Implementing Incrementalization Across Object Abstraction

Michael Gorbovitski, Tom Rothamel, Yanhong A. Liu, Scott D. Stoller
Computer Science Dept., State Univ. of New York at Stony Brook, Stony Brook, NY 11794
{mickg,rothamel,liu,stoller}@cs.sunysb.edu

ABSTRACT
We have implemented an interpreter (InvTS) for a declarative rule language (InvTL) supporting invariant-driven transformations of object-oriented programs. Using a library of rules, it can perform incrementalization across object abstractions, allowing the programmer to write clear, straightforward code while relying on InvTS to generate sophisticated and efficient implementations.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *Code Generation, Optimization*
D.2.1 [**Software Engineering**]: Requirements/Specification
D.2.3 [**Software Engineering**]: Coding Tools and Techniques
D.2.13 [**Software Engineering**]: Reusable Software

General Terms
Performance, Design, Languages

Keywords
Transformation Languages, Incrementalization, Refactoring, Instrumentation, Python, May-Alias Analysis

1. INTRODUCTION
Abstraction is fundamental in computer science. It allows for encapsulation in systems and components by separating the "what" on data and operations on the data from the "how". This enables the assembly of software components into complex software systems.

Operations on the data can be classified into queries or updates. Queries compute results using data, and updates change data.

Implementation of queries and updates can vary significantly. In a straightforward implementation, each operation does its respective query or update and is clearly separated from other queries and updates. Often, this leads to poor performance, as expensive queries are repeatedly executed. Using incremental maintenance of the results of expensive queries, a sophisticated implementation can have good performance. However, the code for incremental maintenance is less modular and more error-prone.

As the number of queries and updates grows, the situation becomes worse, especially when queries and updates cross component boundaries. Considering all complex dependencies and tradeoffs and deciding on how to maintain which results becomes increasingly difficult. This produces code that is significantly more difficult to write, understand, and maintain.

This presents a conflict between clarity and efficiency. Programmers could be significantly more productive if, while writing clear, straightforward implementations, they could rely on automated tools to generate sophisticated and efficient code.

A method to resolve the conflict is presented in [4]. We present a powerful system (InvTS / InvTL) that supports this method. It allows each software component to be written in a clear and modular fashion in any supported object-oriented language. InvTS identifies and analyzes queries and updates across object abstraction. Using a library of rules, it generates sophisticated and efficient code that incrementally maintains results of repeated expensive queries with respect to updates of their parameters.

InvTS is an interpreter for a declarative rule language (InvTL) that supports invariant-preserving transformations of object-oriented programs. InvTL is the language we use to write the incrementalization rules presented in [4].

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

2. RULES
As in most transformation systems, a rule specified in InvTL consists of a pattern that matches and then transforms code. The goal of InvTS is to allow a programmer to write straightforward queries, while InvTS automatically generates sophisticated code that incrementally maintains the values of the queries. Thus, the main parts of a rule are:

1. A pattern that matches a query.
2. Transformations that describe how to incrementally update the result of the query when the values it depends on change.

The pattern that matches the query, coupled with the variable that holds the result of the query is called an invariant, because the rule makes sure that at all points in the program where the query appears, the value of the variable containing the result is always equal to the value of the query. The purpose of a rule is to maintain an invariant. For each possible type of update to the invariant, a rule must specify a way to maintain the invariant. InvTL guarantees that for each update to an invariant, code that maintains it is applied; if there is an update that the rule does not describe, InvTS assumes that the programmer who wrote the rule did not take this type of update into account, and that the rule cannot be safely applied.

We now present a rule that maintains the size of a set in a Java-like language. We assume that the set can only be modified via the "add" and "remove" methods. We also assume that the "size" method of the set class, Set, is $O(n)$, and returns the size of the set. We assume that Set has a "contains" method that is $O(1)$.

```
inv count={F.size()}
if (type(F)==Set)
(
   at {F.add(x)}
   do before {
        if (not F.contains(x)){count=count+1} }

   at {F.remove(x)}
   do before {
        if( F.contains(x)) {count=count-1} }
)
```

This rule shows the "inv", "if", "at", and "do before" clauses. The "inv" clause declares the invariant. The "if" clause is used to check that the pattern that we matched is actually an instance of a Set. Each "at" clause defines a type of update to the invariant, and the "do before" clause specifies the actions needed to maintain the invariant. There also exist "do after" and "do instead" clauses. These insert maintenance code after and instead of the code that updates the invariant, respectively. The "de" clause, not shown above, is used to specify modifications to code that are non-local, such as adding new methods.

3. IMPLEMENTATION

InvTS currently supports Python as the language to be transformed. It applies InvTL rules to the target Python code in the following way:

All rules in the rule library are repeatedly considered and applied until no rule can be applied. To apply the rule to the input, InvTS finds all matches to the query pattern. For each match, it then identifies all places in the source that update the values the query depends on. To do that for Python, InvTS need to perform may-alias and data-flow analysis on the program. It uses Goyal's may-alias analysis [3], which is an efficient implementation of Choi's alias-analysis algorithm [2]. The algorithm is not directly suited to analyzing object-oriented programs, as it is designed for the C language. It is designed to handle pointers with multiple level of indirection. To apply the algorithm, we expand all assignments of objects with more than one field into multiple assignments, thus turning the program into a C-like program. This procedure is limited to object graphs that are not self-referencing. The rule is not applied if any self-referencing structures are encountered in the may-alias set of a variable that the query depends on. Both Choi's algorithm and the expansion procedure require the control flow graph (CFG) of the program, the construction of which is a non-trivial problem for Python. We generate the CFG using the PyPy framework [5].

After identifying all sites that update the values the query depends on, InvTS attempts to match patterns in "at" clauses to all of these sites, and, if successful, applies the transformations specified in the corresponding "de" and "do" clauses.

4. EXPERIMENTS

We applied InvTS to a diverse set of problems. The most interesting application was the automatic optimization of a straightforward implementation of Core RBAC [1] in Python [4]. RBAC is a framework for specifying and determining access permissions to various resources based on roles of users. RBAC can be used to determine whether a given user has certain roles, whether a role is allowed to perform an operation on an object, whether a user is allowed to do the same, and other related queries.

The ANSI standard for RBAC is specified in the Z language. The specification calls for sets to maintain the mappings among users, roles, permissions, objects, and operations. The queries described in the previous paragraph are specified as set comprehensions in Z. As we were implementing it in the most straightforward manner, the queries were implemented as Python comprehensions.

We have applied five rules to incrementalize the straightforward RBAC implementation. These rules incrementally maintain five different types of set comprehensions, such as $\{\boldsymbol{e}$ for v in $\boldsymbol{s}$ if $\boldsymbol{e_1}\}$. An italicized variable denotes a variable; a bold variable denotes an expression without side effects.

InvTS incrementalized seven queries. Even with just seven queries, the size of the source code increased dramatically: the original, straightforward implementation took 125 LOC, while the optimized implementation was 610 LOC. The complexity of the source code also greatly increased. The straightforward RBAC implementation was very easy to implement based on the specification in Z. When we implemented the optimized version by hand, the resulting code had a great number of cross-method dependencies, making the implementation time-consuming and making it difficult to prove its equivalence to the Z specification.

After the incrementalization, the automatically optimized version was considerably faster, taking under a second to process our largest example. The straightforward implementation took over 5 seconds [4].

5. CONCLUSION

We have developed a language and a tool that can be used to easily specify and apply invariant-driven rules. This lets the programmer write clear and straightforward programs that, using a library of InvTL rules, are automatically transformed into efficient and correct programs. This helps resolve the conflict between clarity and efficiency by implementing a systematic method for incrementalization across object abstraction [4].

Besides optimization, InvTS can be used for program instrumentation, code refactoring, verification, and other purposes.

6. REFERENCES

[1] American National Standards Institute, Inc. Role-based access control. ANSI INCITS 359-2004. Approved Feb 19, 2004 , http://csrc.nist.gov/rbac/

[2] J-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointerinduced aliases and side effects. *In Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 232--245, Jan 1993

[3] D. Goyal. Transformational Derivation of an Improved Alias Analysis Algorithm. *Higher-Order and Symbolic Computation, 18, 1/2* , Feb 2005

[4] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, Y. E. Liu. Incrementalization across object abstraction. *In Proceedings of the ACM SIGPLAN 2005 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Oct 2005

[5] PyPy Team. PyPy 0.6 - A Python Interpreter in Python, Jun 2005 , http://codespeak.net/pypy/