Towards an Automatic Complexity Analysis for Generic Programs

Kyle D. Ross

Open Systems Laboratory
Indiana University
kyle@osl.iu.edu

Abstract

Generic libraries, such as the C++ Standard Template Library (STL), provide flexible, high-performance algorithms and data types, along with functional specifications and performance guarantees. It is left as a nontrivial task to the library user to choose appropriate algorithms and data types for a particular problem. In the present paper, we describe an automatic analysis to aid library users in selecting types for use in generic programs. The analysis is based on manipulation and comparison of symbolic complexity expressions, constructed using cost-bound functions and abstract interpretation of program behaviour. Our analysis serves to detect and rectify "performance bugs" by recommending type selections that will improve performance.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Languages Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphim; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems—Computations on discrete structures

General Terms Performance, Languages

Keywords Generic programming, library-oriented software, complexity analysis, performance, concepts, abstract interpretation, cost-bound functions

1. Introduction

The C++ Standard [2] defines the *C++ Standard Library*, including the *Standard Template Library* (STL) [3, 22], a generic library providing containers, searching and sorting algorithms, and iterators to connect the two. Generic library designers are concerned with optimising two aspects of their implementations: Genericity and performance. Mechanisms like C++ templates simultaneously satisfy both of these aims by parametrising data types and functions in a flexible, reüsable way that preserves high performance of the instantiated code.

Pursuant to the focus on performance, the Standard defines upper-bound performance requirements for data structures and algorithms in addition to functional requirements. These informal performance descriptions stipulate guarantees that any Standard-compliant STL implementation must satisfy, allowing the library's user to consider performance implications when making decisions about data types and algorithms.

One of the most important aspects of generic programming is that algorithms and containers are decoupled: Algorithms operate on arbitrary containers, including those provided by STL and user-implemented containers; likewise, user-implemented generic algorithms can operate on the full range of STL containers. It is often the case that STL provides multiple functionally-equivalent algorithms and containers that differ with respect to performance. Each is useful on a certain class of problems, and the library user must make choices about which are appropriate for a given program.

For example, sort is usually the function recommended for sorting. It is required to perform "approximately $\mathcal{O}(N \log N)$ ...comparisons on average" [25.3.1.1(2)], but an implementation can meet this requirement with quicksort [13], which has $\mathcal{O}(N^2)$ worst-case performance. The Standard recommends that another sorting algorithm be used if worst-case performance is extremely important: "If the worst-case behaviour is important, stable_sort or partial_sort should be used."

Likewise, one usually achieves optimal performance by using the container vector. This is true so often that many novice STL programmers use vector almost exclusively. One would be misguided in doing so, however, since there *are* cases where vector is asymptotically inferior to, e.g., list. We will consider such a case in this paper (see Section 1.2). Note that it is more difficult to select the proper container than it is to select the right algorithm: Whereas one can choose a sorting algorithm based on local properties, choosing a particular container affects program complexity non-locally. To achieve optimal performance, one must consider the container's use during the *entire* program.

In order to help STL users avoid mistakes that can arise from "vector-is-always-best"-style assumptions, we present a method utilising abstract interpretation and symbolic cost-bound functions to check a generic program for *performance bugs*—poor program performance caused by improper type or algorithm choices. The *type selection problem* can be defined as follows: Given a program p with a type variable c that is constrained to model certain concepts[1] (e.g., to be a BackInsertionSequence), and *type library* T, find a type $t \in T$ that meets the constraints imposed on c such that p achieves the highest possible performance. We will present a method to aid users in solving this kind of component-selection problem.

Although previous work has considered automatic complexity analysis [4, 6, 5, 10, 11, 19, 20, 25, 24, 28, 29], we study the problem in the context of generic programming and at an ab-

WGP'06 September 16, 2006, Portland, Oregon, USA.

[1] The term *concept* is used in generic programming to refer to an abstraction over types defined by a set of requirements that *modelling* types must satisfy. [3]

stract, *implementation-independent* level—we consider only the *specification* of STL rather than any particular implementation. This allows us to make general recommendations applicable to any C++ compiler with any standard-conforming implementation of the STL. Because we perform an analysis at an abstract level, we manipulate symbolic constants that represent particular implementation factors (e.g., the cost of incrementing a certain iterator) in lieu of the actual constants for a particular combination of library implementation, platform, compiler, etc.

In some cases, we will be unable to make a concrete recommendation. Complexities naturally form a partial order (see Section 2.10), and there are cases where two or more containers have incomparable[2] asymptotic complexity in a particular program; we cannot recommend a particular type in such cases. This is acceptable, however, because one is not, even in principle, able to select one such container as better than the others since this is a platform- and implementation-specific matter.

Although we report here on a prototype implementation for C++ and the STL, the approach is suitable for generic libraries in general. Also, we discuss the analysis only for a single type variable, but it can be used to select multiple data types simultaneously.

1.1 Organisation

The remainder of the paper is organised as follows: Section 1.2 presents a motivating example, Section 2 presents our analysis, Section 3 returns to our example with some preliminary results, Section 4 describes related work, Section 5 discusses future work, and Section 6 concludes.

1.2 Motivating example

To describe our work, we present an example adapted from a popular textbook for STL programmers [17]. In this example, we consider the task of filtering failing students from a sequence—given a sequence of students, we remove those who are failing and put these into a separate sequence. Figure 1 shows the C++ implementation of a function extract_fails that performs this task. This function is parametrised (at the type level) by C, the type of the input sequence; this type C must model the concept BackInsertionSequence and hold values of type Student_info. The function extract_fails removes all failing students, where failure is determined by the predicate fgrade, from students and returns a sequence of these failing students, maintaining stable element order for both the original sequence and the sequence returned.

The decision, then, for a programmer who wishes to use extract_fails is to choose a suitable container type C. Because vector is often the most efficient STL data structure (c.f. Section 1), a naïve STL user might assume that vector is the correct choice for this example, but it is not—vector provides a very efficient data structure when elements are added and removed primarily from the end of the sequence, but it performs badly when insertions and deletions occur in the middle of the sequence such as the deletion (erase) in our example.

Our analysis considers three choices for C: list, vector, and deque. After analysing the performance of each container in the context of the example program, it correctly recommends the use of list. It also determines that there is not enough information to recommend vector over deque or vice versa since these data structures have similar asymptotic complexities for the operations used, and it would be impossible to recommend one over the other without considering the details of a particular combination of library implementation and machine. These results correspond to those from an informal benchmark.

[2] Recall from order theory: If we have a partial order relation $\preceq$ and $x \not\preceq y$ and $y \not\preceq x$ then x and y are said to be incomparable. Consider, e.g., $\mathcal{O}(m^2 + n)$ and $\mathcal{O}(m + n^2)$.

Requires: Type C is a model of BackInsertionSequence. Type C's value type is convertible to Student_info.
Effects: Removes all elements from students for which fgrade returns true.
Returns: A container of type C holding all elements removed from students.
Note: extract_fails is stable—the elements remaining in students and the elements in the container returned are in the same relative order as they originally were in students.

```
template <typename C>
C extract_fails (C& students)
{ C fail ;
  typename C::iterator iter =
    students.begin () ;

  while (iter != students.end ())
  { if (fgrade (* iter))
    { fail.push_back (* iter) ;
      iter = students.erase (iter) ;
    }
    else
      ++ iter ;
  }

  return fail ;
}
```

Figure 1. C++ code for our example: An algorithm to filter failing students from a sequence.

2. Methodology

At a very general level, we must do the following to solve the type selection problem: We must be able to represent programs, types, and complexities. Then, using these representations, we must build complexity expressions describing the behaviour of a program with each type of interest. Finally, we can compare the complexities of the instantiations and emit appropriate diagnostic messages about any performance bugs detected by the analysis.

The first step to solving a problem is choosing a proper representation—we must be able to represent the relevant factors about STL programs including information about types, control structures, and state. In the case of our extract_fails example, we need to represent the types of variables, the series of library calls (e.g., students.begin (), iter.operator!= (students.end ())), the C++ control constructs (e.g., while, if), and state (e.g., the size of students).

Additionally, we must represent certain operational and performance descriptions of the STL itself: For example, how particular algorithms alter the sizes of containers, and the complexity of assorted algorithms as functions of the types and values of their arguments. Operational definitions of the library are necessary because the complexity of calling library functions depends on the values computed during execution [27]. We must also be able to represent complexities. In our example, we must represent, e.g., that the complexity of vector<Student_info>.push_back is $\mathcal{O}(1)$ amortised and that it increases the size of the vector by one element.

Finally, we must represent concept information, viz. which types are models of which concepts, for example that vector<Student_info> models the BackInsertionSequence concept and that its iterator type models RandomAccessIterator.

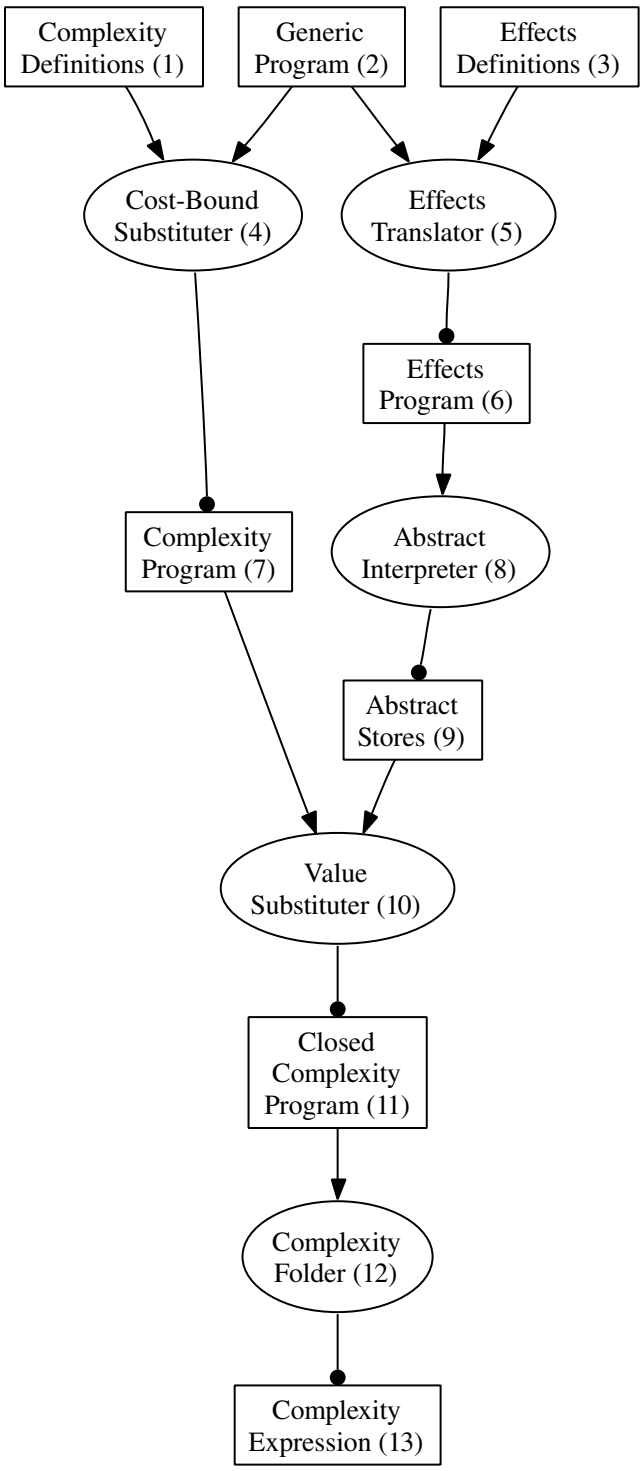

Figure 2. Overview of the complexity analysis.

Once one has represented a program and the relevant library, type, and language information, one can begin the first phase of the analysis. Given the constraints on each type variable (e.g., that C must model BackInsertionSequence) and the library of defined types, one must find all acceptable types with which each type variable can be instantiated. The next step is to replace each library call with an expression representing the complexity of that call. This depends on several factors: The function called, properties of the arguments passed to this function, and the store in which the call is evaluated. After this has been performed, we can produce a symbolic complexity expression for the program by associating folding rules with each language construct, and then recursively applying these rules, which compute the symbolic complexity of program fragment given the complexities of its subterms. For our example, we infer that vector<Student_info>, deque<Student_info>, and list<Student_info> are types that meet the requirements on type variable C. We then must replace each library call with the complexities we represented in the previous step, combine these in a suitable way, and compare the complexities for the three containers to issue recommendations to the user.

By comparing programs' complexities, one can determine the best type (if one exists) to use for a particular type variable and make an appropriate recommendation to a user if some other type is currently selected. In the case where two or more types are incomparable, the analysis can report this to the user, providing choices to consider. For example, if vector and deque are incomparable but each is better than list, the analysis can suggest that the user carefully consider the choice between vector and deque but need not consider list.

Figure 2 shows an overview of the analysis; numbers in parentheses will refer to the labels in this diagram to put each component of the analysis into perspective. The inputs to the analysis are: A generic program (2), complexity definitions for each library function (1), and effects definition for each library function (3).

2.1 Program Representation

$$
\begin{array}{rcl}
< stmt > & ::= & \text{Assign} < id > \Longleftarrow < expr > \\
& | & \text{Temp} < id > \Longleftarrow < expr > \\
& | & \text{Loop} < id > < stmt > \text{End} \\
& | & \text{Branch} < case >^{+} \text{End} \\
& | & < expr > \\
< case > & ::= & \text{Case} < stmt > \text{End} \\
< expr > & ::= & < id > \\
& | & < const > \\
& | & \text{Call} < fId > (< id >^{*}) \\
& | & < mod > . < fId > (< id >^{*}) \\
< module > & ::= & < mId > \\
& | & < mod > . < mId >
\end{array}
$$

Figure 3. Grammar for CSimp, a simple composition language for representing library-oriented programs.

To represent a library-oriented generic program (2), we define a simple composition language [1]. This language must be able to represent: assignments, iterations, conditions, library calls, and (non-library) function calls. For our prototype implementation, we have defined a language, CSimp, the grammar for which is shown in Figure 3. An abbreviated version of the CSimp implementation of our running example is shown in Figure 4. An example input—a library-oriented program viewed as a composition of library components—is shown in Figure 5. Notice that each library call has been numbered; this numbering is used to associate the proper store and arguments with each call.

2.2 Complexity

Next, we replace each library call by a *cost-bound function* (4) [20]—a function that returns bounds on the cost of executing the original function given the same inputs. These functions are declared and are based on the C++ Standard's specification of the STL.

```
c.dflt_cons(fail)

Temp beg_students⟸c.begin(students)
c.iter_t.copy_cons(iter, beg_students)

Temp size_students⟸5
Loop size_students
  Temp end_students⟸c.end(students)
  c.iter_t.!=(iter, end_students)
  Temp st_iter1⟸c.iter_t.deref(iter)
  Call fgrade(st_iter1)
  Branch
    Case
      Temp st_iter2⟸c.iter_t.deref(iter)
      c.push_back(fail, st_iter2)
      Assign iter⟸c.erase(students, iter)
    End
    Case
      c.iter_t.++(iter)
    End
  End
End

c.copy_cons(result, fail)
```

Figure 4. CSimp implementation of the example.

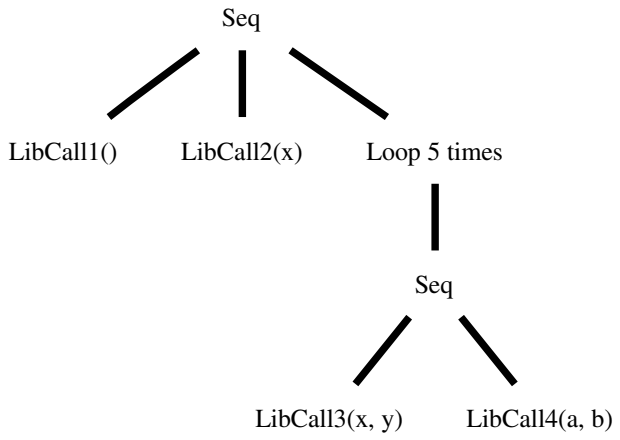

Figure 5. Input to the analysis: A composition of library calls.

2.3 Complexity Representation

In an introductory algorithms course, Big-O ($\mathcal{O}$) notation [16] is often used to describe the complexities of the algorithms studied. While a useful abstraction, Big-O complexities are too imprecise to represent the algorithms and operations provided by STL: In certain cases, the C++ Standard makes requirements on the precise number of operations that an algorithm may (or must) perform; amortised analyses are required to account for memory operation costs when a container must be "physically" relocated.

We have identified that the STL requirements are specified at three distinct levels of precision. *Exact counts* specify precisely how many of a particular kind of operation may or must be performed. *Amortised complexities* [7] require that, on average, each operation in a sufficiently-long sequence has a particular complexity. *Approximate complexities* make requirements on average complexity, but allow for worse complexity in suitably rare cases. Figure 7 illustrates examples of these complexity requirements; angle brackets enclose complexity ranges in the format $<$ best case, worst case $>$.

$$
\begin{array}{rcl}
< comp > & ::= & < comp > \text{ (;) } < comp > \\
 & & \quad \text{Sequence} \\
 & | & < comp > \text{ (.) } < comp > \\
 & & \quad \text{Branch} \\
 & | & < \ < comp > \ , \ < comp > \ > \\
 & & \quad \text{Range} \\
 & | & < comp > \ < op_2 > \ < comp > \\
 & | & < op_1 > \ (< comp >) \\
 & | & \text{constmin} \mid \text{constmax} \\
 & & \quad \text{Symbolic constant} \\
 & | & < expr > \\
 & & \quad \text{Value expression} \\
 & | & (\ < comp > \ ) \\
< op_2 > & ::= & + \mid - \mid * \mid / \\
 & & \quad \text{Operators} \\
< op_1 > & ::= & \text{log} \mid \text{min} \mid \text{max}
\end{array}
$$

Figure 6. Grammar for complexity expressions.

Complexity from C++ Standard	Equivalent complexity expression
Exact operation count: E.g.,accumulate (first, last, init, binary_op): "Computes its result by initialising the accumulator acc with the initial value init and then modifies it with ...acc = binary_op (acc, * i) for each iterator i in the range [first, last) in order." [26.4.1(1)]	λ fst lst init bop . let cbop = compFuncFor(bop) in let elts = fst.container.elts in cbop (init, elts) + cbop(elts, elts) * (lst - fst - 1)
Amortised complexity: E.g., vector<T>::push_back (t): "A vector ...supports (amortised) constant-time insert and erase." [23.2.4(1)]	λ this elt . <constmin, constmax * this.size> (Hand analysis shows that the worst-case complexity of push_back should be linear)
Approximate complexity: E.g., partial_sort (first, middle, last): "It takes approximately (last − first) ∗ log(middle − first) comparisons." [25.3.1.3(2)]	λ fst mid lst . let cmp = (last - first) * log (middle - first) in cmp * <constmin, constmax> (We interpret "approximately" to mean roughly $\mathcal{O}$; the symbolic constants constmin and constmax represent the "approximateness".)

Figure 7. Complexities representation.

To represent complexities for the purpose of our analysis, we define a simple range-based complexity language that designates complexities as pairs of best and worst cases; this is shown in Figure 6. Notice that combinators for sequencing operations (;) and branching (.) are abstract in the sense that they do not commit to a particular "complexity semantics" for these language concepts. Encodings in our language for the examples from the C++ Standard are also shown in Figure 7.

2.4 Complexity Program

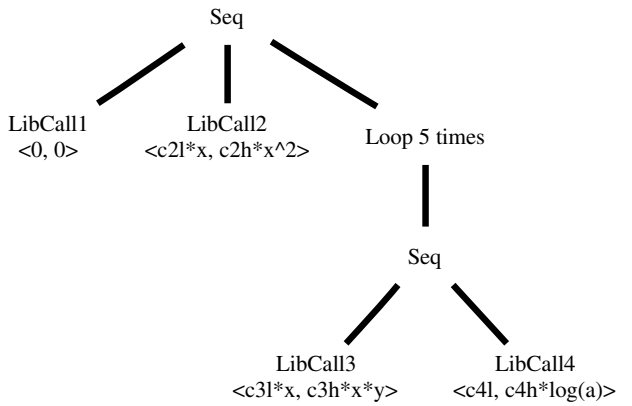

Figure 8. Complexity program: Each library call has been replaced by a complexity expression.

Now that we can represent the complexities of library calls, we construct a tree (7) representing the program where each library call has been replaced by the complexity expression associated with that call. The structure of the tree (i.e., the control structures) remains intact. Figure 8 shows an example complexity program. The variables x, y, and a in the complexity expressions are parameters to the library call the expression represents. The constants (beginning with 'c') are generated symbolic constants representing implementation-specific performance factors.

2.5 Program State

As mentioned in Section 2.2, each cost-bound function represents the complexity of a particular library call and requires the arguments to the original library call as inputs. Because the performance-relevant factors of a data structure (e.g., the size of a container) change over the course of program execution, we must estimate a store for each library call.

2.6 Effects Definitions

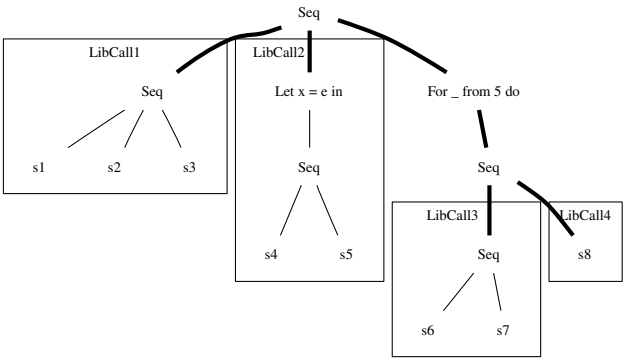

Figure 9. Effects program: Each library call has been replaced by an effects program.

To represent the effect of executing a library call, we define While′, a simple untyped imperative language based on While, a language defined in [23]. While′ adds extensible structures, references, and let bindings. For each library call, we define an effects function that simulates calling the associated library function—both in terms of the result returned and in terms of effects (changes to the store). As with the cost-bound functions in Section 2.4, a tree is constructed (6) representing the program in which each library call is replaced (5) by the associated effects definition. Figure 9 shows an example of such an effects program; the leaves (beginning with 's') represent statements.

For instance, consider the effects definition for post-incrementing a list<T> iterator:

```
template (t)
list-of-<t>-iter.post_increment (this) is
  let temp := this in
    this.position := this.position + 1 ;
    return temp
  end
```

As expected, the position of the iterator is increased, and a temporary iterator representing the old position is returned.

2.7 Store Estimation

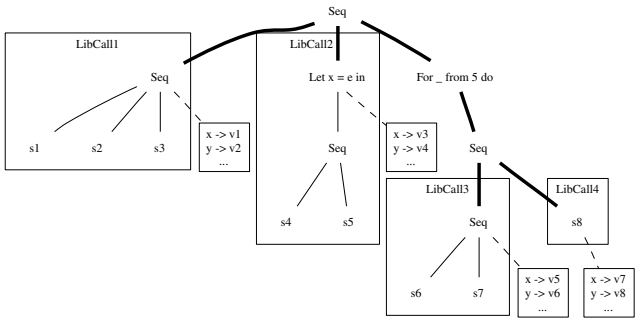

Figure 10. Effects program with abstract stores.

Since we do not have the inputs to the program whose behaviour we are analysing, we use an *abstract interpretation* (8) [8, 14] to approximate the effects of running the program on any possible input. The semantics for the abstract interpretation of While′ is defined in Appendix A. The result of this phase of the analysis is an abstract store (9) associated with each library call. This store associates each variable with a safe approximation of its value. Figure 10 illustrates an effects program with a store for each library call.

2.8 Combining Cost-Bound Functions and State

Combining the complexity program described in Section 2.2 with the abstract environments discussed in Section 2.5, we can replace (10) the arguments to each cost-bound function using the appropriate store (i.e., the store that represents the global state immediately prior to the associated library call) to generate a closed complexity program (11). Figure 11 shows an example of such a program.

2.9 Complexity Simplification

Now, we are ready to perform the final step in the conversion from a program to a complexity expression: By applying a set of simplification rules (12) to the closed complexity program, we can systematically reduce it to a single complexity expression (13). This is performed by associating a particular folding rule with each language construct in our composition language, and then recursively applying these rules, which compute the (symbolic) complexity of

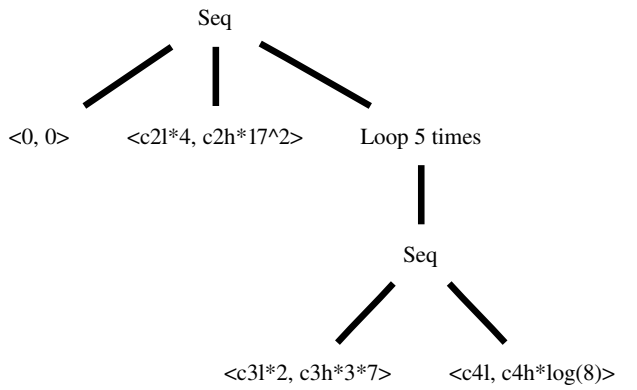

Figure 11. Closed complexity program: Each variable has been replaced by the appropriate abstract value.

a language construct given the complexities of its constituents. For example, consider the following rule for sequencing:

$$\textsc{Sequencing}\quad \frac{\forall i \in [1..n].s_i \rightarrow c_i}{\text{Seq}([s_1..s_n]) \rightarrow c_1(;)..(;)c_n}$$

As expected, the complexity of a sequence is the "sum" of the complexities of the constituent statements. Sum is defined in terms of a sequential complexity combinator, (;). To define (;) and the branch combinator, (.), we are currently using a symbolic range-propagation calculus similar to that presented by Blume and Eigenmann [4]. The complete definition of the simplification may be found in Appendix B.

2.10 Complexity Comparison

Finally, we compare complexity expressions mechanically via symbolic limit analysis—we examine the limits of ratios of complexity expressions as the input size approaches infinity since we are interested in our program's performance as the size of the input increases. At present, we are doing so using Maple [21], a symbolic mathematics package, to perform this limit analysis; we manually select the induction variable.

If c_1 and c_2 are the symbolic complexity expressions for a program instantiated with types t_1 and t_2, respectively, we examine the ratio $lim_{s\rightarrow\infty}\frac{c_1}{c_2}$ where s is an appropriate induction variable (e.g., size in the case of containers). If the limit of this expression is 0 then t_2's complexity dominates t_1's, and, thus, t_1 is a better choice. If the limit is some expression containing symbolic constants, then the complexities are incomparable—implementation-specific factors must be considered to determine which type is a better choice.

This stage of the analysis is presently performed by hand—the implementation produces complexity terms in Maple syntax, but the actual limit analysis must be performed externally.

3. Results

Returning to the motivating example from Section 1.2, we consider some results from our prototype implementation of the analysis. Using the CSimp version of the example program (Figure 4), effects and cost-bound function definitions for deque, list, and vector, and an encoding of the refinement relationship for the containers and associated iterators, the analysis correctly predicts that list<Student_info> is the correct container selection—that it is better than either deque or vector.

If we compare deque against vector, the limit expression converges to a ratio involving the costs of the associated push_back and erase operations:

$$\frac{deque\_of\_student\_info\_\_push\_back\_\_max\_const \\ +\ deque\_of\_student\_info\_\_erase\_\_max\_const}{vector\_of\_student\_info\_\_push\_back\_\_max\_const \\ +\ vector\_of\_student\_info\_\_erase\_\_max\_const}$$

This is the correct result: program complexities naturally form a partial order (consider $\mathcal{O}(m * n^2)$ and $\mathcal{O}(m^2 * n)$), and, therefore, there exist pairs of complexities that are incomparable. In such cases, an analysis should indicate that it is unable to produce a definite result. This is precisely what we would like—in this case, the difference in performance between the data types depends on implementation constants; were one to consider particular implementations of vector and deque, one could examine the costs for these operations in the specific implementation and, thus, make an informed selection. In this particular case, the costs associated with push_back and erase must be compared to determine which data type is the better choice.

4. Related Work

Blume and Eigenmann [4] discuss symbolic range propagation in the context of program parallelisation, where each range represents the set of possible values for a program variable. They present a collection of rewrite rules for simplifying range expressions and then for propagating these ranges through a program. Although their work is applied to a different problem, our complexity simplification serves the same purpose as their rewrite rules.

Sitaraman et al. [28] study both functional and complexity requirements for object-oriented programs in the Resolve framework [24]. They consider object-based abstract data types, specifying complexity requirements using a variant of Big-O notation and assuming a constant-time swap operator. Like we do, they use "value-based performance specifications" [18], but their analysis is not as general as ours: We consider concept-based libraries at a generic level and operate with abstract values representing all possible inputs rather than specific ones.

Cohen and Zuckerman [6] describe a two-language approach to analysing program complexity. They consider concrete implementations, probabilistic branching, and complexity expressions in terms of symbolic constants. Programs in their implementation language, PL, are converted to complexity expressions via a syntax-directed translation. Then, these complexity expressions are interactively manipulated via a second language, EL. Our approach is similar—CSimp corresponds to PL and Maple to EL—but for generic programs and with automated symbolic analysis rather than interactive manipulation.

Dornic, et al. [9] present a type-system-like time system for a purely functional language with higher-order functions. They describe a programming language that has both type and time systems. The time system assigns each expression a time: Unit time is assigned to each primitive operation; plambda is used to provide abstraction at the time level. All recursive functions and, thus also, functions that call recursive functions are assigned the time type long, which limits the applicability and precision of the analysis. By using this time system, each function's time can be checked separately, eliminating the need for a whole-program analysis. In the present version, time (and type) annotations are required, but reconstruction is listed as future work.

Ermedahl and Gustafsson [10, 11] describe an idea to calculate worst-case execution time for real-time programs. They do so by using data flow analysis over an abstract environment to determine the possible values of variables at various control points in the

program. Abstract environments are merged at the control points at the ends of loops, functions, and programs. Since the analysis is in a real-time context, where there are time budget annotations, termination is guaranteed—when the analysis can show that the program may exceed its budget, the analysis is terminated. Our use of abstract interpretation is inspired by this work.

Cohen and Weitzman [5] present a three-tool method for program analysis. All three tools are interactive and are implemented via a combination of definite clause grammars (DCG's) and symbolic simplification in Maple. The first tool "compiles" a program to a time formula, relying on the user to provide the probabilities with which branches are taken. DCG's are used to parse the high-level program, to associate each language construct with a time formula, and to prompt the user for input. Maple is then used to simplify the resulting time formula. The second uses DCG's to generate finite-difference equations which are then solved by Maple's rsolve to obtain bounds on loop trip counts; this is applicable only in simple cases and is not as general as the method presented in [30]. The third tool converts a program to a Markovian transition matrix representation that the user can then manipulate in Maple to estimate average-case performance.

5. Future Work

5.1 Translation

For the purposes of the present work, the programs under scrutiny have been manually translated to CSimp, our library-oriented composition language. It should be possible to extract this information directly from the source program, given a means to distinguish which calls are library calls and which are calls to user functions.

Likewise, the effects definitions, refinement relationship for generic types, and cost-bounds functions have been manually encoded. If the library's requirements were specified in some formal system (e.g., Tecton [15]), it would be possible to automatically generate effects definitions; it would, however, not be possible to extract these definitions from a particular implementation of the library, since this would capture the idiosyncrasies of the specific implementation rather than the behaviours common to all possible implementations.

Applying the analysis to a language that directly supports generic programming, such as G [26], would allow extraction of the refinement relationship.

5.2 Loop Bounds

One of the major limitations of the present analysis is that loop bounds must be determined manually and must be statically known. The reason for this is that the abstract integers introduce infinite ascending chains into the lattice of abstract values. Requiring static bounds for loop iterations avoids possible nontermination of the analysis.

The traditional solution to this problem is to define a widening operator. This would allow our analysis to handle programs with general loops (i.e., loops without static constant bounds). It remains to be seen if an appropriate widening operator could be defined that would maintain an acceptable level of precision for the analysis.

An alternate possibility is to incorporate a loop-bound analysis [12] to automatically determine lower and upper iteration bounds for each loop in the program. These bounds could then be used with the current abstract interpreter.

5.3 Induction Variable Detection

Presently, the induction variables used in the symbolic limit analysis must be selected manually. In order to fully automate the analysis, induction variables detection [30] would be necessary.

5.4 Complexity Calculus

The complexity calculus we use is based on that in [4]. It may be possible to achieve better results by extending or parametrising the calculus to make it more expressive. A calculus incorporating conditions, for instance, would be better able to capture the complexities expressed in the STL specification—some complexities depend on what concepts a particular type models (advance, e.g., provides a more efficient implementation for random-access iterators than for input iterators).

5.5 Interprocedural Analysis

At present, we handle only a single procedure composed of basic blocks, conditions, and loops. It would be useful to extend the analysis to multiple procedures, particularly if user-level generic procedures were allowed.

6. Conclusions

We have presented an analysis based on cost-bound functions and abstract-interpretation approximation of program states for generic programs. If such analyses were integrated into compilers for programming languages supporting generic programming, it could be a great aid to library users in choosing data types for their programs in order to avoid performance bugs.

A. Abstract Interpretation for While′

Following is a denotational semantics for the abstract interpretation of While′, our language for approximating program effects.

We will use the following meta-variables: x is used for variables, p for l-value patterns, f for fields in structures, v for values, e for expressions, s for statements, i for integer constants, and σ for (abstract) stores.

The definitions of $\sqcup$ and $\mathcal{E}$ are not shown.

Patterns ($\mathcal{P} : \widetilde{\text{Pattern}} \rightarrow \widetilde{\text{Lvalue}}\ \text{Set}$)

$$
\begin{array}{lcl}
\mathcal{P}[\![x]\!]\sigma & = & \{x\} \\
\mathcal{P}[\![\text{Deref } p]\!]\sigma & = & \text{let } ps' = \mathcal{P}[\![p]\!]\sigma \text{ in} \\
 & & \text{let } ps'' = \text{map } (\lambda e.\mathcal{E}[\![e]\!]\sigma)\ ps' \text{ in} \\
 & & \{p'' | (\text{Ref } p'') \in ps''\} \\
\mathcal{P}[\![p.f]\!]\sigma & = & \text{let } ps' = \mathcal{P}[\![p]\!]\sigma \text{ in } \{p'.f | p' \in ps'\}
\end{array}
$$

Statements ($\mathcal{S} : \widetilde{\text{Statement}} \times \widetilde{\text{Store}} \rightarrow \widetilde{\text{Store}}$)

$$
\begin{array}{lcl}
\mathcal{S}[\![p :=_1 v]\!]\sigma & = & \sigma[x \mapsto v] \\
\mathcal{S}[\![p.f :=_1 v]\!]\sigma & = & \sigma[x \mapsto \text{structUpdate}(\\
 & & \quad \sigma(x), f \mapsto v)] \\
\mathcal{S}[\![p := e]\!]\sigma & = & \text{let } ps' = \mathcal{P}[\![p]\!]\sigma \text{ in} \\
 & & \text{let } v = \mathcal{E}[\![e]\!]\sigma \text{ in} \\
 & & \text{if card}(ps') = 1 \\
 & & \text{then let } \{p'\} = ps' \text{ in} \\
 & & \quad \mathcal{S}[\![p' :=_1 v]\!] \\
 & & \text{else let } \sigma s = \{\mathcal{S}[\![p' :=_1 v]\!]\sigma \\
 & & \quad |p' \in ps'\} \text{ in} \\
 & & \quad \text{fold } (\sqcup)\ (\sigma s \cup \{\sigma\}) \\
\mathcal{S}[\![\text{Skip}]\!]\sigma & = & \sigma \\
\mathcal{S}[\![s_1 \ ; \ s_2]\!]\sigma & = & \text{let } \sigma' = \mathcal{S}[\![s_1]\!]\sigma \text{ in } \mathcal{S}[\![s_2]\!]\sigma' \\
\mathcal{S}[\![\text{If } e \text{ then } s_1 \text{ else } s_2]\!]\sigma & = & \text{let } b = \mathcal{E}[\![e]\!]\sigma \text{ in} \\
 & & \text{case } b \text{ of} \\
 & & \quad true \ \rightarrow \ \mathcal{S}[\![s_1]\!]\sigma \\
 & & \quad false \ \rightarrow \ \mathcal{S}[\![s_2]\!]\sigma \\
 & & \quad \top \ \rightarrow \ \text{let } \sigma_1 = \mathcal{S}[\![s_1]\!]\sigma \text{ in} \\
 & & \quad\quad \text{let } \sigma_2 = \mathcal{S}[\![s_2]\!]\sigma \text{ in} \\
 & & \quad\quad \sigma_1 \sqcup \sigma_2 \\
\mathcal{S}[\![\text{For } x \text{ from } i \text{ do } s]\!]\sigma & = & \text{let } < i_l, i_h > = i \text{ in} \\
 & & \text{if } i_h = 0 \\
 & & \text{then } \sigma \\
 & & \text{else let } i' = \\
 & & \quad < \max(0, i_l - 1), \\
 & & \quad\quad i_h - 1 > \text{ in} \\
 & & \quad \mathcal{S}[\![\text{Let } x = i \text{ in } (s \ ; \\
 & & \quad\quad \text{For } x \text{ from } i' \text{ do } s)]\!]\sigma \\
\mathcal{S}[\![\text{Let } x \ = \ e \text{ in } s]\!]\sigma & = & \text{let } v = \mathcal{E}[\![e]\!]\sigma \text{ in} \\
 & & \text{let } \sigma' = \sigma[x \mapsto v] \text{ in} \\
 & & \text{let } \sigma'' = \mathcal{S}[\![s]\!]\sigma \text{ in} \\
 & & \sigma''[x \mapsto \sigma(x)]
\end{array}
$$

Expressions ($\mathcal{E} : \widetilde{\text{Expression}} \times \widetilde{\text{Store}} \rightarrow \widetilde{\text{Value}}$)

B. Complexity Simplification Rules

Following are the simplification rules used when folding a complexity program into a complexity expression. Library calls and assignments are labelled in order to match each with the proper store.

We will use the following meta-variable naming scheme: a is used for variables, x for l-values and X for their types, $m.f$ for library functions, f for non-library functions, F for complexity functions, s for statements, e for expressions, l for labels (on library functions and assignments), and i for integer values.

$\mathcal{E}$ is the While$'$ semantic function for expressions from Appendix A. Σ maps labels to abstract stores, as computed by the abstract interpretation step of the analysis. σ_F maps library functions to the corresponding cost-bound functions. The following rules define $\rightarrow$, the complexity simplification on CSimp abstract syntax.

LIBRARY CALL

$$\frac{\Sigma(l) = \sigma \qquad \sigma_F(m.f) = F \qquad \mathcal{E}[\![F(a_1..a_n)]\!]\sigma = c}{\text{LibCall}(m.f, [a_1..a_n])@l \rightarrow c}$$

The complexity of a library call is determined by looking up the complexity function F corresponding to the library function $m.f$ and then evaluating the application of F to $m.f$'s arguments with the proper store.

(NON-LIBRARY) FUNCTION CALL

$$\frac{}{\text{FuncCall}(f, [a_1..a_n]) \rightarrow 0}$$

The complexity of a non-library call is ignored. (The cost of non-library function calls is assumed invariant with respect to the types with which we instantiate the program's type variables.) This assumption could be relaxed if the implementation were extended to an interprocedural analysis (see Section 5.5).

SEQUENCING

$$\frac{\forall i \in [1..n].s_i \rightarrow c_i}{\text{Seq}([s_1..s_n]) \rightarrow c_1(;)..(;)c_n}$$

The complexity of a sequence of statements is the $(;)$-sum of the complexities of the statements.

ASSIGNMENT

$$\frac{\text{typeOf}(x) = X \qquad \text{LibCall}(X.\text{assignment}, [x, a])@l \rightarrow c}{\text{Assign}(x, a)@l \rightarrow c}$$

An assignment is simplified by inferring the proper assignment operator (from the type of the l-value argument) and then treating the assignment as a library call to the proper assignment operator.

TEMP ASSIGNMENT

$$\frac{e \rightarrow c}{\text{TempAssign}(x, e) \rightarrow c}$$

The complexity of a temporary assignment is the complexity of evaluating its expression. (The cost of the assignment is ignored.)

BRANCH

$$\frac{\forall i \in [1..n].s_i \rightarrow c_i}{\text{Branch}([s_1..s_n]) \rightarrow c_1(.)..(.)c_n}$$

The complexity of a branch is the $(.)$-sum of the complexities of the cases.

CASE

$$\frac{s \rightarrow c}{\text{Case}(s) \rightarrow c}$$

The complexity of a case expression is that of the statement evaluated in the case.

LOOP

$$\frac{s \rightarrow c}{\text{Loop}(i, s) \rightarrow i * c}$$

The complexity of the loop is the product of the trip-count bound, i, and the complexity of the body of the loop.

Acknowledgments

Thanks to Todd Veldhuizen, Grégoire Hamon, Marcin Zalewski, and Sibylle Schupp for numerous stimulating and helpful discussions regarding this work and comments on drafts of this paper. Special thanks to Sibylle Schupp for suggesting this research direction and for guidance in developing the approach to the analysis.

References

[1] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola—a small composition language. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing—A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[2] ANSI-ISO-IEC. *C++ Standard, ISO/IEC 14882:1998*, ANSI standards for information technology edition, 1998.

[3] M. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.

[4] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, 1995.

[5] J. Cohen and A. Weitzman. Software tools for micro-analysis of programs. *Software Practice and Experience*, 22(9):777–808, Sept. 1992.

[6] J. Cohen and C. Zuckerman. Two languages for estimating program efficiency. *Commun. ACM*, 17(6):301–308, 1974.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press/McGraw Hill, second edition, 2001.

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

[9] V. Dornic, P. Jouvelot, and D. K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, Mar. 1992.

[10] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proceedings of the 3rd International Euro-Par Conference on Parallel Processing, EURO-PAR'97 (Passau, Germany, August 26-29, 1997)*, volume 1300 of *LNCS*, pages 1298–1307. Springer-Verlag, Berlin-Heidelberg-New York-Barcelona-Budapest-Hong Kong-London-Milan-Paris-Santa Clara-Singapore-Tokyo, 1997.

[11] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Parallel and Distributed Computing Practices*, 1(2):257–262, Mar. 1998.

[12] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 12–21, Washington - Brussels - Tokyo, June 1998. IEEE.

[13] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[14] N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. pages 527–636, 1995.

[15] D. Kapur, D. R. Musser, and X. Nie. The tecton proof system. In V. S. Alagar, L. V. S. Lakshmanan, and F. Sadri, editors, *Formal Methods in Databases and Software Engineering*, Workshops in Computing, pages 54–79. Springer, 1992.

[16] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[17] A. Koenig and B. E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000.

[18] J. Krone, W. F. Ogden, and M. Sitaraman. Verification of performance constraints. Technical Report RSRG-03-04, Clemson University, 2003.

[19] D. Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, Apr. 1988.

[20] Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. *Lecture Notes in Computer Science*, 1474:31–40, 1998.

[21] MapleSoft. Maple 9.5. `http://www.maplesoft.com/`.

[22] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison Wesley, 2nd edition, 2001.

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[24] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part I: the resolve framework and discipline: a research synopsis. *SIGSOFT Softw. Eng. Notes*, 19(4):23–28, 1994.

[25] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[26] J. G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, August 2005.

[27] M. Sitaraman. Compositional performance reasoning. In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors, *Proceedings 4th ICSE Workshop on Component-Based Software Engineering*. IEEE, 2001.

[28] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy. Performance specification of software components. In *Proceedings of the 2001 Symposium on Software Reusability*, pages 3–10. ACM Press, 2001.

[29] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[30] M. Wolfe. Beyond induction variables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Francisco, June 1992.