Annotating User-Defined Abstractions for Optimization

Dan Quinlan[1], Markus Schordan[2], Richard Vuduc[1], Qing Yi[3]

[1]Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94550 USA
{dquinlan,richie}@llnl.gov

[2]Institute of Computer Languages
Vienna University of Technology
Vienna, Austria
markus@complang.tuwien.ac.at

[3]Dept. of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249 USA
qingyi@cs.utsa.edu

Abstract

Although conventional compilers implement a wide range of optimization techniques, they frequently miss opportunities to optimize the use of abstractions, largely because they are not designed to recognize and use the relevant semantic information about such abstractions. In this position paper, we propose a set of annotations to help communicate high-level semantic information about abstractions to the compiler, thereby enabling the large body of traditional compiler optimizations to be applied to the use of those abstractions. Our annotations explicitly describe properties of abstractions that are needed to guarantee the applicability and profitability of a broad variety of such optimizations, including memoization, reordering, data layout transformations, and inlining and specialization.

1 Introduction

We consider the problem of improving the performance of scientific computing applications that rely on user-defined high-level abstractions. Although conventional compilers implement a wide range of optimization techniques, they may miss opportunities to optimize the use of abstractions because they are not designed to recognize and use semantic information about such abstractions. Thus, developers today either accept the idea of an "abstraction penalty" and live with relatively poor performance, or manually rewrite code to use lower-level constructs, obtaining better performance at the cost of maintainability and portability.

In this paper, we propose a set of annotations to help communicate high-level semantic information about abstractions to the compiler, thereby enabling the large body of traditional compiler optimizations (Section 2) to be applied to the use of those abstractions. These annotations explicitly describe properties of the abstraction needed to guarantee the applicability and profitability of particular optimizing transformations.

Our annotations are influenced by our practical experience with optimizations that have had an impact on applications used throughout the U.S. Department of Energy (DOE) research laboratories (Section 3). Though not all-inclusive, these annotations permit a broad variety of classical optimizations, including memoization, reordering, data layout transformations, and inlining and specialization, to be applied to the use of abstractions. This paper generalizes our earlier research on the optimization of array abstractions [17] to arbitrary user-defined abstractions.

Figure 1 shows an example of a user-defined abstraction that exhibits many features of typical unstructured mesh computations. Specifically, the *Mesh* class is a user-defined abstraction, and the *compute()* procedure is a user's algorithm that operates on a *Mesh* object. The *Mesh* class is a container of nodes and edges, where an edge connects two nodes, and *Mesh* provides a means to iterate over either nodes or edges. An important property of *Mesh* is its many-to-one and onto mapping (surjection) of edges to nodes. The procedure *compute()* implements some algorithm that is most naturally expressed as iteration over edges, but

Figure 1. User-defined abstraction example.

```
class Node { // ...
    public: int id (); double eval(double a);
};
class Edge { // ...
    public: Node *node1(); Node *node2();
};
class Mesh { // ...
   public:
     Edge* get_edge(int i );
     Node* get_node(int i);
     int node_size (); int edge_size ();
};

void compute(Mesh& m, double a) {
  for (int i = 0; i < m.edge_size(); ++i) {
        Edge* e = m.get_edge(i);
        // ...
        double x = e->node1()->eval(a);
        double y = e->node2()->eval(a);
        bar (x, y);
        // ...
    }
}
```

Figure 2. *compute()* memoized.

```
void compute_optimized(Mesh& m, double a) {
  vector<double> eval_precomp (m.node_size ());
  for (int i = 0; i < m.node_size(); ++i) {
        Node* n = m.get_node(i);
        eval_precomp[n->id()] = n->eval(a);
  }

  for (int i = 0; i < m.edge_size(); ++i) {
        Edge* e = m.get_edge(i);
        // ...
        double x = eval_precomp[e->node1()->id()];
        double y = eval_precomp[e->node2()->id()];
        bar (x, y);
        // ...
    }
}
```

repeatedly calls an expensive and side-effect free function *eval* on each node. Due to the surjection, we can optimize *compute()* by precomputing (memoizing) *eval* on all nodes, yielding the optimized code of Figure 2. White, *et al.*, showed a 2× speedup by applying this specific memoization to a realistic benchmark that was extracted from a DOE code [14].

Figure 3. Annotations for *Mesh*.

```
class Node : has_value { id = this.id (); }
class Edge : has_value {
   n1 = this.node1(); n2 = this.node2();
   };
operator Node::eval(double a) :
   read {this,a}; modify none; alias none;
class Mesh : has_value{
     nsize = this.node_size (); esize = this.edge_size ();
     nodes(i:0: nsize)= this.get_node(i);
     edges(i:0: esize ) = this.get_edge(i);
     };
   restrict_value { nodes(i).id ≠nodes(j).id; }
   never_alias (edges(i).n1) = edges(i).n2;
   never_alias (edges(i)) = edges(j) : j ≠i ;
   never_alias (nodes(i)) = nodes(j) : j ≠i ;
   must_alias(nodes(j)) = edges(i).n1 or edges(i).n2;
   restrict_value {esize ≥nsize * k1; esize ≤nsize * k2}
```

Figure 3 specifies the properties of *Mesh* as annotations, to enable the translation of Figure 1 into Figure 2. The annotations describe aliasing properties of the data abstractions, and side-effect properties of the function abstraction *eval()*. Moreover, we can specify properties of the data structure (such as lower and upper bounds on the ratio of edges to nodes) that could influence subsequent profitability analysis.

The optimized code in Figure 2 is more complex and difficult to debug than Figure 1, and so the transformation may be undesirable to apply by hand everywhere. Whether the optimization is even profitable will vary by computer architecture. Though automatable, the transformation relies on the semantics of the abstraction, so that a conventional compiler is unlikely to discover such an optimization opportunity or even know whether the additional storage can be tolerated. In the following sections, we use this example to motivate the need for annotations to support the difficult program analysis required to know when such optimizations may be used profitably.

Our annotations, presented in Section 4, provide an open interface for developers to communicate the semantics of their abstractions to the compiler. These annotations complement program analysis when compiler analysis is insufficient to enable optimizations. Whether an abstraction is annotated automatically or specified by the developer, traditional optimizations can be naturally extended to use the annotations and then applied to uses of the abstraction. We are implementing these ideas in the ROSE source-to-source compiler infrastructure (Section 5) [10, 11].

2 Analysis for Traditional Optimizations

We wish to express, through annotations, the results of the analysis needed to apply traditional compiler optimizations to the use of user-defined abstractions. A compiler may readily apply such optimizations to the use of built-in types whose semantics are known. However, it might not do so for abstractions whose semantics must be inferred from an implementation (even if available) using necessarily conservative or limited analysis techniques. Below, we review four broad classes of traditional optimizations, the analysis each requires, and highlight some problems in analyzing user-defined abstractions.

- *Memoization optimizations.* Memoization uses data-flow analysis to determine results of computations that can be saved (memoized) for later reuse, thereby avoiding redundant computation. Examples include common subexpression elimination, loop invariant code motion, strength reduction, and dead code elimination. These optimizations usually require precise information about side-effects of function calls that may affect the expressions being saved, but such side-effects are often non-trivial to identify, even if the implementation is available.

- *Reordering optimizations.* Reordering optimizations requires dependence analysis to determine ordering constraints between each pair of statement (or instruction) instances. Examples include instruction scheduling for better CPU utilization, loop transformations such as loop fusion and blocking for better memory hierarchy performance, and automatic parallelization and communication optimizations for better utilization of multiprocessors.

- *Data layout optimizations.* These optimizations rearrange the layout of data structures to accommodate resource constraints of computers. Examples include register allocation, scalar replacement, and array padding. These optimizations require precise knowledge of the data structures, which may be obscured by abstraction.

- *Abstraction inlining and specialization.* These optimizations reduce the overhead of user-defined abstractions in programs. Such overheads include the cost of making function calls, missed optimization opportunities due to abstraction boundaries, as well as inefficient data grouping and additional indirections due to the necessity of data abstraction. Optimizations in this category include procedure inlining and specialization, data structure splitting, and elimination of indirection.

In addition to data-flow and dependence analysis, pointer analysis is critical to performing all the above optimizations effectively. Polymorphism and dynamic functions complicate the problem of pointer analysis in object-oriented languages, and unconstrained uses of pointers in C and C++ make the problem worse still. Although good linear algorithms exist for pointer analysis of stack-allocated variables, the modeling of the heap remains a challenge, with algorithms ranging from linear to double exponential time. These problems can be alleviated or even solved by using abstractions with user-defined annotations.

For interprocedural analysis (whole-program analysis), we must consider all effects on function parameters, return values, and global variables. When analyzing function calls, different levels of precision can be accomplished with call-strings or assumption sets. For object-oriented programs, the modeling of the heap and the states of objects becomes increasingly important. Data hiding and the encapsulation of object states allows modular program analysis, but in general, the scalability of an analysis for real-world applications must be addressed by storing results of previous analysis passes, especially when libraries are used.

3 Extending Traditional Optimizations *via* Abstraction-Aware Analysis

Given a mechanism to express or compute abstraction semantics (*e.g.*, annotations), an *abstraction-aware analysis* (A^3) combines the semantics of built-in types with the properties of user-defined abstractions to compute more precise information regarding program behavior, thereby facilitating more advanced traditional optimizations. When program analysis becomes abstraction-aware, traditional optimizations are naturally extended to apply to the use of abstractions.

For example, in our running example in Figure 3, the function *eval* is annotated to indicate that it does not modify the variables *a* and *this* and that it creates no aliasing. Thus, calling this function does not change the object's state. This information is crucial for traditional analysis such as *available expressions analysis* or *very busy expression analysis*. In our approach, we extend the analysis to include such function calls that do not change the object's state in the collection of available expressions. Based on the results of available expressions analysis, redundant expressions can be elim-

Optimization	A1	A2	A3	A4
Common subexpression elim.	no	no	no	no
Loop transformations	no	no	no	no
Procedure inlining	no	yes	yes	no
Structure splitting	no	yes	no	yes
AA Scalar replacement	yes	no	yes	yes
AA Loop transformations	yes	no	yes	yes
AA Common subexpr. elim.	yes	no	yes	no
OpenMP parallelization of container iteration	yes	no	yes	yes
Iteration-space narrowing	yes	no	yes	no
Iteration-space partitioning and loop specialization	yes	no	yes	no
Iteration-space tiling	yes	no	yes	yes
Precomputation	yes	yes	yes	no
Array abstraction translation	yes	yes	no	yes
Elimination of indirection	yes	yes	yes	yes

Table 1. Optimizations classified in four different aspects

inated. Hence, we can perform library-aware redundancy elimination on applications that use libraries.

Table 1 lists optimizations that we have identified as crucial for improving the performance of scientific applications used within the DOE laboratories, several of which are applied by White, *et al.*, to DOE codes [14]. This table classifies the optimizations according to the following four properties:

A1. Does the optimization require high-level semantics of the user's abstraction?

A2. Does the optimization eliminate abstraction layers or boundaries?

A3. Does the optimization apply to function abstractions? For example, procedure inlining eliminates a function abstraction. Note that this optimization applies to the application code, not the library (the function remains in the library).

A4. Does the optimization apply to data abstractions or use semantics of data abstractions? For example, when applying structure splitting, the data abstraction of the original structure is eliminated and replaced by new data structures.

A special category in Table 1 neither requires high-level abstraction semantics, nor eliminates abstraction layers. This category includes most of the optimizations in conventional compilers, such as common subexpression elimination, scalar replacement, and loop transformations.

Three optimizations are explicitly denoted as abstraction-aware (AA) optimizations. For these optimizations, the corresponding traditional optimizations—scalar replacement, loop transformations, and common subexpression elimination without annotations—are essential as well in improving the performance of our applications. Other optimizations use high-level semantics in various ways; for instance, OpenMP parallelization uses the high-level semantics of STL containers and their iterators [9].

Most conventional compilers implement optimizations that do not require elimination of any abstractions. Some also implement non-trivial optimizations that do not require high-level semantics of abstractions but do eliminate user-defined abstractions, such as structure splitting. In contrast, all the abstraction-aware optimizations require annotations, especially when the required properties cannot be established by a conventional analysis. More advanced transformation capabilities are required for a compiler infrastructure to permit optimizations, such as array abstraction translation as we have demonstrated using ROSE [10].

We have also identified optimizations for which we did find useful abstraction-aware extensions, but which mostly applied in the back-ends of compilers. For example, *code selection*, which can be performed by Bottom Up Rewrite Systems (BURS) like burg or iburg, may apply to abstraction use but is not useful at the AST level in our experience due to the need for absolute values as weights for the selectable instructions.

4 Annotating Abstractions

Annotations enable abstraction-aware analysis. They may be specified explicitly by the programmer when they cannot be generated automatically. Separate annotations may be generated for each library used by a program, and saved as external descriptions for use during optimization (whether or not the source is available). In this section, we develop a framework of annotations informed by the discussion of abstraction-aware analysis and optimization from Section 3.

In most languages that support abstractions in user-defined types, the abstractions can be separated into two categories: function abstractions and data abstractions. Additionally, languages such as C++ support object-oriented abstractions, where different function and data abstractions can relate to each other through subtype relations and through inheritance.

Function (or procedure) abstractions represent algorithms that operate on data. The operations might be as simple as returning the value of a field within a compound data structure, or as complex as sorting elements in a container. Their semantics can be expressed in terms of what restrictions the input data must satisfy before entering the operation, what data are being modified by the operation, and what properties and relations the resulting data would satisfy after the operation.

Data abstractions are encapsulated collections of values that relate to each other. The semantics of data abstractions are normally expressed in terms of properties and invariants that must be satisfied by the data stored in the abstraction. For example, in a singly-linked list, pointers connecting elements must be acyclic. Because implementation details of abstractions are not visible to the outside, such properties can often be described in terms of abstract attributes of the abstraction. These attributes are abstract in that they do not necessarily have concrete storage in the abstraction.

Figure 5 shows the grammar and examples of some annotations that we developed for optimizing loops that operate on user-defined array abstractions [17]. The annotations in Figure 5 are preliminary and need to be extended in many ways. However, as shown in the following, these annotations serve as informative examples to illustrate what semantics need to be described by a complete annotation language.

4.1 Function Annotations

In Figure 5(b), all annotations except (1) and (5) are function annotations. These annotations describe semantics of functions that operate on the $floatArray$ and *Range* data abstractions, which in turn are described in (1) and (5). The semantics of these functions can be separated into the following categories.

- *Restrictions on the inputs of the operation.* In Figure 5(a), three annotations, *read*, *allow-alias*, and *restrict-value* are used to describe input data of a function abstraction. As shown in examples (2), (4), (6) and (8) in Figure 5(b), *read* lists all the memory locations that are accessed by the operation; *allow-alias* describes restrictions on aliasing relations between locations of the input data—everything not listed in *allow-alias* cannot be aliased with other inputs; *restrict-value* describes relations between values of input data. Note that *restrict-value* can also be used to describe relations between input data and results, as illustrated in examples (6) and (8).

- *Modification side-effects.* In Figure 5(a), the $modify$ annotation describes modification side effects, specifically, what data (variables) are modified by the operation. The items listed by $modify$ must include all memory storage reachable from the function parameters and global variables.

- *Relations between results and inputs.* In Figure 5(a), the annotations *new-array*, $modify$-*array*, *restrict-value*, and *alias* can all be used to describe relations between the results and the inputs of an operation. The annotations *new-array* and $modify$-*array* are specific to array abstractions. The *restrict-value* annotation describes relations between values of inputs and results. The *alias* annotation describes the aliasing relations between inputs and results.

- *Rewrite annotations.* In Figure 5, the *inline* annotation is essentially a transformation directive that eliminates abstraction boundaries. It describes an operation by replacing it with a collection of equivalent operations. The *inline* annotation therefore can be seen as a transformation specification for rewriting function abstractions.

4.2 Data Annotations

In Figure 5(b), examples (1) and (5) describe the semantics of user-defined data abstractions. Specifically, they describe properties of the $floatArray$ and *Range* classes. These properties can be separated into the following categories.

- *Data attributes.* In Figure 5(b), example (1) uses the *is-array* annotation to specify that the $floatArray$ class has three data attributes: *dim*, *len* and *elem*, where *dim* is a single scalar value, and *len* and *elem* are collections of values that are indexed by a sequence of integer parameters. Similarly, example (5) uses the *has-value* annotation to specify that the *Range* class has three data attributes: *stride*, *base* and *len*, where all the attributes are scalar values. Data attributes conceptually model the values stored within a data abstraction. However, internally the values may be implicitly represented in various forms and concrete storage may not be found for the specified attributes.

- *Properties of attributes and relations among them.* In Figure 5(b), when example (1) uses *is-array* to describe $floatArray$, it implicitly conveys that $floatArray$ has FORTRAN90 array semantics; that is, the *dim* and *len* attributes describe the shape of the array, the *elem* attribute returns the elements within the array, and no elements of the

```
<annot> ::= <annot1> | <annot1>;<annot>
<annot1> ::= class <cls_annot>
    | operator <op_annot>
<cls_annot> ::= <clsname>:<cls_annot1>;
<cls_annot1>::=
    <cls_annot2> | <cls_annot2> <cls_annot1>
<cls_annot2>::= <arr_annot>
    | inheritable <arr_annot>
    | has-value { <val_def> }
<arr_annot>::= is-array{ <arr_def>}
    | is-array{define{<stmts>}<arr_def>}
<op_annot> ::= <opdecl> : <op_annot1> ;
<op_annot1> ::=
    <op_annot2> | <op_annot2> <op_annot1>
<op_annot2> ::= modify <namelist>
    | new-array (<aliaslist>){<arr_def>}
    | modify-array (<name>) {<arr_def>}
    | restrict-value {<val_def_list>}
    | read <namelist>
    | alias <nameGrouplist>
    | allow-alias <nameGrouplist>
    | inline <expression>
<arr_def> ::=
    <arr_attr_def> | <arr_attr_def> <arr_def>
<arr_attr_def> ::= <arr_attr>=<expression>;
<arr_attr> ::= dim  | len (<param>)
    | elem(<paramlist>)
    | reshape(<paramlist>)
<val_def> ::= <name>; | <name>;<val_def>
    | <name> =  <expression> ;
    | <name> =  <expression> ; <val_def>
```

(a) grammar

```
(1) class floatArray:
inheritable is-array { dim = 6;
        len(i) = this.getLength(i);
        elem(i$x:0:dim-1) = this(i$x);
        reshape(i$x:0:dim-1) = this.resize(i$x); };
(2) operator floatArray::operator =
(const floatArray& that):
modify-array (this) {
      dim = that.dim; len(i) = that.len(i);
      elem(i$x:1:dim) = that.elem(i$x); };
(3) operator +(const floatArray& a1,double a2):
new-array () { dim = a1.dim; len(i) = a1.len(i);
             elem(i$x:1:dim) = a1.elem(i$x)+a2; };
(4) operator floatArray::operator ()
(const Range& I):
restrict-value { this = { dim = 1; } };
            result = {dim = 1; len(0) = I.len;}; };
new-array (this) { dim = 1; len(0) = I.len;
            elem(i) = this.elem(i*I.stride + I.base); };
(5) class Range: has-value {stride; base; len; };
(6) operator Range::Range(int _b,int _l,int _s):
modify none; read {_b,_l,_s}; alias none;
restrict-value { this={base =_b;len=_l;stride=_s;};};
(7) operator floatArray::operator() (int index) :
inline { this.elem(index) };
restrict-value { this = { dim = 1; };};
(8) operator + (const Range& lhs, int x ) :
modify none; read {lhs,x}; alias none;
restrict-value { result={stride=lhs.stride;
              len = lhs.len; base = lhs.base + x; };};
```

(b) example

Figure 5. Annotation language

array are aliased. Similarly, the attributes described in example (5) for the *Range* class must satisfy $len \geq 0$. The *allow-alias* and *restrict-value* annotations in Figure 5 need to be extended to describe such properties. Figure 3 shows some of the extensions to these annotations.

- *Relation between attributes and functions*. In Figure 5, each attribute declaration must be followed by a definition, which includes function calls necessary to extract the attribute values from the data abstraction. Furthermore, functions that operate on data abstractions are described in terms of their effects on the attributes. Examples of such annotations are discussed in more detail in Section 4.1.
- *Rewrite annotations*. In Figure 5, the *is-array* annotation specifies a collection of definitions that can be used to replace an array abstraction under certain conditions. Specifically, we can replace the array abstraction with a more efficient equivalent implementation. Such annotations are very similar to the *inline* directive in example (7) and are used solely for optimization purposes.

4.3 Object-oriented annotation

In object-oriented languages, user-defined abstractions can inherit from each other and have sub-type relations among one another. For example, in C++, a derived class can adapt the behavior of its superclasses by re-implementing virtual functions. Thus, an abstraction annotation language must model relationships between different abstractions. In Figure 5(b), the *inheritable* annotation in example (1) specifies that the array semantics described by the *is-array* annotation can be inherited by subclasses of $floatArray$. In general, an annotation language needs not only to support the inheritance of semantic properties, but also to provide mechanisms to specify how behaviors of abstractions are adapted by inheritance.

4.4 Discussion

The annotations in Figure 5 are not a complete annotation language, which is the topic of our future research. This section uses our prior experience to summarize and to speculate on what properties of user-defined abstractions need to be described by an annotation language in order to significantly extend the applicability of compiler analysis and optimizations.

An annotation language is not complete unless we can verify that the specified properties reflect the implementations of abstractions. Otherwise, misinformed annotations would lead compilers to generate incorrect programs. We believe that identifying what properties need to be annotated is a significant step toward verifying such properties, as program analysis often needs external annotations from programmers to be effective. Our future research includes both identifying additional annotations that would benefit compiler optimizations and developing program analysis techniques to verify such annotations.

5 ROSE Infrastructure

We are implementing our work on optimizing user-defined abstractions within ROSE, a U.S. Department of Energy (DOE) project to develop an open-source compiler infrastructure for optimizing large-scale (on the order of a million lines or more) DOE applications [10, 11]. The ROSE framework enables tool builders who do not necessarily have a compiler background to build their own source-to-source optimizers. The current ROSE infrastructure can process C and C++ applications, and we are extending it to support FORTRAN90 as part of on-going work.

The ROSE infrastructure provides several components to build a source-to-source optimizer. A complete C++ front-end is available that generates an object-oriented abstract syntax tree (AST) as an intermediate representation. Optimizations are performed on the AST. The AST preserves the high-level C++ language representation so that no information about the structure of the original application, its abstractions in particular, is lost. A C++ back-end can be used to unparse the AST and generate C++ code. The user builds the "mid-end" to analyze or transform the AST, and ROSE assists by providing a number of mid-end components, including a predefined traversal mechanism, an attribute evaluation mechanism, whole-program analysis capabilities, transformation operators to restructure the AST, and predefined optimizations. ROSE also provides support for library annotations whether they be contained in pragmas, comments, or separate annotation files.

6 Related Work

Several projects address optimization of user-defined abstractions, including our own prior work on loop optimizations for array abstractions [10, 17]. Among these, our goals are most similar in spirit to the Broadway compiler by Guyer and Lin [6]. Broadway contains a specific annotation language suitable for optimizing the use of C libraries written in an object-oriented style. We consider a broader set of possible annotations that can directly express (a) relationships between function and data abstractions, and (b) the unique characteristics of object-oriented languages.

Other compiler projects can optimize library use, especially in the context of *Telescoping Languages* [8]. The SUIF compiler [1], MAGIK compiler [5], and MPC++ [7, 4] all provide a programmable level of control over the compilation, but require implementation within the compiler itself. In contrast, Schupp, *et al.*, develop for C++ an expression simplifier, which users extend for optimizing their abstract data types through annotations inserted into the code [12]. Similarly, users of CodeBoost (for C++) tag variables in the source, with tags interpretable by its rule-based rewrite system [3]. We share the design goal of developing annotations which may be provided by library developers who do not have a compiler background.

Template Meta-Programming can also optimize user-defined abstractions [13], but only when optimizations are isolated within a single statement. Loop fusion across statements, which requires dependence analysis, is beyond this technique.

Other approaches embed semantic knowledge within the compiler. Wu, *et al.* [15], proposed *semantic inlining*, which allows their compiler to treat user-defined abstractions as primitive types in Java. Artigas, *et al.* [2], devised an *alias versioning* transformation that creates alias-free regions in Java programs for optimizing loops over Java primitive arrays and array abstractions. Wu and Padua [16] investigated automatic parallelization of loops operating on user-defined containers, but require the compiler to know the semantics of all operators. In contrast to these approaches, we target general abstractions by allowing the programmer to communicate explicitly with the compiler.

7 Conclusions and Future Work

This paper discusses the features of an annotation language that we believe to be essential for optimizing user-defined abstractions. These features should capture semantics of function, data, and object-oriented abstractions, express abstraction equivalence (*e.g.*, a class represents an array abstraction), and permit extension of traditional compiler optimizations to user-defined abstractions. Our future work will include developing a comprehensive annotation language for describing the semantics of general object-oriented abstractions, as well as automatically verifying and inferring the annotated semantics.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The suif compiler for scalable parallel machines. In *in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995.

[2] P. V. Artigas, M. Gupta, S. Midkiff, and J. Moreira. Automatic loop transformations and parallelization for Java. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.

[3] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Proc. Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[4] S. Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.

[5] D. R. Engler. Incorporating application semantics and control into compilation. In *Proc. USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, USA, October 1997.

[6] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 39–52, Berkeley, CA, Oct. 3–5 1999. USENIX Association.

[7] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and implementation of metalevel architecture in C++—MPC++ approach. In *Proc. Reflection '96 Conference*, April 1996.

[8] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *J. Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.

[9] D. Quinlan, M. Schordan, Q. Yi, and B. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In *16th Annual Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Oct. 2003.

[10] D. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen. Classification and utilization of abstractions for optimization. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods*, Paphos, Cyprus, October 2004.

[11] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proc. Joint Modular Languages Conference*, 2003.

[12] S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. User-extensible simplification—type-based optimizer generators. In *Proc. International Conference on Compiler Construction*, volume LNCS 2027, pages 86–101, Genova, Italy, April 2001. Springer-Verlag.

[13] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[14] B. S. White, S. A. McKee, B. R. de Supinski, B. Miller, D. Quinlan, and M. Schulz. Improving the computational intensity of unstructured mesh applications. In *Proc. International Conference on Supercomputing*, Boston, MA, USA, June 2005.

[15] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Improving Java performance through semantic inlining. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, Mar 1999.

[16] P. Wu and D. Padua. Containers on the parallelization of general-purpose Java programs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Oct 1999.

[17] Q. Yi and D. Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.