(\bar{d})$ *as a subexpression, then* $\mathit{mbody}(m, C_0) = \bar{x}.e_0$ *and* $\#(\bar{x}) = \#(\bar{d})$ *for some* $\bar{x}$ *and* e_0*.*

As with FJ, several theorems follow immediately from Theorem 1 and Theorem 2. FJ-ADE is *type sound*, in that all terminating program executions either compute a value or get stuck at an incorrect cast. Furthermore, cast-free programs do not get stuck and thus always proceed to produce a value if they terminate. Since these theorems follow so directly, the precise definitions and theorem statements are omitted.

Finally, strict type checking abides by its name: strict type checking is strictly more strict than transitional type checking.

Theorem 3. *(Strict Checking). Suppose CT is strict-OK. If* $\texttt{strict}; \Gamma \vdash e : C$*, then* $\texttt{trans}; \Gamma \vdash e : C$*. Further, CT is also trans-OK.*

5.2 Safe transitions

This section shows how to use the strict and transitional modes of FJ-ADE to evolve interfaces safely. There are two properties given which show when it is safe to add a deprecated or encouraged method, thus entering a transition period. Following, there are two theorems showing that, when a program strictly checks, it is safe to remove deprecated methods as well as to upgrade encouraged methods to abstract methods. Finally, there are four theorems showing that when the four described safe changes are made, the resulting programs not only type check but continue to behave identically.

The notation needs extra precision, because these properties all involve two programs. There are two versions of each relation and function, one for each program under discussion. To disambiguate between the two versions when it is not clear from context, the program is used as a subscript. For example, $\mathit{abstract}_{CT}(C_0)$ means that C_0 is abstract in program *CT*, and $\mathit{str}; \Gamma \vdash_{CT'} x : e$ means that *x* type checks in program *CT′* with checking mode *str*.

All of these properties discuss a single program being updated from one version to the next. However, as discussed in Section 2, the properties are carefully written to support updating single *classes* when that class is going to be used in many different programs.

Specifically, the two introduction theorems, require only transitional type checking plus properties of the superclasses of the modified class. Thus, transitional changes can be introduced safely so long as the superclasses of the changed class are immediately available. Further, the requirements on superclasses are weak enough that the superclasses can themselves be modified according to the introduction theorems without invalidating the requirements of the introduction theorems.

The two removal theorems, to contrast, require that all interesting programs be strictly checked before it is safe to perform the removal. This is potentially a lot of work, but the programs do not need to be tested all at once. They can be tested one by one throughout the transition period, as each collaborating development group finds time.

THEOREM 4. *(Deprecation Introduction). Let CT be any class table that is trans-OK, class A be a class in CT, and m be a method of class A. Suppose that if m overrides a method, then that method is either encouraged or deprecated,* i.e. *if A* `extends` *B then $\neg mavail(m, B,$* `false`*,* `false`*). Define CT' as the same class table as CT except that m is deprecated in class A. Given these assumptions, whenever* `trans`$;\Gamma \vdash_{CT} e : C$*, it is also true that* `trans`$;\Gamma \vdash_{CT'} e : C$*. Further, CT' is trans-OK.*

THEOREM 5. *(Encouragement Introduction). Let CT be any class table that is trans-OK, and let A be a class in CT which does not define or inherit a non-encouraged method named m,* i.e. *it is the case that $\neg mavail(m, A,$* `false`*,* `true`*). Define CT' to be the the same class table as CT except that A has the following additional method definition:*

$$B\ m(\bar{B}\ \bar{x})\ \texttt{encouraged}$$

Then, whenever `trans`$;\Gamma \vdash_{CT} e : C$*, it is also true that* `trans`$;\Gamma \vdash_{CT'} e : C$*. Further, CT' is trans-OK.*

THEOREM 6. *(Deprecation Removal). Let CT be any class table that is strict-OK, and let A be a class in CT which defines a method named m that is deprecated. Define CT' to be the same class table as CT except that m is removed from A. Then, whenever* `strict`$;\Gamma \vdash_{CT} e : C$*, it is also true that* `strict`$;\Gamma \vdash_{CT'} e : C$*. Furthermore, CT' is strict-OK.*

THEOREM 7. *(Encouragement Upgrade). Let CT be a class table that is strict-OK, and let A be a class in CT which has the following method definition:*

$$B\ m(\bar{B}\ \bar{x})\ \texttt{encouraged}$$

Define CT' to be the same class table except that the above method definition is replaced by this one:

$$B\ m(\bar{B}\ \bar{x})\ \texttt{abstract}$$

```
abstract class A {
  abstract int foo(int x);
}

class B extends A {
  /**
   * @deprecated
   */
  int foo(int x) {
    return x+1;
  }
}

class Client {
  void run() {
    A a = new B();
  }
}
```

Figure 14: Removing a method can cause a class to become abstract. Instantiating such a class should cause a deprecation warning.

Then, whenever `strict`$;\Gamma \vdash_{CT} e : C$*,* `strict`$;\Gamma \vdash_{CT'} e : C$*. Furthermore, CT' is strict-OK.*

THEOREM 8. *Let CT and CT' be as in Theorem 4. If $e \longrightarrow_{CT} e'$ and* `trans`$;\Gamma \vdash e : C$*, then $e \longrightarrow_{CT'} e'$.*

THEOREM 9. *Let CT and CT' be as in Theorem 5. If $e \longrightarrow_{CT} e'$ and* `trans`$;\Gamma\ \vdash e : C$*, then $e \longrightarrow_{CT'} e'$.*

THEOREM 10. *Let CT and CT' be as in Theorem 6. If $e \longrightarrow_{CT} e'$ and* `strict`$;\Gamma \vdash e : C$*, then $e \longrightarrow_{CT'} e'$.*

THEOREM 11. *Let CT and CT' be as in Theorem 7. If $e \longrightarrow_{CT} e'$ and* `strict`$;\Gamma \vdash e : C$*, then $e \longrightarrow_{CT'} e'$.*

6. WEAKNESSES IN CURRENT TOOLS

Today's practical deprecation checkers do not flag all code that can fail if a deprecated method is removed. Instead, they detect only direct accesses to deprecated features. This section examines four general categories of checks that a full transition checker should include. Along the way, this section examines the level of support of each category in Sun `javac` version 1.5.0_06 and Eclipse 3.2.

Method invocation

In strict mode, the T-INVK rule does not allow a message-send expression to invoke a method that is deprecated. Current checkers capture this familiar rule.

Post-abstract methods

In strict checking mode, the T-NEW rule does not allow instantiating a post-abstract class, *i.e.* a class that might be abstract after a transition phase is left. An example is given in Figure 14. Class `B` is not abstract currently, but it will be come abstract once the deprecated method `foo` is

```
class A {
  void frob() {
     System.out.println("frobbed!");
  }
}

class B extends A {
  int accesses = 0;

  /**
   * @deprecated
   */
  void frob() {
    accesses += 1;
    super.frob();
  }
}

class Client {
  void run() {
    A a = new B();
    a.frob();
  }
}
```

Figure 15: Deprecating a method that overrides a concrete method can result in invariants being broken.

removed. Thus, while B is not abstract currently, it will be after its deprecated method is removed. A proper transition checker should issue a warning for code that instantiates B, because such code will no longer function if the deprecated method is removed. No warning is given, though, by javac or Eclipse.

Deprecated methods and overriding

Not all overrides of abstract, encouraged, and deprecated methods are allowed. Deprecated methods should only override other deprecated methods, abstract methods, and encouraged methods.

An example problem appears in Figure 15. The code in class C type checks by considering method A.foo, but at run time it invokes B.foo. If method B.foo is removed, then the behavior of the program will change and B's invariants might be broken. If this behavior change is truly acceptable, then B.foo should be removed instead of deprecated. Again, javac and Eclipse do not issue a warning for this code.

Encouraged methods and overriding

The requirements on encouraged methods are discussed in Section 3. Anti-deprecation is not supported at all in existing tools.

7. RELATED WORK

There has been substantial work supporting interface evolutions that are refactorings [4, 1, 10, 15]. When such work applies, the benefit can be immense, because the transition period can be shortened or even eliminated. Nonetheless, many desirable interface changes are not refactorings at all. For example, not all uses of deprecated methods can be rewritten to use non-deprecated methods. Sometimes the basic functionality is being removed. For such changes, some kind of transition period is necessary, and checking tools can help entering and leaving those transition periods safely.

There has been work on language features to help manage or eliminate incompatibilities due to interface upgrades. The reuse contracts of Steyaert, *et al.*, allow detection of a variety of upgrade problems when given only the new version of an interface and not the old one [20]. The override keyword of C# and Scala prevents accidental override of newly added methods in a superclass [7, 22, 13, 16]. The present work focuses more on managing the transition periods than on detecting or ameliorating problems after an interface changes.

Interface definitions of various kinds have long supported recording deprecation. Two examples for programming languages are Java's deprecation annotations [3] and Eiffel's obsolete keyword [8]. The present work uses the same concept but adds anti-deprecation.

Dig and Johnson have quantitatively studied the kinds of interface changes that occurred during the lifetime four software systems [6]. The authors start with the developers' change logs and the version control systems for each software system, and then use these data sources to identify the relative frequency of several kinds of API changes. For example, they classify over 80% of the API changes as some kind of refactoring. Software science such as this provides invaluable input for those designing transition mechanisms that are to be useful in practice.

8. FUTURE WORK

The present work is entirely theoretical. It remains future work to try the encouraged annotations and the new checking rules in practice. Two platforms are promising for such a study: Eclipse and Scala Bazaars [17, 19]. Eclipse, as previously discussed, is a very widely used platform with many components developed independently. Scala Bazaars is a code-sharing network for Scala users. Users share Scala code compiled to Java bytecodes, and the compiled libraries all too frequently become incompatible due to seemingly trivial changes in the inter-library interfaces.

The theoretical work is also not complete. First, there are still interface evolutions that are impossible to check with FJ-ADE. For example, the checker does not support changes in constructor signatures nor changes in the classes that are inherited. It remains future work to investigate transition checking rules that are more general.

Additionally, this theory's checker produces a coarse result: either all changes may proceed, or none. Future work will check each change individually instead of having a bulk strict versus trans checking mode. One formalism that looks promising is to replace the checking mode with a *hold* set, where the *hold* set includes the set of changes which may not yet progress. If checking succeeds with a method left out of *hold*, then that one method may be updated even if the others are not ready. Given such a mechanism, new transition periods can begin while old ones are in the middle, without adding an additional obstacle to the old transition period.

Finally, a number of techniques complement evolution checking. Detection remains important: how do developers become aware that they are making an interface change? Interfaces themselves can be more flexible: *e.g.* there could be a construct, analogous to `instanceof`, for dynamically testing whether an object implements an `encouraged` method. Organizational questions arise as well. For example, is it helpful in practice to record a default "interface evolution rate" at the package level, or should every change have its own rate recorded, or should the tools avoid this question entirely?

9. CONCLUSION

Interface evolution is a recurring practical problem. This article investigates one technique, static checking for deprecation and anti-deprecation, which can make interface evolution more graceful. Even these simple method-level evolutions exhibit some subtlety, and the formal study brings out weaknesses in existing tools.

This work is only a beginning, though. Checking tools can potentially check more than method additions and removals. Furthermore, checking tools themselves are just one tool in the toolbox for developers to address interface evolution.

10. ACKNOWLEDGMENTS

Thank you to the anonymous reviewers. Your careful reading and feedback have made this article much more readable and relevant.

11. REFERENCES

[1] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proc. of Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[2] Douglas Bell. *Software Engineering: A Programming Approach*, chapter 6: Modularity. Addison Wesley, 3rd edition, 2005.

[3] Gilad Bracha, James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 3rd edition, 2005.

[4] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *Proc. of International Conference on Software Maintenance (ICSM)*, 1996.

[5] Jim des Rivières. Evolving Java-based APIs. `http://www.eclipse.org/eclipse/development/java-api-evolution.html`.

[6] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proc. of International Conference on Software Maintenance (ICSM)*, September 2005.

[7] ECMA. *ECMA-334: C# Language Specification.* European Association for Standardizing Information and Communication Systems (ECMA), second edition, December 2002.

[8] ECMA. *ECMA-367: Eiffel: Analysis, Design and Programming Language.* European Association for Standardizing Information and Communication Systems (ECMA), 2nd edition, June 2006.

[9] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, not objects. In *Companion to the ACM conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, New York, NY, USA, 2004. ACM Press.

[10] Johannes Henkel and Amer Diwan. Catchup! Capturing and replaying refactorings to support API evolution. In *Proc. of International Conference on Software Engineering (ICSE)*, 2005.

[11] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. of Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999.

[12] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In *Proc. of eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2000.

[13] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, 2004.

[14] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[15] Jeff H. Perkins. Automatically generating refactorings to support API evolution. In *Proc. of Program Analysis for Software Tools and Engineering (PASTE)*, September 2005.

[16] Scala web site. `http://scala.epfl.ch`.

[17] Scala Bazaars web site. `http://www.lexspoon.org/sbaz`.

[18] Alexander Spoon. Anti-deprecation: Towards complete static checking for api evolution (extended version). Technical Report LAMP-REPORT-2006-004, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[19] Alexander Spoon. Package universes: Which components are real candidates? Technical Report LAMP-REPORT-2006-002, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[20] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proc. of Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1996.

[21] Bill Venners. A conversation with Erich Gamma, part III. `http://www.artima.com/lejava/articles/designprinciples.html`, June 2005.

[22] Visual C# web page. `http://msdn.microsoft.com/vcsharp/`.

A Generic Lazy Evaluation Scheme for Exact Geometric Computations

[Extended Abstract]

Sylvain Pion
INRIA, Sophia-Antipolis, FRANCE
Sylvain.Pion@sophia.inria.fr

Andreas Fabri
GeometryFactory, Grasse, FRANCE
andreas.fabri@geometryfactory.com

ABSTRACT

We present a generic C++ design to perform efficient and exact geometric computations using lazy evaluations. Exact geometric computations are critical for the robustness of geometric algorithms. Their efficiency is also critical for most applications, hence the need for delaying the exact computations at run time until they are actually needed. Our approach is generic and extensible in the sense that it is possible to make it a library which users can extend to their own geometric objects or primitives. It involves techniques such as generic functor adaptors, dynamic polymorphism, reference counting for the management of directed acyclic graphs and exception handling for detecting cases where exact computations are needed. It also relies on multiple precision arithmetic as well as interval arithmetic. We apply our approach to the whole geometric kernel of CGAL.

Keywords

computational geometry, exact geometric computation, numerical robustness, interval arithmetic, lazy evaluation, generic programming, C++, CGAL

1. INTRODUCTION

Non-robustness issues due to numerical approximations are well known in geometric computations, especially in the computational geometry literature. The development of the CGAL library, a large collection of geometric algorithms implemented in C++, expressed the need for a generic and efficient treatment of these problems.

Typical solutions to solve these problems involve exact arithmetic computations. However, due to efficiency issues, good implementations make use of arithmetic filtering techniques to benefit from the speed of certified floating-point approximations like interval arithmetic, hence calling the costly multi-precision routines rarely.

One efficient approach is to perform lazy exact computations at the level of geometric objects. It is mentioned in [13] and an implementation is described in [7]. Unfortunately, this implementation does not use the generic programming paradigm, although the approach is general. This is exactly the novelty of this paper.

In this paper, we devise a generic design to provide the most generally applicable methods to a large number of geometric primitives. Our design makes it easy to apply to the complete geometry kernel of CGAL, and is extensible to the user's new geometric objects and geometric primitives.

Our design thus implements lazy evaluation of the exact geometric objects. The computation is delayed until a point where the approximation with interval arithmetic is not precise enough to decide safely comparisons, which may hopefully never be needed.

Section 2 describes in more detail the context and motivation in geometric computing, as well as the basics of a generic geometric kernel parameterized by the arithmetic, and what can be done at this level. Then, Section 3 discusses our design in detail, namely how geometric predicates, constructions and objects are adapted. Section 4 illustrates how our scheme can be applied to the users' own geometric objects and primitives. We then provide in Section 5 some benchmarks that confirm the benefit of our design and implementation. Finally, we list a few open questions related to our design in Section 6, and conclude with ideas for future work.

2. EXACT GEOMETRIC COMPUTATIONS AND THE CGAL KERNEL

2.1 Exact Geometric Computations

Many geometric algorithms such as convex hull computations, Delaunay triangulations, mesh generators, are notoriously prone to robustness issues due to the approximate nature of floating-point computations. This is due to the dual nature of geometric algorithms: on one side numerical data is used, such as coordinates of points, and on the other side discrete structures are built, such as the graph representing a mesh.

The bridges between the numerical data and the Boolean decisions which allow to build a discrete structure, are called the geometric *predicates*. These are functions taking geometric objects such as points as input and returning a Boolean or enumerated value. Internally, these functions typically perform comparisons of numerical values computed from the input. A classical example is the `orientation` predicate of three points in the plane, which returns if the three points are doing a left turn, a right turn, or if they are collinear (see Figure 1). Using Cartesian coordinates for the points, the orientation is the sign (as a three-valued function: -1, 0, 1) of the following 3-dimensional determinant which reduces to a 2-dimensional one:

$$\begin{vmatrix} 1 & 1 & 1 \\ p.x() & q.x() & r.x() \\ p.y() & q.y() & r.y() \end{vmatrix} = \begin{vmatrix} q.x() - p.x() & r.x() - p.x() \\ q.y() - p.y() & r.y() - p.y() \end{vmatrix}$$

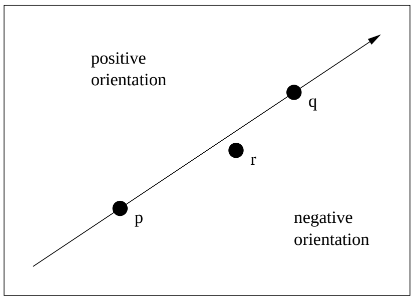


Figure 1: The orientation predicates of 3 points in the plane.

Many predicates are built on top of signs of polynomial expressions over the coordinates of the input points. Evaluating such a function with floating-point arithmetic is going to introduce roundoff errors, which can have for consequence that the sign of the approximate value differs from the sign of the exact value. The impact of wrong signs on the geometric algorithms which call the predicates can be disastrous, as for example it can break some invariants like planarity of a graph, or make the algorithm loop. Didactic examples of consequences can be found in [12] as well as in the computational geometry literature.

Operations building new geometric objects, like the point at the intersection of two lines, the circumcenter of three non-collinear points, or the midpoint of two points, are called geometric *constructions*. We will use the term geometric *primitives* when referring to either predicates or constructions.

In order to tackle these non-robustness issues, many solutions have been proposed. We focus here on the *exact geometric computation paradigm* [16], as it is a general solution. This paradigm states that, in order to ensure the correct execution of the algorithms, it is enough that all decisions based on predicates are taken correctly. Concretely, this means that all comparisons of numerical values need to be performed exactly.

A natural way to perform the exact evaluation of predicates is to evaluate the numerical expressions using exact arithmetic. For example, since most computations are signs of polynomials, it is enough to use multi-precision rational arithmetic which is provided by libraries such as GMP [8]. Note that exact arithmetic is also available for all algebraic computations using libraries such as CORE [10] or LEDA [5], which is useful when doing geometry over curved objects. This solution works well, but it tends to be very slow.

2.2 The Geometry Kernel of CGAL

CGAL [1] is a large collection of computational geometry algorithms. These algorithms are parameterized by the geometry they apply to. The geometry takes the form of a *kernel* [9, 4] regrouping the types of the geometric objects such as points, segments, lines, ... as well as the basic primitives operating on them, in the form of functors. The CGAL kernel provides over 100 predicates and 150 constructions, hence uniformity and genericity is crucial when treating them, from a maintenance point of view.

CGAL provides several models of kernels. The basic families are the template classes `Cartesian` and `Homogeneous` which are parameterized by the type representing the coordinates of the points. They respectively use Cartesian and homogeneous representations of the coordinates, and their implementation looks as follows:

```
template < class NT >
struct Cartesian {
  // Geometric objects
  typedef ...          Point_2;
  typedef ...          Point_3;
  typedef ...          Segment_2;
  ...
  // Functors for predicates
  typedef ...          Compare_x_2;
  typedef ...          Orientation_2;
  ...
  // Functors for constructions
  typedef ...          Construct_midpoint_2;
  typedef ...          Construct_circumcenter_2;
  ...
};
```

These simple template models already allow to use `double` arithmetic or multi-precision rational arithmetic for example. CGAL therefore provides a hierarchy of concepts for the *number types*, which describe the requirements for types to be pluggable into these kernels, such as addition, multiplication, comparisons... The functors are implemented in the following way (here the return type of the predicate is a three-valued enumerated type, moreover some `typename` keywords are removed for clarity):

```
template < class Kernel >
class Orientation_2 {
  typedef Kernel::Point_2     Point;
  typedef Kernel::FT          FT;
public:
  typedef CGAL::Orientation   result_type;

  result_type
  operator()(Point p, Point q, Point r) const
  {
    FT det = (q.x() - p.x()) * (r.y() - p.y())
           - (r.x() - p.x()) * (q.y() - p.y());
    if (det > 0) return POSITIVE;
    if (det < 0) return NEGATIVE;
    return ZERO;
  }
};

template < class Kernel >
class Construct_midpoint_2 {
  typedef Kernel::Point_2     Point;
public:
  typedef Point               result_type;

  result_type
  operator()(Point p, Point q) const
  {
    return Point( (p.x() + q.x()) / 2,
                  (p.y() + q.y()) / 2 );
  }
};
```

As much as conversions between number types are useful, CGAL also provides tools to convert geometric objects between different kernels. We shortly present these here as they will be referred to in the sequel. A kernel converter is a functor whose function operator is overloaded for each object of the source kernel and which returns the corresponding object of the target kernel. Such conversions may depend on the details of representation of the geometric objects, such as homogeneous versus Cartesian representation. CGAL provides such converters parameterized by converters between number types, for example the converter between kernels of the `Cartesian` family:

```
template < class K1, class K2, class NT_conv =
                Default_conv<K1::FT, K2::FT> >
struct Cartesian_converter {
  NT_conv cv;

  K2::Point_2
  operator()(K1::Point_2 p) const
  {
    return K2::Point_2( cv(p.x()), cv(p.y()) );
  }
  ...
};
```

Related to this, CGAL also provides a way to find out the type of a geometric object (say, a 3D segment) in a given kernel, given its type in another kernel and this second kernel. This is in practice the return type of the function operator of the kernel converter described above.

```
template < class O1, class K1, class K2 >
struct Type_mapper {
  typedef ...  type;
};
```

The current implementation works by specializing on all known kernel object types like `K1::Point_2`, `K1::Segment_3`. A more extensible approach could be sought, although this is not the main point of this paper.

2.3 A Generic Lazy Exact Number Type

In order to speed up the exact evaluation of predicates, people have observed that, given that the floating-point evaluation gives the right answer in most cases, it should be enough to add a way to detect the cases where it can change the sign, and rely on the costly multi-precision arithmetic only in those cases. These techniques are usually referred to as arithmetic filtering.

There are many variants of arithmetic filters, but we are going to focus on one which applies nicely in a generic context, and is based on interval arithmetic [3], a well known tool to control roundoff errors in floating-point computations. The idea is that we implement a new number type which forwards its operations to an interval arithmetic type, and also remembers the way it was constructed by storing the history of operations in a directed acyclic graph (DAG) [2]. Figure 2 illustrates the history DAG of the expression $\sqrt{x}+\sqrt{y}-\sqrt{x+y+2\sqrt{xy}}$.

When a comparison is performed on this number type and the intervals overlap, then the DAG is used to recompute the values with an exact multi-precision type, hence giving the exact result. CGAL provides such a lazy number type called `Lazy_exact_nt<NT>` parameterized by the exact type used to perform the exact computations when needed (such as a rational number type). Somehow, this can be seen as a wrapper on top of its template parameter, which delays the computations until they are needed, as hopefully they won't be needed at all.

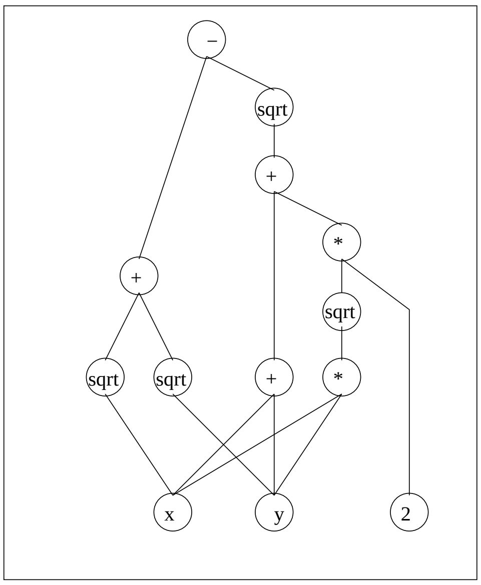


Figure 2: Example Dag: $\sqrt{x}+\sqrt{y}-\sqrt{x+y+2\sqrt{xy}}$.

This solution works very well. It can however be further improved in terms of efficiency. Indeed we note that there are several overheads which can be optimized. First, a node of the DAG is created for each arithmetic operation, so it would be nice to be able to regroup them in order to diminish the number of memory allocations as well as the memory footprint. Second, rounding mode changes for interval arithmetic computations are made for each arithmetic operation, so again, it would be nice to be able to regroup them to optimize away these mode changes.

These remark have lead to a new scheme mentioned in [13], and the description of an implementation has also been proposed in [7]. The idea is to introduce a DAG at the geometric level, by considering geometric primitives for the nodes. The next section describes such an optimized setup. Our design differs from the one in [7] in that we followed the generic programming paradigm and extensive use of templates to make it as easily extensible as possible.

3. DESIGN OF THE LAZY EXACT COMPUTATION FRAMEWORK

The previously described design of lazy computation is based only on genericity over the number type. In this section, we make use of the genericity at the higher level of geometric primitives, in order to provide a more efficient solution. We first describe how to filter the predicates. Then we extend the previous idea of `Lazy_exact_nt` to geometric objects and constructions.

3.1 Filtered Predicates

Performing a filtered predicate means first evaluating the predicate with interval arithmetic. If it fails, the predicate is evaluated again, this time with an exact number type. As all predicates of a CGAL kernel are functors we can use the following adaptor:

```
template <class EP, class AP, class C2E, class C2A>
class Filtered_predicate
{
  typedef AP    Approximate_predicate;
  typedef EP    Exact_predicate;
  typedef C2E   To_exact_converter;
  typedef C2A   To_approximate_converter;

  EP  ep;
  AP  ap;
  C2E c2e;
  C2A c2a;

public:

  typedef EP::result_type  result_type;

  template <class A1, class A2>
  result_type
  operator()(const A1 &a1, const A2 &a2) const
  {
    try {
      Protect_FPU_rounding P(FE_TOINFTY);
      return ap(c2a(a1), c2a(a2));
    } catch (Interval_nt_advanced::unsafe_comparison) {
      Protect_FPU_rounding P(FE_TONEAREST);
      return ep(c2e(a1), c2e(a2));
    }
  }
};
```

Function operators with any arity should be provided. This is currently done by hand up till a fixed arity, and will be replaced when variadic templates become available in C++.

Note that `Protect_FPU_rounding` changes the current rounding mode of the FPU to the one specified as argument to the constructor, and saves the old one in the object. Its destructor restores the saved mode, which happens at the return of the function or when an exception is thrown.

The class `Filtered_kernel` is hence obtained from a kernel `K` by adapting all predicates of `K`. This is currently done with the preprocessor. The geometric objects as well as the constructions remain unchanged.

```
template < class K >
struct Filtered_kernel {

  // The various kernels
  typedef Cartesian<double>            CK;
  typedef Cartesian<Interval_nt>       AK;
  typedef Cartesian<Gmpq>              EK;

  // Kernel converters
  typedef Cartesian_converter<CK, AK>  C2A;
  typedef Cartesian_converter<CK, EK>  C2E;

  // Geometric objects
  typedef CK::Point_2                  Point_2;
  ...
  // Functors for predicates
  typedef Filtered_predicate<AK::Compare_x_2,
                             EK::Compare_x_2,
                             C2E, C2A> Compare_x_2;
  ...
};
```

3.2 Lazy Exact Objects

Performing lazy exact constructions means performing constructions with interval approximations, and storing the sequence of construction steps. When later a predicate applied to these approximations cannot return a result that is guaranteed to be correct, the sequence of construction steps is performed again, this time with an exact arithmetic. Now the predicate can be evaluated correctly.

The sequence of construction steps is stored in a DAG. Each node of the DAG stores (i) an approximation, (ii) the exact version of the function that was used to compute the approximation, (iii) and the lazy objects that were arguments to the function. So the out-degree of a node is the arity of the function.

The example illustrates that lazy objects can be of the same type, without being the result of the same sequence of constructions. a, m, and b are all point-ish. Therefore we have a template handle class, with a pointer to a node of the DAG. In our example, only the latter are of different types.

We will now explain some of the classes in Figure 4 in more detail.

`Lazy_exact` is the handle class. It also does reference counting with a design similar to the one described in [11]. It has `Lazy_exact_nt` as subclass, which provides arithmetic operations. Note that this framework handles arithmetic and geometric objects in a unified way. For example a distance between geometric objects yields a lazy exact number, and

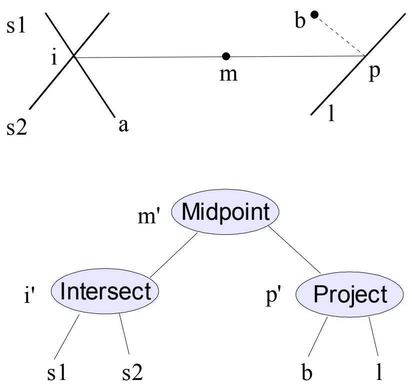


Figure 3: The Dag represents the midpoint of an intersection point and the vertical projection of a point on a line. Testing whether a, m, and b are collinear has a good chance to trigger an exact construction.

a lazy exact number can become the coordinate of a point.

The class `Construction` is an abstract base class. It stores the approximation, and holds a pointer to the exact value. Initially, this pointer is set to `NULL`, and it is the virtual member function `update_exact` which later may compute the exact value and then cache it.

The subclass `Construction_2` is used for binary functions. Similar classes exist for the other arities. These classes store the arguments and the exact version of the function. The arguments may be of arbitrary types. In the case of lazy exact geometric objects or lazy exact numbers the arguments are handles as described before.

```
template <class AC, class EC, class LK, class A1, class A2>
class Construction_2
  : public Construction<AC::result_type, EC::result_type, E2A>
  , private EC
{
  typedef AC              Approximate_construction;
  typedef EC              Exact_construction;
  typedef LK::C2E         To_exact_converter;
  typedef LK::C2A         To_approximate_converter;
  typedef LK::E2A         Exact_to_approximate_converter;
  typedef AC::result_type  AT;
  typedef EC::result_type  ET;

  A1 m_a1;
  A2 m_a2;

  const EC& ec() const { return *this; }

public:

  void
  update_exact()
  {
```

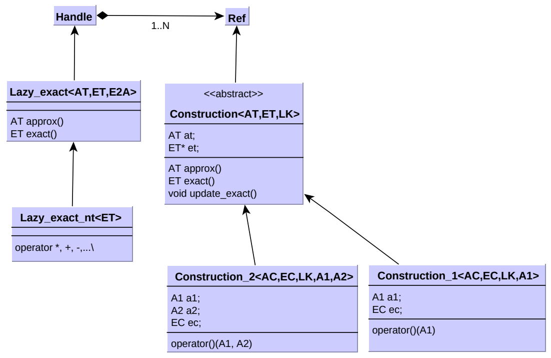


Figure 4: The class hierarchy for the nodes of the Dag.

```
    this->et = new ET(ec()(C2E()(m_a1), C2E()(m_a2)));
    this->at = E2A()(*(this->et));
    // Prune lazy dag
    m_a1 = A1();
    m_a2 = A2();
  }

  Construction_2(const AC& ac, const EC& ec,
         const A1& a1, const A2& a2)
    : Construction<AT,ET,E2A>(ac(C2A()(a1), C2A()(a2)),
                              m_a1(a1), m_a2(a2)
  {}
};
```

The constructor stores the two arguments. It then takes their approximations and calls the approximate version of the functor.

In case the exact version of the construction is needed, this gets computed in the `update_exact` method. It fetches the exact versions of the arguments, which in turn may trigger their exact computation if they are not already computed and cached. From the exact lazy object one computes again the approximate object, as the object computed with the approximate version of the functor has a good chance to have accumulated more numerical error.

Finally, the Dag is pruned. As the nodes of the Dag are reference counted, some of them may get deallocated by the pruning. Most often `A1` and `A2` will be lazy exact objects. For performance reasons their default constructors generates a handle to a shared static node of the Dag.

Also, we use private derivation of the exact construction `EC`, instead of storing it as data member, in order to benefit from the empty base class optimization.

The other derived classes store the leaves of the Dag. There is a general purpose leaf class, and more specialized ones, for example for creating a lazy exact number from an `int`. They are there for performance reasons.

3.3 The Functor Adaptor

So far we have only explained how lazy constructions are stored, but not how new nodes of the Dag are generated.

The following functor adaptor is applied to all the constructions we want to make lazy. It has function operators for other arities.

```
template <class LK, class AC, class EC>
class Lazy_construct
{
  typedef LK                  Lazy_kernel;
  typedef AC                  Approximate_construction;
  typedef EC                  Exact_construction;
  typedef LK::AK              AK;
  typedef LK::EK              EK;
  typedef EK::FT              EFT;
  typedef LK::E2A             E2A;
  typedef LK::C2E             C2E;
  typedef AC::result_type     AT;
  typedef EC::result_type     ET;
  typedef Lazy_exact<AT, ET, E2A> Handle;

  AC ac;
  EC ec;

public:

  typedef Type_mapper<AT,AK,LK>::type result_type;

  template <class A1, class A2>
  result_type
  operator()(const A1& a1, const A2& a2) const
  {
    try {
      Protect_FPU_rounding P(FE_TOINFTY);
      return Handle(new Construction_2<AC, EC, LK, A1, A2>
                                      (ac, ec, a1, a2));
    } catch (Interval_nt_advanced::unsafe_comparison) {
      Protect_FPU_rounding P(FE_TONEAREST);
      return Handle(new Construction_0<AT,ET,LK>
                               (ec(C2E()(a1), C2E()(a2))));
    }
  }
};
```

The functor first tries to construct a new node of the DAG. If inside the approximate version of the construction an exception is thrown, we perform the exact version of the construction, and only create a leaf node for the DAG.

3.4 Special-Case Handling

The generic functor adaptor works out of the box for all functors that return lazy exact geometric objects or a lazy exact number.

Functors returning objects which are not made lazy are an easy to handle exception. An example in CGAL is the functor that computes a bounding box with `double` coordinates around geometric objects, whose width is not required to be tight. As the intervals corresponding to the coordinates of the approximate geometric object are already 1-dimensional bounding boxes, we never have to resort to the exact geometric object. The functor adaptor is trivial.

Some functors of CGAL kernels return a polymorphic object. For example, the intersection of two segments may be empty, or a point, or a segment. In order not to have a base class for all geometric classes, CGAL offers a class `Object`[1] which is capable of storing typed objects. The problem we have to solve is that the lazy exact functor must not return a lazy exact `Object`, but instead must return an `Object` holding a lazy geometric object. This is solved by looping over all CGAL kernel types, to try to cast, and if it works to construct the lazy geometric object and put it in an `Object` again.

[1]The `Object` class is comparable to `boost::any`.

Less trivial cases are functors which pass results of a computation back to reference parameters, or which write into output iterators. They need a special functor as well as special `Construction` classes. It is not hard to write them, but the problem is that they must be dispatched by hand, as we have no means of introspection. One solution would be to introduce functor categories.

3.5 The Lazy Exact Kernel

We are ready to put all pieces together, by defining a new kernel which has an approximate and an exact kernel as template parameters.

```
template < class AK, class EK >
struct Lazy_kernel {

  // Kernel converters
  typedef Lazy_kernel<AK, EK>          LK;
  typedef Approx_converter<LK, AK>     C2A;
  typedef Exact_converter<LK, EK>      C2E;
  typedef Cartesian_converter<EK, AK>  E2A;

  // Geometric objects
  typedef Lazy_exact<AK::Point_2,   EK::Point_2>   Point_2;
  typedef Lazy_exact<AK::Segment_2, EK::Segment_2> Segment_2;

  // Functors for predicates
  typedef Filtered_predicate<EK::Compare_x_2, AK::Compare_x_2,
                             C2E, C2A> Compare_x_2;
  ...

  // Functors for constructions
  typedef Lazy_construct<LK, AK::Construct_midpoint_2,
                         EK::Construct_midpoint_2>
          Construct_midpoint_2;
  ...

  typedef Lazy_Construct_returning_object<LK, AK::Intersection_2,
                                          EK::Intersection_2>
          Intersection_2;
};
```

In the current implementation we use the preprocessor to generate the typedefs from a list of types, and we use the Boost MPL library for dispatching the special cases. `Approx_converter` simply fetches the stored approximate object. Similarly `Exact_converter` fetches the exact approximate object, possibly triggering its computation.

4. EXTENSIBILITY

We have to distinguish between different levels of extensibility.

When CGAL kernels get extended by geometric objects and constructions this needs changes in the lazy construction framework if the new constructions have "new" interfaces, e.g., two output iterators, followed by two reference parameters to return a result. This would need a new node type for the DAG, a new functor, and hard wired dispatching in the lazy kernel. Otherwise there is nothing to do.

When the CGAL user wants to extend the lazy kernel with his own geometric objects and constructions, he first has to add them to the kernel that then gets into the lazy computation machinery, as described in [9]. Then, what we stated in the previous paragraph applies.

The `Curved_kernel` and the `Lazy_curved_kernel` of CGAL which provide primitives on circles and circular arcs [14, 6], are examples for both.

5. BENCHMARKS

We now run a simple benchmark that illustrates the benefit of our techniques. We compare the running time and memory consumption of various kernel choices with the following algorithm:

- generate 2000 pairs of 2D points with random coordinates (using `drand48()`).
- construct 2000 segments out of these points.
- intersect all pairs of segments among these, and store the resulting intersection points.
- shuffle the resulting points
- iterate over consecutive triplets of these points, and compute the `orientation` predicate of these.

Figure 5 provides the resulting data for a choice of four different kernels:

- `SC<Gmpq>` stands for the simple Cartesian representation kernel parameterized with `Gmpq`, which is a C++ wrapper around the multi-precision rational number type provided by GMP,
- `SC<Lazy_exact_nt<Gmpq>>` uses the lazy exact evaluation mechanism at the arithmetic level,
- `Lazy_kernel<SC<Gmpq>>` is our approach for performing lazy exact evaluations at the geometric object level,
- `Lazy_kernel<SC<Gmpq>> (2)` is similar to the previous one, but it does not include the additional optimization which consists in eliminating rounding mode changes, which is allowed by the consecutive interval computations,
- finally, `SC<double>` is the simple Cartesian representation kernel parameterized with `double`. It is given for reference as it is not robust in all cases. It shows what the optimal performance could be.

Benchmarks have been performed using the GNU `g++` compiler versions 3.4 and 4.1 with the `-O2` optimization option. The machine was a Pentium-M laptop at 1.7 GHz, equipped with 1 GB of RAM and 1 MB of cache, running the Fedora Core 3 Linux distribution. The memory consumption is the same for these two compiler versions, however timings differ significantly. Timings are given in seconds and memory in megabytes.

The results show that our approach wins almost a factor of 10 on memory over the basic lazy evaluation scheme. It is also between 2 and 3 times faster. However, it remains 4 times slower than the approximate floating-point evaluation, but of course it is guaranteed for all cases.

Kernel	time g++ 3.4	time g++ 4.1	mem
`SC<Gmpq>`	71	70	70
`SC<Lazy_exact_nt<Gmpq>>`	9.4	7.4	501
`Lazy_kernel<SC<Gmpq>>` (2)	4.9	3.6	64
`Lazy_kernel<SC<Gmpq>>`	4.1	2.8	64
`SC<double>`	0.98	0.72	8.3

Figure 5: Benchmarks comparing different kernels.

The benchmark also illustrates the gain obtained thanks to the elimination of rounding mode changes, which is now allowed by the regrouping of operations on intervals.

Another data point illustrating the improvements is that we measured the number of DAG nodes allocated. For `SC<Lazy_exact_nt<Gmpq>>`, 29 million nodes were allocated, while for `Lazy_kernel<SC<Gmpq>>` only 2.5 million nodes were needed. So we have won a factor of more than 10, due to the regrouping allowed by our design.

Note that the algorithm we chose uses random data, hence it does not produce many filter failures, so almost no exact evaluation is performed. Another thing worth noticing is that it uses relatively simple 2D primitives. More complex primitives, especially in higher dimensions, should show more benefits to the method. Finally, real-world geometric applications tend to produce more combinatorial output, hence the relative runtime cost of primitives is smaller, so the slow down factor is lower in those cases. First such experiments on a 3D surface reconstruction algorithm have shown a factor of 6 improvement on memory consumption and a speed up factor of 3.

6. OPEN DESIGN QUESTIONS

Here is a list of open questions related to our framework.

The first question concerns the regrouping of expressions. Our framework asks the user to pass it functors specifying the level at which the regrouping of expressions is made. In CGAL this is not a problem since the primary interface of the kernel towards the geometric algorithms is a list of functors. However it has the drawback of not being automatic. We can think of approaches based on expression templates [15] which would automatically detect sequences of operations and regroup them. Unfortunately, expression templates are limited to single statement expressions and they tend to slow down compilation times considerably. Could there be a way to extend the automatic regrouping to more than single statements? Maybe the `auto` keyword recently proposed for addition to the C++ language will allow to propagate this through several statements? Or maybe the Axiom feature part of the proposal for concepts in C++ could be used to specify this kind of transformation.

Another question is if similarly delayed computations are used in other areas, and if yes, then is it possible to find out a common design, more general than the one we propose.

7. CONCLUSION AND FUTURE WORK

We have presented in this paper a generic framework which implements lazy exact geometric computations, motivated by the needs for robustness and efficiency of geometric algorithms. This framework allows to delay the costly exact evaluation using multi-precision arithmetic when the faster interval arithmetic suffices.

The proposed design is easily extensible to new geometric primitives – predicates and constructions –, as well as new geometric objects. It is based on a template family for representing lazy objects, as well as generic functor adaptors which produce them.

Future work in this area will consist of various added special-case optimizations as well as generalizations. It is for example possible to refine the filtering scheme by growing the precision little by little instead of switching directly to full multi-precision computation in case of insufficiency of precision of the intervals. We also would like to study possibilities of merging the `Filtered_predicate` and `Lazy_construct` functor adaptors. Possible optimizations for specific cases also can be done, using faster schemes than interval arithmetic (so-called static filters). Moreover, the current way of providing a full kernel is by a list of types for the objects and functors, which is provided through the use of the preprocessor, we will therefore try to provide a better design on this particular point.

Finally, we plan to make our implementation part of a future release of CGAL, whose entire geometry kernel already benefits from it.

8. ACKNOWLEDGMENTS

The work reported in this paper has been supported in part by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

9. REFERENCES

[1] *CGAL User and Reference Manual*, 3.2 edition, 2006.

[2] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy solution to imprecision in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 73–78, 1993.

[3] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

[4] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Schirra, and S. Pion. 2D and 3D kernel. In C. E. Board, editor, *CGAL User and Reference Manual.* 3.2 edition, 2006.

[5] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, Jan. 1996.

[6] I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards an open curved kernel. In *Proc. 20th Annu. ACM Sympos. Comput. Geom.*, pages 438–446, 2004.

[7] S. Funke and K. Mehlhorn. Look – a lazy object-oriented kernel for geometric computation. *Computational Geometry - Theory and Applications (CGTA)*, 22:99–118, 2002.

[8] T. Granlund. GMP, the GNU multiple precision arithmetic library. `http://www.swox.com/gmp/`.

[9] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes Comput. Sci.*, pages 79–90. Springer-Verlag, 2001.

[10] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. *The CORE Library Project*, 1.2 edition, 1999. http://www.cs.nyu.edu/exact/core/.

[11] L. Kettner. Reference counting in library design – optionally and with union-find optimization. In *Workshop on Library Centric Software Design*, october 2005.

[12] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes Comput. Sci.*, pages 702–713. Springer-Verlag, 2004.

[13] S. Pion. *De la géométrie algorithmique au calcul géométrique.* Thèse de doctorat en sciences, Université de Nice-Sophia Antipolis, France, 1999. TU-0619.

[14] S. Pion and M. Teillaud. 2D circular kernel. In C. E. Board, editor, *CGAL User and Reference Manual.* 3.2 edition, 2006.

[15] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[16] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

APPENDIX

A. BENCHMARK CODE

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Lazy_kernel.h>
#include <CGAL/Gmpq.h>
#include <CGAL/Lazy_exact_nt.h>
#include <CGAL/intersections.h>
#include <CGAL/Timer.h>
#include <CGAL/Memory_sizer.h>
using namespace CGAL;

// Choosing a kernel:
//typedef Simple_cartesian<Gmpq>                 K;
//typedef Simple_cartesian<Lazy_exact_nt<Gmpq> > K;
//typedef Lazy_kernel<Simple_cartesian<Gmpq> >   K;
typedef Simple_cartesian<double>               K;

typedef K::Point_2    Point;
typedef K::Segment_2  Segment;

Point   random_point()   { return Point(drand48(), drand48()); }
Segment random_segment() { return Segment(random_point(), random_point()); }

int main() {
  int loops = 2000, init_mem = Memory_sizer().virtual_size();
  Timer t; t.start();

  std::cout << "Generating initial random segments: " << loops << std::endl;
  std::vector<Segment> segments;
  for (int i = 0; i < loops; ++i)
    segments.push_back(random_segment());

  std::cout << "Counting intersections [brute force algorithm]: " << std::flush;
  std::vector<Point> points;
  for (int i = 0; i < loops-1; ++i)
    for (int j = i+1; j < loops; ++j) {
      Object obj = intersection(segments[i], segments[j]);
      if (const Point* pt = object_cast<Point>(&obj))
        points.push_back(*pt);
    }
  std::cout << points.size() << std::endl;

  // we shuffle the points, as consecutive points have good chance to come
  // from the same segments, hence filter failures in orientation() later...
  std::random_shuffle(points.begin(), points.end());

  std::cout << "Performing orientation tests" << std::endl;
  int negative_ort = 0, positive_ort = 0, collinear_ort = 0;
  for (int i=0; i < points.size()-2; ++i) {
    Orientation o = orientation(points[i], points[i+1], points[i+2]);
    if (o < 0)      ++negative_ort;
    else if (o > 0) ++positive_ort;
    else            ++collinear_ort;
  }
  std::cout << "orientation results : (-) = " << negative_ort
                              << "    (+) = " << positive_ort
                              << "    (0) = " << collinear_ort << std::endl;

  t.stop();
  std::cout << "Total time   = " << t.time() << std::endl;
  std::cout << "Total memory = " << ((Memory_sizer().virtual_size() - init_mem) >>10)
                                 << " KB" << std::endl;
}
```

A Generic Topology Library

René Heinzl
Christian Doppler Laboratory
Gusshausstrasse 27-29
Vienna, Austria
heinzl@iue.tuwien.ac.at

Michael Spevak
Institute for Microelectronics
Gusshausstrasse 27-29
Vienna, Austria
spevak@iue.tuwien.ac.at

Philipp Schwaha
Christian Doppler Laboratory
Gusshausstrasse 27-29
Vienna, Austria
schwaha@iue.tuwien.ac.at

ABSTRACT

We present a generic topology library that is based on topological space and combinatorial properties. A notation is introduced whereby data structures can be described by their topological cell dimensions and internal combinatorial properties. A common interface for different types of data structures is presented. Various issues of iteration of these data structures can be explained from the topological properties. Using this multi-dimensional topology library we introduce new possibilities for functional programming in the field of scientific computing.

1. INTRODUCTION

In this work we investigate internal topological and combinatorial properties of data structures and the effect on their interfaces. Generic interfaces to data structures have proven to be highly successful means of generic programming. With the great achievement of accessing all data structures in a minimal but concise way, generic programming has emerged. A detailed analysis of generic programming is given in [1], where this topic is introduced from a theoretical point of view, namely category theory. A lot of insight is gained through this approach and a solid base has been achieved with this theory. Our work deals with the basic nature of topological spaces related only to data structures and is based on GrAL [2]. This is not as general as the category theory approach, but the basic features and issues are exposed.

Usually programmers have to know the **specific properties** of data structures to achieve the best performance of an algorithm. A simple example is the iteration and data access within a `std::vector`, which is constant, whereas the insertion or deletion uses linear time. This is relevant for the actual run-time behavior of all implemented algorithm applied to it. Closely related to this issue is the fact that the C++ STL algorithms use the most basic iteration mechanism for the access to data structures, the `forward` iterator mechanism. The optimal way of iteration of containers can often not be achieved, because linear iteration is simply not optimal [1], such as traversing a `std::map` or higher-dimensional topological structures, e.g., `boost::graph` from the Boost Graph library [3]. We introduce (Section 4.1) a unified data structure definition, where only the dimension and the combinatorial properties of topological spaces are specified. This can also be accomplished automatically at compile-time, based on requirements of algorithms.

Modern application design requires the utilization of data structures in several dimensions. Especially the field of scientific computing uses different topological elements to discretize partial differential equations (PDE). Various approaches are available such as the STL containers, the BGL, and for grids the GrAL [2]. However, a **standardized interface** to these data structures is missing. We introduce a basic interface (Section 4.2.3) for different dimensions of data structures based on topological and combinatorial properties.

A major issue of generic programming is the treatment of **data structure iteration and data access** [4], but the upcoming C++0x standard does not yet include this insight [5]. Therefore we use the property map concept [4], which is presented in Section 5 to utilize an extra data space. Briefly, the combination of iteration and access leads to a miscategorized algorithm specialization.

Our search for a general data structure library for the needs of scientific computing has shown that the topological structures of different STL containers and BGL mechanisms can be abstracted and generalized to a multi-dimensional generic topology library (GTL). We do not only separate the data access and iteration [4], but also provide a formal description of the underlying topological space with emphasis on the combinatorial properties:

$$\boxed{\text{topological space} + \text{data type} = \text{data structure}}$$

With a formalization of the topological properties and the iteration mechanism this approach renders a new possibility of the functional programming paradigm (Section 7) which is emerging in C++ [6, 7]. Up to now, functional expressions lack the support of a unique interface for all different kinds of data structure iteration. As we present in a generic discretization library for the discretization of various partial differential equations (GDL [8]), the full power of functional programming is revealed with consistent topological data structure. Note, the GTL is not restricted to applications for scientific computing, simple iterations can be specified elegantly as well.

2. MOTIVATION

Our motivation for developing generic libraries is derived from the need in high performance applications in the field of scientific computing, especially in Technology Computer

Aided Design (TCAD). Briefly, TCAD deals with the assembly of large equation systems by utilizing discretized partial differential equations from different fields of physics. All types of PDEs (elliptic, parabolic, hyperbolic) have to be considered for the various types of problems from the fields of semiconductor simulation [9]. Different grid types and dimensions of topological elements, linear and nonlinear solvers with their associated numerical issues have to be considered during application development and demand great care to ensure high software quality while also addressing performance issues.

Our institute has a long history in developing such applications [10, 11, 12, 13, 14]. In early years only one- and two-dimensional data structures were used, due to the limitations of computer resources. The imperative programming paradigm was sufficient for this type of task [11]. With the improvement of computer hardware and the advent of the object-oriented programming paradigm, the shift to more complex data structures was possible. More complexity is added when modeling requires a change of the underlying topological data structure, usually from regular to irregular grids. Additional complexity is introduced by changes in the solver mechanisms or through the use of different types of data, e.g., vectorial or tensorial data [15]. The most drastic changes usually result from a change in the discretization scheme, or the mathematical problem formulation itself that is derived from PDEs [9, 16].

The main motivation for the GTL was the circumstance that a detailed analysis of the tools developed at our institute has shown the following distribution between the amount of source code for data structures and algorithms:

Name	Year	DS	Algorithm	Reference
`MINIMOS`	1980	60 %	40 %	[9]
`S*AP`	1989	60 %	40 %	[12]
`MINIMOS-NT`	1996	70 %	30 %	[13]
`ELSA`	1999	70 %	30 %	[17]
`WSS`	2000	90 %	10 %	[14]

Most of these applications use data structures such as `list` and `array` as well as triangles, quadrilaterals, tetrahedra, cuboids, each with their own different access and storage mechanisms, and iteration operations. Although these tools use the C++ STL to some extent, the overall application design is not based on generic libraries. For this reason, the number of source lines is growing quickly due to the complex requirements of two and three-dimensional problems. The currently used applications exceed the limit of maintainability greatly.

This was the start for our own analysis related to data structures and different programming paradigms in TCAD. Our analysis then revealed that, up to now, none of the investigated libraries (BGL, GrAL) can be used directly. For lower-dimensional applications (0D, 1D) the libraries suffer from higher-dimensional information, such as incidence or adjacence. Applications, based on libraries, which use different types of grids (triangles, tetrahedra, cubes) were always outperformed by manually tuned applications. However, for the field of scientific computing, it is essential to abstract from the iteration mechanism, dimensionality, and type of the underlying cell complex.

3. FORMAL SPECIFICATION

This section introduces the basic notation of topological spaces and cell complexes in our approach. In Figure 1 we present an overview of the terms used.

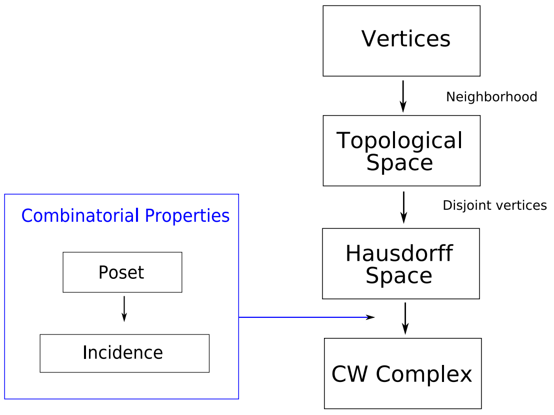


Figure 1: Basic mathematical formalism.

Of particular interest are the combinatorial properties of a CW-complex to characterize different data structures of arbitrary dimensions. Hence, we introduce the formal specification of a CW-complex [18] first. A complete introduction of all terms is available in [2, 18].

Definition: ***CW-Complex*** $\mathcal{C}$***, [18]***
A pair $(\mathcal{T}, \mathcal{E})$, with $\mathcal{T}$ a Hausdorff space and a decomposition $\mathcal{E}$ into cells is called a CW-Complex, iff the following axioms are satisfied:

- mapping function: for each n-cell $c \in \mathcal{E}$ a continuous function $\Phi_e : D^n \rightarrow \mathcal{T}$ exists, which transforms D^n homeomorphically onto a cell c and S^{n-1} in the union of maximal $(n{-}1)$ dimensional cells. D^n represents an n-dimensional ball and S^{n-1} represents the $n-1$ cell complex.
- finite hull: the closed hull(c) of each cell $c \in \mathcal{E}$ connects only with a finite number of other cells.
- weak topology: $A \subset \mathcal{T}$ is open, iff each $A \cap \text{hull}(c)$ is open.

An n-cell describes the cell with the highest dimension:

- zero-dimensional (0D) cell complex: vertex
- one-dimensional (1D) cell complex: edge
- two-dimensional (2D) cell complex: triangle

For this work, the most important property of a CW-complex can be explained by the usage of different n-cells and the consistent way of attaching sub-dimensional cells to the n-cells. This fact is covered by the mapping function. From now, we use an abbreviation to specify the CW-complex with its dimensionality, e.g., a 1-cell complex describes a one-dimensional CW-complex. An illustration of this type of cell complex is given in Figure 2.

In the regime of data structures the requirements of a CW-complex, the finite hull and weak topology, are always satisfied due to the finite structure. The underlying topology of a CW-complex used in computer data structures is always generated from the power set $\mathcal{P}(X)$. For this reason, the topological space cannot be used directly to characterize the different data structural properties. An example is the topological space of a random access container specified by

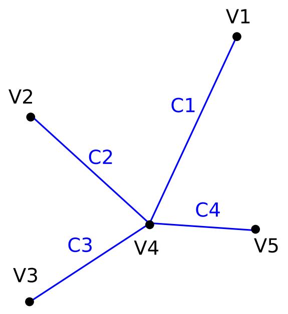


Figure 2: Representation of a 1-cell complex with cells (edges, C) and vertices (V).

the following code line:

```
std::vector<int> container(3);
```

The topological space $\mathcal{T}$ is described by the power set which models the arbitrary access of this container.

$$\mathcal{T} = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\}$$

For this reason we introduce the concept of a topological neighborhood [18].

Definition: ***Neighborhood***
A subset $A \subseteq X$ of a topological space $\mathcal{T}$ is a neighborhood of an element $p \in X$, iff it contains an element O of $\mathcal{T}$ that contains p.

$$A \subseteq X \text{ neighborhood of p} \iff \exists \mathcal{O} \in \mathcal{T} : \text{p} \in \mathcal{O}, \mathcal{O} \subseteq \mathcal{A}$$

A *base of neighborhoods* at $p \in X$ is a set of neighborhoods of p such that every neighborhood of p contains one of the base neighborhoods. We introduce the notion of *bn* which describes the number of elements of the base of neighborhoods. Different data structures can be uniquely characterized by this number. To illustrate this term we present the following list data structure:

```
std::list<int> container(4);
```

$\mathcal{T}$ is also described by the power set but the base of neighborhood can be used to characterize the list. The following sub-set of the topology represents the base of neighborhood of the list:

$$\mathcal{T}_i = \{\{0,1\}, \{1,2\}, \{2,3\}, \{3,4\}\}$$

Next, we introduce the combinatorial properties of a cell complex. These properties are responsible for the internal layout of data structures, as well as for the iteration mechanisms of these data structures.

With the assumption of cell complexes and the base of neighborhoods we introduce the following term:

Definition: ***Adjacence and Incidence***
Given two sets $a, b \in \mathcal{T}$, we define a binary adjacence relation $\mathcal{R}_{\text{adj}}(a,b)$ with the following properties:

$$\mathcal{R}_{\text{adj}}(a,b) :\iff a \cap b \neq \emptyset$$

As a special case of adjacence we define the incidence relation $\mathcal{R}_{\text{in}}(a,b)$:

$$\mathcal{R}_{\text{in}}(a,b) :\iff a \cap b = a \vee b$$

The incidence relation gives the possibility of an iteration of a topological spaces, using only the definition of a base of neighborhoods which separates the combinatorial properties of our underlying topological spaces.

To define higher-dimensional cell complexes, a mechanism is introduced which handles the internal structures of cells. The topological space of, e.g., a triangular grid is described by the vertex on cell information. The number of elements of a sub-set does not give any information about the internal structure of this element. The sub-set $\mathcal{T}_j = \{1,2,3,4\}$ can describe a tetrahedron in three dimensions or a quadrilateral in two dimensions. In order to be able to distinguish these different element, we introduce the concept of a poset:

Definition: ***Poset, [19]***
A poset $(\mathcal{S},<)$ is a finite set $\mathcal{S}$, together with a partial order relation.

In the case of a cell complex, the partial order relation is described by incidence. A Hasse diagram can be used to visualize the poset of a cell. Any two elements are connected by a line, if they are comparable.

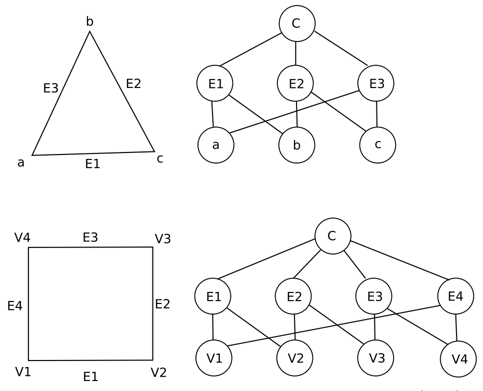


Figure 3: A Hasse-diagram for a triangle cell (top) and a quadrilateral cell (bottom).

With the Hausdorff property of the CW-complex we can uniquely characterize cells or faces by their set of vertices. We define $\{a,b,c\}$ as the element which exactly contains the vertices a,b,c.

Another important property is the locality of the cell complex. Two different properties can be distinguished, which represent the arbitrary and the iterative access of data structures.

Definition: ***Global Cell Complex***
A cell complex $\mathcal{C}$ which is homeomorphic to the following combinatorial structure of cells [2], where i_d represents the dimensional ticks:

$$\{[i_1, i_1+1] \times .. \times [i_d, i_d+1] \mid 0 \leq i_j \leq m_j\}$$

is called a cell complex with *global* properties. Here the topological incidence relation is apparent from the fact that global information is explicitly available. This property is important because of the fact, that a global cell complex describes the *random access* container types.

Definition: *Local Cell Complex*
Conversely, a cell complex which cannot be described globally is called a local cell complex.

In scientific computing, neighborhood information of a local cell complex has to be stored explicitly. Due to the non-trivial construction of instances of cell complex types, we refer to literature [20]. Related to data structures, a local cell complex models different types of lists, trees, or maps.

4. GENERIC TOPOLOGY LIBRARY

In this section we introduce the basic idea of the underlying cell complex for data structures. The classification of each data structure is using the dimension of the cells. Figure 4 shows a 0-cell complex. In this special case, cells and vertices are identical. No neighborhood information is given, only the cells are depicted. This topological structure covers most of the STL data structures. The differences between each of the data structures such as `std::vector` and `std::list` can be found in the *base of neighborhoods* and the *incidence relation* or, in other words, in their combinatorial properties.

The internal mechanism and utilization of the internal structure of the data structure is not possible due to the 0-cell complex, which means that no higher incidence or adjacence (see Section 3) is available directly. No data can be stored on edges or cells easily.

Generic algorithms cannot always use the internal structure of, e.g., `std::map` or `boost::graph` without modification. Copying a map or graph could be much more efficient, if the algorithms were aware of different internal topologies of data structure, such as the tree structure of a map.

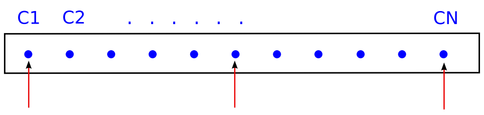


Figure 4: Iteration over cells within a 0-cell complex.

As already mentioned, applications designed in the field of scientific computing need higher-dimensional data structures as well as higher-dimensional iteration operations. Consider, for example, a 1-cell complex (Figure 5) and a 2-cell complex (Figure 6).

In the case of the 1-cell complex, the basic concept of *incidence* is mostly covered. There are only edges and vertices, and most of the operations on these two elements can be implemented with basic methods. For higher-dimensional cell complexes, e.g., a 2-cell complex, the incidence relation becomes more complex. There are various permutations of incidence relations which all lead to a different iteration. All vertices connected to a triangle, or all edges which are part of the triangle can be traversed. Also adjacent iteration can be derived easily.

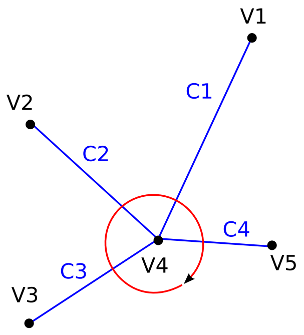


Figure 5: Iteration over edges for a 1-cell complex.

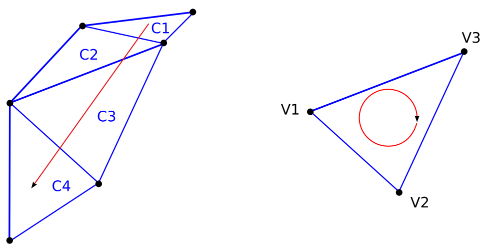


Figure 6: Iteration over cells and incident vertices of a cell for a 2-cell complex.

4.1 Topological Properties of Data Structures

We can now show, based on the formal definitions in Section 3, that we can derive a consistent categorization of different data structures and therewith a homogeneous interface which does not restrict the dimensionality or iteration mechanism of the data structures. In the following table we characterize common data structures with their combinatorial properties. The used terms are:

- dim: dimension of the cell complex
- locality: refers to the local or global combinatorial properties of the underlying space
- bn: represents the number of the elements of the base of neighborhoods of the cell

SLL stands for single-linked-list whereas DLL means double-linked-list. A global defined cell complex does not require a base of neighborhood due to the fact, that the neighborhood is implicitly available.

data structure	dim	locality	bn
array/vector	0	global	
SLL/stream	0	local	2
DLL/binary tree	0	local	3
arbitrary tree	0	local	4
graph	1	global	
regular grid	2	global	
irregular rid	2	local	4
regular grid	3	global	
irregular grid	3	local	5

The next code snippet presents our topological data structure definition. The first number stands for the actual dimension, the tags *global* and *local* stand for the combinatorial property, and the final number specifies the number of elements of the base of neighborhoods.

```
topology<0, global  >  topo;  // array
topology<0, local, 2>  topo;  // SLL/stream
topology<2, global  >  topo;  // regular grid
topology<2, local, 4>  topo;  // irregular grid
```

For a 0-cell complex the STL iterator traits can be used to classify the data structure easily:

```
topology<0, random_acess>   topo;  // global
topology<0, forward>        topo;  // local, 2
topology<0, bidirectional>  topo;  // local, 3
```

Based on this formulation, an automatic mechanism is possible to derive optimal data structures based on the requirements of algorithms.

To show the implementation with the GTL and equivalence of the data structure compared to the STL vector a simple code snippet is presented:

Equivalence of data structures

```
typedef topology<0, random_access> topo_t;
typedef long                       data_t;

typedef cell_t<topo_t, data_t>     container_t;
container_t   container;

// is equivalent to

std::vector<data_t>       container;
```

Here, the separation of the topological structure specification can be clearly observed.

4.2 Finite Cell Complexes

This section deals with the analysis of the data structures from the STL and BGL and generalize these expressions to arbitrary-dimensional data structures. We show that all different data structures model a common interface and each dimension can use specializations to obtain the best performance.

4.2.1 The 0-Cell Complex

A typical representative of a 0-cell complex is the topological structure of a simple array. The C++ STL containers such as `vector` and `list` are representatives and are schematically depicted in Figure 7. The points represent the cells on which data values are stored.

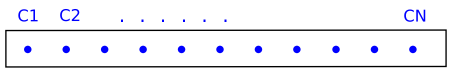


Figure 7: Representation of a 0-cell complex with a topological structure equivalent to a standard container.

Iteration and data access is used simultaneously in the basic iterator concept of the STL. The next code snippet presents these facts, where the forward iteration `++it` is used to traverse the cells. The `*it` is used to access the value attached to the cell at position `it`.

C++ STL approach

```
std::vector<int>            container;
std::vector<int>::iterator  it;

it = container.begin();
++it;                  //topological traversal
int value = *it;       //data access
```

On the one hand side, the iterator concept is one of the key elements of the C++ STL. It separates the actual data structure access from algorithms. Thereby the implementation complexity is significantly reduced. On the other hand side, it combines iteration and data access. The improvements of separating the iteration and data access are outlined with a cursor and property map concept [4]. A possible application of this approach is demonstrated in the next code snippet:

Separated iteration and data access

```
vector<bool>              container;
vector<bool>::iterator    it;
property_map              pm(container);

it = container.begin();
++it;                     //iteration
bool value = pm(*it);     //data access
```

The `std::vector<bool>::iterator` can be modeled by a random access iterator [4], whereas the data access returns a temporary object which can be used efficiently [21] with modern compilers. Additionally, this mechanism offers the possibility of storing more than one value corresponding to the iterator position. This feature is especially useful in the area of scientific computing, where different data sets have to be managed, e.g., multiple scalar or vector values on a vertex, face, or cell.

Based on the formal classification of Section 3 we analyze the combination of iteration and data access in more detail. The following list overviews the basic iterator traits [22]:

- input/output
- forward
- bidirectional
- random access

As we have seen, there is a unique and distinguishable definition possible for all of these data structural properties. On the one hand side, the backward and forward compatibility of the new iterator categories are a major problem [23]. On the other hand side, problems are encountered, if we integrate the iterator categories into our topological specification. In the following the replacement for the input and output traits is listed:

- incrementable
- single pass
- forward

The combinatorial property of the underlying space of these three categories is the same: a 0-cell complex with a local topological structure, defined by the following code snippet:

```
topology<0, local, 2>  tp;
```

The old iterator properties have only used two different categories which specify the data behavior, namely the input and output property.

The difference between these three categories can be described by:

- incrementable: this is a topological property only
- single pass: this is a data property only
- forward: this combines the incrementable and single pass properties

Only the incrementable property can be described by a topological property, whereas the other two categories are data dependent.

4.2.2 *The 1-Cell Complex*

This type of cell complex is usually called a graph. Figure 2 presents a typical example. A cell of this type of cell complex is called an *edge*. Incidence and adjacence information is available between edges and vertices.

We give examples on simple algorithms based on graphs using the BGL. The BGL implements comprehensive and high performance graph treatment capabilities including the associated adjacence and incidence relation. Iteration and data access are separated by the already mentioned cursor and property map concept [3]. The next code snippet presents an iteration using mechanisms of the BGL. In this algorithm all edges are traversed.

BGL iteration

```
typedef adjacency_list<vecS, vecS> Graph;
Graph gr(number_of_points);

// edge initialization

edge_iterator eit, eit_end;

for (tie(eit, eit_end) = edges(gr);
     eit != eit_end; ++eit)
{
  test_source1 += source(*eit, gr);
  test_source2 += target(*eit, gr);
}
```

With the GTL the same functionality can be accomplished as demonstrated in the following code snippet. The `global` keyword is used to highlight the global structure of the graph, which means, that the internal data layout is prepared for a dense graph storage.

GTL iteration

```
typedef topology<1, global> topo_t;
topo_t topo(number_of_vertices);

// cell initialization

cell_on_vertex_it covit, covit_end;

for (tie(covit, covit_end) = cells(topo);
     covit != covit_end; ++covit)
{
  test_source1 += source(*covit, topo);
  test_source2 += target(*covit, topo);
}
```

4.2.3 *The ND Cell Complex*

We extend the 0-cell and 1-cell complex types to arbitrary-dimensional cell complexes. In this work we restrict the topological spaces to the most important to scientific computing: the local (Figure 8) and the global cell complex (Figure 9). Based on our cell complex types the following cell types are available:

- 0-cell: vertex
- 1-cell: edge
- 2-cell: triangles, quadrilaterals
- 3-cell: tetrahedra, cubes

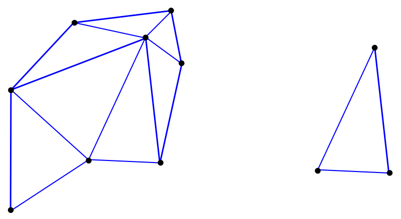


Figure 8: Local cell complex (left) and a cell representation (right). Vertices are marked with black circles.

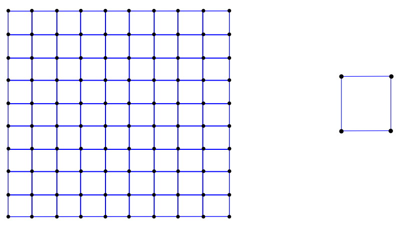


Figure 9: Global cell complex (left) and a cell representation (right).

The following code snippet presents the implementation of an arbitrary topology with the structure of a local 2-cell complex. The stored data is based on scalar values using a `double` for representation.

Iteration with our approach

```
typedef topology<2, local, 4>        topo_t;
topology_traits<topo_t>::iterator    it;

typedef data<scalar, double>         data_t;
data_t                               data;
data_traits<data_t>::value           value;

it = topo.vertex_begin();

++it;                 // iteration
value = data(*it);    // access
```

The next example presents an iteration mechanism starting with an arbitrary cell iterator evaluated on a cell complex, which is an instance of a topological cell complex. Then a vertex on cell iterator is initialized with a cell of the complex. The iteration is started with the `for` loop. During this loop an edge on vertex iterator is created and initialized with the evaluated vertex. This edge iterator starts the next iteration. The corresponding graphical representation is given in Figure 10. The necessary *valid()* mechanism models a *circulator* concept [24]. The objects marked depict the currently evaluated objects. In the first iteration state the vertex `v1` is used and the iteration is performed over the incident edge, then the iteration continues with the remaining vertices.

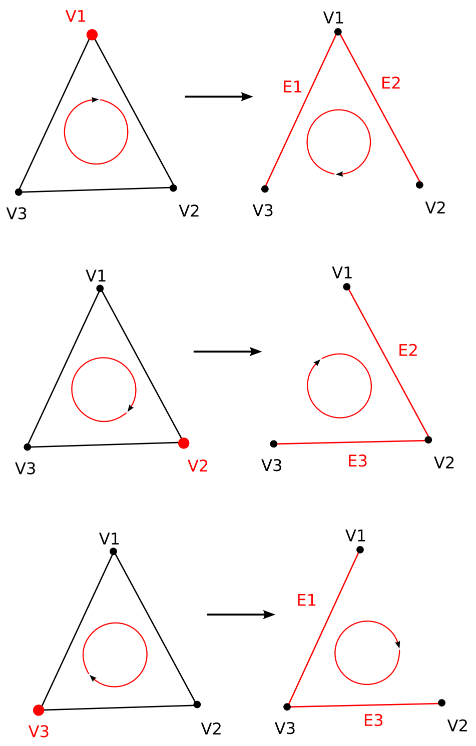


Figure 10: Incidence relation and iteration mechanism.

A more complex iteration

```
cell_iterator  ce_it  =  topo.cell_begin();

vertex_on_cell_iterator  vocit(*ce_it);
for(;vocit.valid(); ++vocit  )
{
  edge_on_vertex_iterator  eovit(*vocit);

  for  (;eovit.valid(); ++eovit  )
  {
     //operations  on  edges
  }
}
```

As can be seen, the iteration mechanism can be used independently of the used dimension or type of cell complex. The iteration is initialized with a cell iterator only. Three different objects have to be assured by the cell complex: vertices, edges, and cells. All cell complex types which support these three objects can be used for this iteration.

5. DATA ACCESS

We use the *property map* concept by a functional access mechanism called *data accessor*. The data accessor implementation also takes care of accessing data sets with different data locality, e.g., data on vertices, edges, facets, or cells. This locality is specified by the given key `key_d`. During initialization the data accessor `da` is bound to a specific cell complex with that key. The `operator()` is evaluated with a vertex of the cell complex as argument. The next code snippet presents this assignment briefly.

Data assignment

```
string  key_d   = "user_data";
data_t  da  =  scalar_data(topo,  key_d);

da(vertex)  =  1.0;
```

In the following code snippet, a simple example of the generic use of this accessor at run-time is given, where a scalar value is assigned to each vertex in a domain. The data accessor creates an assignment which is passed to the `std::for_each` algorithm.

Data assignment

```
da_t  da  =  scalar_data(topo,  key_d);

for_each
(
 topo.vertex_begin(),
 topo.vertex_end(),
 da  =  1.0
);
```

Another example is given, where the data accessor is combined with the topological structure to completely specify a container. The data accessor can be used independently.

Equivalence of data structures

```
typedef  topology<0, random_access>  topo_t;
typedef  long                        data_t;
container_t<topo_t,  data_t>         container;

//  is  equivalent  to

std::vector<data_t>        container;
```

6. GTL ARCHITECTURE

The GTL is based on a layered concept, which means that the iteration mechanism and data access mechanisms are orthogonal (Figure 11). The lowest layer represents the concepts for cell, vertex, and the poset information. The other part of the lowest layer implements the data storage. It can be observed that data can be handled independently of the topological information and iteration. The second layer provides the incidence relation and the data accessor mechanisms.

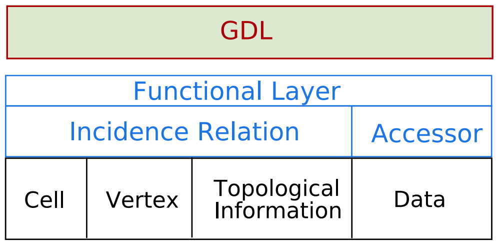


Figure 11: Conceptual view of the GTL.

The highest level in the GTL is based on meta and functional programming for a convenient usage of the different iteration mechanisms. To illustrate these mechanisms different examples are presented. The first snippet shows a simple functional iteration:

Functional iteration

```
typedef topology<2, random_access>  topo_t;
typedef long                        data_t;
container_t<topo_t, data_t>         container;

gtl::iterate<vertex_on_cell>
[
  std::cout << _1 << std::endl
](container) ;
```

With the GDL, different algorithms can be used as well as presented in the next example. Here different topological containers can be traversed and the data is accumulated and printed.

GTL iteration with GDL mechanisms

```
typedef topology<2, random_access>  topo_t;
typedef long                        data_t;

container_t<topo_t, data_t>         container;
container_t::data_accessor          da;

gtl::iterate<cell>
[
   std::cout <<
   gdl::sum<vertex_on_cell>(0.0)[da(_1)]
   << std::endl
](container) ;
```

7. OUTLOOK

The field of scientific computing requires an efficient notation of equation systems, has to construct equations, and has to abstract from the iteration mechanisms of different underlying objects. Various algorithms in the field of scientific computing only depend on the combinatorial properties of the underlying space. Using only combinatorial information results in more stable algorithms.

By providing a concise interface to different kinds of data structures, a new type of equation specification is made possible. In this way algorithms and equations can be specified independently from dimension or topological cell complex types.

To show the requirement for the equation specification we use a simple equation system resulting from a self-adjoint PDE type. Figure 12 presents a local patch of a 1-cell complex on which the equation is evaluated.

The data `Aij` represents the area of the dual graph (Voronoi graph). Using a finite volume discretization scheme [9] a generic Poisson equation $\mathrm{div}(\varepsilon\ \mathrm{grad}(\Psi)) = \varrho$ can be formulated in two spatial dimensions as:

$$\sum_j D_{ij}\ A_{ij} = \varrho \tag{1}$$

$$D_{ij} = \frac{\Psi_j - \Psi_i}{d_{ij}}\ \frac{\varepsilon_i + \varepsilon_j}{2} \tag{2}$$

D_{ij} stands for the projection of the dielectric flux onto the cell/edge c_i that connects the vertices v_i and v_j. The direct transformation of each equation element can be observed clearly when considering the following source code:

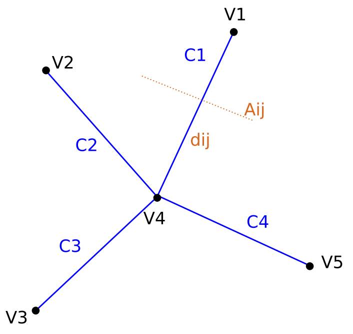


Figure 12: Cell complex with corresponding data.

Generic Poisson equation

```
value =
(
  gdl::sum<vertex_edge>
  [
     gdl::diff<edge_vertex>
     [
       Psi(_1)
     ] * A(_1)/d(_1) *
     gdl::sum<edge_vertex>[epsilon(_1)] / 2
   ] - rho(_1)
)(vertex);
```

The term `Psi` represents the distributed data set, `A` the Voronoi area, `d` the distance of two points, `rho` the right hand side, and `epsilon` some material property. It is important to stress that all data sets have to be evaluated in their right data locality, that is `Psi`, `epsilon`, and `rho` on vertices and `A`, `d` on the incident edges. The example uses the unnamed function object `_1` only. The data accessor implementation handles the correct access mechanism. The GDL implements mechanisms to derive the correct data locality of each unnamed object. An in-depth discussion is given in [8] The complex resulting from this mapping is completed by specifying the current vertex object `vertex` at run-time.

8. CONCLUSION

We have shown that the specific properties of different data structures can be specified by means of topological and combinatorial properties. An automatic derivation of optimal data structures based on the requirements of algorithms is possible.

Based on the topological properties the iterator traits can be derived automatically from combinatorial properties of the corresponding data structure. A concise iteration mechanism for different dimension is presented which includes the STL containers as well as higher-dimensional cell complex types. Different issues of the currently used iterator mechanism can be easily explained.

The full power of functional programming is revealed, when it performs different types of topological traversal with our approach.

9. ADDITIONAL AUTHORS

Siegfried Selberherr, email: selberherr@iue.tuwien.ac.at

10. REFERENCES

[1] G. D. Reis and J. Jarvi, "What is Generic Programming?" in *Library Centric Sofware Design, OOPSLA*, San Diego, CA, USA, October 2005.

[2] G. Berti, "Generic Software Components for Scientific Computing," Dissertation, Technische Universität Cottbus, 2000.

[3] J. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley, 2002.

[4] D. Abrahams, J. Siek, and T. Witt, "New Iterator Concepts," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. N1477 03-0060, 2003.

[5] D. Gregor, J. Willcock, and A. Lumsdaine, "Concepts for the C++0x Standard Library: Iterators," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. N2039=06-0109, June 2006.

[6] *Boost Lambda Library*, Boost, http://www.boost.org. [Online]. Available: http://www.boost.org

[7] *Boost Phoenix*, Boost, 2004, http://spirit.sourceforge.net/. [Online]. Available: http://spirit.sourceforge.net/

[8] M. Spevak, R. Heinzl, P. Schwaha, T. Grasser, and S. Selberherr, "A Generic Discretization Library," in *Library Centric Sofware Design, OOPSLA*, Portland, OR, USA, October 2006.

[9] S. Selberherr, *Analysis and Simulation of Semiconductor Devices.* Wien–New York: Springer, 1984.

[10] S. Selberherr, A. Schütz, and H. Pötzl, "MINIMOS—A Two-Dimensional MOS Transistor Analyzer," *IEEE Trans. Electron Dev.*, vol. ED-27, no. 8, pp. 1540–1550, 1980.

[11] S. Halama, C. Pichler, G. Rieger, G. Schrom, T. Simlinger, and S. Selberherr, "VISTA — User Interface, Task Level, and Tool Integration," *IEEE J.Techn. Comp. Aided Design*, vol. 14, no. 10, pp. 1208–1222, 1995.

[12] R. Sabelka and S. Selberherr, "A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures," *Microelectronics Journal*, vol. 32, no. 2, pp. 163–171, 2001.

[13] IμE, *MINIMOS-NT 2.1 User's Guide*, Institut für Mikroelektronik, Technische Universität Wien, Austria, 2004, http://www.iue.tuwien.ac.at/software/minimos-nt.

[14] T. Binder, A. Hössinger, and S. Selberherr, "Rigorous Integration of Semiconductor Process and Device Simulators," *IEEE Trans.Comp.-Aided Design of Int. Circ. and Systems*, vol. 22, no. 9, pp. 1204–1214, 2003.

[15] W. Benger, "Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model," Dissertation, Freie Universität Berlin, 2004.

[16] P. A. Markowich, C. Ringhofer, and C. Schmeiser, *Semiconductor Equations.* Wien-New York: Springer, 1990.

[17] A. Sheikholeslami, E. Al-Ani, R. Heinzl, C. Heitzinger, F. Parhami, F. Badrieh, H. Puchner, T. Grasser, and S. Selberherr, "Level Set Method Based General Topography Simulator and its Applications in Interconnect Processes," in *Intl. Conf. on Ultimate Integration of Silicon*, Bologna, Italy, July 2005, pp. 139–142.

[18] K. Jänich, *Topologie.* Heidelberg: Springer, 2001.

[19] A. J. Zomorodian, "Topology for Computing," in *Cambridge Monographs on Applied and Computational Mathematics*, 2005.

[20] J. R. Shewchuk, "Delaunay Refinement Mesh Generation," Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1997.

[21] J. Siek and A. Lumsdaine, "Mayfly: A Pattern for Lightweight Generic Interfaces," in *Pattern Languages of Programs*, July 1999.

[22] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[23] M. Zalewski and S. Schupp, "Changing Iterators with Confidence. A Case Study of Change Impact Analysis Applied to Conceptual Specifications," in *Library Centric Sofware Design, OOPSLA*, San Diego, CA, USA, October 2005.

[24] A. Fabri, "CGAL- The Computational Geometry Algorithm Library," 2001, citeseer.ist.psu.edu/fabri01cgal.html. [Online]. Available: citeseer.ist.psu.edu/fabri01cgal.html

A Generic Discretization Library

Michael Spevak
Institute for Microelectronics
Disastrous 27-29
1040 Vienna, Austria
spevak@iue.tuwien.ac.at

René Heinzl
Christian Doppler Laboratory
Gusshausstrasse 27-29
1040 Vienna, Austria
heinzl@iue.tuwien.ac.at

Philipp Schwaha
Christian Doppler Laboratory
Gusshausstrasse 27-29
1040 Vienna, Austria
schwaha@iue.tuwien.ac.at

ABSTRACT

We present a generic library which provides means to specify partial differential equations using different discretization schemes, dimensions, and topologies. Due to the common interfaces for simulation domains as well as numerical algebra we have an overall high inter-operability.

1. INTRODUCTION

One of the major topics in the field of scientific computing is the solution of differential equations. The field of differential equations covers various sub-fields of varying complexity and has different requirements on the underlying simulation domain as well as the mathematical formalism. In the most complex cases we face a system of coupled partial differential equations.

As mathematical structures such as scalar fields on a simulation domain do not have a direct mapping to data structures of a computer, discretization schemes and numerical methods have to be employed. During the last decades a vast number of different tools for the solution of differential equations has been developed. In general, the methods that need to be performed have not changed. Some of them have to be used together with other techniques. Based on a data structure which represents a cell complex, an equation system is assembled. After the solution of the equation system is computed the data are mapped back to the cell complex.

The main part of our work is the re-factoring and separation of the program structures needed for the discretization as well as the assembly of differential equations. By investigating numerous tools we found that various parts of code have been re-implemented in each of these tools. The tediously implemented domain-specific improvements introduced by each of the tools were not reusable in any way and had to be recoded repeatedly.

Typical programs operate on two different structures, namely a simulation domain and a matrix data structure. In the process of re-factoring and re-organizing of available code it is crucial to define all the external interfaces explicitly. The formalization of topological mechanisms allows the implementation of different discretization schemes independently from the actual representation of the topological data structure. The generic topology library (GTL) [7] provides an implementation of such a data structure which can be parametrized for arbitrary dimensions and cells.

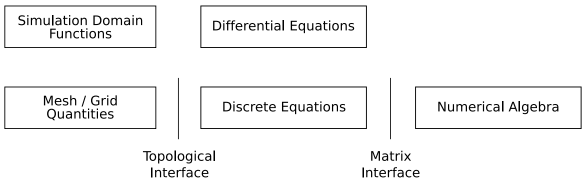


Figure 1: Interface dependencies of discretization schemes, solving algorithms and matrix interfaces and the simulation domain

The interface for the simulation domain requires access to the topological structure as well as the defined functions. The GTL models these interfaces and provides the necessary functionality to store values on the simulation domain, which are given discretely on the single topological elements. The topological functionality is mostly needed to fulfill the requirements of the discretization schemes such as the finite element method [14] or the finite volume method [12]. All of these schemes need a set of neighboring elements based on the topological property of **incidence**. Two elements are incident if one of the sets is a subset of the other.

The formalization of matrix access mechanisms for the use in numerical algebra provides the inter-operability and exchangability needed for different solver mechanisms [8]. This allows a comparison of different numeric algebra software packages under the same circumstances, which is usually not possible without a considerable amount of manual work.

There are many different interfaces [1, 8] available for the efficient assembly of equation systems. In general the interface can be reduced to only a few requirements. Even if there are solvers available which support highly specific storage structures such as band matrices, symmetric matrices or diagonal matrices, using different compressed matrix formats, the basic operations of the solvers are the same. The main problem of all of these solution mechanisms is the lack of a high level standard interface which reduces the detail of knowledge required by the end user.

We present a simple interface which makes the internal matrix structure completely interchangeable and due to orthogonality of the concepts we can reduce this effort of interfacing from $O(m \times n)$ to $O(m + n)$. The generic discretization library (GDL) introduces interfaces and makes them applicable to a huge number of problems. Its main aims are to formalize the way of discrete mathematical formula specification and to provide a formalized way of coding mathematical expressions.

The intended range of applications covers typical PDE methods as encountered in electrodynamics, mechanics, diffusion processes, fluid mechanics, and chemical reactions as well as typically discrete phenomena such as particle dynamics or even graph based problems.

The main focus of the design of the GDL was spent on specifying the mathematical formalisms. There have been approaches towards domain specific languages in this field [9, 2, 3, 11] but most of them are very specific and only work in special cases such as finite element or finite volume discretization schemes. We construct the base framework of topological operations, which covers different discretization schemes as well as purely discrete phenomena.

2. REQUIREMENTS RESULTING FROM DISCRETIZATION SCHEMES

The main aim of discretization schemes is to yield a numeric representation of a differential equation by projecting it onto a discrete simulation domain. Typically, the discretization of differential equations is achieved on elements of an underlying cell complex, e.g. on vertices. This results in an algebraic equation being assembled for each of the discrete elements of the complex. The assembled algebraic equations do not only involve values on single vertices, but also depend on neighboring elements. It is therefore impossible to solve the equations at each vertex locally and a set of coupled equations is obtained.

The discretization schemes we have investigated are the finite element method, the finite volume method, and as specialized case the finite difference method [13]. In particular, we show the different iteration mechanisms necessary for implementation.

The finite element method uses shape functions on the highest dimensional elements. Each global shape function is located on a vertex. Therefore shape-functions have non-zero values in all cells containing this vertex. The main aim of this discretization scheme is to find a weighted residual formulation. For the formulation of this discretization scheme the following operations have to be performed.

```
On all cells (c) in Neighborhood(v)
 On all vertices (w) in Neighborhood(c)
  ...
```

Finite volume schemes and finite difference schemes as well as the required topological iteration mechanisms are discussed in [4].

3. EXTERNAL INTERFACES

The main topological requirements on the simulation domain is an iteration over incident elements. A typical example of such an iteration is to find all incident edges of a vertex or vice versa. These operations use iterator-like structures [7] for all permutations of different topological elements. Apart from incidence and iteration, discretization schemes need the property of orientation. All elements have a standard orientation, e.g. an edge provides a source and a sink vertex. We define an orientation function $\mathcal{O}(a, b)$ between an edge and a vertex which returns $+1$ if a vertex coincides with the source and -1 if the vertex coincides with the sink (Figure 2). A full reference of the topological iteration possibilities can be found in [7].

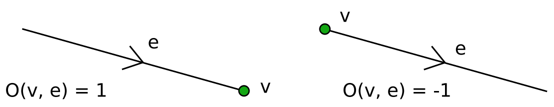


Figure 2: Orientation of an edge. The orientation function returns either $+1$ or -1 depending on whether the vertex is the sink or the source of the edge.

An interface to linear solvers has to provide several operations like the solution of a linear equation system, the inversion of a matrix or the retrieval of eigenvalues. Non-linear solver mechanisms are usually based on linear solvers and therefore can be matched to this interface.

All of these methods have in common that they are capable of handling matrices covering an arbitrary number of entries. At the time of initialization of the matrix data structure, the number of rows and columns of the matrix as well as the number of right hand side vectors has to be specified. In order to define the 10×10 matrix structure for a linear solver with three right hand side vectors we call the following constructor.

```
matrix_t msi(10, 3);
```

All matrix elements can be accessed by a function object. This interface can be used in order to obtain the values of the right hand side, the solution, eigenvalues or eigenvectors.

```
matrix_t::entry_accessor entry(msi);
matrix_t::rhs_accessor rhs(msi);

entry(0, 1) = 12.5;
rhs(2, 2) = 3.4;
```

Algorithms and data structures for linear algebra can be formulated independently. While simulation tools fill the entries of the matrix using properties of the simulation domain and the discretized differential equations, solver mechanisms operate on the matrix in order to provide the solution of the discretized problem.

4. A FUNCTIONAL CALCULUS FOR DISCRETIZATION

We have already listed the operations which are necessary to formulate discretization schemes. Based on the example of the Laplace operator we introduce the notions necessary for a fast as well as efficient implementation.

Due to space considerations we omit the means of matrix assembly. However we have to state that differential equations can be formulated using the associated differential operators. All differential equations $\mathcal{L}(f) = 0$ can be represented by their operator $\mathcal{L}(f)$. The underlying matrix mechanisms can be employed to determine all entries of the discretized differential equation and assemble the differential equations.

For the following considerations we only show the application of the differential operator.

4.1 Specification of Mathematical Formulae

The simplest expressions of our calculus are data which are stored on topological elements. Each of the discrete formulae has to access different values which are associated with the topological elements.

We assume a simulation domain where values are stored on vertices. Each vertex has one value ψ stored on it. If a function evaluation of the expression ψ has to be performed in a distinct vertex, we obtain this stored value.

If we limit the calculus to simple evaluations of the values of data, it unnecessarily impoverishes the resulting calculus without leading to a significant simplification of expressions. However, if we can combine these expressions using operators, we obtain a huge variety of combinations which cover a very broad range of expressions.

So far, the operations used do not differ very much from a typical functional approach such as the Boost Lambda Library or Boost Phoenix [5]. The major difference of mathematical expressions occurring in discretization to typical functional expressions is that the location of evaluation does not change in functional expressions. In the following example that shows the evaluation of the Laplace operator we see that for the calculation we do not only need values on the vertex of interest but we also have to access values within its neighborhood. We investigate the typical finite volume formulation of this differential operator.

$$\operatorname{div}(\operatorname{grad}(\psi_i)) = \sum_j \frac{\psi_j - \psi_i}{d_{i,j}} \cdot A_{i,j} \tag{1}$$

Even though this formulation is common for finite volume formulations, it is not directly implementable in a computer. First and most importantly, the ranges of the sum as well as the indices i and j are not defined explicitly. A lot of information is implicit and is only valid within the framework of the specific discretization scheme. For the sake of generality, however, we can not keep domain specific notations.

There is only one iteration operation to adjacent vertices, there are data $A_{i,j}$ which seem to be evaluated on both of these vertices. More precisely, the formulation states: Doubly indexed data are stored on an edge (which is defined by the vertices i and j), singly indexed data fields are located on vertices. Even though the summation index j is specified, the kind of iteration (namely vertex-vertex adjacency over edges) is assumed implicitly. From the implementation point of view this specification can only be implemented using further assumptions.

Expression (1) can be re-organized in order to show a consistent iteration scheme which allows a formulation free of domain specific notational abbreviations. We use a sum as well as a difference based on topological iteration. The indices ve and ev denote a local neighborhood iteration from a vertex to an edge and vice versa (3, 4). We explicitly name the occurring topological elements, the initial vertex is called v, the edges are called e and the vertices derived by

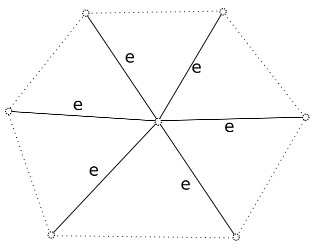


Figure 3: Exterior iteration loop. Starting from the base vertex, the incident edges are determined and traversed. The values of A as well as d are evaluated on the edges.

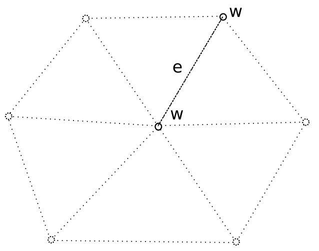


Figure 4: The interior iteration loop. The vertices which are incident with the edges are used for data access.

iteration are called w.

$$\operatorname{div}(\operatorname{grad}(\psi(v))) = \sum_{\mathrm{ve}(v,e)} \frac{A(e)}{d(e)} \Delta_{\mathrm{ev}(e,w)} \psi(w)\ . \tag{2}$$

Even though this formulation provides explicit naming of the topological elements we can see that this is not necessary for most of them. In the outer sum the data accessed are located only on e, in the difference the data accessed are located only on w. Even though the accessed data are not always located on the actually traversed element, this holds true in most of the cases. Therefore we implicitly access the element that is actually being traversed in the innermost loop. This allows us to reduce our formulation without losing generality.

$$\operatorname{div}(\operatorname{grad}(\psi)) = \sum_{\mathrm{ve}} \frac{A}{d} \Delta_{\mathrm{ev}} \psi\ . \tag{3}$$

However, the difference of elements is directly connected to the order in which the elements are traversed. Due to the fact that the implementation of the cell complex is based purely on topological neighborhood information, the order is completely free and the subtractions may be performed in an arbitrary order. We overcome this problem by generalizing the difference to more general summation processes using appropriate multiplicative factors to determine if a value has to be added or subtracted.

In order to define the sign, we use the orientation function $\mathcal{O}(a,b)$, (see Section 2) which is passed to the edge as well as the vertex (Figure 2). As a consequence we have to access the edge and pass it to the orientation function. As the edge orientation function uses the edge which is not the element of the innermost loop, we have to use an explicit name e.

$$\operatorname{div}(\operatorname{grad}(\psi)) = \sum_{\mathrm{ve}} \frac{A}{d} \sum_{\mathrm{ev}(e,w)} \psi \cdot \mathcal{O}(e,w)\ . \tag{4}$$

With this calculus we can handle a wide range of mathematical formulations which orthogonally use the methods of topological iteration and conventional functional expressions.

5. APPLICATIONS

This section describes the basic structure of a generic discretization library as well as its application to differential equations using different discretization schemes. Based on the functional calculus we establish a set of functional expressions to specify arithmetic operations and function application to data in combination with topological iteration methods. We provide a few basic examples which show the application of the mentioned discretization schemes.

5.1 Finite Volumes

We show a possible procedural implementation of the Laplace operator first. It uses iterators which are gradually replaced by a functional approach in order to obtain a formulation which is close to the expression.

Using a purely generic approach with functional elements for data access applied to (4) yields:

```
double laplacian;
vertex_edge veit(v);
for(; veit.valid(); ++veit) {
  double inter = 0;

  edge_vertex evit(*veit)
  for(; evit.valid(); ++evit) {
    inter += psi(*evit) * Orient(*evit, *veit);
  }

  inter *= A(*veit) / d(*veit);
  laplacian += inter;
}
```

Conventional formula assembly using loops and accumulation implies the use of loop counters or iterators, as well as intermediate results, which are avoided in the mathematical formulation. We explicitly need to name the iterators `evit, veit` as well as the variable `inter`. This does not only force us to find names for these variables in each implementation but also introduces redundant information.

Apart from this fact, we can see that in most cases we only need the iterator of the innermost loop explicitly. This enables us to condense the formulation using function objects which provide accumulation in combination with topological iteration. For the sake of simplicity we omit the orientation function at this point.

```
sum<vertex_edge>(ZERO)[
  A(_1) / d(_1) * sum<edge_vertex>(ZERO)[
    psi(_1) ]]
(v);
```

Although this formulation does not cover the complete information, the expression already contains the semantic information of the original formula. In contrast to the Phoenix 2 library we use the unnamed function object `_1` for the element of the innermost loop. This means, that in the outer sum `_1` denotes the edge, whereas in the inner sum `_1` denotes the vertex.

Indeed, there are two significant problems with this formulation: First, the return value of functions is hard to determine, because it is not given explicitly. In addition, more general accumulation routines require some kind of neutral value to start the accumulation. For this reason, we explicitly insert the neutral element, in our case `ZERO`. As there are many kinds of accumulation operations (sums, products, all, exist) the value can not be coupled to the type but has to be specified explicitly.

The second problem arises when we introduce the orientation function. Even though such a function can be made available as a functional expression, the second argument, namely the edge is not available directly. In analogy to the Phoenix library we use a named variable `_e` to keep the local element available in the inner loops. These local variables are passed to the function objects using the local calling stack.

```
laplacian = sum<vertex_edge>(ZERO)[
  A(_1) / d(_1) * sum<edge_vertex>(ZERO, _e)[
    psi(_1) * Orient(_1, _e) ]]
(v);
```

This functional expression is equivalent to the mathematical formulation (4) and transforms the semantics into the C++ programming language.

5.2 Finite Elements

The finite element scheme uses an integral formulation in order to assemble partial differential equations. For each two points belonging to a common cell $\mathcal{C}$ (e.g. a triangle) an integral is evaluated. This integral determines a local summand for the differential operator.

```
sum<vertex_cell>(ZERO, _v)[
  sum<cell_vertex>(ZERO, _c)[
    psi(_1)*int(_c, _1, _v) ]]
(v);
```

All finite element formulations using shape functions which are located on the vertices can be re-formulated in this manner. For higher order finite elements, this method can be easily generalized to general neighborhood operations.

All equation-specific properties of the formulation can be encapsulated in the term `int(_c, _1, _v)`. The integral term returns the value of an integral, where $\mathcal{L}$ is the given differential operator

$$\mathrm{int}(\mathcal{C}, i, j) := \int_{\mathcal{C}} \mathcal{L}\psi_i(x) \cdot \psi_j(x) \cdot dV \ . \qquad (5)$$

The arguments passed to this term are the cell which is the domain of integration. This integral formulation can be evaluated either using an analytical formula or numerically. After the integrals are evaluated and the resulting linear equation is evaluated, it is entered into the global matrix.

6. IMPLEMENTATION

The GDL is based on the Phoenix2 library and excessively uses the interfaces provided there. The GDL provides three different kinds of function objects: Data accessors, functions and accumulators. We briefly show how accumulation is implemented using the Phoenix2 library. Data accessors are used in formulae in order to access values which are stored with respect to a topological element. These accessors can be adapted to the underlying property map and do not use topological features.

In the accumulation objects, topological information is combined with information of the data stored. We show the implementation of the following line.

```
sum<IteratorTag>(Initial)[Summand]
```

First, we use several object generators in order to beautify the code and to save manual effort for explicitly coding data types. For the implementation as Phoenix2 objects, we provide a class which implements a function `eval` as well as a meta-function `apply` which returns the return type of `eval` depending on the arguments passed.

The evaluation function requires the so called environment, which contains function arguments, as well as further objects for the summand and for the initial value, which are also implemented as Phoenix 2 data structures. In the following implementation, the types of the variables are omitted due to space considerations.

We obtain the first element of the passed environment and construct a GTL style iterator. Then we initialize the result value with the result value of the function object `init`. In the next snippet, we perform the iteration combined with the evaluation of the summand.

```
eval(Env & env, Initial & init, Summand & summand)
{
  base_elem(at<0>(env.args));
  Iterator iter(base_elem);
  result = init.eval(env);

  while(iter.valid()){
     result += summand.eval(newenv(env, *iter));
     ++iter;
  }
```

For the evaluation of the summand, we have to pass the value of the iterator. If we use a vertex on cell iterator of the GTL, this function is passed a cell. The summand, however, has to be passed a vertex. For this reason we introduce a function `newenv` which transforms the environment. All other variables of the environment are preserved, only the first variable is changed.

We briefly measure the loss of performance due to the achieved level of abstraction. This was tested for the calculation of a finite volume difference approximation of a Laplace operator. A three dimensional mesh is used and compared with respect to compile time as well as run time for the functional implementation as well as for its imperative analogon.

We found that the run time for evaluation of functional expressions was within the specified range of Phoenix2 which is about 1 per cent. The abstraction penalty was under one per cent. However resource use for the compilation of large functional expressions is not negligible. The evaluation of large functional data structures also requires large amounts of RAM. For a more in-depth benchmark of functional structures we refer to [6].

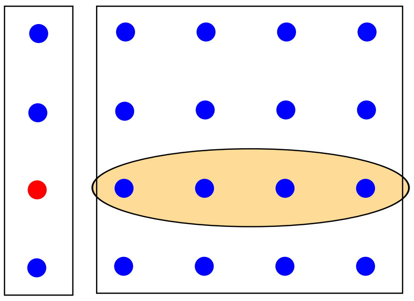


Figure 5: Two dimensional iteration. A two dimensional field of vector is iterated. The GDL provides general access mechanisms.

7. OUTLOOK

The GDL is not restricted to the operations used in scientific computing. Almost every branch of computer science deals with data structures with a more complicated underlying topology, and therefore can be reformulated to use an interface which is provided by the GTL. We show applications of this library which were not intended at the beginning but can also be performed using the GDL.

As a typical example we show an array data type (e.g. `std::vector`). If we can specify GTL style iterators which can be easily provided for each type of array we have the ability to access the underlying data structure via the GDL interfaces. Even though a vector can be specified by means of the GTL explicitly, one might have an implementation which relies on `std::vector`.

Such an expression is listed in the following snippet. A vector is traversed and all its elements are summed up. This is still possible using standard algorithms which is shown for comparison. For this reason we use some of the GTL functionality to represent the topological structure of the vector.

```
vector<int> vec;
sum<vector>(ZERO)[_1](vec);
accumulate(vec.begin(), vec.end(), ZERO, _1 + _2);
```

The following operations are not possible with standard algorithms but can be specified using functional environments like FC++ or Phoenix2 [10, 5]. We give a GDL as well as a Phoenix2 implementation. An iteration over a two-dimensional field is performed. (Figure 5)

```
vector<vector<int> > vec2;

sum<vector>(ZERO)[sum<vector>(ZERO)[_1]](vec2);

std::accumulate(vec2.begin(), vec2.end(), 0,
  _1 + phoenix::accumulate(_2, 0.0,
    lambda()[_1 + _2]))
```

In the following we show a simple example which exceeds the power of available functional frameworks. We perform an iteration over a container. During this iteration we accumulate the product of the values stored in the N elements which are topologically closest to the initial element. The set of these N elements is called a meta-cell (Figure 6). A the `meta_cell<N>` iterator of the GTL which provides the required functionality.

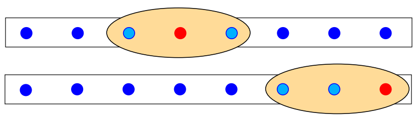


Figure 6: Metacell iteration. In this iteration the the 2 closest elements of a given element are traversed.

```
vector<int> vec;

sum<vector>(ZERO)[
  product<meta_cell<3> >(ZERO)[_1]]
(vec);
```

Even though FC++ and Phoenix2 provide container access, such operations can not be performed in an easy manner without rewritings of some components for this special case. Using the iteration data structures of the GTL, one can use arbitrary subsets for accumulation or iteration.

The applicability of the GDL strongly depends on the availability of topological iteration mechanisms on the underlying data structures. In most cases it is possible to establish such a layer. If data structures model the GTL interfaces it is also possible to specify general functional behavior via the GDL.

8. CONCLUSION

We have shown that the presented library closes the gap between the field of discretized equations and scientific application development. Apart from syntactic difficulties of C++, which complicate the formulation, the specified formulae are identical. Compiler error messages with a higher semantic level could even help the application designer to detect problematic code.

The library offers a possibility of very compact and minimalistic formulation. Even though some expressions can still be shortened, the use of the GDL reduces the effort of specification enormously. This does not only increase the speed of specification, but it also reduces the probability of typical errors.

The consequent use of the library does not only lead to a minimal effort of specification, but it also makes the programmer aware of the topological structures which are required for the discretization schemes. For this reason the framework supports direct implementation of mathematical formulations. Programmers and mathematicians have proved with well defined interfaces and the functional losses resulting from the explanation of formalisms is greatly reduced.

9. ADDITIONAL AUTHORS

Tibor Grasser, email: grasser@iue.tuwien.ac.at
Siegfried Selberherr, email: selberherr@iue.tuwien.ac.at

10. REFERENCES

[1] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. *Proc. Supercomp. '90*, pages 2–11, 1990.

[2] W. Bangerth, R. Hartmann, and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*. http://www.dealii.org.

[3] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – A General Purpose Object Oriented Finite Element Library. Technical Report ISC-06-02-MATH, Institute for Scientific Computation, Texas A&M University, 2006.

[4] G. Berti. *Generic Software Components for Scientific Computing*. PhD thesis, Technische Universität Cottbus, 2000.

[5] Boost. *Boost Phoenix2*, 2006. http://spirit.sourceforge.net/.

[6] R. Heinzl, P. Schwaha, M. Spevak, and T. Grasser. Performance Aspects of a DSEL for Scientific Computing with C++. In *Proc. of the POOSC Conf.*, Nantes, France, July 2006.

[7] R. Heinzl, M. Spevak, P. Schwaha, and S. Selberherr. A Generic Topology Library. In *Library Centric Sofware Design, OOPSLA, accepted*, Portland, OR, USA, October 2006.

[8] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. K., R. B. L., K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An Overview of the Trilinos Project. *ACM Trans. on Math. Software*, 2005. For TOMS special issue on the ACTS Collection.

[9] Kevin Kramer, W. Nicholas, G. Hitchon, University of Wisconsin. *Semiconductor Devices, a Simulation Approach*. Prentice Hall Professional Technical Reference, 1997.

[10] B. McNamara and Y. Smaragdakis. Functional Programming in C++ using the FC++ Library. *SIGPLAN*, 36(4):25–30, Apr. 2001.

[11] C. Prud'homme. A Domain Specific Embedded Language in C++ for Automatic Differentiation, Projection, Integration and Variational Formulations. *Sci. Comp.*, page submitted, 2005.

[12] S. Selberherr. *Analysis and Simulation of Semiconductor Devices*. Springer, Wien–New York, 1984.

[13] J. C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. Chapman and Hall, 1989.

[14] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method*. McGraw-Hill, Berkshire, England, 1987.

The SAGA C++ Reference Implementation

Lessons Learnt from Juggling with Seemingly Contradictory Goals

Hartmut Kaiser
Louisiana State University
Baton Rouge
Louisiana, USA
hkaiser@cct.lsu.edu

Andre Merzky
Vrije Universiteit
Amsterdam
Netherlands
andre@merzky.net

Stephan Hirmer
Louisiana State University
Baton Rouge
Louisiana, USA
shirmer@cct.lsu.edu

Gabrielle Allen
Louisiana State University
Baton Rouge
Louisiana, USA
gallen@cct.lsu.edu

ABSTRACT

The Simple API for Grid Applications (SAGA) is an API standardization effort within the Open Grid Forum (OGF). OGF strives to standardize grid middleware and grid architectures. Many OGF specifications are still in flux, and multiple, incompatible grid middleware systems are used in research or production environments. SAGA provides a simple API to programmers of scientific applications, with high level grid computing paradigms which shield from the diversity and dynamic nature of grid environments.

The SAGA specification scope will be extended in the coming years, in sync with maturing service specifications. SAGA is defined in SIDL (Scientific IDL). A C++ language binding is under development, language bindings for FORTRAN, Java, Python and C are planned.

Implementing the SAGA API specification is an interesting and challenging problem itself, due to the dynamic environment presented by current grids. Nevertheless, the perceived need of the grid community for a high level API is great enough to tackle that problem *now*, and not to wait until the standardization landscape settles. This paper describes how the C++ SAGA reference implementation tries to cope with these requirements – we believe there are lessons to learn for other API implementations.

1. INTRODUCTION

Relatively few existing grid-enabled applications exploit the full potential of grid environments. This is mainly caused by the difficulties faced by programmers trying to master the complexities of grids (see section 2). Several projects concentrate on the development of high-level, application-oriented toolkits that free programmers from the burden of adjusting their software to different and changing grids. The Simple API for Grid Applications (SAGA) [1] is a prominent recent API standardization effort which intends to simplify the development of grid-enabled applications, even for scientists with no background in computer science, or grid computing. SAGA was heavily influenced by the work undertaken in the GridLab project [2], in particular by the Grid Application Toolkit (GAT) [3], and by the Globus Commodity Grid [4]. The concept of high level grid APIs has proved to be very useful in several projects developing cyberinfrastructures, such as the SURA Coastal Ocean Observing Program (SCOOP) which uses GAT to interface to large data archives [5] using multiple access protocols.

The C++ implementation of the SAGA API presented in this paper leverages the experience we obtained from developing the GAT and will provide a reference implementation for the OGF standardization process. As the SAGA API is originally specified using the Scientific Interface Description Language (SIDL) [6], the implementation also represents a first attempt to develop the SAGA C++ language bindings. It has a number of key features, described later in detail:

- Synchronous, asynchronous and task oriented versions of every operation are transparently provided.
- Dynamically loaded adaptors bind the API to the appropriate grid middleware environment, at runtime. Static pre-binding at link time is also supported.
- Adaptors are selected on a call-by-call basis (late binding, supported by a object state repository), which allows for incomplete adaptors and inherent fail safety.
- Latency hiding (e.g. asynchronous operations and bulk optimizations) is generically and transparently applied.
- A modular API architecture minimizes the runtime memory footprint.
- API extensions are greatly simplified by a generic call routing mechanism, and by macros resembling (SIDL) [6] used in the SAGA specification.
- Strict adherence to Standard-C++ and the utilization of Boost [7] allows for excellent portability and platform independence.

2. REQUIREMENTS

As mentioned in the introduction, the SAGA C++ reference implementation must cope with a number of very dynamic requirements. Additionally, it must provide the "simple" and "easy-to-use" API the SAGA standard is intended to specify. We describe the resulting requirements in some detail motivating our SAGA implementation design described in section 3.

We identified several main characteristics the SAGA C++ reference implementation must provide, if any of these properties is missing, acceptance in the targeted user community will be severely limited:

- It must cope with evolving grid standards and changing grid environments.

- It must be able to cope with future SAGA extensions, without breaking backward compatibility.
- It must shield application programmers from the evolving middleware, and X should allow various incarnations of grid middleware to co-exist.
- It must actively support fail safety mechanisms, and hide the dynamic nature of resource availability.
- It must be portable and, both syntactically and semantically, platform independent.
- It must allow these and other latency hiding techniques to be implemented.
- It must meet other end user requirements outside of the actual API scope, such as ease of deployment, ease of configuration, documentation, and support of multiple language bindings.

3. GENERAL DESIGN

The implementation level requirements of the SAGA reference implementation as described in the previous section directly motivate a number of design objectives. Our most important objective was to design a state-of-the-art Grid application framework satisfying the majority of user-needs while remaining as flexible as possible.

This flexibility and extensibility of the implementation, is then central to the design, and dominates the overall architecture of the library (see figure 1). As a summary: only components known to be stable, such as the SAGA "look & feel" and the SAGA utility classes, are statically included in the library – all other aspects of the API implementation, such as the core SAGA classes and the middleware compile time and run time bindings, are designed to be components which can be added and selected separately.

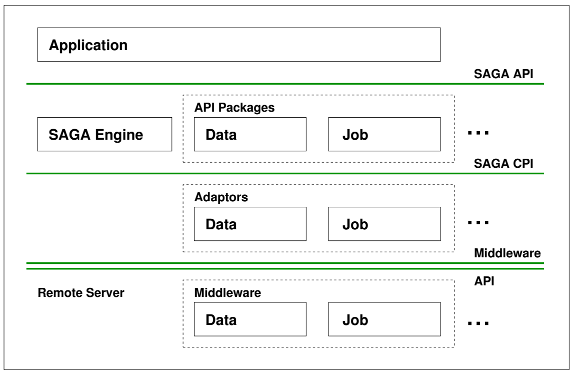


Figure 1: Architecture: A lightweight engine dispatches SAGA calls to dynamically loaded middleware adaptors. See text for details.

3.1 Design Objectives

Although the Simple API for Grid Applications is, by definition *simple* for application developers, this doesn't imply that the implementation itself has to be simple. We made a major effort to build as much logic and functionality as possible into the SAGA library core, providing all the needed common functionality. This enables the user to extend it with minimal effort. On the other hand, the library *is* designed to be easy to build, use, and deploy.

A major design objective was to maximize decoupling of different components of the developed library to provide as much *flexibility*, *adaptability* and *modularity* as possible.

As the SAGA implementation is expected to be used on different platforms and operating systems we strive for maximal implementation *portability*.

The API should be *extensible* with minimal effort: ideally, adding a new API class is orthogonal to all other properties of the implementation.

3.2 The Overall Architecture

To meet these goals we decided to decouple the library components in three completely orthogonal dimensions – the user of the library may use and combine these freely and develop additional suitable components usable in tight integration with the provided modules.

3.2.1 Horizontal Extensibility – API Packages

Our implementation uses the grouping of sets of API functions as defined by the SAGA specification to define *API packages*. Current packages are: file management, job management, remote procedure calls, replica management, and data streaming. These modules depend only on the SAGA engine, the user is free to use and link only those actually needed by the application, minimizing the memory footprint. It is straightforward to add new packages (as the SAGA specification is expected to evolve) since all common operations needed inside these packages (such as adaptor loading and selection, or method call routing) are imported from the SAGA engine. The creation of new packages is essentially reduced to: (1) add the API package files, and declare the classes, (2) reflect the SAGA object hierarchy (see section 4.1.2), and (3) add class methods.

The declaration and implementation of the API methods is simplified by macros, which essentially correspond directly to the methods SIDL specification (see section 4.6). We are considering (partly) automating new package generation, by parsing the SIDL specification and generating the class stubs and class method specifications. Additionally, this approach will also allow us to generate other SAGA language bindings from the SIDL specification, such as for C and FORTRAN.

3.2.2 Vertical Extensibility – Middleware Bindings

A layered architecture (see figure 1) allows us to vertically decouple the SAGA API from the used middleware. Separate adaptors, either loaded at runtime, or pre-bound at link time, dispatch the various API function calls to the appropriate middleware. These adaptors implement a well defined *Capability Provider Interface* (CPI) and expose that to the top layer of the library, making it possible to switch adaptors at runtime, and hence to switch between different (even concurrent) middleware services providing the requested functionality.

The top library layer dispatches the API function calls to the corresponding CPI function. It additionally contains the *SAGA engine* module, which implements: (1) the core SAGA objects such as session, context, task or task_container – these objects are responsible for the SAGA look & feel, and are needed by all API packages, and (2) the common functions to load and select matching adaptors, to perform generic call routing from API functions to the selected adaptor, to provide necessary fall back implementations for the synchronous and asynchronous variants of the API functions (if these are not supported by the selected adaptor).

The dynamic nature of this layered architecture enables easy future extensions by adding new adaptors, coping with emerging grid standards and new grid middleware.

3.2.3 Extensibility for Optimization and Features

Many features of the engine module are implemented by intercepting, analyzing, managing, and rerouting function calls between the API packages, (where they are issued) and the adaptors (where they are executed and forwarded to the middleware). To generalize this management layer, a PIMPL [8] (Private IMPLementation) idiom was chosen, and is rigorously used throughout the SAGA implementation. This PIMPL layering allows for a number of additional properties to transparently implemented, and experimented with, without any change in the API packages or adaptor layers. These features include: generic call routing, task monitoring and optimization, security management, late binding, fallback on adaptor invocation errors, and latency hiding mechanisms. The decoupling of these features from the API and the adaptors succeeds, essentially, because these properties affect only the IMPL side of the PIMPL layers.

The engine module is thus fully generic, and loosely coupled to both the API and adaptor layers. Any engine feature, all optimizations, latency hiding techniques, monitoring features etc. are implemented in generically, and are orthogonal to the API and adaptor extensions.

4. IMPLEMENTATION DETAILS

The following section will describe certain implementation details of the SAGA C++ reference implementation. As will be described, the implementation gains its flexibility mainly from the combined application of C++'s compile time and runtime polymorphism features, i.e. template's and virtual functions respective.

4.1 General Considerations

To achieve maximum portability, platform independence and code reuse, the SAGA C++ reference implementation relies strictly on the Standard C++ language features, and uses the C++ Standard and Boost libraries where possible.

4.1.1 The SAGA task model

A central concept of the SAGA API design is the SAGA task model [9], prescribing the form of synchronous and asynchronous method calls. Essentially, each method call comes in three variants: as a *synchronous call* (executed immediately), as a *asynchronous call*, and as a *task call*. The latter versions of the calls return a `saga::task` class instance. A `saga::task` thus represents an asynchronously running operation, and has an associated state (`New, Running, Finished, Failed`). Task versions of the method calls return a `New` task, asynchronous versions return a `Running` task. For symmetry reason, we added a fourth, synchronous version of method calls, returning a `Finished` task. The realization of the `saga::impl::task` class bases on a implementation of the *futures* paradigm, a concurrency abstraction first proposed for MultiLisp [10]. The C++ rendering of the SAGA task model is shown in figure 2.

While we tried to absolutely minimize the use of template's in the API layer, it was decided to implement the different flavors of the API functions using function templates (see figure 2). This makes the whole SAGA C++ implementation *generic* with respect to the synchronicity model, being another reason for providing two types of the synchronous function flavors: a direct and a task based one.

```
SAGA task model

string dest = "any://host.net//data/dest.dat";
file   file  ("any://host.net//data/src.dat");

// normal sync version of the copy method
file.copy (dest);

// the three task versions of the same method
task t1 = file.copy <task::Sync>  (dest);
task t2 = file.copy <task::ASync> (dest);
task t3 = file.copy <task::Task>  (dest);

// task states of the returned saga::task
// t1 is in 'Finished' or 'Failed' state
// t2 is in 'Running'              state
// t3 is in 'New'                  state

t3.run  ();
t2.wait ();
t3.wait ();
// all tasks are 'Finished' or 'Failed' now
```

Figure 2: The SAGA task model rendered in C++

4.1.2 The Object Instance Structure

As already mentioned, the SAGA API objects are implemented using the PIMPL idiom. Their only essential member is a `boost::smart_ptr<>` to the base class of the implementation object instance[1], keeping it alive. This makes them very lightweight and copyable without major overhead, and therefore storable in any type of container.

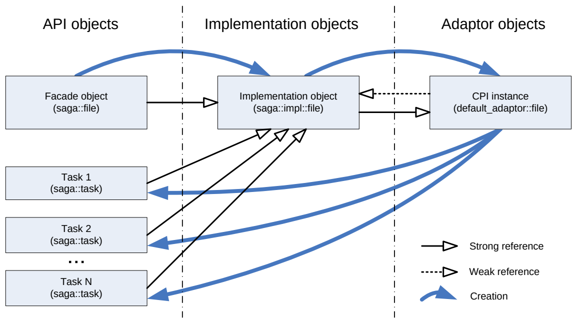


Figure 3: Object instance structure: Copying a API object instance means sharing state, returned tasks keep implementation alive.

As shown in figure 3, any API object instance creates the corresponding impl instance holding all the instance data of the SAGA object instance Copying of an API instance therefore shares this state between the copied instances. This behavior is consistent with anticipated handle based SAGA language bindings (e.g. in C or FORTRAN), where copying the handle representing a SAGA object instance naturally

[1]We refer to the implementation side of the PIMPL layer as *impl classes* in this document

means sharing the internal instance data as well[2].

Due to the shared referencing after copies, the impl instances can be kept alive by objects which depend on their state – for example, a task keeps the objects alive for which they represent a asynchronous method call (see figure 3).

The call sequence for creating a SAGA API object instance is shown in figure 4. Whenever needed, the implementation creates a CPI object instance implemented in one of the adaptors. The process of adaptor selection and CPI instantiation is injected into the API packages by the macros mentioned before (see section 3.2.1).

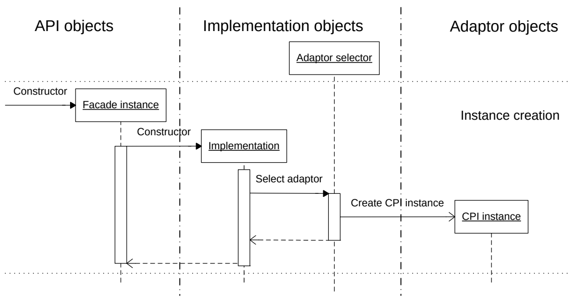


Figure 4: Object creation: Sequence diagram depicting the creation of all components as showed in figure 3. Note, how the call is intercepted by a SAGA engine module component to select a appropriate adaptor.

4.2 Inheritance and PIMPL

An interesting problem in the strict application of the PIMPL mechanism lies in the API object hierarchy: the `saga::file` class for example inherits the `saga::ns_entry` class, which inherits the `saga::object` class. Additionally, the SAGA specification requires all these classes to implement additional interfaces. Now, the PIMPL paradigm requires all class instances to own exactly *one* impl pointer[3], and are built using single inheritance only, otherwise we would face object slicing problems when copying around the base classes only. The solution is (1) to add the required interfaces to the most derived classes by duplication the interface functions, and (2) to up-cast the impl reference stored in the base class whenever needed.

API classes access the impl pointer through `get_impl()`, which, in derived classes, implies a static up-cast for the base class' impl pointer.

The implementation objects resemble the API object hierarchy. These are also derived from a common base class and contain, somewhere in their own hierarchy, similar objects to the API objects. The `saga::impl::file` class[4] inherits the `saga::impl::ns_entry` class, which inherits the implementation specific `saga::impl::proxy` class, which is derived from the common `saga::impl::object` class. Thus, the class hierarchy on the implementation side of the PIMPL paradigm reflects the API side of the class hierarchy, ensuring the correct casting behavior in the `get_impl()` methods.

[2]A polymorphic `saga::object::clone()` method is, however, part of the SAGA API, and allows for explicit deep copies of API objects, forcing the instance data to be copied as well.

[3]In fact the impl pointer stored in any `saga::object` instance is a `boost::smart_ptr<saga::impl::object>`, i.e. a reference to the very base class of the implementation object hierarchy.

[4]The `saga::impl::file` class for example is the implementation equivalent to the `saga::file` class, as we kept all API classes in `namespace saga` and all corresponding implementation classes in `namespace saga::impl`.

Constructors in the saga::file hierarchy

```
// saga::file constructor
file::file ([args])
:  ns_entry (new saga::impl::file ([args]))  {}

// saga::ns_entry constructor
ns_entry::ns_entry (saga::impl::ns_entry* impl)
:  saga::object (impl) {}

// saga::object constructor
//  'impl_' is a boost::smart_ptr<saga::impl::object>
object::object (saga::impl::object* impl)
:  impl_ (impl) {}
```

Figure 5: Realizing inheritance in PIMPL classes (simplified). Only the `saga::object` base class owns an impl pointer.

4.3 State Management

Section 4.1.2 discussed object state, in relation to state sharing of objects after shallow copies. Here we describe the object state management of the SAGA implementation in more detail, since state management is a central element on several layers. On a different layer, the adaptors represent operations on the object instances, and need to maintain state as well. At the adaptor level this is complicated by the fact that the object state can (and in general will) be changed by several adaptors (remember: adaptors are selected at runtime, and may change for each API function invocation). For state management, we hence distinguish between three types of state information.

- *Instance data* represent the state of API objects (e.g. file name, file pointer etc.). These are predefined and not amendable by the adaptor as they represent common data either passed from the constructor, or needed for consistent state management on the API level.
- *Adaptor data* represent the state of CPI objects (e.g. open connections) and are shared between all instances of all CPI object types implemented by a single adaptor and corresponding to a single adaptor instance.
- *Adaptor-instance data* represent the state shared between all CPI instances created for a single API object and implemented by the same adaptor (e.g. remote handles).

The lifetime of any type of the state information is maintained by the SAGA engine module, which significantly simplifies the writing of adaptors.

All three types of state information are carefully protected from race conditions potentially caused by the multi-threaded nature of the implementation. We provide helper classes simplifying the correct locking of the instance data. This uniform state management enables object state persistency in the future, with minimal impact on the code base.

4.4 Generic Call Routing

The essential idea of the implemented generic API call routing mechanism is to represent the calls as abstract objects, and to redirect their execution depending on several attributes and the adaptor suitability. For example, an asynchronous method call for a `saga::file` instance is preferably directed to a asynchronous file adaptor, or, if such is not available, to a synchronous file adaptor, wrapping it in a thread, or, returns an error otherwise (`NotImplemented`).

This routing mechanism allows for (1) trivial (synchronous) adaptor implementations, (2) late binding (differents adaptor can be selected for each call, even on the same API object instance), (3) variable adaptor selection strategies (based on adaptor meta data, user preferences, and heuristics), and (4) latency hiding (bulk optimization [11], or automatic load distribution over multiple adaptors). Figure 6 is depicting the injection of the call routing mechanism by the SAGA engine.

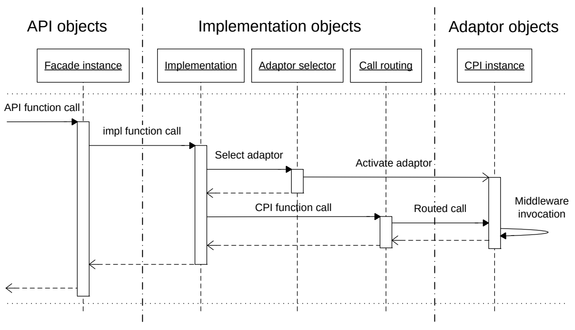


Figure 6: API function call: Diagram illustrating the execution sequence through the different object instances during a call to any adaptor supplied function.

All SAGA API methods come in synchronous and asynchronous flavors (see section 4.1.1). To avoid, that adaptors need to implement both flavors, we provide fallback implementations in the SAGA engine. The synchronous behavior is modelled by executing the the asynchronous implementation and waiting for it to finish. The asynchronous wraps the synchronous implementation into a thread representing the asynchronous remote operation.

Even if this approach has a couple of drawbacks (it is not really asynchronous, the middleware call still blocks, causing lock problems if implemented badly, and tasks are not able to survive the application life time), the mechanism simplifies adaptor implementations greatly, as most of the existing grid middleware is *not* fully asynchronous anyway.

4.5 Adaptor Selection

The selection of suitable adaptors at runtime is a central component in the implementation (see figure 6). It is, a very simple mechanism: on loading, the adaptor components register their *capabilities* in the adaptor registry. If a method is to be executed, the adaptor selector searches that registry for all suitable adaptors, orders them, and tries them one-by-one, until the method invocation succeeds. The adaptor selection is routed through SAGA engine, generically implementing this for any API function.

To overcome the limitations of this approach (several CPI instances have to be created, remote operations add additional latencies), our library allows adaptors to specify additional, key/value based meta data, and also allows to exchange the adaptor selection component.

4.6 Utilization of Macros

Our SAGA implementation makes extensive use of C++ preprocessor macros. This might be perceived as a design flaw, at least by some readers, and we were very hesitant to utilize macros extensively. However, the benefits for the end user and other programmers(!) seem currently to outweigh the problems, such as limited debugging abilities. mentioned in section 3.2.1, We are using Boost.Wave [7] features to pregenerate partially macro expanded sources to overcome the disadvantages of plain macros, hence simplifying debugging and improving readability.

5. IMPLEMENTATION PROPERTIES

This section summarizes the properties of our SAGA implementation from an end user perspective, gives an overview about the lessons learnt, and motivates further developments and extensions.

5.1 Uniformity over Programming Languages

The SAGA API specification is language independent. One of the consequences of this is that it does not use templates, which were thought too difficult to express uniformly over many languages. Also, the specification tries to be concise about object state management, and hence also expresses semantics for shallow and deep copies. Our implementation follows the SAGA API specification closely. It is also designed to accommodate wrappers in other languages. A Python wrapper for our library is in alpha status, and we plan to add wrappers to provide bindings to C, FORTRAN, Perl, and possibly others. In the past we found it very useful to be able to write Python adaptors for the GAT [3], a predecessor of SAGA, and we will provide similar support here as well.

5.2 Genericity in Respect to Middleware, and Adaptability to Dynamic Environments

The dynamic nature of grid middleware is addressed in our implementation by the described adaptor mechanism which binds to diverse middleware. Late binding, fall back mechanisms, and flexible adaptor selection allow for additional resilience against an dynamic and evolving run time environment. Adaptors need to deploy mechanisms like resource discovery, and need to implement fully asynchronous operations, if the complete software stack is to be able to cope with dynamic grids.

5.3 Modularity ensures Extensibility

Section 4.6 described how the SAGA implementation will be able to cope with the expected evolution and extension of the SAGA API. The adaptor mechanism allows for easy extensions of the library, providing additional middleware bindings. The task of adaptor writing requires massively more effort than the implementation of the presented library. Ideally, middleware vendors will *implement* adaptors for SAGA, and deliver them as part of their client side software stack. This would be a major step towards wide spread grid applications.

5.4 Portability and Scalability

Heterogeneous distributed systems naturally require portable code bases. Our library implementation is very portable, as we strictly adhere to the C++ standard and portable libraries. We currently develop the library on Windows, Linux and MacOS concurrently, covering three major target platforms without any problems. However, the portability of our SAGA implementation depends on the portability of the adaptors, and hence on the portability of the grid middleware client interfaces, being the much greater problem if compared to the library code itself.

Distributed applications are often sensitive to scalability issues, in particular in respect to remote communications. This equally applies to SAGA, so that scalability concerns are naturally raised in respect to SAGA implementations as well. Even if the SAGA API is not targeting high performance communication schemes, but tries to stick to simple communication paradigms, our design allows for zero-copy implementations of the SAGA communication APIs, and for fast asynchronous notification on events – both are deemed critical for implementing scalable distributed applications.

5.5 Simplicity for the End User

SAGA is *designed* to be simple. However, simplicity of an API is not only determined by its API specification, but also by its implementation: simple deployment and configuration, resilience against lower level failures, adaptability to diverse environments, stability, correctness, and peaceful coexistence with other programming paradigms, tools and libraries are some of the characteristics which need attention while implementing the SAGA API.

A modular implementation helps to keep a library implementation itself simple. Features as the generic call routing, or the adaptor selection are hidden in the engine module. Modeling these central properties as modules increases the readability and maintainability of the code significantly.

Due to its notion of tasks the SAGA API implicitly introduces a concurrent programming model. Our C++ language binding of the API, allows to combine that model with arbitrary mechanisms for managing concurrent program elements (thread safety, object state consistency, etc.).

6. FUTURE WORK

As mentioned, work on appropriate middleware adaptors will undoubtedly require significant resources in the future. This motivates us to work on simplifying adaptor creation, integration and maintenance, and seek support and contributions from the OpenSource community, and from grid middleware vendors. We will develop other language bindings on API and adaptor level, and apply further generic latency hiding techniques.

7. CONCLUSION

We have described the C++ reference implementation of the SAGA API, which is designed as a generic and extensible API framework: it allows for the extension of the SAGA API, easily usable for other APIs); it allows for run-time extension of middleware bindings, and it allows for orthogonal optimizations and features, such as late binding, diverse adaptor selection strategies, and latency hiding. The used techniques enable these features, amongst them the application of the PIMPL paradigm for a complete class hierarchy and generic call routing.

These implementation techniques incur a certain overhead, however, in grid environments the runtime overhead is usually vastly dominated by communication latencies, so that ***this** overhead does not matter*. The lesson learned is that distributed environments *allow* for fancy mechanisms, which are too expensive in local environments. Fail safety and latency hiding mechanisms are more important than, for example, virtual functions, late binding, and additional abstraction layers.

8. ACKNOWLEDGMENTS

We thank Thorsten Schütt for his work on Futures; Michel Zsandstra and Vladimir Prus for their initial work on dynamic loading; the SAGA groups in OGF for their work on the SAGA specification; the Center for Computation & Technology at Louisiana State University, the Vrije Universiteit Amsterdam, and the XtreemOS project for funding this work.

9. REFERENCES

[1] SAGA Core Working Group. Simple API for Grid Applications – API Version 1.0. Technical report, OGF, 2006. `http://forge.ggf.org/sf/projects/saga-core-wg`.

[2] Gridlab: A Grid Application Toolkit and Tsetbed. `http://www.gridlab.org/`.

[3] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 2004.

[4] G. v. Laszewski, I. Foster, J. Gawor, and P. Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):645–662, 2001.

[5] D. Huang and G. Allen and C. Dekate and H. Kaiser and Z. Lei and J. MacLaren. getdata: A Grid Enabled Data Client for Coastal Modeling. In *High Performance Computing Symposium (HPC 2006)*, April 3–6 2006.

[6] SIDL. Scientific Interface Definition Language. `http://www.llnl.gov/CASC/components/babel.html`.

[7] Boost C++ libraries. `http://www.boost.org/`.

[8] H. Sutter. Pimples–Beauty Marks You Can Depend On. *C++ Report*, 10(5), 1998. `http://www.gotw.ca/publications/mill04.htm`.

[9] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. v. Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications – High-Level Application Programming on the Grid. *Computational Methods in Science and Technology: special issue "Grid Applications: New Challenges for Computational Methods"*, 8(2), SC05, November 2005.

[10] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] S. Hirmer, H. Kaiser, A. Merzky, A. Hutanu, and G. Allen. Seamless Integration of Generic Bulk Operations in Grid Applications. In *Submitted to International Workshop on Grid Computing and its Application to Data Analysis (GADA'06)*, Agia Napa, Cyprus, 2006. Springer Verlag.

A Parameterized Iterator Request Framework for Generic Libraries

Jacob Smith
Texas A&M University
thechao@cs.tamu.edu

Jaakko Järvi
Texas A&M University
jarvi@cs.tamu.edu

Thomas Ioerger
Texas A&M University
ioerger@cs.tamu.edu

Abstract

The iterator abstraction is central to many generic libraries, such as the Standard Template Library (STL). Generic algorithms are commonly specialized with regard to the kinds of iterators available. There is, however, no mechanism for selecting the kind of iterator that a container should provide for a particular algorithm. We propose a framework for explicitly requesting iteration schemes from containers by using a collection of tag classes. This is a new axis of parameterization for STL-like generic libraries. The motivation for the framework comes from our work with the CCTBX and TEXTAL Protein Crystallography (PX) libraries. The models of the data in this domain provide multiple, complex iteration schemes, and the efficiency of many algorithms depends crucially on selecting a suitable scheme. The framework allows individual algorithms to access the preferred iteration scheme over the container it uses. We describe the framework with examples in the context of the STL and the PX libraries.

1. Introduction

Generic programming emphasizes *algorithm specialization*, which essentially means providing many implementations for the same functionality. A specialization of a generic algorithm places more requirements on its inputs and can make more assumptions on them, possibly enabling a more efficient implementation. A simple example is finding an element in an unsorted versus in a sorted sequence: the former requires linear run-time with respect to the length of the sequence, the latter only logarithmic since the assumption about sortedness allows an implementation using the binary search strategy.

In generic libraries, such as the Standard Template Library [13] (STL), algorithm specialization is used for many algorithms whose inputs are types conforming to the STL's iterator concepts. For example, the distance function—for measuring the distance between two iterators—is defined for all types that meet the requirements of the INPUTITERATOR *concept*. (We assume the reader is familiar with the established terminology of generic programming, including terms such as "concept", "modeling", and "refinement" — see for example [3].) The least specialized version of distance is implemented by counting the number of times the first iterator is incremented to reach the second iterator. A different specialization of this distance-computing algorithm—operating in constant time—is provided for RANDOMACCESSITERATORS. Due to the random access capability of the iterators, the implementation of this specialization is a simple subtraction. Typically algorithm specialization takes place automatically: the generic library selects the best available specialization for the types of the inputs to the algorithm.

STL algorithms operate on sequences described as pairs of iterators. The source of a sequence is often a container which provides mechanisms (such as the begin and end functions) to present the contents of the container as a sequence. When using these mechanisms the container provides *the most powerful iterator types it can offer to enable the most efficient algorithms to be defined for it*. For example, v.begin() for a v with type std::vector gives a RANDOMACCESSITERATOR, whereas l.begin() for an l with type std::list is only capable of providing a BIDIRECTIONALITERATOR. The result is that the two lines below eventually invoke different implementations of the distance function, the first being a constant time operation with respect to the distance between the iterators, and the second linear:

```
distance(v.begin(), v.end());
distance(l.begin(), l.end());
```

Algorithm specialization along the hierarchy of iterator concepts is not the only opportunity for specialization. In particular, in a design where algorithms also operate on containers and not solely on iterators, the selection of the iteration scheme for a particular container can be subjected to specialization, and can result in performance gains. Note that in many domains the norm of passing data to generic algorithms is via containers, not iterators; graphs and matrices serve as examples of such data. In Section 4.3 we describe data structures in the domain of *protein crystallography* where this is true as well.

When requesting a sequence from an STL container (with the begin and end member functions), the container typically provides iterators that conform to the most refined iterator concept possible. This can be less than ideal. It might be more efficient to provide a *less capable* iterator scheme—it is possible for the container to implement the less capable iterator in a more efficient way. Similar situations occur frequently. For example, consider a generic image type that represents a two-dimensional raster image with an arbitrary number of channels, parametrized over the value type of the channels. Examples of concrete instances of such an image type include a one-bit black and white mask, an RGB or CMYK bitmap, or images with a larger number of color channels. Such an image type can be implemented as an array whose size is the product of the width, height, and the number of channels. Assume we lay out the image data in this array as a list of raster lines, where a raster line is a list of pixels, where a pixel is a list of values from each channel. Consider visiting each channel value and performing some independent operation on it. If the only iteration scheme provided by the image type directly models the hierarchy

LCSD '06 October 22nd, Portland, Oregon.
ACM [to be supplied].

"raster line, pixel, channel value", a function visiting each channel value requires three nested loops:

```
raster_line_iterator rtr = image.begin();
raster_line_iterator rnd = image.end();

for ( ; rtr!=rnd; ++rtr ) {
  pixel_iterator ptr = rtr→ begin();
  pixel_iterator pnd = rtr→ end();

  for ( ; ptr!=pnd; ++ptr ) {
    channel_iterator ctr = ptr→ begin();
    channel_iterator cnd = ptr→ end();

    for ( ; cnd!=ctr; ++ctr ) some_operation (*ctr);
  }
}
```

This hierarchy may not, however, be necessary for the operation performed with the channel values. In such a case, this iteration scheme will perform a non-trivial amount of unnecessary work. A more direct and efficient mechanism for visiting all channel values is to iterate over the underlying contiguous memory directly, ignoring the hierarchical structure:

```
channel_iterator ctr = begin<channel>(image);
channel_iterator end = end<channel>(image);

for ( ; cnd!=ctr; ++ctr )
    some_operation (*ctr);
```

The begin<channel> and end<channel> functions are requests for this non-hierarchical iteration scheme (see Section 3). This iteration scheme is considerably faster; we report timing results in Section 4.1.

Requesting a simpler iterator scheme for efficiency is the "dual" of algorithm specialization over iterator schemes. If equivalent algorithmic functionality can be provided with the same complexity guarantees with a simpler iterator, then it is preferable to use the simpler iterator as it will have improved performance. Section 4.2 discusses the algorithmic differences between "Las Vegas" and sequential InputIterators. For example, the exact same code (the STL's find algorithm) has drastically different performance characteristics depending upon which iteration scheme is used.

In this paper, we propose a lightweight framework for algorithms to request a particular iterator scheme from a container or a sequence source. The library consists of a small number of functions forming the API for the client of the library, and requires the algorithm and container implementations to follow a small set of conventions, a relatively light burden for the library developers.

The core of the framework is a set of tag **struct**s which we call *iterator tags*. The global functions begin and end parametrized with a tag and a container give access to the iterator schemes that a container provides. In essence, the proposed framework suggest a new degree of parametrization to STL-like generic libraries. We suspect that it is possible to identify a set of iterator tags that could be established similar to the iterator concepts in the STL. We do not suggest such a set in this paper, but describe a handful of useful iterator schemes. We identify situations where parameterizing the iteration scheme provides notable benefits. In describing the framework, we assume some familiarity with *template metaprogramming*, as described, e.g., in [1].

2. Background and Motivation

In the STL [14], access to the iterator of a container is provided through the member functions begin and end. Some containers support iteration backwards with the members rbegin and rend. Additionally, STL containers overload these functions for the case where the container is a **const** object. Any single container can thus provide up to four different iterator *types* (but essentially only two iteration *schemes*). For any particular STL container, these schemes are always the most capable iterators that the container can offer, for example, std::vector provides RandomAccessIterators, and std::list BidirectionalIterators [14]. The STL containers are thus closely tied to the iterator scheme they implement and to the iterator concept the return types of their begin and end functions provide. Even if a container could provide multiple iteration schemes over its data, the STL defines no generic interface for accessing them.

In order to take advantage of alternative iterator schemes, we need a mechanism to access such schemes. A straightforward mechanism for doing this would be to provide a specific function name for each iteration scheme, as is already done with rbegin and rend. Dedicating a specific function name for each iteration scheme does not, however, work well with generic programming: function names become hard-wired in the implementations of generic algorithms. When writing a generic algorithm, the iteration scheme is not necessarily known. If we assume a generic algorithm is parametrized over a container type, and it uses a particular member function of the container to access an iteration scheme, then the use of the algorithm is limited to containers which implement the particular function name for requesting the iterator. Using distinct member functions for each iterator scheme also goes against the principle of specialization. Algorithm specialization automatically selects the best available implementation for an algorithm, but gracefully degrades to a slower version if the requirements of the faster ones are not met. Similarly, we wish to allow a request for a particular iterator scheme, but settle for a less specialized one, if the exact requested one is not supported by a particular container.

The STL implements algorithm selection using *tag dispatching*: a fixed set of tag **struct**s which each correspond to a particular iterator concept. The tag of any iterator type can be accessed via the iterator_traits machinery. The iterator_traits<Iter>::iterator_category expression is guaranteed to denote the tag of the type Iter, if Iter conforms to one of the iterator concepts [14]. Similar to iterator categories, we use tags to refer to different iterator schemes. Our iterator request framework defines the global functions begin and end that take a container type as their function parameter, and additionally a type argument specifying the requested iterator tag. In this way the iterator tag is not a fixed part of the signature. It can be, for example, a type parameter at the call site to the begin and end functions—a generic algorithm can itself be parametrized over the tag, allowing the caller of the algorithm to specify the tag requested in the interior of the algorithm. This allows clients of the generic algorithm to "reach through" the algorithm to specify functionality.

3. Iterator Request Framework

The iterator request framework consists of a family of begin and end functions, a metafunction that computes the type of the iterators returned by the begin and end functions, and some helper functions and metafunctions. We first describe the metafunction iterator, shown in Figure 1, that specifies the type of the iterators for a given iterator tag–container pair. By default, a metafunction called iterator<Tag, Container>::type resolves to the member type iterator in Container. The default is thus to access the iterator member type in the STL containers. To make the iterator types of other iteration schemes accessible, a generic library must specialize the iterator template for the relevant iterator tag–container type pairs.

The second metafunction, const_, is for convenience; it provides access to the type of the iterators implementing the constant version of the requested iteration scheme. The default is, analogously, the const_iterator member type of the container parameter.

In addition to the above metafunctions, the interface to the library includes the functions begin, end, and const_begin, and const_end. The first two functions provide both constant and non-constant access to the iterators. The latter two functions are included to aid in situations when a constant iterator scheme is

```
template < typename Tag, typename Container >
struct iterator {
        typedef typename Container::iterator type;
};
template < typename Arg >
struct const_ {
        typedef typename Arg::const_iterator type;
};
template < typename Tag, typename Container >
struct const_< iterator<Tag,Container> > {
        typedef typename Container::const_iterator type;
};
```

Figure 1. The iterator and const_ metafunctions for the framework.

needed, but where the current context is non-constant. All four functions are parametrized over both an iterator tag and a container type. Their return types are computed with the iterator metafunction discussed above, and shown in Figure 1.

Figure 2 shows all versions of the begin interface functions; the implementations of the end functions are analogous. All the interface functions merely forward the calls to appropriate tagged_begin or tagged_end functions, implementing the three different versions of begin and end with only two "back-end" functions. This means less work for the container implementer.

```
template < typename Tag, typename Container >
typename iterator<Tag,Container>::type
begin ( Container& ctr ) {
        return tagged_begin(ctr,Tag());
}
template < typename Tag, typename Container >
typename const_<iterator<Tag,Container> >::type
begin ( Container const& ctr ) {
        return tagged_const_begin(ctr,Tag());
}
template < typename Tag, typename Container >
typename const_<iterator<Tag,Container> >::type
const_begin ( Container const& ctr ) {
        return tagged_const_begin(ctr,Tag());
}
```

Figure 2. The set of begin functions for the framework.

The tagged_begin and tagged_end functions are shown in Figure 3. The default versions of these functions forward to the container's member functions begin and end, using the current STL convention. The second function is the same as the first, except that the computed return type is constant. It is up to the container or algorithm implementer to overload these functions to return the desired iterator for a particular iterator tag.

```
template < typename Tag, typename Container >
typename iterator<Tag,Container>::type
tagged_begin ( Container& ctr, Tag ) {
        return ctr.begin();
}
template < typename Tag, typename Container >
typename const_<iterator<Tag,Container> >::type
tagged_begin ( Container const& ctr, Tag ) {
        return ctr.begin();
}
```

Figure 3. The set of tagged_begin functions for the framework.

In sum, to add a new iterator scheme, one metafunction must be extended with a new class template specialization (iterator) and with four function template overloads (the const and non-const versions of the tagged_begin and tagged_end).

4. Examples

In this section we demonstrate the use and benefits of the iterator request framework with three examples. The first one is the image example discussed in Section 1, for which we present some run-time performance information; the second is in the context of the STL; and the third is our motivating example taken from the computational protein crystallography context.

4.1 Timings for images

In Section 1 we presented two alternative schemes of iterating over the pixels of an image. To demonstrate the importance of being able to select the most suitable iteration scheme for such image containers, we measured the performance difference of the two different iteration schemes, which we refer to as *hierarchical* and *linear*. Our implementation of the image was a wrapper around the std::vector. Access to the iterators are through the framework's begin and end functions, using either of the tags hierarchical or linear. Each of the hierarchical iterators stores a pointer to its parent iterator and an integer to the offset from the parent's offset. That is, the raster iterator stores a pointer to the image and the raster-line it is representing. A pixel iterator stores a pointer to its parent raster-line iterator and an offset into that raster-line to the pixel. And the channel iterator stores a pointer to its parent pixel iterator and an offset into the pixel.

Size, Channels	1	3	6
128×128	1.75	2.15909	2.24436
256×256	1.74011	2.15009	2.23694
512×512	1.73558	2.14746	2.24312
1024×1024	1.73084	2.15235	2.25039

Figure 4. The ratios of the execution time of the hierarchical iteration scheme over the execution time of the linear iteration scheme, measured by timing the execution times of a function essentially equivalent to the STL's fill. The columns are the number of channels, and the rows are the number of pixels.

We measured the performance of iterating through images with varying width, height, and the number of channels using both iteration schemes. Our test algorithm was a variation of the STL fill algorithm: we assigned the value "127" to each channel value in the image. The measured data are depicted in Figure 4, which shows the ratios of execution time of the hierarchical iterator scheme implementation to that of the linear iterator scheme implementation. The hierarchical iterator is generally about twice as fast as the linear iterator on a PowerBook5,6 G4 at 1.67 GHz with 2 GB of RAM.

4.2 Alternate run-time characteristics for std::find

In the above example with images, the selected iterator scheme affected the implementation of the algorithm. In this section, we show how changing the iterator scheme can affect the run-time performance, even run-time behavior, of the same piece of code. In particular, we take a linear deterministic algorithm and convert it into a Las Vegas algorithm. A Las Vegas algorithm is a randomized algorithm that terminates when some stopping condition is met or a certain number of iterations have occurred.

The input arguments of the STL find algorithm must be INPUTITERATORS, at minimum. The find algorithm implements a straightforward sequential search. The run-time performance for find is dependent upon the distribution of the data in the sequence being iterated over, leading possibly to the worst-case behaviour being realized frequently. In such a case, a randomized algorithm, e.g. a Las Vegas algorithm, can possibly guarantee a better average complexity of iterator increments and dereferences.

With the iterator scheme selection framework we can parametrize a generic algorithm over the iteration scheme, allowing the client to choose the iteration scheme to be used in the find algorithm. In the following example, we associate the iterator tag linear with the sequential iterator scheme and the las_vegas with the Las Vegas iteration scheme:

```
template <typename Tag, typename Container>
void uses_find ( Container const& ctr,
  typename Container::value_type const& v ) {
  ...
  std::find(begin<Tag>(ctr), end<Tag>(ctr), v);
  ...
}
```

Note that the uses_find function does not need to change in order to change the iterator scheme for std::find; the Tag type parameter tunnels through to find, as demonstrated by the following code fragment:

```
std::vector<int> a(1000,0), b(1000,1);
a.insert(a.end(), b.begin(), b.end());
a_function_that_calls_find<linear>(a, 1);
a_function_that_calls_find<las_vegas>(a, 1);
```

4.3 Protein Crystallography

The impetus for the iterator request framework was from designing generic algorithms for computational protein crystallography (PX). This section describes how without such a framework writing generic code leads to unacceptable trade-offs: the programmer must either depend on library-specific data structures, or accept an unreasonable loss of efficiency—a performance penalty of up to a factor of 30. We first briefly introduce the field of protein crystallography, followed by the discussion on implementing one of the key algorithms of PX libraries. We then demonstrate the use of our framework that avoids the above trade-off, achieving simultaneously generality and efficiency.

4.3.1 Background information on PX

In computational protein crystallography, one of the fundamental data structures is the *electron density* container. This is a container which represents the "presence" of an electron at a particular point in space—literally, the probability of an electron being at a given point in space. The electron density is computed by taking the Fast Fourier Transform (FFT) of a set of *reflections*, where a reflection is a five-tuple representing a 3D reflection-plane, an amplitude, and a phase [8, 12]. The data comes from bombarding a crystal of a protein with X-Rays. The electron density is used in algorithms and programs to help the crystallographer construct a model of the protein under investigation, to be used in drug discovery, determining novel structures, and so forth [12].

The libraries we primarily work with are TEXTAL [9] and CCTBX [7]. TEXTAL is a tool chain that automatically builds protein models from electron density data [9]. CCTBX is a library of algorithms and other tools to aid in the development of PX software [7]. Within CCTBX and TEXTAL—as with other PX libraries not considered here [4, 5]—the electron density container has numerous different representations [7–9, 12], leading to numerous ways of iterating over the data.

In TEXTAL and CCTBX the iterators used are also coordinates, i.e. the type which represents an iterator with the normal semantics (increment, dereference), is also a type which implements the semantics of a 3-dimensional vector, or an offset, depending on the iterator. Access to the data in an electron density container is then provided either by dereferencing the iterator in the normal way, or by "passing" the iterator as a coordinate to the electron density container, usually by the **operator** []. The two concepts are mixed to provide programmers' the syntactic convenience of iterators and, on the other hand, coordinate access to the same data when that is more natural. There are a large number of coordinate systems—and their equivalent iterator schemes—for the electron density container which arise from the various descriptions of the topology of the electron density data.

The large number of iteration schemes arises because there are four coordinate systems and three so called "symmetry" representations. The coordinate systems are know as the *linear-array*, *grid*, *fractional*, and *cartesian* system. Of these, the grid coordinate system—and it's equivalent iteration scheme—is the default system for the CCTBX and TEXTAL libraries. The grid-coordinates essentially represent the data as a 3-dimensional array. The linear-array is a 1-dimensional array, and the underlying structure which holds the data. The fractional and cartesian coordinate systems are 3-dimensional systems which are used to conveniently represent the underlying topology, and "real" space, respectively. All of the coordinate systems and iteration schemes are necessary, and should thus be accessible to the client of these libraries [4, 5, 7, 9].

For each coordinate system there are three levels of increasing symmetry: *non-symmetric*, *translation-independent*, and *asymmetric*. The different symmetry concepts provide different levels of sophistication to the representation of the actual underlying topology of the space [8]. In this paper non-symmetric means to not use symmetry relations; translation-independent means to consider an entire unit-cell; and asymmetric means to use the most refined space group. Figure 5 depicts the symmetry classes with a simplified example. The four coordinate systems and the three symmetries amount to ten—we exclude the two higher symmetries for the linear-array coordinate system—different RANDOMACCESSITERATORS possible for any electron density container.

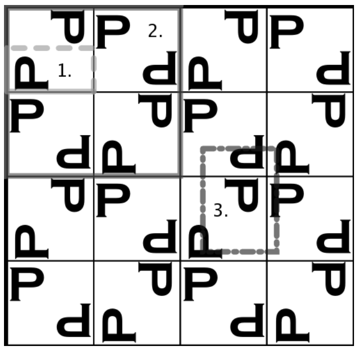


Figure 5. The three types of symmetry. The image represents a finite subset of the infinite symmetric plane. Box 1 (half solid, half dash line) represents the asymmetric symmetry. Box 2 (using a solid line) represents the translation-independent symmetry. Box 3 (using —. line) represents a non-symmetric subset. In theory algorithms that operate on the asymmetric unit should be faster because they cover less data; in practice, the cost of discovering the proper symmetry operator to map the data back into the asymmetric unit more than offsets this.

Each iterator type is useful depending upon the algorithm in question. For example, since negative values are not well defined for electron density, some algorithms [9] set negative density values to zero; this can only be done with the linear-array or grid coordinate systems, that provide mutability. Real-space refinement, a method to make the modeled protein fit better into the electron

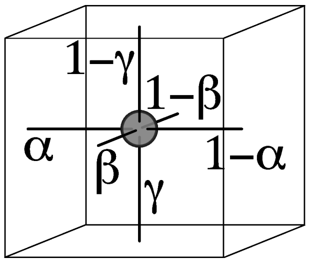


Figure 6. The eight point interpolation algorithm. (1) Given an arbitrary point in space (the disc), (2) find the eight surrounding grid iterators/coordinates, and (3) measure the distance from the given point to the "lower-left" iterator. The distances—α, β, γ—are used to weight the values of the eight coordinates to find the linearly interpolated value. Finding the eight surrounding points may require finding symmetric copies which are possibly "far away" in the the underlying data, necessitating costly computations. For the non-symmetric case no computations are needed, for translation-independent data the computations are a modulus operations, but for asymmetric data searching a list of 4×4 rotation-translation matrix operators is required. Because TEXTAL and CCTBX do not have efficient algorithms to find the operator, asymmetric algorithms are almost always slower than their translation-independent equivalents, even though there is less data.

density data, is a critical technique in PX that relies heavily on the cartesian coordinates. This is because cartesian coordinates have an intuitive notion of distance and direction [6, 11]. Translation-independent and asymmetric symmetries, which are most naturally expressed in the fractional coordinate system, are useful in isolating a unique model [2].

4.3.2 Density Interpolation Algorithm

The density interpolation (DI) function is a critical algorithm used in PX. It computes a electron density value at an arbitrary coordinate in space based on the known stored electron density values surrounding that coordinate. The DI algorithm is invoked, e.g., in the inner loop of the real-space refinement algorithm [6] that fits a model into the electron density data, and must therefore be efficient. This necessitates several different iterator schemes for accessing data in the electron density container. In particular, the non-symmetric symmetry can dramatically benefit from using the linear-array coordinate system.

The iteration selection framework gives access to many iterator schemes, which allowed us to implement an efficient DI algorithm in a generic fashion. The following pseudo-code outlines the computation of an interpolated electron density value at a given coordinate:

```
INPUT: coordinate, P; electron density, E
OUTPUT: linearly interpolated value, V
grid_point ←convert_to_grid_coordinate_system (P)
grid_iters[8] ←get_iterators_surrounding (grid_point)
values[8] ←get_values_of (grid_iters)
distances[6] ←get_distances_to_grid_iters (grid_point, grid_iters[0])
V ←linearly_weight_values_based_on_distance (distances, values)
```

The algorithm differentiates between a grid coordinate system *point*, and a grid coordinate system *iterator*. The former is some arbitrary vector in 3-dimensional space defined by the grid coordinate system. The latter is a dereferenceable and mutable iterator of the grid iterator scheme, pointing to a value stored in memory. To compute the interpolated density, the algorithm first converts the coordinate from its given coordinate system (cartesian, fractional, etc.) to the grid coordinate system. The algorithm then computes the coordinates for the eight grid coordinate system iterators which surround the given point. Then, the values of the iterators are acquired—this can be a non-trivial computation, potentially exploiting the asymmetric, translation-independent, or non-symmetric symmetries. The distance from the point whose value is being interpolated to the "lower-left" grid iterator is calculated, and six weights are computed from the components of the distance as depicted in Figure 6.

4.3.3 Generic Density Interpolation using Iterator Selection

The most efficient way to compute the density interpolation is with the non-symmetric symmetry using the linear-array iterator scheme. Our benchmark for the fastest DI implementation is the TEXTAL function, InterpolateDensity, which makes these assumptions about the data. In addition, TEXTAL's InterpolateDensity is highly optimized: for example, it hand-unrolls all the loops that compute the weighted-average. To acquire the values of the grid coordinates surrounding the point to be interpolated, the TEXTAL code converts the lower-left grid coordinate to a linear-array iterator, then uses pre-computed offsets to find the other seven surrounding grid points, similar to Figure 7. By using precomputed offsets, the algorithm saves the recomputation of linear offsets into the underlying linear-array for the other seven grid iterators.

The TEXTAL InterpolateDensity code is dependent upon the particular selection of symmetry. Furthermore, different stages of the algorithm use different iterator schemes. TEXTAL encodes the concrete iterator types directly into the InterpolateDensity function, which is a non-generic function that only works with the TEXTAL's data structure representing the electron density map.

As described above, the interpolation algorithm consists of two parts: acquiring the values of the surrounding points, and computing the value in the current coordinate as a weighted average of the values of the surrounding points. The former part must be encoded differently for different symmetries, the latter part works for any symmetry. In order to make the code generic, we must first factor the symmetry-dependent value-acquisition code out of the symmetry-independent weight-averaging code. The weight-averaging code uses only the grid iterator scheme. The value-acquisition code uses different iteration schemes depending upon the symmetry. In our implementation of the value-acquisition code in Figure 7 for non-symmetric data, we use the linear-array iteration scheme. To acquire a linear-array iterator we use our framework with the tag linear. We then convert the input grid coordinate to an offset with the function linearize.

The resulting code is nearly identical to the TEXTAL code, except for the call to acquire the alternate iterator scheme. Without the ability to specify the linear-array iterator scheme we would be unable to take advantage of the "pre-computed offsets" optimization described above. This very localized change allows us to write both symmetry-independent and generic code, making the algorithm usable with any electron density container: the algorithm is parametrized over the coordinate type, the electron density container, and the symmetry type. This means that the client can pass in a point from any coordinate system (cartesian, fractional, grid, linear), and implicitly or explicitly specify any type of symmetry (non-symmetric, translation-independent, asymmetric), and get an interpolated value.

We compared our implementation to the density interpolation algorithm to that of TEXTAL's; to the comparable but less optimized standard density interpolation routine in CCTBX called nonsymmetric_eight_point_interpolation; and to the CCTBX's routine basic_map::get_value which is more general than the above two. The CCTBX's basic_map::get_value function supports different symmetries in the value acquisition through the use of vir-

```
   ALGORITHM: get_corner_values
   INPUT: Electron Density, E; Grid Coordinate, X; Values, values
 3 data ←begin<linear>(E)
   data += linearize(X)
   values[0] ←*data
 6 values[1] ←*(data+1)
   data += stride(E,0)
   values[2] ←*data
 9 values[3] ←*(data+1)
   data += stride(E,1)
   values[6] ←*data
12 values[7] ←*(data+1)
   data −= stride(E,0)
   values[4] ←*data
15 values[5] ←*(data+1)
```

Figure 7. Part of our implementation of the electron density interpolation algorithm. Here, we acquire the values of the eight surrounding coordinates of a given point. We use the iterator selection framework to choose the linear-array iterator for this purpose (see line 3). This code uses pre-computed offsets stored in the electron density container, and accessed by the function stride. The iterator is offset by the linearized grid coordinate X to get to the value in the lower-left corner; the other seven values are calculated from pre-computed offsets into the linear-array. Using the linear-array iterator, instead of a grid iterator, dramatically boosts the speed of this implementation. Except for a change to line 3, this code is nearly identical to the corresponding part of TEXTAL's InterpolateDensity function.

tual functions and overloading. Using a dual-processor, hyper-threaded, 3.04 GHz Pentium IV, with 2 GB of RAM, our generic C++ implementation of the algorithm for density interpolation runs about 5–10% faster than TEXTAL's InterpolateDensity. The speed increase is due to suggestions for additional optimizations—which we implemented—in the TEXTAL code-base. Compared to CCTBX's nonsymmetric_eight_point_interpolation routine, our implementation is about 3–5 times faster, and it is about 30 times faster than CCTBX's basic_map::get_value. We have written the necessary adaptors to use our generic algorithm with TEXTAL's emapT C-**struct**, and with CCTBX's numerous electron density container implementations.

5. Conclusion

The proposed iterator request framework allows containers to provide multiple iterator schemes in an easily accessible way. The iterator scheme is requested using a tag class, and not by a dedicated function name corresponding to the iterator scheme. Thus, the iteration scheme can be a parameter in a generic algorithm; a generic algorithm does not have to hard-wire the iterator scheme it uses. A single algorithm can even use several iteration schemes from the same container. Selecting a different iteration scheme becomes a simple change to the tag used to request the iterator.

The motivation for the framework comes from our experiences with implementing generic algorithms for the domain of computational protein crystallography. In that domain, the majority of the generic algorithms operate on containers, rather than pairs of iterators. Moreover, the containers support numerous iteration schemes, the choice of which has a dramatic impact on performance of the algorithms. The framework allowed us to write our algorithms in a fully generic way, and apply the most appropriate iteration scheme for each calling context. We found the framework crucial for writing efficient code for the complex data-structures in the domain.

The iterator selection framework is a generalization of the family of functions, such as begin, end, rbegin, rend, where a fixed signature is used to refer to an iterator scheme. Abstracting the iterator scheme in the begin and end functions is a new and beneficial axis of parametrization in generic libraries. Essentially, by varying the iteration scheme, we can get drastically different behavior and performance from the exact same algorithm or code.

In this paper we discussed several iterator schemes and their corresponding tags in context of isolated examples. Our next step is to continue to analyze the iterator tags and identify a set of generally applicable and "standardized" tags and their refinement hierarchy. We believe that with a full "concept analysis" a small number of useful tags, analogous to the STL's iterator hierarchy, can be developed.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[2] P. D. Adams, R. W. Grosse-Kunstleve, L.-W. Hung, T. R. Ioerger, A. J. McCoy, N. W. Moriarty, R. J. Read, J. C. Sacchettini, N. K. Sauter, and T. C. Terwilliger. *PHENIX*: building new software for automated crystallographic structure determination. *Acta Crystallographica Section D*, 58(11):1948–1954, Nov 2002.

[3] M. H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[4] A. T. Brünger. *X-Plor Version 3.1: A System for X-Ray Crystallography and NMR*. Yale University Press, 1993.

[5] K. Cowtan. The clipper project. *Joint CCP4 and ESF-EACBM Newsletter on Protein Crystallography*, 40, 2002.

[6] K. Gopal, T. Romo, E. Mckee, K. Childs, L. Kanbi, R. Pai, J. Smith, J. Sacchettini, and T. Ioerger. Textal: Automated crystallographic protein structure determination. *Proceedings of the Seventeenth Conference on Innovative Applications of Artificial Intelligence*, pages 1483–1490, 2005.

[7] R. W. Grosse-Kunstleve, N. K. Sauter, N. W. Moriarty, and P. D. Adams. The computational crystallography toolbox: crystallographic algorithms in a reusable software framework. *J. Appl. Cryst.*, 35:126–136, 2002.

[8] T. Hahn. *International Tables for Crystallography, Volume A: Space Group Symmetry*. Spring, 2002.

[9] T. R. Ioerger, T. Holton, J. A. Christopher, and J. C. Sacchettini. TEXTAL: A pattern recognition system for interpreting electron density maps. In *Proceedings of AAAI*, pages 130–137.

[10] T. R. Ioerger and J. C. Sacchettini. Automatic modeling of protein backbones in electron-density maps *via* prediction of c$^{\alpha}$ coordinates. *Acta Crystallographica Section D*, 58(12):2043–2054, Dec 2002.

[11] T. R. Ioerger and J. C. Sacchettini. Textal system: Artificial intelligence techniques for automated protein model building. *Methods in Enzymology*, 374:244–270, 2003.

[12] G. Rhodes. *Crystallography Made Crystal Clear, Third Edition : A Guide for Users of Macromolecular Models*. Academic Press, 2006.

[13] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, Apr. 1994. `http://www.hpl.hp.com/techreports`.

[14] B. Stroustrup. *The C++ Programming Language (Third Edition and Special Edition)*. Addison-Wesley Publishing Co., New York, NY, USA, 1997.

Pound Bang What?

John P. Linderman
AT&T Labs–Research
jpl@research.att.com

ABSTRACT

The author and other data mining researchers at AT&T run `perl` scripts and their attendant library modules on web sites with different architectures and operating systems. On most of these sites, we are not at liberty to modify `perl` or the collection of modules considered "standard". On at least one of these sites, we maintain separate *test* and *production* environments. The challenge is: To what extent is it possible to run the same scripts and library modules, *without modification*, in all these environments? By employing a site-specific *wrapper* command instead of invoking `perl` directly, we have been able to hide the differences among the environments, making the scripts and libraries more portable and often more efficient as well. The wrapper concept has made it possible to react quickly to changes made by site administrators. The implementation is simple and the benefits are not limited to library modules written in `perl`.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*portability*

General Terms

Standardization, Languages, Design

Keywords

Perl, portability, modularity

1. THE PROBLEM

As part of a program of research in data mining, we maintain a number of web pages, primarily written in `perl`, on a web site associated with AT&T Research. A web server was already operating there, and it was accessible to the AT&T intranet. But most of us do our research and program development on other machines, often personal computers where we have complete control. The location of the `perl` interpreter and our module libraries is often different on our personal machines and the research website, so when we moved scripts from our personal machines to the web server, we had to change a few lines in the script to accommodate the differences.

As business units came to rely on the web services, we realized that we needed a backup system for the research server, which is subject to occasional down time. This *production server* is more reliable, but less accessible to researchers, and effectively beyond our control. This reliable server uses a more structured software release process, with separate *test* and *production* hierarchies on the same server. Further changes were required as the scripts moved from the research website to the test environment, and then again to the production environment.

Such changes were largely mechanical, but they were a nuisance at best, and if the changes were not made correctly, the scripts would fail. Because the scripts were known to differ from site to site, one could not verify that the "same version" was in two places by such simple mechanisms as comparing two checksums. What could be done to make it possible to run the scripts *without modification* in all the different environments?

2. COMMAND EXECUTION

Both the problem and our solution are related to the way that commands are identified and executed in operating systems derived from `UNIX`. We will therefore take a detailed look at command execution.

The basic `execve` system call for executing a command takes three arguments, a *command file name*, an array of arguments to be passed to the command, and an array of strings that constitute the *environment* in which the command is executed. Significantly, the environment is inherited by all other commands that the original command might invoke. The strings in the environment are, by convention, of the form

```
NAME=value
```

One of the most important environment variables is `PATH`, whose value is a list of directory names separated by colons. Most shells, and some commonly used front ends to `execve`, look at the *command file name*, and, if the name does not include a directory component, look through the directories in the `PATH` list for the first that contains an executable

OOPSLA'06 Portland, Oregon, October 22-26, 2006

file with the given name. The complete path name is then passed to the `execve` system call.

For example, given

```
PATH=/bin:/usr/bin:/usr/local/bin:.
```

then a command file name of

```
/usr/local/bin/perl
```

will be used, unmodified, because it already specifies the directory containing the `perl` command. However, if the command file name is simply `perl`, then directories in the `PATH` list are searched, in the order in which they occur, for an executable file named `perl`. Assuming `/bin` contains no such file, but `/usr/bin` does, then

```
/usr/bin/perl
```

will be the command name passed to `execve`. Many of the front ends to the `execve` system call do not explicitly mention the environment array. Whatever environment array is in effect when the front end call is made is passed along to the `execve` call.

The first argument in the array of arguments passed to the executed command is, by convention, the name of the command itself. Most executed commands ignore this first argument except, perhaps, for display in diagnostic messages. We will see another use for this argument later.

The first argument to `execve`, a command file name, **f**, must either be a compiled "binary" file, or an ordinary "text" file whose first line begins with the characters `#!`. Binary files are simply executed, with the arguments and environment discussed above. Files starting with `#!`, usually pronounced "pound bang", are more interesting.

2.1 execve and #!

If executable text file **f** begins with the characters `#!`, the first term following the `#!` is treated as the name of an *interpreter* command. `execve` does no `PATH` searching, so the command name is usually a full path name. We will refer to the interpreter command file as **interpreter**. The effect of `execve` is very much like what would happen if

```
interpreter f arguments-to-f
```

were executed by a typical shell. That is, the command file name is passed as an argument to the interpreter identified in that command file, and any arguments to the original command are passed along following the command file name. A single, optional, argument **a** can follow the interpreter name on the `#!` line, and it will result in something similar to shell command

```
interpreter a f arguments-to-f
```

When `execve` is invoked by a shell, it can be difficult to determine who is doing what. So to completely understand `execve` and `!#`, let us eliminate the intervention of any shell. We start with a very simple `C` program, `args.c`.

```
#include <stdio.h>
int main(int argc, char *const argv[])
{
    int i;
    for (i = 0; i < argc; ++i) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

It simply prints *all* of its arguments, including the often ignored first argument. We will assume it has been compiled into file `args` in the current directory. If we run

```
./args a b c
```

the result will be

```
./args
a
b
c
```

This is really just telling us what the shell put into the argument array that ended up in an `execve` call. To eliminate the actions of the shell, we employ another very simple `C` program, `invoke.c`,

```
#include <unistd.h>
int main(int argc, char *const argv[])
{
    char *array[] = {
        "command",
        "arg1",
        NULL
    };
    char *env[] = { NULL };
    execve(argv[1], array, env);
}
```

`invoke` just passes an argument vector of known contents and a completely empty environment to the command file whose name is supplied as its first argument. If we execute

```
invoke args
```

we get

```
command
arg1
```

as we would expect. The shell orchestrated the execution of `invoke`, but we know exactly how `invoke` executed command `args`. Finally, we create an executable text file

```
args.int
```

containing just one line

```
#!  args  -x    # comment?
```

It is usually a bad idea to use a relative path name following a `#!`, but we know that a command named `args` is in this directory, so we can get away with it. If we now execute

```
invoke ./args.int
```

we get

```
args
-x    # comment?
./args.int
arg1
```

So the low-level effect of calling `execve` with a command file starting with `#!` is to *replace* the first argument array entry with the original command file name, pushing the interpreter file name (and optional argument, if any) onto the argument array, and then executing the interpreter with this argument array.

Note that it is the kernel[1], not a shell, that is processing the `#!` line. White space between the `#!` and the interpreter

[1]The `!#` construct of `execve` is not directly supported by Microsoft operating systems. It can be made available by using `UNIX` compatibility packages such as `UWIN`[2]. Even where `!#` is not supported, our wrapper can be invoked explicitly, with many of the advantages we discuss.

name is ignored, as is white space between the interpreter name and optional argument. But the argument is *everything* else that follows the interpreter name, untouched, as a single argument, with no notion of comments or multiple arguments. An interpreter *might* know what to do with an argument containing embedded white space, but most familiar interpreters will not. If anything follows the interpreter name, it is usually just a single token.

While we have the tools at hand for inspecting low level operations, consider

```
ln -s args.int argsym
invoke argsym
```

This creates a *symbolic link* to `args.int` from `argsym`, so a reference to `argsym` turns into a reference to `args.int`. We then invoke the command via this symbolic link. The result is

```
args
-x    # comment?
argsym
arg1
```

Although we know it is the `args.int` command that is actually executed, it is the name by which it was invoked, `argsym`, that appears in the `execve` argument list. For our purposes, this will be very convenient.

3. POUND BANG WHAT?

One of the items that forced us to change scripts as we moved them from site to site was the location of the `perl` interpreter. Although it was often found in `/usr/bin/perl`, this was not always true, and on some sites, `/usr/bin/perl` was a release that was too old to support all the language constructs our scripts relied on. In some places, the scripts started with

```
#!/usr/bin/perl
```

in others

```
#!/usr/local/bin/perl
```

and in still others

```
#!/usr/common/bin/perl.new
```

On sites under our control, we could install a suitable release of `perl` in `/usr/local/bin/perl`, but we did not control all the sites where the scripts would be running. Understandably, system administrators on shared systems take a proprietary view of the contents of "standard" directories like `/usr/bin` or `/usr/local/bin`. They support many communities of users, and no single community can dictate, for example, what release of `perl` is installed in `/usr/local/bin`. We cannot use the "real" location of `perl` in the `#!` line if we hope to run the scripts without modification on all machines.

So we started with the premise that we are more likely to achieve our goals by requesting the creation of a normal home directory, say `/home/vip` for our *very important project*, rather than negotiating for names in standard command directories. Of course, it may be that `/home/vip` is already in use on a machine we would like to port to, or that `/home` is not where home directories reside on the machine in question. In that case, scripts will have to change. But the string `/home/vip` is sufficiently unusual that we can probably get away with wholesale substitution for it, replacing it with a path to a home directory we *can* control. If we cannot achieve absolute portability, minimizing and simplifying the changes is a good second best.

So we will assume that `/home/vip` is the root of a hierarchy we can control, without special permission from the administrators. And we will boldly go ahead and assume that `/home/vip/wrapbin/perl` is going to appear on our `#!` lines, and see where that takes us.

We do not have to do a complete `perl` installation in the `/home/vip` hierarchy, although that is not terribly difficult if no acceptable release of `perl` is already available. We can make a symbolic link to some suitable release of `perl`, for example

```
ln -s /usr/bin/perl /home/vip/wrapbin/perl
```

The symbolic link will reference different commands on different sites, but that is not visible in the `#!` line, which we have now standardized.

4. PERL LIBRARY MODULES

That is not a bad start, but our scripts also rely on `perl` library modules that are not part of the standard `perl` distribution. We *could* (and, initially, did) begin each script with a

```
use lib qw( /home/vip/lib );
```

which causes `perl` to look for modules in the `/home/vip/lib` directory before it searches the standard `perl` library directories. But now we have put `/home/vip` into each script twice, and we would prefer not to sprinkle funny names about any more widely than necessary. More to the point, this does not address the need to have a test and production environment on the same machine. Presumably, test versions of some modules differ from the production versions. So a fixed directory will not support two or more environments on the same machine.

Speaking of "environments", though, we can use the

```
PERL5LIB
```

environment variable to determine where non-standard modules will be looked for. Like the `PATH` environment variable, which specifies a list of directories where a command file might be found, `PERL5LIB` specifies a list of directories where `perl` will look for modules. Assume we create `Test` and `Prod` directories under `/home/vip` for the test and production environments. We can put all the production-quality modules under `Prod/lib`, and set environment variable

```
PERL5LIB=/home/vip/Prod/lib
```

for the production environment. We can put test versions of modules into `Test/lib`, and set

```
PERL5LIB=/home/vip/Test/lib:/home/vip/Prod/lib
```

in the test environment, so we preferentially pick up test versions but find standard versions in the production environment, if there is no test version present.

Now we can get rid of the `use lib` in all our scripts. If one of our `perl` scripts invokes other `perl` scripts via backticks or the `system` command, these commands inherit the environment with the extended `PERL5LIB` variable, so they, too, will find the appropriate library modules.

5. MORE ENVIRONMENT VARIABLES

Our scripts invoke commands other than `perl` scripts. Like `perl` itself, these may be in different places on different sites, so we carefully avoid full path names. Anticipating the need to have distinct test and production versions of these, as well, we will want separate `PATH` environment variables for these environments.

```
PATH=/home/vip/Prod/bin:...
PATH=/home/vip/Test/bin:/home/vip/Prod/bin:...
```

`/home/vip/Prod/bin` serves two purposes. It is a repository for commands that we write, and it can hold symbolic links to pre-existing commands in much the same way that we have (so far) used

```
/home/vip/wrapbin/perl
```

to select a suitable release. It is not necessary to make a symbolic link for *every* command we might invoke. Judicious selection of the other directories in the search `PATH` will often result in the "right" command being found by default. For example, if our production `PATH` starts with

```
/home/vip/Prod/bin:/bin:/usr/bin:...
```

then any commands found in `/bin` or `/usr/bin` do not have to be linked into our production `bin` if the standard search would find the command we wish to invoke.

What is true of `PATH` and `PERL5LIB` also applies to load library paths, and command-specific environment variables. We have a whole collection of environment variables that might have to be set to distinguish the test and production environments. That is not such a horrible price to pay to make the scripts and modules totally portable. But it is a bit clumsy. You would like to have the script do something sensible and predictable, like default to the production environment, if the script were invoked without having done the environment variable setup. And what if we want to have different releases of `perl`, itself, in the test and production environments? We have hard-wired

```
#!/home/vip/wrapbin/perl
```

into our scripts. How do we distinguish between the test and production releases?

6. WRAPPERS

Maybe `/home/vip/wrapbin/perl` could be a tiny shell script that looks at something in the environment, sets the entire collection of environment variables accordingly, and then, with `PATH` already set appropriately, invokes `perl`, allowing the `PATH` to determine which release will be selected.

```
#!/bin/sh
export PATH=/home/vip/Prod/bin:/bin:/usr/bin ...
export PERL5LIB=/home/vip/Prod/lib
 ...
if test -n "$TESTING"
then
       PATH=/home/vip/Test/bin:$PATH
       PERL5LIB=/home/vip/Test/lib:$PERL5LIB
        ...
fi
perl "$@"
```

This would package up the environment variable setting, and make it possible to have different releases of `perl` accessible. This does not *quite* work, because the command following a `#!` must be a compiled command, not another `#!` command. However, it is simple enough to compile a tiny *wrapper* that invokes a shell script like the one above. The only downside is that the invocation of the wrapper to invoke the shell to invoke the real command adds to the overhead of executing the real command.

7. PERFORMANCE

The command

```
perl -e 0
```

simply evaluates the expression `0`. If we measure how long it takes to run, what we are really measuring is the time it takes to get `perl` started. We ran this baseline test with three different commands. One was the actual `perl` interpreter we use on the machine where the tests were run. We will refer to this as the *unwrapped* invocation. Another was a compiled wrapper we use on our website. It sets a number of environment variables, then executes `perl`, which, given the `PATH` it establishes, is the same interpreter used in the unwrapped invocation. We will call this the *C wrapper*. The third command is a tiny `C` program that executes a shell script similar to the one in the previous section, the *shell wrapper*.

100,000 iterations of each command took between 150 and 280 seconds on the machine we used for the measurements. The unwrapped version was fastest, averaging about 154 seconds over ten such runs. The `C` wrapper was second, with an average around 187 seconds, and the shell wrapper came in around 279 seconds. Absolute times vary from machine to machine, but the ratios are more consistent. The `C` wrapper adds about 20%, the shell wrapper about 80%.

Whether a few milliseconds matter depends on the nature of the application. If the `perl` scripts run for more than a second, the startup overhead is obviously insignificant. If the scripts run for only a fraction of a second, the overhead matters more. The test, doing as little as it did, is a worst-case scenario.

In fact, we discovered that the wrappers can actually improve performance. Many of our scripts used the `perl POSIX` library module to extract information about the host they were running on. For example

```
use POSIX qw( uname );
$ENV{HOST} = (uname())[1];
```

sets the `HOST` environment variable from the second element of the array returned by the `uname` routine. Suppose we modify the shell wrapper by adding

```
export HOST=`hostname`
```

This accomplishes the same thing, so we can compare the modified shell wrapper executing the `0` expression, and the unwrapped version invoking `uname` as shown above. The `C` wrapper was already setting `HOST`, so there is no need to rerun it. With these modifications, the shell wrapper times increased to an average of around 391 seconds, but the unwrapped times jumped to 795 seconds.

The `POSIX` module is large, nearly 19,000 bytes. Even though we need only one method from the module, the entire module must be read and parsed. By eliminating the need to use the module, we more than pay for the additional overhead of the wrappers.

8. BEYOND PERL

With the basic "stuff the environment and run" functionality in place, it becomes evident that the wrapper is useful for commands other than `perl`. Establishing a special `PATH` is useful to *any* script that may invoke other commands, so one might like to have a "wrapperized" `awk` and `python` and so on. We do not have to construct a different wrapper for each command. The `C` wrapper, for example, uses the *basename* of the command name, dropping any directories that might have been included in the name, as the default command name to be executed. After making all the changes to the environment, the last thing it does it to call `execvp`, the `PATH`-searching front end to the `execve` call, with just this basename as the command name. If we invoke the `C` wrapper as

```
/home/vip/wrapbin/perl
```

then the basename is `perl` and we end up invoking whatever version our `PATH` determines. If we simply make a symbolic link to

```
/home/vip/wrapbin/perl
```

from

```
/home/vip/wrapbin/awk
```

then we can use this `awk` command in our `#!` lines and enjoy all the environment variables, like `PATH` and `HOST`, set by the wrapper. We could do the same for other interpreters, like `python`, but to do a proper job, we would want to set interpreter-specific variables like

```
PYTHONPATH
```

which is the `python` equivalent of `PERL5LIB`. There are `perl` variables that are of no interest to `python`, and vice versa. This results in a certain amount of "environment bloat", but as long as there is no *disagreement* about the meaning of any given environment variable, it is unlikely to be noticed, and we will see later why it is advantageous to have a single wrapper rather that one wrapper per interpreter.

We built our `C` wrapper so that if variable

```
WRAP_COMMAND
```

is present in the environment, its value will be used in preference to the basename.

```
WRAP_COMMAND=awk /home/vip/wrapbin/perl awk-script
```

will therefore invoke `awk` without the need to create a symbolic link, and

```
WRAP_COMMAND=printenv /home/vip/wrapbin/perl
```

is a favorite way to see the environment variables that the wrapper sets. This relies, of course, on an executable version of the command being found in the search `PATH`. The wrapper can set a `umask`, for example, but if you try to see what it is by invoking

```
WRAP_COMMAND=umask /home/vip/wrapbin/perl
```

it will not work, because `umask` is a shell builtin, not an executable binary.

Similar features can be added to the shell wrapper, but they require changes to, and symbolic links to, both the compiled wrapper and the shell scripts it invokes.

8.1 Symbolic Links and Hard Links

There are two kinds of links we can use to give alternative names to the wrappers. Where *symbolic* links store a path to another file, *hard* links assign another name to an existing file. There is a tiny bit of extra overhead for the operating system to chase the path associated with a symbolic link. So why use symbolic links where a hard link could be used?

When we want to modify the wrapper, it is important to do so *atomically.* That is, commands referring to the wrapper must see the old version or the new version, but not some file where part of the new version has been copied, but part of the old version still remains. The shell command

```
mv new-file file
```

employs the `rename` system call, which is atomic. The atomicity is achieved by changing the directory entry for `file` to reference a new file. No file copying is involved.

If we have been using symbolic links, the `rename` atomically changes all the references. However, if `file1` is a *hard* link to `file`, then after the `rename`, `file1` continues to be a name for the *old* version. The old version continues to be a valid executable command, but it may be that the old and new versions are logically incompatible, for example, by using different formats for some log file. What is more, there is no atomic way to make `file1` a hard link to the new `file`. `file1` must first be removed, then linked, and there is a "window", however tiny, between the removal and the linking where a command referencing `file1` would fail. By using symbolic links, we make atomic update easy.

9. TEST VERSUS PRODUCTION

Our `C` wrapper makes it easy to do conditional environment variable assignments based on the value, if any, of environment variable `WRAP_SELECT`. Our convention is that the production environment is to be used unless `WRAP_SELECT` has a value containing the letter `t`. This means it is easy to exercise the test environment by setting `WRAP_SELECT=t`. This is fine for interactive testing, but we do not always have complete control over our environment, for example, when a script is invoked from a website.

Recall that the wrapper has access to the name of the command in which the wrapper name followed the `#!`. Our wrapper looks for an occurrence of the string `/test/` in that command name, and sets `WRAP_SELECT` to `t` if the string is found. The wrapper can use the `getcwd` function to determine the name of the directory in which it is executing, and our wrapper performs a similar check for the directory name. The identity of the user executing the command could also be checked.

It is therefore not necessary for `WRAP_SELECT` to be set when the wrapper is invoked. The wrapper can check other aspects of the broader environment in which it finds itself to set the variable, as appropriate.

10. UNEXPECTED BENEFITS

When we made our latest move to a new web server, the system administrators decided to use a new mechanism, `cgi-wrap`, to execute user web scripts. Simple test scripts ran correctly under `cgi-wrap`, but in actual use, we noticed that some long-running scripts were exiting before they completed, and large files were being truncated. We discovered that `cgi-wrap` was setting resource limits to constrain runaway scripts. The system administrators were very cooperative, and offered to raise the limits. But a few of our scripts spawn jobs that may run for days, and write files that are hundreds of megabytes. Limits large enough to accommodate these jobs would not do much to constrain ordinary scripts.

Instead, we modified the `cgi-wrap` source so that "soft" limits were set, but "hard" limits were not. This opened up the possibility of resetting the soft limits. We then modified our wrapper to remove the limits that `cgi-wrap` imposed. No modifications were needed to the individual scripts, and scripts that did not employ the wrapper inherited limits that were quite reasonable for ordinary web applications.

We also discovered that some of the redirects to the new website were interacting badly with the `CGI perl` module, resulting in "not found" errors when users submitted forms. We found that we could correct the problem by performing a string substitution on the `SCRIPT_NAME` environment variable. We modified a few scripts to get the users back on the air. But then we realized that the changes were easy to do in the wrapper. By making the changes there, we fixed the problem for all scripts, without modifying them directly.

11. OTHER POSSIBILITIES

The benefits described in the previous section are due, in part, to the wrapper acting as a "executable point of contact" for all scripts. Changes made there take effect in every script that invokes the wrapper, without having to change the scripts.

One could do performance monitoring by bracketing the final `execvp` system call with a `fork` and a `wait`, and then logging the results of a `getrusage` call. Such logging could be turned on and off dynamically, based on script name or time of day or day of the week.

The real joy is that it is not necessary to anticipate all the things you might want to do. You can add (and remove) functionality as inspiration dictates, and the changes come and go, globally and instantaneously, as you install various implementations of the wrapper.

12. RELATED WORK

12.1 Commands that Invoke Commands

Commands whose function is to make inheritable changes to the environment and then execute a different command in that modified environment are not new. The `nohup` and `nice` commands are examples from the earliest days of the `UNIX` operating system.

```
nohup nice sort -o out largefile &
```

might be used to sort a large file at reduced processor priority (`nice`) and to ignore the signal that will be sent if the session ends before the sort is complete (`nohup`).

Conventional wrapper commands like `nohup` take, as arguments, the name of the command to be executed, and the arguments to that command. This makes them unusable with the `#!` construct, where arguments are limited in number and length. Our wrapper usually determines the command to be executed from the name by which the wrapper is invoked. It can only invoke commands whose basenames are also a name for the wrapper.

The commands invoked by a conventional wrapper are designed to do something sensible in the absence of any wrapper. Our wrapper is associated with a suite of commands that anticipate operating in environments where no single default will be effective everywhere. The suite *relies* on the wrapper to configure a suitable environment for execution.

12.2 Configuring for Multiple Environments

There are configuration tools like `autoconf`[3] and `iffe`[1] that facilitate porting commands to disparate environments. The model is to configure a single command once, using a combination of user-specified options and automatic detection of system-specific differences. These tools could be used to configure a suite of commands for a given machine, but they fail to address the need to have different test and production environments on a single machine. Configuration tools might be valuable for porting our wrapper, particularly if the nature of the wrapper were more formally expressed. But we would not choose to configure individual commands, even if that were possible. We like the single, executable, point of contact that a wrapper provides.

13. SUMMARY

The use of a small, compiled *wrapper* can contribute significantly to the portability and testability of scripts. The additional overhead will be insignificant for most applications, and performance *improvements* have been observed. By acting as an *executable point of contact* for all scripts, a wrapper provides wholesale protection from changes in the operating environment, and offers opportunities for dynamic logging and measurement.

14. REFERENCES

[1] Glenn S. Fowler, David G. Korn, John J. Snyder, and Kiem-Phong Vo. Feature-based portability. In *Very High Level Languages Symposium (VHLL)*, pages 197–207, October 1994.

[2] David G. Korn. UWIN — UNIX for Windows. In *The USENIX Windows NT Workshop 1997*, pages 133–145, 1997.

[3] David Tilbrook and Russell Crook. Large scale porting through parameterization. In *USENIX Conference Proceedings*, pages 209–216, Summer 1992.