Representation-Independent Program Analysis

Michelle Mills Strout
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439
mstrout@mcs.anl.gov

John Mellor-Crummey
Rice University, Dept. of
Computer Science - MS 132
P.O. Box 1892
Houston, TX 77251-1892
johnmc@cs.rice.edu

Paul Hovland
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439
hovland@mcs.anl.gov

ABSTRACT

Program analysis has many applications in software engineering and high-performance computation, such as program understanding, debugging, testing, reverse engineering, and optimization. A ubiquitous compiler infrastructure does not exist; therefore, program analysis is essentially reimplemented for each compiler infrastructure. The goal of the OpenAnalysis toolkit is to separate analysis from the intermediate representation (IR) in a way that allows the orthogonal development of compiler infrastructures and program analysis. Separation of analysis from specific IRs will allow faster development of compiler infrastructures, the ability to share and compare analysis implementations, and in general quicker breakthroughs and evolution in the area of program analysis. This paper presents how we are separating analysis implementations from IRs with analysis-specific, IR-independent interfaces. Analysis-specific IR interfaces for alias/pointer analysis algorithms and reaching constants illustrate that an IR interface designed for language dependence is capable of providing enough information to support the implementation of a broad range of analysis algorithms and also represent constructs within many imperative programming languages.

1. INTRODUCTION

The past decade has witnessed a proliferation of compiler infrastructures; however, no compiler infrastructure has become universally adopted. The principal reason is that different compiler projects have requirements that are not addressed by a single infrastructure. For example, some projects require access to the compiler infrastructure and robust support for a particular language. In other cases, different research goals may be best supported with different intermediate forms. For instance, an infrastructure for analysis and transformation of application binaries is not well suited to support source-to-source transformation of high-level languages, and vice versa.

For projects whose analysis and transformation requirements are not met by an existing compiler infrastructure, the challenges are daunting. Existing compiler infrastructures are monolithic in design—one cannot readily use a single component of an infrastructure without adopting the entire infrastructure. As a result, groups focusing on analysis and transformation algorithms typically build fragile infrastructures for one or a few languages. Groups targeting multiple languages often implement only the simplest analyses and transformations. Since each compiler infrastructure typically has a unique intermediate representation (IR) and analysis and transformation are heavily integrated with the IR, each compiler infrastructure requires duplication of effort to implement a specialized version of existing analysis and transformation algorithms.

We are experimenting with an approach for decoupling program analysis from the IR. In the OpenAnalysis toolkit, we create analysis-specific interfaces between analysis algorithms and IRs for imperative programming languages. This approach separates the choice of program analysis implementation from the selection of a compiler infrastructure. Analysis algorithms interact with IRs using a well-defined set of interface methods, and infrastructure developers create an interface to their infrastructure that implements the methods for any subset of analysis algorithms.

Figure 1 provides an overview of the relationship between components in the OpenAnalysis tookit, clients, and IRs. The IR-specific implementation of analysis-specific interfaces are in the grey box. Analyses within the toolkit use the IR-specific implementations through an analysis-specific, *IR-independent* interface by making queries on program constructs such as statements through opaque handles. Some analyses share requirements, and thus the IR-specific implementation of all IR interfaces is a single component to enable implementation reuse. This is accomplished in C++ through the use of multiple inheritance and abstract base classes.

The analysis-specific IR interfaces contain abstractions that appear in most imperative programming languages. Imperative (or procedural) programming languages support the manipulation of state through the execution of a sequence of statements. In low-level representations such as assembly, the state is stored in registers and memory. Higher-level languages and their IRs store state in variables and heap allocations. In general, the evaluation of expressions generates values, which are stored into program state through assignment statements. The sequence of statements can be manipulated with control-flow constructs such as for loops and if statements. Procedures accept references to program state or values generated from expressions as parameters.

OpenAnalysis represents the following program control constructs with opaque handles: procedure definitions, statements, expressions, and labels. An example of a query involving program control might be a request for an iterator over all statements in a particular procedure. Some analyses require iteration over statements in a hierarchical fashion, for example to iterate over the statements in the true or false branch of an `if` statement. For example, to generate a control-flow graph, the CFG analysis manager creates basic blocks that contain statement handles to the IR. Also, the IR-specific interface implementation categorizes statement handles as simple, multiway branches, gotos, and so forth.

The main objective of the OpenAnalysis toolkit is to separate analysis from the intermediate representation (IR) in a way that eliminates analysis reimplementation for each compiler infrastructure. We also aim to satisfy the following subgoals:

1. The interface between an analysis algorithm and any imperative programming language IR should be relatively easy to implement.

2. The IR interface should support a significant range of analysis implementations.

3. Simple procedures should exist for developing analysis implementations and contributing them.

4. Using analysis results generated by analysis implementations within OpenAnalysis should be straightforward.

The ability to implement only the IR interface for an analysis of interest supports the first subgoal. The IR interfaces clearly specify what information the source IR must provide to OpenAnalysis to perform a specific analysis or set or related analyses. Another feature enabling quick startup is the tendency of abstract interfaces for the various analyses to share many concepts. This enables the efficient incremental implementation of analysis-specific IR interface capabilities for a particular IR. Overlap occurs because the analyses are all applicable to imperative programming languages.

The software architecture supports the other subgoals and the underlying theme that OpenAnalysis should be easy to start using. Since analyses can also be clients, using abstract interfaces for analysis results enables interaction between analysis results generated by the OpenAnalysis toolkit and analysis results generated in an IR-specific fashion. Abstract interfaces for analysis results also enable interchangeable variants of an analysis. Thus, when analyses that provide more accuracy are developed, they can be plugged in as the manager for that analysis type.

The OpenAnalysis toolkit is being used to analyze Fortran 90 programs represented in Rice's Open64/SL compiler, C++ programs represented in Lawrence Livermore National Laboratory's ROSE compiler, and application binaries for a wide variety of processor architectures including MIPS, Alpha, x86, Itanium, and Sparc. As such, OpenAnalysis is a crucial component for the HPCToolkit project at Rice and the automatic differentiation tools ADIC and OpenAD at Argonne National Laboratory.

This paper focuses on the details of designing analysis-specific IR interfaces and discusses how such interfaces can express imperative programming language constructs for a broad range of applicable analysis implementations. We present the analysis-specific IR interfaces for *alias analysis* algorithms and the data-flow analysis *reaching constants*.[1] Within the context of these two analyses we describe generic IR interface abstractions such as memory reference expressions, locations, expressions, and constants. We evaluate the abstractions in terms of what imperative programming language constructs they are capable of expressing. We also describe some of our experiences in developing analysis implementations using an IR-independent approach and present issues that require future research.

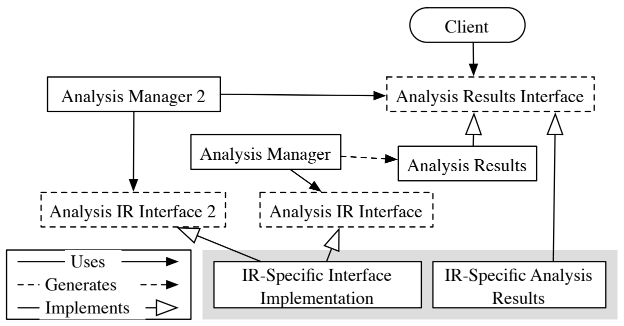

Figure 1: Overview of OpenAnalysis Software Architecture

2. IR INTERFACE FOR ALIAS ANALYSIS

In Imperative programming languages program state is manipulated through memory references. Program state is represented in OpenAnalysis with the location abstraction. Registers, local variables, global variables, and heap allocations are examples of locations. The goal of alias analysis is to determine which locations each memory reference may access. Memory references may access the same location (i.e., may alias) as a result of (1) accessing subsets of the same array, (2) constructs such as union in C and common blocks and equivalence in Fortran, (3) pointers, and (4) reference parameters. This section describes how the location and memory reference expression abstractions are implemented in OpenAnalysis and how they support generic alias analysis of imperative programming languages.

2.1 Location Abstraction

Figure 2 shows the class hierarchy for the Location abstraction in OpenAnalysis. Named locations such as local and global variables are represented with a `NamedLoc` and have a `SymbolHandle` associated with them. Dynamically allocated locations are represented with the `UnnamedLoc` subclass and associated with the statement that the allocation occurs. Invisible locations refer to locations within a procedure that are accessed by dereferencing nonlocal locations or parameters to the procedure. Invisible locations are conceptually the same as the invisible locations described in [7], extended parameters in [19], and nonvisibles in [17].

For named locations, information about static aliasing and the symbol scope is included within the `NamedLoc` data structure. Static aliasing includes information about which

[1]When combined with transformation this analysis is commonly referred to as constant propagation.

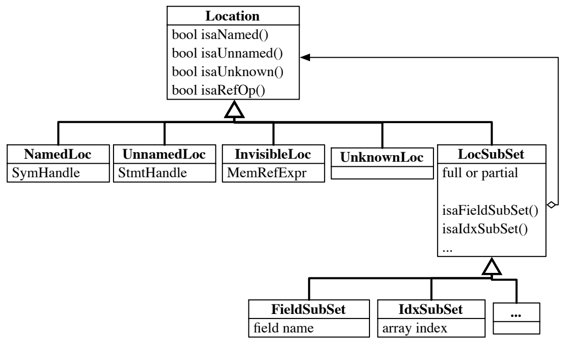

Figure 2: Location Class Hierarchy

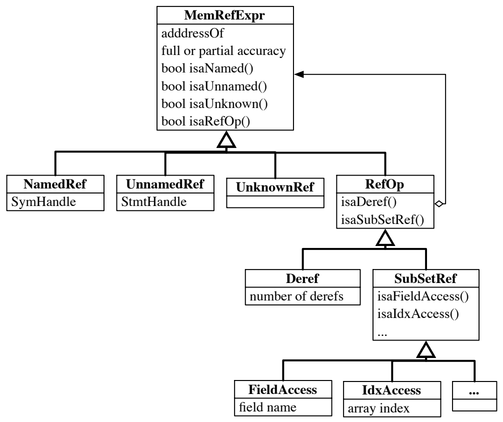

Figure 3: Memory Reference Expression Class Hierarchy

other locations a symbol must or may overlap with because of a language construct such as Fortran equivalence, common blocks, and C unions.

Aliasing due to accessing subsets of the same datastructure is expressed with location subsets. Location subsets are implemented with the `LocSubSet` class, using the Decorator pattern [8]. The concept of locations and location subsets is similar to the concept of location blocks and location sets introduced by Wilson in [19]. Our Location abstraction is, however, more general than the one presented by Wilson et al. The design of the `Location` class hierarchy enables adding subclasses that express different location subset concepts.

2.2 Memory References

As stated previously, locations (i.e. program state) are accessed through memory references. Table 1 shows example statements in various imperative languages and the memory references they contain. Memory references are associated with an opaque handle called a `MemRefHandle` that uniquely identifies a memory reference in the intermediate representation and in OpenAnalysis. One example of an appropriate `MemRefHandle` is the address of the IR object that represents the memory reference. The IR can cast the value in the handle to the appropriate pointer type, and OpenAnalysis sees unique values amongst all handles.

The memory reference expression (implemented with the class hierarchy shown in Figure 3) provides an avenue for a specific IR to express generic information about a memory reference. The third column in Table 1 specifies the memory reference expression objects that would appropriately describe the memory references in the second column. For example, the memory reference `q` in the Fortran 90 statement "`... = q`" is described as one dereference to a named reference to `q`. Notice that this description is similar to the description for the memory reference `*p` in the C statement "`... = *p`". This is due to the fact that even though the syntax and probably the IR for Fortran 90 and C are different, the semantics are similar and can be described as such to OpenAnalysis. As a final note, the Sparc code performs the same computation as the C and Fortran 90 statements; however, the IR is unable to provide any pointer assignment pairs or address computation descriptions since Sparc is not typed.

2.3 Alias Analysis

In OpenAnalysis, alias analysis managers are responsible for providing alias analysis results that indicate which locations a memory reference *may* or *must* access. The alias results must also indicate whether any pair of memory references may or must alias each other. Aliasing between two memory references is indicated if they both may or must reference the same location or locations that may or must overlap. When aliasing is due to accessing subsets of the same array, the fact that a partial `LocSubSet` of an array may overlap with the array itself or another subset is detected. For overlapping due to C unions or the Fortran `equivalence` keyword, the `NamedLoc` datastructure maintains a set of symbols that may or must overlap with it. For pointers, the alias analysis must maintain a static estimate of the dynamic relationships between memory references involving dereferences and locations. Aliasing due to reference parameters requires analysis of the memory references passed to all calls of a procedure.

While developing the alias IR interface, we surveyed the information used by a number of alias analysis algorithms that provide a broad spectrum of the tradeoff between accuracy and efficiency [13, 4, 7, 19, 2, 18, 17, 3, 15]. Based on the needs of these analyses, the analysis-specific IR interface for alias analysis includes an iterator over all the statements in a procedure, an iterator over all of the memory references in a statement, the set of memory reference expressions that describe a given memory reference handle, the location abstraction associated with a particular symbol, an iterator over the pointer assignments that occur in the statement, and iterators over procedures, procedure calls, and procedure call parameters. Table 1 shows the information that the IR must provide to the alias analysis and the results of an alias analysis manager in terms of which locations each memory reference may access. Note that the `&x` and `a` memory references do not actually map to locations. These memory references are actually address computations.

An alias analysis that does not handle dereferences can map all of the memory reference expressions that are named to the corresponding named locations and map all memory reference expressions that are dereferences to the unknown location or alternatively all accessible locations. A more advanced alias analysis can use the pointer assignment pairs to

Statement	Provided by IR			Alias Results
	Memory References	Memory Reference Expressions	Ptr Assign Pairs	May Locations
C				
`int x, *p;`				

`p = &x;`	`p`
	`&x`
		NamedRef(`p`, DEF)
		NamedRef(`x`, USE, address of)
			`< p, & x >`	NamedLoc(`p`)
				address computed

`... = *p;`	`p`
	`*p`
		NamedRef(`p`, USE)
		Deref(1, NamedRef(`p`, USE), USE)
				NamedLoc(`p`)
				NamedLoc(`x`)

Fortran 90				
`integer a`				
`integer, pointer :: q`				

`q => a`	`q`
	`a`
		NamedRef(`q`, DEF)
		NamedRef(`a`,USE, address of)
			`< q, a >`	NamedLoc(`q`)
				address computed

`... = q`	`q`	Deref(1, NamedRef(`q`, USE), USE)		NamedLoc(`a`)
Sparc Assembly				

`sub %fp, 4, %l1`	`%fp`
	`%l1`
		NamedRef(`%fp`, USE)
		NamedRef(`%l1`, DEF)		

`st %l1, %l2`	`%l1`
	`%l2`
		NamedRef(`%l1`, USE)
		NamedRef(`%l2`, DEF)		

`ld [%l2], %l3`	`%l2`
	`[%l2]`
	`%l3`
		NamedRef(`%l2`, USE)
		Deref(1,NamedRef(`%l2`, USE), USE)
		NamedRef(`%l3`, DEF)		

Table 1: Examples in C, Fortran 90, and Sparc Assembly with the memory references for each statement.

more accurately analyze what locations a dereference might access. In the C example in Table 1, a more accurate alias analysis can determine that `*p` references location `x`. In the Fortran 90 example, a more accurate alias analysis can determine that `q` references location `a`.

Some alias analysis algorithms also use information about whether a symbol is local to the current procedure. The important scoping distinction is whether a symbol is accessible by name *only* within the current procedure or whether it is also accessible outside the current procedure. Often if a variable is local to a procedure, then it is accessible only within the current procedure. One exception is languages that allow embedded procedures. In implementing IR interfaces for such a language, local variables in the source IR can be classified as local to OpenAnalysis algorithms only if that local variable does not appear in any embedded procedures.

The final proof that this relatively thin interface is sufficient for the cited analysis algorithms would be an implementation of each of these algorithms within OpenAnalysis. Currently the only alias analysis algorithm detects aliasing due to reference parameters but has all memory references involving a dereference map to the `Unknown` location as a conservative estimate. Future work includes validating this interface with more alias analysis implementations and developing a type abstraction for OpenAnalysis to enable type-based alias analysis algorithms. The proof that analysis-specific IR interfaces work for any imperative programming language is stronger in that IR interface implementations exist for a number of compiler infrastructures.

2.4 Handling Difficult Language Constructs

A common approach to handling complicated memory references is to canonicalize the program into a simplified form or translate the IR into a simplified form [7]. Such translations make it difficult or impossible to map analysis results back to the original program. Our approach involves each memory reference in the original IR being uniquely associated with an opaque handle, thus solving the problem of mapping analysis results back to the original IR. We argue that the memory reference expression (MRE) abstraction is capable of conservatively representing challenging language features such as pointer arithmetic, array operations, and function pointers.

ID	PL	Statement	Memory References
1	C	`x = *(p+q);`	`x, p, q, *(p+q)`
2	C	`d->c->b = 5;`	`d, d->c, d->c->b`
3	C	`p = &a;`	`p, &a`
4	F90	`A = B + C`	`A, B, C (arrays)`
5	Sparc	`ld [%l0],%o0`	`%l0, [%l0], %o0`
6	C	`*(r<*q?r:*q) =0;`	`r, q, *q, *(r<*q?r:*q)`

Table 2: Example memory references.

The key idea is to associate each sub-memory reference within a statement with an opaque memory reference handle to avoid breaking the references into their component parts. Examples 1, 2, 5, and 6 in Table 2 show how complex memory references can be viewed as multiple sub-memory references. The semantics of the original IR then provides direction for generating MREs. For example, statement 1 in Table 2 involves pointer arithmetic. Assume that `p` is a pointer; then the MRE for `*(p+q)` should differ based on whether `q` is an integer or another pointer. If `q` is another pointer, then the expression is invalid according to the C99 standard; and if the compiler doesn't flag that as an error upon parsing, then the IR interface implementation can represent the reference with an `UnknownRef`. If `q` is an integer, then the `*(p+q)` reference can be represented as a dereference to `p` with partial accuracy if the assumption is made that adding q will not cause overstepping array bounds, or as an `UnknownRef` to be on the safe side. Array operations in Fortran90 such as those in statement 4 can be represented as a fully accurate `NamedRef` to each array (references to array elements can be conservatively represented as partially accurate named references to an array). Statement 6 involves a ternary operator and can be represented with two memory reference expressions: a single dereference to the named reference `r` and a double dereference to the named reference `q`.

Since many alias analysis algorithms do not use the concept of type, the current memory reference expression abstractions can represent a wide range of memory references including, references to entire arrays, references to fields within a structure, and assignments and calls to function pointers. Function pointer assignments such as `foo = bar` can be represented as `foo = &bar`. Then calls to a function pointer such as `foo()` can be represented as a dereference to the named reference `foo`.

Another advantage of generating a view of the original IR for program analysis, instead of a canonicalized or simplified IR, is that the view can be a conservative estimate of the behavior. The abstractions passed through the various analysis-specific IR interfaces need not be detailed enough for code generation. In the context of alias analysis, this approach enables expressing memory references at more or less accuracy to analysis implementations to encourage quick initial IR interface development while enabling future refinements. For example, the main memory reference for statement 2 in Table 2 can be accurately represented as a field reference to `b` that is part of a dereference to a field reference to `c` that is part of a dereference to `d`, or the memory reference can be represented with partial accuracy as two dereferences to `d`.

3. IR INTERFACE FOR REACHING CONSTANTS

Reaching constants is a data-flow analysis that determines which memory references have a constant value at compile time. Since data-flow analysis involves facts about the data in the program state, data-flow information is associated with the location abstraction in OpenAnalysis. For reaching constants, the data-flow information consists of tuples that map a location to a constant. A memory reference is constant if all the locations that it may reference are assigned the same constant value. The analysis-specific IR interface for reaching constants is interesting because it illustrates the abstract use of constants and operators as well as being an example of a data-flow analysis algorithm.

OpenAnalysis can perform reaching constants using opaque handles to operands, memory references, statements, and constants and abstractions for expressions, constants, and memory reference expressions. Specifically, the constant value abstraction requires the implementation of an equality operator only. The IR interface for reaching constants also requires that the IR can generate a new opaque constant given an opaque handle to an operator and one or two opaque constants as operands. The key point is that OpenAnalysis does not need to know the datatype for the constants or the semantics of the operator. More generally, various analyses will need only a limited interface associated with a constant value abstraction.

For expressions there is an expression tree abstraction. Other analyses that require an expression tree abstraction include array data dependence analysis and symbolic analysis. Expression trees contain a generic expression node type that is then subclassed to wrap handles that can occur in an expression. Example expression nodes include a call node that contains an expression handle that represents a function call, an operator node that contains an operator handle and pointers to children within the expression tree, and a memory reference node that contains a memory reference handle.

As a data-flow analysis, reaching constants is implemented by using a data-flow analysis framework that is part of OpenAnalysis. Therefore, it is necessary to implement only initialization, transfer, and meet functions. For data-flow analysis algorithms in general, design decisions with respect to the analysis-specific IR interface will affect to what extent the analysis implementation is IR independent. For example, if the transfer function is specified as part of the IR interface, then a significant piece of the analysis must be done in an IR-specific fashion. Our reaching constants IR interface includes methods that (1) indicate whether a statement involves the assignment of an expression to a memory references, (2) provide an iterator over all such assignments in the statement, (3) provide the expression abstraction for any expression, (4) provide an opaque constant value given the handle to a constant symbol or value, (5) evaluate an opaque operator handle given two opaque constant values, and (6) return iterators needed for other statements such as all the memory references that indicate uses or defines within a statement.

4. TOOLKIT STATUS

The OpenAnalysis project was initially begun to jumpstart two emerging compiler activities at Rice University. Both activities needed control-flow graph (CFG) analysis capabilities but were based on intermediate representations with very different levels of abstraction. The first activity involved a tool for recovering information about loop nesting structure from application binaries. The second involved the Open64/SL infrastructure for source-to-source program analysis and transformation [1], which uses an abstract syntax tree-level intermediate form. Three existing compiler projects at Rice had relevant technologies to contribute: a MATLAB compiler effort had code for building CFGs from structured control flow, the scalar compiler group had developed a sophisticated strategy for constructing a control-flow graph for scheduled assembly code that permits branches in branch instruction delay slots [5], and the DSystem compiler infrastructure [16] had a sophisticated implementation of an interval analysis algorithm for identifying flowgraph cycles and nesting of both reducible and irreducible loops [9].

However, none of the code could be reused directly in the new tools. The principal obstacle was that each depended intimately on the details of the particular intermediate representation for which it was developed. The key step for leveraging and integrating these components was to design an interface through which the analysis algorithms could obtain necessary information from the intermediate representation, yet be insulated from the details of any particular IR by a layer of abstraction.

Using this approach, Rice researchers created a broadly applicable control-flow graph construction and analysis package. Today, this package is being employed to construct control flow graphs for two abstract syntax tree-level representations: Rice's Open64/SL infrastructure for Fortran 90 and Lawrence Livermore National Laboratory's ROSE compiler infrastructure for C++. The OpenAnalysis flowgraph package also is the cornerstone of bloop—an open-source, multiplatform binary analyzer that is part of Rice University's HPCToolkit performance analysis tools [11]. Using OpenAnalysis's representation-independent flowgraph analysis infrastructure, bloop can recover loop nesting structure for application binaries for a wide variety of processor architectures including MIPS, Alpha, x86, Itanium, and SPARC. The OpenAnalysis flowgraph package is also used in Argonne's automatic differentiation tools ADIC and OpenAD.

Upon being adopted by researchers at Argonne, OpenAnalysis underwent a facelift. The control flow graph functionality and some simplified call graph functionality were converted over to the more modular software architecture overviewed in Figure 1. We have also implemented prototype analysis managers for intraprocedural reaching definitions, ud and du-chains, reaching constants, interprocedural alias analysis, side-effect analysis, and a domain-specfic analysis needed for automatic differentiation called activity analysis. We also have begun prototyping data-flow analysis frameworks for analysis over a control-flow graph, ICFG, and call graph.

5. LIMITATIONS AND ISSUES

The current instantiation of OpenAnalysis requires the whole program to do correct analysis, does not handle function pointers while constructing the call graph, uses inefficient algorithms to manipulate the location abstraction, and interacts only with compiler infrastructures written in C++. We plan to address these issues in future work. Even with these limitations OpenAnalysis is being actively used within a number of research projects.

While developing the IR interface implementation for Open64/SL, we observed that IR interface implementations can result in the creation of a subsidiary IR to support the necessary queries. For example, in Open64's Whirl IR many of the opaque handles are valid only within the context of a procedure; therefore, the IR interface implementation must iterate over all the constructs within the IR and maintain a mapping of handles to procedure context. We also observed that adding new analysis-specific IR interface functionality sometimes leads to a redesign of the ad hoc subsidiary IR. If each analysis requires the construction of a subsidiary IR, but there is some overlap, the IR interface implementations themselves could become quite tangled. More work is needed to develop effective approaches to implementing IR interfaces in an incremental fashion.

Language independence can also be a limitation. Many alias analysis implementations take advantage of language-specific assumptions to improve the accuracy of alias analysis. For example, in Fortran 90 the possible targets of a pointer must be specified. In Fortran 77 and Fortran 90, if reference parameters are passed the same location, then assignment is not allowed. The alias IR interface within OpenAnalysis does not provide a way to communicate such information with the alias analysis implementation. One open question is how these assumptions can be codified separately from alias analysis implementations but still be used to improve accuracy. In existing alias analysis implementations these assumptions are strongly coupled with the IR or spread throughout the analysis implementation.

6. RELATED WORK

In [14], Moonen presents a software architecture that provides language-independent data-flow analysis. The goal of language independence is the same as in our current work. Moonen's approach is different in that he converts program representations to a data-flow representation language. Although he handles the problem of mapping the results back to the original representation in situations where the ASF+SDF tool [12] is used, this will still be an issue in the general case. Also, in order to convert a program representation to his data-flow representation language, it is necessary to perform alias analysis. We enable language-independent alias analysis as well.

GENOA [6] and StarTool [10] are two program analysis tools that have developed an adaptation level between intermediate representations and analysis. Both tools require that the specific intermediate representation be an instantiation of an abstract syntax tree, whereas the OpenAnalysis toolkit is already able to handle a wider range of program representations including executables. The GENOA tool works with an AST-based interface that attempts to make information available for all possible analyses resulting in a monolithic interface between the intermediate representation and analysis implementations. In the StarTool, Hayes et al. took a more client-driven approach; the adaptation level was smaller and simpler because it focused on the needs of the StarTool analysis. We are taking a similar approach in the OpenAnalysis toolkit by developing a separate IR interface for each analysis.

7. CONCLUSIONS

An IR-independent analysis toolkit enables higher productivity of compiler researchers in several ways: (1) enabling the use of common analyses in a plug-and-play fashion with the compiler infrastructure most appropriate to researchers' goals, (2) providing analyses at all representation levels within a particular compiler infrastructure, (3) providing analysis frameworks, (4) encouraging toolkit extension and enabling compiler researchers to share their analyses with others, and (5) providing analysis implementations that realize different accuracy versus efficiency tradeoffs. OpenAnalysis provides all of these benefits without requiring commitment to a particular compiler infrastructure. Thus, compiler researchers can focus on such issues as ease of use, robustness, and language coverage while selecting the appropriate compiler infrastructure for their projects.

Analysis-specific, IR-independent interfaces are the key to providing language-independent program analysis. It is possible to conservatively represent complex language constructs starting from basic imperative programming constructs that lie at the intersection of all such languages. A broad range of analysis implementations of difficult analyses such as alias analysis is possible through analysis-specific interfaces that use a subset of imperative language construct abstractions. Our experience with the OpenAnalysis toolkit indicates that sharing analysis implementations between compiler infrastructures is possible and a promising approach for eliminating the need to duplicate analysis implementation work.

8. ACKNOWLEDGMENTS

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

9. REFERENCES

[1] Open64/SL web page, 2003. http://www.hipersoft.rice.edu/open64/.

[2] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the pressence of pointers. In D. Gelertner, A. Nicolau, and D. Padua, editors, *Lecture Notes in Computer Science, 892*. Springer-Verlag, 1995.

[3] B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 57–69. ACM Press, 2000.

[4] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[5] K. Cooper, T. Harvey, and T. Waterman. Building a control-flow graph from scheduled assembly code. Technical report, Rice University TR02-399, 2002.

[6] P. T. Devanbu. GENOA: A customizable language- and front-end independent code analyzer. In *International Conference on Software Engineering*, pages 307–317, 1992.

[7] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley Professional Computing Series. Addison Wesley, 1995.

[9] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(19):557–567, 1997.

[10] J. Hayes, W. G. Griswold, and S. Moskovics. Component design of retargetable program analysis tools that reuse intermediate representations. In *the 2000 International Conference on Software Engineering, (ICSE 2000)*, June 2000.

[11] HiPerSoft: A Center for High Performance Software Research @ Rice. HPCToolkit web page, 2000-2004. http://www.hipersoft.rice.edu/hpctoolkit/.

[12] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[13] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, New York, NY, USA, 1991. ACM Press.

[14] L. Moonen. A generic architecture for data flow analysis to support reverse engineering. In *the 2nd International Workshop on the theory and Practice of Algebraic Specifications (ASF+SDF'97)*, 1997.

[15] E. M. Nystrom, H. S. Kim, and W. M. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th Static Analysis Symposium*, August 2004.

[16] Rice University Parallel Compiler and Tools Group. The DSystem compiler infrastructure. http://www.cs.rice.edu/ dsystem.

[17] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):105–186, 2001.

[18] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41. ACM Press, 1996.

[19] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 1–12. ACM Press, 1995.