Author manuscript, published in "Domain-Specific Program Development, Nashville : États-Unis d'Amérique (2008)"

hal-00350271, version 1 - 6 Jan 2009

Final:Realizing Domain-Specific Compiler Optimizations via Generic Programming

Xiaolong Tang
Texas A&M University
xiaolong@cs.tamu.edu

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

Introduction Contemporary mainstream programming languages offer mechanisms for expressing domain-specific abstractions, but these languages and their compilers fall short in support for optimizing the use of such abstractions. Optimizations rules in compilers are typically hard-coded and apply only for "low-level" operations of built-in types; there are no mechanisms for defining domain-specific optimization rules or strategies that would apply to a specific set of user-defined types. Addressing this discrepancy has been an active area of research (1; 2; 3; 4; 5; 6; 7). Approaches, such as `Active Libraries` (8) and `Simplicissimus` (2), exploit compilers' abilities to perform computations, e.g. template metaprogramming. These approaches can be applied to new domains with relative ease, but are limited in that they are flow-insensitive and do not integrate well with other compiler optimizations; the computations (and thus domain-specified optimizations) must occur in the front-end where flow information is typically not accessible. In contrast, lightweight optimization tools, such as `CodeBoost` (3), and full-fledged compilers, such as `Telescoping languages` (9) and the `BroadWay` compiler (6), allow more powerful domain-specific optimizations by devising their own program analyses and optimizations, whose functionalities and implementations overlap those of traditional compilers; the effort of developing and maintaining domain-specific optimizations for a given domain may be more involved with these approaches.

Assuming a system that supports expressing domain specific user-defined optimizations, the programming tasks that are necessary to enable such optimizations are, first, to describe the domain-specific transformation rules for specific types and their operations, and, second, to annotate the programs to indicate where the transformations should apply. Regarding the first task, it is not realistic to expect all programmers to provide type-specific optimization rules for all new user-defined types. Authors of widely used libraries could, however, engage to such an effort if the effort was reasonable and amortized over many uses of the abstractions offered by the library. Regarding the second task, the burden of annotating programs to enable domain-specific optimizations should be very light, preferably incurring no burden at the client side, when using abstractions that support domain-specific optimizations. Our work studies the potential of *generic programming* for easy expression of generic optimization rules in libraries, and very lightweight annotations to enable those rules: both the expression of the rules, as well as the annotations are part of the normal course of generic programming, that encourages the explicit expression of abstractions' semantic properties. Furthermore, we explore a mechanism to make domain-specific optimizations flow-sensitive, so that they can be applied when traditional optimizations, such as inlining and copy propagation, reveal more optimization opportunities.

Background: Generic Programming Yields Generic Optimizations Generic programming is a paradigm for designing and implementing reusable software libraries. It allows for classifying types based on their capabilities, that is, the operations they support and the semantics of these operations. This classification is the key to economical expression of domain-specific optimizations: they need not be specified for operations of each user-defined type individually, but rather at once to classes of types that have the essential characteristics for a particular transformation. The central notion of generic programming is *concept*, essentially the (algebraic) description of requirements on types. A type that satisfies the requirements described by a concept is said to *model* that concept. We tie domain-specific optimization rules to concepts; the modeling mechanism is what enables these optimizations.

Our work is in the context of C++, in particular its forthcoming extension, the "concepts" language feature (10). The language construct `concept` defines a set of requirements for a type or types, operations that must be defined and a set of accessible "associated" types. The language construct `concept_map` establishes modeling relationship between a type (or types) and a concept. A special `axiom` construct, used inside concept definitions, allow programmers to express semantic constraints as equations that operations of a concept's operations must satisfy. The use of axioms has been previously explored for optimizations and tests (11; 12); in the former use the equations of axioms are viewed as rewriting rules. Transformation rules induced from equations in a concept's axioms are generic, as the equations hold for all types that model a concept. Listing 1 shows a concept defining *monoid*s—mathematically defined to be a set with one *associative*

binary operation and one identity element obeying *the axiom of identity element*. This axiom is denoted by the Identity axiom in the programmatic expression of monoids as the concept Monoid, and give rise to a transformation rule. The two concept maps in Listing 2 justify the transformation, e.g., for x+0 for any expression x of type int and for s+string("") for any expression s of type string.

```
concept Monoid<typename Op, typename T> {
  T identity_element(Op, T);
  axiom Identity(Op op, T x) {
    op(x, identity_element(op, x)) == x;
    op(identity_element(op, x), x) == x;
  }
}
```

Listing 1. Monoid concept definition

```
concept_map Monoid<plus<int>, int> {
  int identity_element(plus<int> op, int x)
  { return 0; }
}
concept_map Monoid<plus<string>, string> {
  string identity_element(plus<string> op, string x)
  { return string(""); }
}
```

Listing 2. concept map definition

Domain-Specific Compiler Optimizations We report on our work in utilizing C++ concepts, and their axioms, in communicating domain-specific optimization opportunities to a compiler. Our goal is to enable high-level effective optimizations with very little extra burden to the programmer. In particular, we discuss our approach to permit and apply high-level domain specific transformation rules throughout the compiler—particularly, in the middle-end.

The outline of our approach is as follows:

- We induce generic rewriting rules from invariants declared inside axioms of concepts, and thereafter use them to directly optimize generic code;
- We instantiate generic rules to arrive at type-specific ones for optimizing non-generic code;
- We utilize flow information to enable high-level optimizations;

Our contributions include developing an approach for domain-specific optimizations based on generic programming, and the tools and techniques to effect them, as well as demonstrating their effectiveness and applicability with a prototype.

First, we refer to the report (11) regarding how axioms give rise to generic optimization rules, and how to effect them in generic code. For example, Interpreting as rewrite rules the axiom Identity's equations in Listing 1 from left to right gives rise to the *generic right identity rule* 1, and the *generic left identity rule* 2.

$$\mathsf{op}(\mathsf{x},\ \mathsf{identity\_element}(\mathsf{op},\ \mathsf{x})) \rightarrow \mathsf{x} \tag{1}$$

$$\mathsf{op}(\mathsf{identity\_element}(\mathsf{op},\ \mathsf{x}),\ \mathsf{x}) \rightarrow \mathsf{x} \tag{2}$$

In our current work, we focus on applying generic optimization rules to non-generic code. The central idea is to use the generic rules in a "generative" role, to transform generic optimization rules into rules applicable for specific user-defined (and built-in) types. We accomplish this transformation in a traditional compiler utilizing its operations for eliminating abstractions.

Specifically speaking, in terminology of guaranteed optimization (13), we can view *parametrization* and *encapsulation* as abstracting operations, and *template instantiation* and *inlining* as their reverse de-abstracting operations. The above de-abstracting operations are well implemented in traditional compilers. Thus, considering functions are basic manipulation units in the compiler, we embed rewriting rules into individual functions, called *rule functions*, so that the compiler could directly manipulate them. Figure 1 illustrates the process to handle the axiom Identity for the concept Monoid in Listing 1. Note that two rule functions, _axiom_1_lhs(for *lhs*) and _axiom_1_rhs(for *rhs*), are generated for one single rule 2.

To enable generic optimization rules to be used in the compiler's middle end, we utilize the de-abstraction operations on the rules themselves, and, to render flow-sensitive transformations, we examine the resulting rules after de-abstractions for their flow patterns and conduct rewriting on these patterns. For example, in Listing 1, the Identity axiom actually defines this pattern for the type string: given one operand defined to be empty, the operation operator+ could be simplified. And it is clear that an optimizing compiler could reveal this pattern in many cases: *use-def* links could see through statements and find definitions for operations, and *inlining* through functions.

The key mechanism our framework uses is carrying over concept knowledge to the middle-end, so that optimizers could exploit semantics of high-level abstractions in manipulating user-defined transformations and improving high-level flow information. Besides, rearranging traditional optimizations account for more optimization opportunities, because some optimizations dis-aggregate high-level abstractions too soon.

hal-00350271, version 1 - 6 Jan 2009

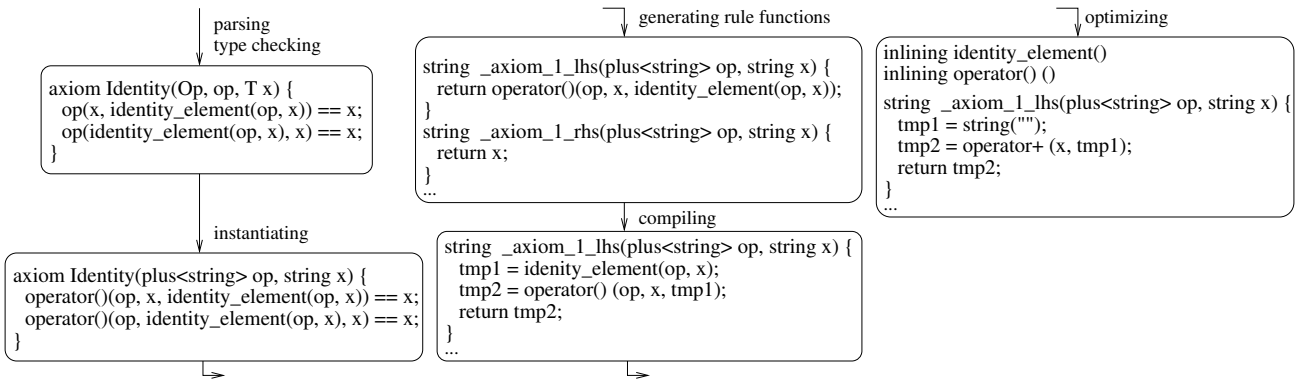

Figure 1. The procedure to compile and optimize axioms

Conclusion and Future Work We have built up a prototype on `ConceptGCC` and our preliminary experiments can demonstrate the effectiveness and relatively effortless specification of select domain-specific compiler optimizations. Domain-specific optimizations are enabled in a simple, non-disruptive manner: once the optimization framework is in place, what a user needs to do to enable a set of optimizations is as simple as declaring a particular concept map, like that for std::string or LiDIA::bigint in Listing 2.

This position paper reports work in progress, and we continue to study the expressiveness of axioms as a foundation for domain-specific optimization rules, and investigate asymmetries of the current compiler in handling high-level abstractions and built-in entities.

References

[1] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5), June 1995.

[2] Sibylle Schupp, Douglas Gregor, David R. Musser, and Shin-Ming Liu. Semantic and behavioral library transformations. *Information & Software Technology*, 44(13):797–810, 2002.

[3] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the codeboost transformation system for domain-specific optimization of C++ programs. In *Source Code Analysis and Manipulation*, pages 65–74. IEEE Computer Society, 2003.

[4] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231, New York, NY, USA, 2003. ACM Press.

[5] Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, 2001.

[6] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, Feb 2005.

[7] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

[8] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. *CoRR*, math.NA/9810022, 1998.

[9] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, 2005.

[10] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, 2006.

[11] Xiaolong Tang and Jaakko Järvi. Concept-based optimization. In *ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07)*, October 2007.

[12] Anya Helene Bagge and Magne Haveraaen. Axiom-based transformations: Optimisation and testing. In *Proceedings of the 8th Workshop on Language Description, Tools and Applications (LDTA 08)*, Budapest, Hungary, April 2008.

[13] Todd L. Veldhuizen and Andrew Lumsdaine. Guaranteed optimization: Proving nullspace properties of compilers. *Lecture Notes in Computer Science*, 2477:605–627, 2002.

hal-00350271, version 1 - 6 Jan 2009