OSUIF: SUIF 2.0 With Objects

Andrew Duncan, Bogdan Cocosel, Costin Iancu, Holger Kienle, Radu Rugina
Urs Hölzle, Martin Rinard
Department of Computer Science
University of California
Santa Barbara, CA 93106
http://www.cs.ucsb.edu/~osuif

Abstract. OSUIF is an extension to SUIF 2.0 that provides support for the compilation of object-oriented languages. OSUIF extends standard SUIF in three main areas: symbol table, intermediate language, and exception handling. The resulting system should be able to support compilers for many (but not all) object-oriented languages. The two initial OSUIF front ends will support C++ and Java.

1. Introduction

The original SUIF system [SCG94] is a research compiler for procedural languages like C and Fortran. While these languages cover a large spectrum of compiler research, many software developers (including the SUIF developers) use object-oriented languages today. As a result, compilation of object-oriented languages has become a relevant research area.

Object-oriented languages like C++ and Java are quite different from procedural languages. The most striking language difference may be the support for polymorphism that allows pieces of code to be reused with objects of different types. This polymorphism can be exploited either statically (at compile time) in the form of parameterized types, or dynamically (at runtime) in the form of dynamic dispatch. In addition to this basic language difference, object-oriented languages encourage a different programming style, so that typical object-oriented programs have runtime characteristics that are very different from procedural programs. In particular, methods (procedures) are much smaller and consequently calls are more frequent [CG94]. Additionally, programs tend to allocate more frequently [DDZ94] and use sophisticated run-time systems with support for exceptions, threads, or garbage collection.

SUIF is a natural tool for researchers to analyze and optimize the execution of object-oriented programs. However, while it would certainly be possible to write a SUIF front end for an object-oriented language, such a compiler would probably not represent the best vehicle for research in compilers for object-oriented languages because much of the source-level semantics would be lost during the translation to the SUIF intermediate language, thus making it difficult to implement high-level optimizations that rely on high-level semantic information.

OSUIF's solution to this problem is quite similar to the approach used in SUIF to support the automatic parallelization of programs. Instead of translating array accesses immediately into sequences of low-level RISC-like intermediate instructions, SUIF keeps them as high-level array access instructions so that analysis and transformation passes can better optimize the programs. Similarly, OSUIF defines a small number of high-level constructs to retain the high-level structure of the source program as long as necessary. Once the passes specific to object-oriented features have performed their work, the high-level instructions are lowered to standard SUIF so that other passes can complete the compilation.

In the remainder of this paper we briefly describe the major parts of the OSUIF design: extensions to the symbol table, new intermediate instructions, and support for exceptions. We then describe how SUIF 2.0's design for extensibility helps us implement OSUIF as a pure SUIF extension, i.e., without modifying the base SUIF system. Finally, we conclude with a summary of the OSUIF architecture as a part of SUIF and how it will be used by a client.

2. Extensions to the Symbol Table

The SUIF symbol table is the central data structure that represents most of a program's semantics, including the definitions of scopes, types, and variables. To support object-oriented languages, the symbol tables must represent the more elaborate semantics of these languages. In particular, the symbol table must represent subclass relationships (i.e., inheritance) and the associated name lookup rules. Additionally, the symbol table must also represent subtype relationships, i.e., assignment compatibility. (Subclass and subtype relationships are not necessarily identical, as demonstrated by Java's dual class and interface hierarchies.)

OSUIF represents class types as extensions of SUIF's group_type (which encompass structs and unions). In essence, class types allow the user to add methods to a group_type in addition to fields. Similarly, a new method_type extends SUIF's procedure_type, allowing the user to distinguish the first argument as the receiver and associating the method with the class defining it. By virtue of being extensions (subclasses) of base SUIF classes, OSUIF types or methods can be manipulated by later non-OSUIF passes because these passes can simply ignore the extra information contained in the OSUIF objects. For example, a field in an OSUIF class type looks just like a field in a C struct to a scalar optimization pass.

However, program semantics cannot easily be captured with data structures alone: a large part of the semantics is expressed through the interpretation of these data structures. For example, while it is fine to represent a link between a subclass and its superclass as a pointer, this pointer does not capture the meaning of the inheritance link. In C++, for example, it could be a public or private inheritance link, or a virtual *vs.* nonvirtual link. While one can add flags that record whether the link is public or private, this wouldn't solve the problem. When determining which methods of the superclass are visible in a subclass, for example, the actual semantics of the language are encoded in the name resolution algorithm that interprets these flags, not in the data structure itself. In summary, a properly abstracted symbol table representation consists not only of state but also behavior (algorithms).

Unfortunately, object-oriented languages disagree on the precise semantics of inheritance, subtyping, or dynamic dispatch. Since the type systems of object-oriented languages are much more complex than those of procedural languages like C or Fortran, it is much harder to find a consistent, common framework that encompasses all object-oriented languages. In contrast, standard SUIF can accommodate both C and Fortran in a single type system because the two languages are similar enough. Owing to the diversity of object-oriented languages, OSUIF takes a different approach.

Instead of trying to contain the union of all object-oriented programming languages, OSUIF provides an extensible framework that abstracts away the details of the particular language and is designed to be extended by each front end. Thus, the OSUIF symbol table itself contains no language-specific information. For example, class_symbol_table is an abstract class that declares some virtual functions (e.g., lookup_method) that must be implemented by a client with a specific source language in mind. A Java compiler, for example, would implement a subclass java_class_symbol_table that implements all the required functionality and possibly adds Java-specific state and behavior to symbol tables. The simplified OSUIF inheritance tree in Figure 1 shows that we will be adding a new symbol table, a new symbol and two new related types (shown in italic).

In a typical object-oriented language, the declaration of a new class introduces a new scope. For this reason we associate with each class declaration a new symbol table. The services of the (already-existing) root class symbol_table can be used to store fields or functions associated with the entire class. The intermediate class group_symbol_table is used for expressing compound structures. It provides methods for adding and removing fields, and retrieving fields by name. Our derived class adds support for adding and removing methods, and for retrieving them not only in the immediate class scope, but also those of its ancestor classes. Most of these facilities will not be called directly by an OSUIF user, but through methods provided in the descendants of group_type.

The SUIF class group_type provides services for adding, removing, and querying about fields in its symbol table. OSUIF provides a subclass that maintains lists of ancestors and descendants, and an interface to its symbol table, for access to the methods declared or defined in the class. The presence of (interface) inheritance means that an instance of a class C may substitute in a context expecting an instance of an ancestor of C. OSUIF will provide an interface for expressing type compatibility; by implementing them a front-end provides the meaning. Subsequent passes can, for example, carry out class hierarchy analysis [DGC95] without having to know details about the source language.

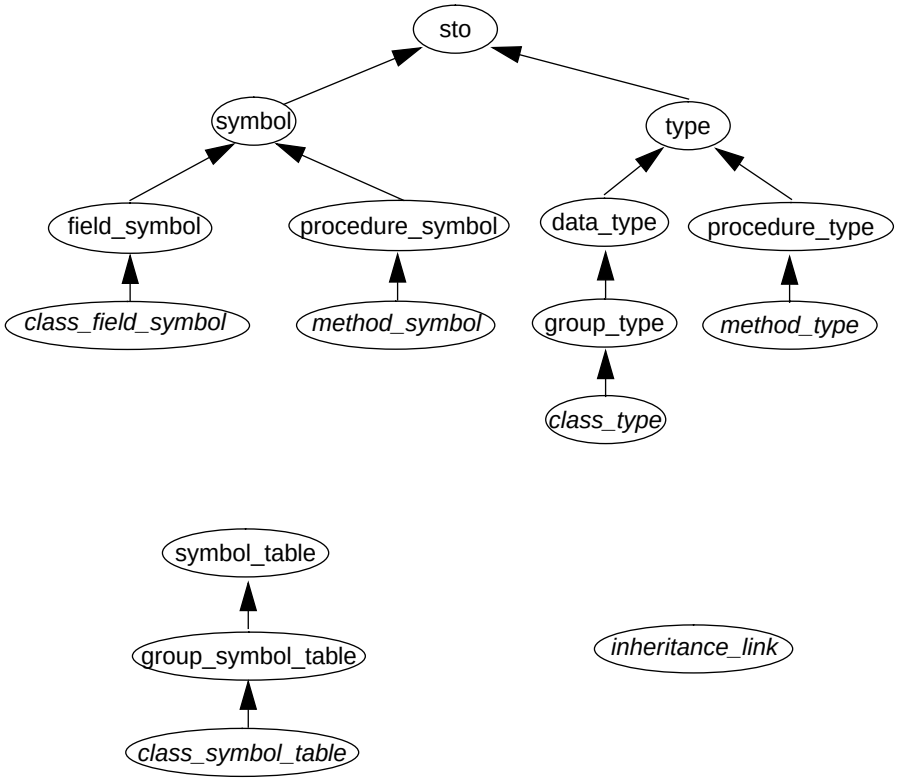

Figure 1. Simplified OSUIF inheritance hierarchy

In addition to the basic features of name, argument types and result types provided by the procedure_type class, method_type contains additional functionality such as access to the receiver type (which can also be viewed as the first argument). Similarly, method_symbol adds additional information to its base procedure_symbol.

Many object-oriented languages introduce access modifiers, for instance public or private. These modifiers make sense when attached to variables. In its extensions to the SUIF symbol classes, OSUIF provides a mechanism for attaching access modifiers to fields or methods. These modifiers will have no effect on passes that do not expect to find them.

Access modifiers are also used in some object-oriented languages to describe the inheritance relationship itself. OSUIF adds a new class, inheritance_link, to describe the inheritance relation, and attaches access modifiers to the link itself. The independent existence of inheritance_link as an OSUIF class also allows more flexibility in implementing the semantics of inheritance.

(The SUIF 2.0 system provides a mechanism, called *qualification,* for attaching modifiers to types entered into a symbol table. However this mechanism is not appropriate for expressing the access or inheritance properties described above. Qualifiers are intended to describe the behavior of storage, for example const, volatile, or register. Qualifiers are associated with types; access modifiers with variables.)

3. Extensions to the Intermediate Language

The most distinctive feature of the object-oriented methodology is dynamic dispatch: a single call site in the source text can represent many possible destinations, the choice to be determined at runtime. In C++, the programmer can specify whether a method is polymorphic (will use dynamic dispatch) or not via the keyword virtual; by contrast in Java all instance methods are (at least potentially) polymorphic. A compiler for an object-oriented language may have considerable latitude in how it implements dynamic dispatch. For this reason OSUIF provides for the expression of message sends at a high level.

OSUIF provides the essential facilities for dynamic dispatch by extending the SUIF class call_instruction. A method call keeps information about its static receiver type (i.e., the decalred type of its receiver). It is the responsibility of a front-end (translator to OSUIF) to make sure the method call node is constructed with the proper receiver. Subsequent passes have access to the method's signature through the inherited instruction interface, and to the receiver through the new interface. It will be the responsibility of a back end (translator from SUIF) to generate code to resolve the method dispatch correctly, based on the state of the receiver at runtime. OSUIF *per se* will only support one receiver per call (i.e., no multiple dispatch as in CLOS or Dylan). However, it should be possible to extend SUIF to express this kind of semantics by using the OSUIF classes as starting points.

This treatment of dynamic dispatch in OSUIF is comparable to the way array instructions are represented in SUIF. OSUIF does not prescribe any particular implementation for method calls, such as virtual function tables; the choice of dispatch implementation is left to subsequent passes that lower OSUIF to ordinary SUIF. For example, based on information collected by earlier passes an implementation might use row-displacement compression of dispatch table [DH95] or inline caching [DS84] instead of standard virtual function tables. Thus, the OSUIF user may experiment with substituting several different lowering phases into the compilation pipeline and compare the static and dynamic properties of the resulting code.

The constructors of a class are distinguished from other methods in several ways. For example, although they are statically dispatched and are not associated with a pre-existing instance, as with a class method (in Java or C++ a static method), they do have a pointer to the (new) current instance as with an instance method. Further, they mark a place in the program where the exact type of a variable is known. This is useful knowledge that we want to preserve. Similarly, destructors are special calls; recognizing them easily helps us reason about the behavior of a program. OSUIF provides an extension to the instruction class that expresses constructors and destructors explicitly.

4. Extensions to the Control Flow

Although not, strictly speaking, part of the object-oriented paradigm, exception handling is recognized as a valuable tool for making programs more robust, providing a structured way to express the processing of exceptional events at runtime. Moreover, for an intermediate language to support many common OO languages, it must provide a way to express the handling of exceptions. However, the semantics of exceptions vary considerably across popular languages.

For example, in C, a programmer could handle runtime exceptions using signals, or with setjmp() and longjmp(). Java has a library interface class Throwable, and a hierarchy of subclasses, any of which may be thrown at runtime. In C++, any type of value may be thrown as an exception. In Ada, exceptions are simple named entities. Ada and Eiffel provide a single clause that catches exceptions; in Java and C++ there are multiple catch blocks, that use parametric polymorphism to determine how to handle the thrown value. Java and Eiffel add runtime code to keep track of the call chain, which is reported if the exception causes the program to terminate; C++ does not. Eiffel adds a retry command that restarts the routine in which the exception occurred, but without executing any scope initialization code, allowing a routine to attempt to preserve its class invariant before bailing out. Other languages can simulate this with nested loops. In some languages it is possible to return normally after handling an exception, without finishing the routine; in others a routine that generates an exception must successfully retry its body or pass the exception.

In keeping with the design philosophy described above, OSUIF will provide a framework for expressing the essential and common behavior of exceptions. Clients of OSUIF may extend these classes to add support for more features as desired, and implement later passes that translate these into specific instructions. Such passes may investigate for example the trade-offs between optimizing the execution speed of exceptions *vs.* reducing the penalty that an exception-handling system incurs even when they are not used.

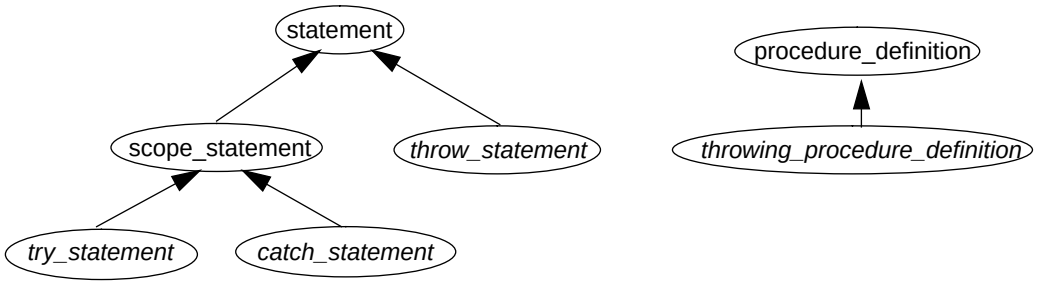

Figure 2. OSUIF exception hierarchy

SUIF already provides a class called scope_statement, which represents the introduction of a nested scope into the flow of control. OSUIF represents a try block by deriving from this class. In addition to the scope_statement's body, we provide a list of catch statements and a can_handle( ) predicate that takes a type.

In order to support the exception specification (throw clause) semantics provided in C++ and Java, in which a function declares what exceptions it may throw, OSUIF derives a new class exception_throwing_definition from procedure_definition. It behaves just like a conventional procedure body; its addition is provision for storage of a list of throw types, and services for adding to, removing from, and querying about the list.

To express the actual raising of an exception, we derive directly from statement a new class throw_statement, and provide accessors for its single optional argument, and implement an is_rethrow() predicate that returns true if there is no argument. To catch a thrown exception, we use a class derived from scope_statement. It adds information about the type of exception it catches, and about its parent try statement.

Note that such predicates as can_throw() or can_handle() depend on a general compatibility tester for types. As discussed above, this is a facility that the OSUIF client must provide by extending the symbol table classes to suit the specific needs of the language semantics.

5. Additional Services

In addition to a full set of extensions to SUIF 2.0, OSUIF will provide some tools that use these extensions to perform common types of analysis and manipulation of programs in SUIF intermediate representation. These may include class hierarchy analysis [DGC95], member lookup [RS97], construction of virtual tables [DH95], and the stack tracing associated with exceptions. In addition, many object-oriented languages are expected or required to run with garbage collection. OSUIF may provide passes that examine the suitability of intermediate code for being garbage collected [DMH92] — for example, verifying that pointers are always pointing to the head of heap objects, or identifying temporary pointer variables.

6. Related Work

Most previous work on intermediate representations for programming languages has focused on representing procedural semantics. Such frameworks can certainly be used for encoding and analyzing programs that have object-oriented features, but a considerable part of the semantic information will be lost in translating to the intermediate form. The Vortex project [DDG96] at the University of Washington uses an intermediate form that provided support for such object-oriented features as inheritance hierarchy, message sends and polymorphism, instance variable access, and run-time type inquiry. This system uses the concept of a generic function [BDG88] to represent the concepts of procedures, methods, and multimethods. A unified back-end is coupled to any number of language-specific front-ends; among the source languages supported are Cecil, C++, Java, and Modula-3.

The Illinois Concert system [CDG97] also supports the representation of high-level object-oriented information, and extends these to also manage the additional complexity that comes with concurrency and parallel programming.

7. Summary

The OSUIF libraries give the SUIF user the ability to express fundamental features of object-oriented programming languages. The classes defined in the libraries are extensible, so they may be tailored to the specific needs and semantics of a source language. OSUIF provides extensions to the basic SUIF symbol tables and intermediate representations. This collection of classes will be distributed in the same form as the standard SUIF library, will use the same unified Make process, and can be compiled to a shared object library. An OSUIF client may extend those classes to properly model the semantics of a given source language, and then build a front-end that depends on those extensions. In addition, OSUIF provides a library of routines and passes (libOO) that use the OSUIF extensions to perform common object-oriented analyses and manipulations, for example class hierarchy analysis, member lookup, construction of virtual tables, and the stack tracing associated with exceptions. At the highest level, research tools to perform object-oriented program analysis can be written using all of these levels of the SUIF system.

OSUIF is an early application of the extensibility provided by SUIF 2.0. With a little bit of work, the instances of these new classes can provide the same fundamental services as the SUIF base classes — in particular, they will be able to identify their runtime class, and store and retrieve themselves from disk. In this way a user can extend the behavior of the SUIF library in ways not specifically planned for in the library, without giving up the services offered by the base library. This extension can happen at the "user level," without the need for priveleged access to the base SUIF code. In addition, the substitutability that inheritance of interface provides helps pre-existing SUIF passes to behave correctly on the users new SUIF programs.

At present, the basic OSUIF interfaces are designed, and most of the basic functionality is implemented. A full alpha implementation should be ready by the end of 1997. The project is expected to last for two more years after that. In the second year the OSUIF to SUIF lowering passes will be implemented, as will other libOO utilities. In the third year, we will be implementing global analyses and other passes using the OSUIF framework.

8. Acknowledgements

This work is funded by DARPA contract MDA904-97-C-0225 as part of the National Compiler Infrastructure (NCI).

9. References

[BDG88] D.G. Bobrow, L. G. DeMichael, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices*, vol. 28 (Special Issue), September 1988.

[CDG97] Andrew Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Supporting High Level Programming with High Performance: The Illinois Concert System. In *HIPS '97 Conference Proceedings* — High-level Parallel Programming Models and Supportive Environments (Workshop at IPPS '97), April, 1997.

[CG94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *21st Annual ACM Symposium on Principles of Programming Languages*, pp. 397-408, January 1994.

[DDG96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *OOPSLA '95 Conference Proceedings, ACM SIGPLAN Notices*, vol. 31, pp. 83-100, October 1996.

[DDZ94] David Detlefs, Al Dosser, and Benjamin Zorn. *Memory Allocation Costs in Large C and C++ Programs.* In Software — Practice and Experience, vol. 27, pp. 527-542, June 1994.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[DH95] Karel Driesen and Urs Hölzle. Minimizing Row Displacement Dispatch Tables. In *OOPSLA '95 Conference Proceedings, ACM SIGPLAN Notices*, vol. 30, pp141-155, October 1995.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In *PLDI Conference Proceedings, ACM SIGPLAN Notices*, vol. 27, pp. 273-282, July 1992.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, pp. 297-302, 1984.

[RS97] G. Ramalingam and Harini Srinivasan. A Member Lookup Algorithm for C++. In *PLDI Conference Proceedings, ACM SIGPLAN Notices*, vol. 32, pp. 18-30, May 1997.

[SCG94] Stanford Compiler Group. *SUIF: A Parallelizing and Optimizing Research Compiler.* Tech. Rep. CSL-TR-94-620, Computer Systems Lab, Stanford University, May 1994.