Interface Compilation: Steps toward Compiling Program Interfaces as Languages

Dawson R. Engler
Computer Science Laboratory
Stanford University
Stanford, CA 94305, U.S.A
engler@stanford.edu

Abstract

Interfaces — the collection of procedures and data structures that define a library, a subsystem, a module — are syntactically poor programming languages. They have state (defined both by the interface's data structures and internally), operations on this state (defined by the interface's procedures), and semantics associated with these operations. Given a way to incorporate interface semantics into compilation, interfaces can be compiled in the same manner as traditional languages such as ANSI C or FORTRAN.

This paper makes two contributions. First, it proposes and explores the metaphor of *interface compilation*, and provides the beginnings of a programming methodology for exploiting it. Second, it presents MAGIK, a system built to support interface compilation. Using MAGIK, software developers can build optimizers and checkers for their interface languages, and have these extensions incorporated into compilation, with a corresponding gain in efficiency and safety. This organization contrasts with traditional compilation, which relegates programmers to the role of passive consumers, rather than active exploiters of a compiler's transformational abilities.

1 Introduction

This paper presents MAGIK, a system that can be used to incorporate both application semantics and control into compilation. MAGIK is motivated by two sets of insights. First, programmer-defined data structures and functions define a semantically rich (albeit syntactically poor) language, built on top of the language the programmer uses to define them. Unfortunately, these *meta languages* have not had optimizers: optimization occurs at the lower-level of the programming language, but not at the higher-level defined by their interface. Our hypotheses is that since high-level operations are heavyweight (e.g., they deal with file I/O, window manipulations, transactions, and thread creation), optimizations which understand their semantics offer the hope of significant speed improvements, potentially exceeding the impact of all other compiler optimizations. Second, programming has historically been passive: with the exception of restricted local code transformations provided by macro systems, programmers are limited to writing code, while the power to transform the code has been reserved for compilers. Our belief is that giving programmers safe, ready access to the compilation process will significantly improve the scope of programmer capabilities.

The MAGIK system has been built to test these beliefs. MAGIK provides a simple, modular mechanism for programmers to dynamically incorporate extensions into the MAGIK compiler. User extensions, written in ANSI C, are dynamically linked into MAGIK during compilation. Extensions are given access to MAGIK's intermediate representation (IR) through a set of interfaces that allow them to easily create, delete, and augment IR at compile time. Both this IR and MAGIK are built on top of the lcc compiler [6], which is used to compile the source language (ANSI C). The control MAGIK gives to programmers enables a broad class of optimization and code transformations. This paper presents seven such extensions and sketches many more.

This paper concentrates on two abilities provided by MAGIK. First, it provides a way for implementors to include domain-specific semantics into compilation. Using this ability, implementors can build both interface optimizers (for speed) and interface checkers (for safety). Interface optimizers exploit application-specific knowledge in order to obtain performance improvements. Such optimizers are applicable to a wide range of interfaces: "bignums", message passing and I/O libraries, math libraries, matrix transformations for graphics, even simple queue operations. From a compiler perspective this ability is useful in any situation where providing a compiler "builtin" would allow more aggressive optimization. From an implementor perspective they are useful in situations where an interface implementor could look at a call or sequence of calls to the implementation and craft specialized call(s) that exploited local uses. For example, a file system implementor can write a optimizer that exploits knowledge of file system operations to perform optimizations such as hiding disk latency by both inserting disk block prefetching commands, and transforming synchronous file I/O operations into asynchronous ones. Interface checkers use application-specific knowledge to enforce stricter semantic checks. For example, by requiring that system call error codes be checked (or inserting such checks) or by ensuring that assertion conditions do not have side effects.

The second main ability MAGIK provides is an easy, modular way to do general code transformations with full access to source information. Using this ability, programmers can instrument code, augment it (e.g., by introducing software fault isolation code [19] or garbage collection reference counters) or enforce invariants about it (e.g., that no pointer casts are allowed). Unlike object code modifiers such as ATOM [16], MAGIK clients are tightly integrated with the source compiler. Performing transformations during the translation from high-level source language to machine code has two important characteristics. First, it provides access to the full semantics of the high-level language, information that source transformers can exploit (or require) during code transformation. Second, the IR produced from these user transformations is a first class citizen, optimized no differently than the IR produced by the compiler itself. As a result, the compiler optimizes these transformations as it would any other code.

This paper is organized as follows. Section 2 explores the metaphor of interface compilation (IC) further and presents a primitive methodology for it. Section 3 provides an overview of the MAGIK system and Section 4 present examples of how it can be used to support IC. Section 5 discusses issues in the current system and directions for future work. Finally, Section 6 discusses related work and Section 7 concludes.

2 Metaphor: Interfaces as languages

This section articulates a primitive methodology for interface compilation. Our goal is to provide interface builders with specific, operational guidelines for deciding *how* to compile their interfaces. The reader should keep in mind that while the latter half of this paper presents a system for implementing such decisions, the patterns below are applicable to any implementation scheme used to support interface compilation, whether it be a domain-specific language, an extensible type system, or an extensible compiler. All of these require that interface writers know how to exploit compilation. Thus, two basic questions need elaboration: when should it occur to an implementer to apply interface compilation? And what rules, common patterns and heuristics can be followed to proceed effectively?

In order to apply compilation to interfaces, we need an intuitive model of what an interface without interface compilation means. An "interface" is the direct use of one of a variety of abstraction mechanisms provided by a programming language. In a language such as C, an interface would be a set of data structures and simple functions that operate on these data structures, typically collected in a "header file." In more advanced languages, an interface could be implemented using abstract data type (object method) invocations, more elaborate protocols for remote procedure calls, or even synchronization scenarios involving distributed message passing. Code that uses an interface (as opposed to implementing it) will be referred to as "client code" in the following text and the code produced by compiling the abstraction directly as "traditional compilation."

There are two useful intuitions to keep in mind during the following discussion. First, an implementor shown a clients's use of an interface will invariably see ways of optimizing and checking that use. Interface compilation allows the implementor to provide a set of rules for codifying these improvements and a mechanism (e.g., an open compiler) for implementing them. Second, traditional compilation, unsurprisingly, provides a useful supply of ideas for how to compile interfaces. Many optimizations and checks in that realm have analogues in this one. For example, optimizing by special-casing operations ("strength reduction") and reusing intermediate results ("common sub-expression elimination") have equivalent operations, as we shall discuss below. The difference between interface compilation and normal compilation is that interface compilation occurs at a much higher level. For example, optimizing file input and output operations rather than loads and stores. The challenge in applying interface compilation is in pulling high-level semantics into compilation.

The rest of this section concentrates on understanding interface compilation. We do so by examining it from two perspectives. The first examines capabilities that simply cannot be obtained using traditional interfaces without interface compiler support. The second examines specific ways that these capabilities are lost during the mapping of interface abstractions into low-level code. The rest of the paper then presents a system for implementing our methodology, along with many concrete examples.

2.1 A mechanical perspective on interface weakness

Interfaces suffer from two problems: (1) their implementations using traditional abstractions are blind to the context and manner in which they are used and (2) they are "passive" in the sense that they cannot modify the client code surrounding their uses, e.g. for optimization purposes. An interface's inability to analyze and transform client code prevents static detection of errors, precludes specialization to the calling context, and generally prevents many forms of adaptation to use. Interface compilation gives interface implementors a mechanism both to obtain a clearer view of the context surrounding the uses of an interface and to modify this context and uses. Below we explore more concrete effects of passivity and blindness, along with the fixes that interface compilation provides.

Interface passivity: Interfaces cannot modify client code. This inability renders interfaces unable to insert code to do runtime checking to detect the profitability or safety of optimizations, unable to specialize calls to a specific use (e.g., generating specialized code for functions called with constant arguments), and often unable to insert sensible error handling because requisite contextual information has been lost. Most of the following patterns require the ability to rewrite client code.

Blindness to callsite: Interface functions are blind to the callsite context in which they were called. Thus, an interface cannot statically check that client uses are legal, nor can it exploit callsite information to specialize uses.

One natural framework to exploit callsite information is to view interface functions as compiler "builtins." Compilers have long used such a mechanism to optimize frequently used routines. Interface compilation shifts the ability to include builtin from compiler writers to system implementors, making it more widely applicable. Some examples of using this ability are compile-time evaluation of constant-argument calls to referentially transparent functions, and specialization of memory copy routines to pointer alignment and size.

More generally, interface implementors can exploit callsite information to construct partial evaluators for important routines. Standard examples include specialization of output routines and remote procedure call marshaling code.

A final scheme is to implement callsite-specific memoization, which allows an interface to associate and specialize the memoization cache with each callsite rather than treating all calls as generic. Using this ability, memory allocators can craft allocator functions taylored to a particular object type and size, I/O libraries can implement program-counter determined prefetching, etc.

Blindness to global context: Interfaces cannot see the global context in which they are called. They have no way to detect whether they are being invoked in a threaded environment, or if a procedure's operands have changed since the last use, etc. Access to data flow and "whole program" information enables many correctness checks and optimizations. For example, an interface optimizer for remote procedure call (RPC) could exploit global information to aggregate a sequence of RPC calls into a single message (improving throughput). Additionally, it could improve latency by using information about the definition of RPC operands and use of RPC results to replace synchronous RPC with asynchronous calls, pushing calls higher in the program text, and checking for RPC completion before use. Similar optimizations can be done for file I/O.

A second example in this spirit is a library-specific extension that optimizes across calls to a graphics library. Given a sequence of calls that manipulate a matrix, this optimizer can reuse intermediate results, eliminate intermediate copies, and perform cache optimizations across calls. A "big num" package can optimize across calls to its operations in a similar manner.

A final, real-world, interface-checking example comes from Bishop and Dilger [2], where they describe a system that uses global information to detect race conditions in privileged Unix applications. Many "root" programs have "time-to-check-to-time-of-use" (TOCTTOU) bugs where they check to see if a particular user has access to a file and, if he does, begin writing on it. Unfortunately, since this check-action pair is not atomic, a malicious user can remove the file after it was checked and create a "symbolic link" to a file he was not allowed to access, which the privileged program will then begin writing on. Global information allows these calls to be bound together and, thus, security errors to be detected [2].

Since interface operations tend to be quite costly, optimizing across multiple interface calls has the potential for enormous benefits. These optimizations benefit from the ability to transform client code in order to exploit flow graph mutation techniques for improved information. For example, they can split flow graphs to improve the data flow properties of a path by removing the pollution of joins ("path splitting") or optimizing a sequence of calls on one path ("software pipelining").

Blindness to client data structures: Interfaces cannot analyze or modify client data structures. Data structure traversal allows the definition of structure independent routines for sorting, searching, marshaling, and printing. Control of data structure layout can improve performance by allowing extensions to group member fields that are used close together into the same cache line, improving cache behavior. It can also enhance usability by enabling extensions to abstract away such details as endianness by automatically rearranging structures to be endian neutral. Data structure redefinition can improve speed on machines that do not provide sub-word memory instructions by allowing an extension to replace sub-word sized structure elements with word-sized ones. Data structure augmentation allows functionality enhancements such as automatic addition of bookkeeping fields needed by reference counting garbage collectors.

Blindness to compiler context: Interfaces cannot access compiler information. Compilers calculate much useful information, which should be made available to interfaces. Two simple extensions we have built in this spirit are an extension that, given a pointer to a type, returns the alignment of that type (this is useful for memory allocators) and an extension that takes a single argument and indicates whether it is a constant expression (useful in making inline decisions).

2.2 A mapping perspective on interface weaknesses

From another perspective, mapping an interface "meta language" onto a lower-level implementation language causes problems whenever there is a semantic mismatch between the two levels. Typically, interfaces are semantically richer than the base language used to implement them. Thus, the invariants or restrictions imposed by this richer language will be inexpressible in the lower language, preventing its compiler from checking or exploiting them. In the former case, the traditional compiler is prevented from checking the constraints; in the latter the compiler is prevented from exploiting context to perform interface-specific optimizations. For example, operations using a Unix file handle after it has been "closed" are illegal. However, since a traditional compiler will not know this invariant, it cannot check for such illegal uses. Similarly, a compiler that is oblivious to the semantics of a set of trigonometric functions cannot replace sequences of calls to them with numerically equivalent, but cheaper, sequences. Below, we discuss common mapping mismatches and how interface compilation can fix them.

Same concept, different result: High-level concepts (such as allocation and deallocation) may not map well onto the similar events within the base language, causing constraints lost in the process to be unchecked except with manual inspection (with predictable results).

A common mismatch is that destruction maps poorly to language abilities. For example, on Unix, the system call close kills a file handle (a capability used to read and write a file). If there is no way to express this in the base language, then subsequent uses of the handle will only be caught at runtime. Consider the following C code, which opens a file, closes it, and then attempts to write to this closed file:

```
static char buf[1024];
int fd;

fd = open(filename);
/* ... */
close(fd);   /* free file handle */
/* ... */
read(fd, buf, 4096); /* use to read */
```

The compiler has no way to know that the object named by the integer value of fd has been killed and, thus, that the subsequent use of fd by read is illegal. Similarly, it cannot statically detect the buffer overrun that will happen when read is invoked since it does not know that read's third argument indicates how many bytes to copy into the buffer. None of these checks are hard to understand, nor, given a means of extending compilation, difficult to incorporate.

Pervasive, poorly supported dynamic typing: The lack of compiler support discussed above requires that interfaces resort to implementing ad hoc, dynamic type-checking schemes themselves (e.g., explicitly checking that parameters obey interface invariants). Manual, runtime checking increases both complexity and errors. Moreover, the use of dynamic checks when errors are apparent at compile time violates the general rule that the earlier errors are caught the better.

When checking must be dynamic (irrespective of compiler support), such as system calls verifying the legality of their arguments, the current inability for implementors to extend type checkers means that they must manually place boilerplate at the beginning of each function to check argument legality. Further, since the compiler does not understand these checks, it cannot warn when they are erroneously elided. As an example, consider the Unix read call, which has the following C function prototype:

```
/* copy 'nbytes' of data from file named by
   'fd' into buffer 'buf'. */
int read(int fd, void *buf, size_t nbytes);
```

Among the many security conditions that read's implementation must check are (1) that fd is legal, (2) that the current user can read from the file, (3) that the virtual address range [buf, buf+nbytes) can be written to. Any of these checks is easily forgotten. A simple solution uses the compiler to verify that system call functions check the validity of all user-supplied pointers. Such automatic enforcement can of course be made sophisticated. For example, system call routines taking both a pointer and a parameter named (say) nbytes, can be checked to ensure that they verify the virtual memory range [ptr, ptr+nbytes).

Unchecked logic errors: Many interface invariants are logic conditions, which often go unchecked because they cannot be expressed in the base language's type system. For example, the reversal of sink and source parameters to copy routine, or the element size and object size to a generic sorting routine can type check perfectly but still have disastrous results. The ability to augment typing with conventions for indicating intent (such as name checking of arguments) can eliminate several classes of such errors.

Lost optimization opportunities: Mapping high-level concepts onto a low-level programming language frequently loses semantic information and, thus, loses optimization opportunities. For instance, consider a user-defined trigonometric library. Since the compiler does not understand the semantics of the library's functions, it cannot do even simple optimizations such as computing the result of function calls with constant arguments at compile time, never mind more semantically-aware reductions such as exploiting trigonometric identities and relationships to replace sequence of calls to the library to compute a formula with a numerically equivalent but cheaper computation. Interface optimizers allow these semantics to be incorporated into compilation.

The patterns from the previous subsection (in particular exploiting callsite information and global information) can be viewed as ways to exploit richer semantics.

Inability to enforce a more restrictive context: The inability to disallow certain programming constructs prevents interfaces from enforcing necessary restrictions on their clients. Many subsystems used by a programmer may require that application code be written to assume a more restrictive execution environment. For example, in a multithreading context, formally legal, unrestricted access to global data must be serialized, while in the context of an operating system kernel running with interrupts disabled, code paths must not exceed a small number of instructions. Interface checkers allow such restrictions to be enforced.

A useful restriction pattern is *language subsetting*, which preserves the syntax of the base language, but makes some constructs (such as pointer casts) illegal and (possibly) inserts runtime checks to catch errors allowed by the base language (e.g., null pointer dereferences, arithmetic over- and under-flow, etc.). A concrete example is to use language subsetting to support thread stack relocation. Because relocation moves a stack to another address range or, even to another machine, pointers to a relocated stack will be invalid. Most systems simply decree that programmers should not address local variables (i.e., create pointers to stack memory). Given interface checkers, a compiler can automatically enforce these rules.

2.3 Relation to traditional compilation

A methodology to develop interface compilation further is to enumerate traditional compiler optimizations, attempting to find their analogues in this new domain. Either the technique will apply, in which case we gain an operational guideline for how to compile interfaces, or the technique will not apply, in which case the difference between interfaces and actual languages will become clearer. Analogues explored in this paper include: common subexpression elimination, flow graph mutations, memoization, strength-reduction, prefetching, load/store coalescing. Additional analogues include using runtime invariant checking to ensure that optimistic assumptions hold and instruction scheduling (here, overlapping long-latency interface calls with others). A concrete application of the above approach is to notice that file reads and writes are simply expensive loads and stores and thus, that traditional optimizations can be used on them: pushing reads early, prefetching, coalescing operations, making writes asynchronous, etc.

While interface optimization has strong similarities to traditional compilation, it has two important differences. First, interface operations are extremely expensive compared to the operations that traditional compilers are typically applied to. (An ALU add takes nanoseconds on a modern processor, while a call to the read system call may take tens of milliseconds.) Second, related operations are stretched farther across both the lexical landscape of program text and temporal time line of program execution. These two differences imply that a more active approach to compilation may be appropriate. The first difference means that the overhead of runtime monitoring and adaptation can be more easily recouped than in the context of traditional compilation. The second difference implies that there is more need for such techniques.

Traditionally, compilers have mostly been passive: given a program, they try to derive properties by static analysis and, typically, when that fails, they give up. Interface compilers can be more active. For example, given an invariant that must be checked either for safety or to enable an optimization, but which cannot be determined statically, inject code into the application to do runtime verification that the invariant holds. Active compilation changes the driving question of a compiler writer from "how do I optimize given limited information?" to "what other information do I need and how do I get it?"

Interface compilation can be viewed as an example of *orthogonal programming*, where the actions of a mass of software are controlled from functions that reside on the outside of it, at right angles. [1] Done appropriately, orthogonal programming aids modularity by isolating the "meta" functionality of checking and control from the brute implementation. Consider an interface checker that works at the level of interface's specification: it requires no knowledge of implementation internals, allowing it to be used across different interface implementations. Neither do these internals need to know about the checker, allowing it to be transparently added and extended. For example, a Unix system call usage checker can be readily constructed by third parties and used across different Unix implementations. And, rather than breaking when interfaces themselves differ, the checker can warn of subtle differences (or even insert code to correct for them).

[1] Another example is "aspect programming" where aspects of synchronization and related concepts are controlled by code outside, in a sense, the actual program doing the work [11].

3 System Overview

Magik provides a framework to extend compilation. User extensions are implemented as dynamically-linked functions. User extensions come in two classes: *code extensions* and *data structure extensions*. Code extensions are invoked at every function definition and are able to enumerate, add, delete, and modify Magik's IR as it makes the transition from source language to machine code. Data structure extensions are invoked at every data structure definition and are able to add, delete and modify structure elements. Since compiler internals are in flux, implementation portability is provided by isolating extensions from internal IR details via a set of standardized interfaces; multiple interfaces are provided, specialized to the main domains Magik is used in.

A given compilation may use many different extensions. To make the system usable, it is crucial that extension composition is modular. The two main requirements of modularity are that extensions be able to inspect the code produced by others and that extensions can be obliviously composed. Magik meets these requirements by providing three different extension types, *transformers*, *optimizers*, and *inspectors*, that correspond to the three main functional uses of extensions. Transformers are used to perform code transformations that do not depend on integration with global optimization (e.g., partially evaluating a C printf call). Optimizers are used to perform iterative optimization and are repeatedly invoked during global optimization until no IR modifications occur. (Optimizers differ from both transformers and inspectors in that they may be invoked multiple times.) Inspectors are similar in functionality to transformers except their placement in the extension pipeline ensures that they see all IR that will be compiled to code.

The main implementation limitation of the Magik system is that optimizers have only weak data flow information. This restriction comes from the fact that the compiler Magik is built on, lcc, has no global optimization framework. We are investigating methods of removing this limitation (e.g., by using the SUIF compiler system [1]).

An operational overview of the extension process is as follows:

1. Programmers implement extensions using the Magik libraries; these extensions are compiled to object code. The location of this code is either specified to Magik using command-line flags or by embedding the location in source files. For instance, header files can specify an extension to optimize the interfaces they define.
2. Magik compiles high-level source (ANSI C) to its internal IR in the traditional manner. As Magik encounters extension location directives (either as compiler flags or embedded in source) it uses the dld dynamic linker [8] to dynamically link the named extensions into the compiler proper.
3. When Magik encounters a function in the source language, it compiles the function to IR, and then

invokes all code extensions, beginning with transformer extensions. At this point the extensions can augment, modify and delete parts of the IR. As part of the global optimization loop, MAGIK calls each optimization extension. These extensions have access to any data flow information computed by the compiler (e.g., use and def sets, values of procedure parameters, etc.) To ensure that code produced by any extension is visible to all others (a requirement for modular composition of different extensions) MAGIK loops through the extensions until no more modifications occur to the IR. A nice result of this organization is that the code produced by an extension is optimized as aggressively as the code produced from application source. After all optimization extensions have run, and no modifications occur, MAGIK runs inspector extensions in their specified order.

4. When MAGIK encounters a structure definition, it generates a symbol table entry, and then invokes all structure extensions, which can add, modify and delete structure entries. Typically these extensions are also paired with code extensions that augment data structure field uses and definitions.

5. MAGIK emits code.

MAGIK's lowest-level IR interface, based closely on that the the underlying compiler (lcc) is terse, simple and portable. Structurally, the IR is a tree language. Leaves are variables, labels, or constants; internal nodes represent operations performed on them (e.g., addition, indirection, jumps, function calls). When operands are created they are associated with a type selected from MAGIK's base types (shown in Table 1). Thereafter, types are implicit: operations infer their own types based on the type of their operands. Any conversions required by ANSI C are performed by MAGIK (e.g., as required by ANSI C a character variable will be converted to an integer before addition with an integer).

User-created IR (type: I_IR) is of a different type than native IR (type: X_IR). This distinction is helpful because user-constructed IR typically requires preprocessing before it can be sensibly incorporated into lcc's internal representation. By exploiting static typechecking, MAGIK can prevent users from blithely intermixing the different representations.

The interfaces are presented in the following tables: routines to allocate, lookup and manipulate symbols in Table 4, routines to construct IR in Table 5, and routines to navigate the IR in Table 2. Higher-level interfaces are discussed in Section 4.

We expect MAGIK to evolve with further experience. To aid iterative design, the current implementation has emphasized simplicity at all levels. MAGIK is built on top of the lcc retargetable ANSI C compiler [6], and uses its IR language as its fundamental interface [7] (higher-level interfaces are crafted on top of this). The regularity and small size of lcc's IR has been a major asset. Importantly, since mapping other IR's to the MAGIK IR and back should be straightforward, it can be realistically used as a basis for defining a standardized, compiler-independent, extension interface similar in availability to ANSI C's standardized libraries.

Type	C name
V	void
C	signed char
UC	unsigned char
S	signed short
US	unsigned short
I	int
U	unsigned
L	long
UL	unsigned long
F	float
D	double
P	void *

Table 1: MAGIK types (superset of lcc's types).

While the implementation exploits lcc's infrastructure, there is no fundamental tie to lcc. As experience with the system and its uses grows, reimplementations will occur in more aggressive compilers (or, alternatively, MAGIK will be used to enhance the optimization framework of lcc).

One of the common uses of MAGIK is to incorporate new functions as "built-ins" into the compiler. Since there can be tens or (at aggressive sites) hundreds of builtins, it is critical that the extension process itself is efficient. To achieve the required efficiency, MAGIK dynamically links extensions rather than isolating them in sub-processes that communicate via shared memory. In most cases this process has no significant impact on compilation speed. For implementations that wish to remove all overhead (at some cost in reduced flexibility) MAGIK provides an interface that can be used to statically link extensions into the compiler proper (similar to the process of adding device drivers to most operating systems).

The following two sections discuss the interfaces MAGIK provides for incorporating application semantics (e.g., interface optimization and checking) and for general code transformation.

4 Extensible compilation: patterns of use

MAGIK provides mechanisms that allow applications to construct extensions that can exploit interface semantics for improved semantic checking, optimization, and general transformations. The two main constructs of interest are functions and data structures. In the case of functions, clients are mainly interested in two pieces of data-flow information: the location of calls in relation to each other, and the definitions and uses of each call's operands and results. Clients also require semantic information about each call site's operands: their type, whether they are constants, and if so, what their values are. In the case of data structures clients are primarily interested in definitions and uses of structures and their fields.

To make IR manipulations easier, MAGIK exploits the limited information needed in this area to provide a default interface that is simpler than the general MAGIK IR. It includes basic block structures, function calls, details about function arguments and results (e.g., whether they are constants, their type, possible values, etc.) and information about structure accesses. A library of routines are provided that allow clients to add, modify, delete and augment function calls and code easily. Additional routines are provided to search for particular functions and lists of functions in the IR (easing IR navigation), traverse argument lists, and routines that compute the set of variables defined and used by a given call site. Table 6 presents MAGIK's interface for finding, manipulating, and constructing call sites. Table 7 presents MAGIK's interface for finding IR tree patterns, and both structure and structure field uses.

Clients that need access to the full power of MAGIK's IR can, of course, use it; the layering provided by default is intended as syntactic sugar rather than a barrier.

The following subsections present MAGIK's semantic interface and numerous example clients. These uses illustrate how interface implementors can use MAGIK to build interface checkers and optimizers. The first client uses MAGIK to add a compiler "builtin" output function that is implicitly aware of its operand types (eliminating the need for printf-style format strings).

The next three clients are interface checkers. The first adds more rigorous error handling of Unix system calls by inserting checks around call sites that ignore a system call's return value. The next client uses global information to warn of possible race conditions in "signal handlers" by searching for calls to non-reentrant functions. The final client warns of side-effects in debugging "assertions."

The final three clients are interface optimizers. The first optimizes RPC call sites by using partial evaluation to generate specialized argument marshaling code. The second specializes memory copy to its argument type and size. The final rearranges client data structures for improved space consumption.

4.1 Adding type-aware functions

ANSI C suffers from the lack of a graceful mechanism to handle poly-typed functions. Programmers are typically reduced to specifying argument types using a manually-constructed type string. This methodology is clumsy and error prone. One of the more painful effects of this lack is that C is one of the few languages in use that does not have type-aware I/O routines.

Figure 1 presents a MAGIK extension that adds a type-aware output routine, output. It works by rewriting all calls to the poly-typed function it defines (output) to call printf using a type string (typestring) it constructs from the type of output's arguments. An operational view is as follows:

1. The extension iterates over all calls to output using the MAGIK functions FirstCall and NextCall.

2. For each callsite, it builds up a printf-style type string by iterating over output's argument list (using the MAGIK functions FirstArg and NextArg) and appending the type of each argument to typestring.

3. After typestring has been constructed, the extension uses RewriteCall to modify the call site to call printf instead of output and inserts typestring as the first argument.

A sample usage:

```
void example(int i, int j) {
   output("i = ", i, "j = ", j);
}
```

While some languages (such as C++) support this capability for simple scalars, our extension can be easily modified to print the fields in aggregate types, freeing programmers from having to tediously write data structure-specific output routines (this functionality was elided for brevity). Extensible code synthesis is powerful. Example uses include the automatic generation of routines to translate data structures between "in-core" and on-disk representations and the construction of linked-list, hash-tables, and associative arrays specialized to particular data structure types. A similar technique is used in Subsection 4.5 to construct an efficient argument marshaling routine for a remote procedure call system.

4.2 Safe signal handlers

This extension uses global information to enforce a more restrictive programming context than that of the base language. Unix signal handlers represent primitive thread of controls. Unfortunately, they are used by many programmers who are unfamiliar with the dangers of threaded programs. A common mistake made is to call non-reentrant library functions from these handlers. If the application was suspended in the middle of a call to the same function (or to a function that manipulates state it depends on) the application program will, non-deterministically, exhibit incorrect behavior.

To help prevent this class of problems we have defined an extension that prevents calls to non-reentrant functions in a signal handler (the extension's code is elided for brevity). The extension works as follows:

- To trigger checking all signal handlers adhere to the naming convention of prefixing their name with "sig_" (e.g., sig_protection_fault).

- The extension scans for all functions beginning with this prefix and, for each callsite, checks that the call is either to one of a list of known reentrant functions or to a function that is prefixed with sig_. Any call that does not satisfy these requirements is flagged.

- To ensure that only checked handlers are installed as signal handlers it also looks for handler installation calls and checks that they only install functions beginning with the sig_ prefix.

```
/* Add a type-aware output function. */
int RewriteOutput(X_IR c) {
#  define MAXARGS 64
   X_IR a;

   /* Foreach callsite, rewrite the output call. */
   for(c = FirstCall(c, "output");
       c != NULL;
       c = NextCall(c, "output")) {

      /* String to hold derived typestring. */
      char typestring[MAXARGS*2+1] = {0};

      /* Foreach argument, create a typestring. */
      for(a = FirstArg(c); a != NULL;
                           a = NextArg(a)) {
         switch(OpType(a)) {
         case I: strcat(typestring, "%d "); break;
         case P:
           /* Print strings differently
                  than pointers. */
           if(RawPtrType(NodeType(a)) == C)
              strcat(typestring, "%s ");
           else
              strcat(typestring, "0x%p ");
           break;
         /* ... */
         default: panic("Bogus type");
         }
      }
      /* Add newline */
      strcat(typestring, "\n");
      /* Change call to output to call to printf. */
      RewriteCall(c, "printf");
      /* Add typestring as first argument. */
      PushArg(c, Cnststr(typestring));
   }
   return MAGIK_OK;
}
```

Figure 1: Routine to add a type-aware output routine to C

```
/* Add checks to unchecked system calls. */
int RewriteUnix(X_IR c) {
   /* list of all calls we insert checks for */
   char *unixcalls[] = {"read","write","seek",
                                 /* ... */ 0};
   I_IR res, err, stmt;
   char *n;

   /* foreach callsite, rewrite the output call */
   for(res = NULL, c = FirstCallV(c, unixcalls);
        c != NULL; c = NextCallV(c, unixcalls)) {

      n = CallName(c);

      /* If result used, assume it is checked. */
      if(Uses(c))
         continue;
      else
         warn("unchecked system call <%s>\n",n);

      /* Create temp to hold returned value */
      if(!res)
         res = Temp(inttype, MAGIK_REG);

      /* Create IR to assign the return value
         to res. */
      stmt = AddStmt(c,
                  Asgn(res, ImportExprRef(c)));

      /* Create a call to error routine; expects
         syscall's name and return code. */
      err = Call("error", voidtype, Cnststr(n),
                res, NULL);
      /* Insert check for syscall failure. */
      AddStmt(stmt,
             IfStmt(Lt(res, Cnsti(0)), err));
    }
   return MAGIK_OK;
}
```

Figure 2: Extension that places error checks around unchecked system calls.

4.3 Safe system calls

C and Unix are notorious for using integer error codes to indicate exceptional conditions. C and Unix programmers are notorious for not checking these codes. This problem is a significant one, especially with the prevalence of network computing (where file I/O operations have to be retried with some frequency). Figure 2 presents an extension that inserts error condition checks around unchecked Unix system calls and prints out errors that occur.

The extension works as follows:

1. It iterates over all calls to the functions listed in the array unixcalls using the MAGIK functions FirstCallV and NextCallV.
2. For each call site it checks if the result of the call is used (using the MAGIK routine Uses). Unfortunately, a use does not guarantee that the call's result is checked — for simplicity, we elide more aggressive checking.
3. For call sites that do not use the result of the system call, the extension creates IR to check the system call's return value and, if it is an error, call an error procedure (error) to print it out. It then inserts this IR into the original IR using AddStmt.

4.4 Safe assertions

Debugging assertions check invariants at runtime. The presence of side-effects in assertion expressions causes difficult-to-find bugs that show up when the assertions (and thus their side-effects) are disabled for "production use." Figure 3 presents an extension that conservatively guards against side-effects in assertions by scanning for function calls and assignments.

4.5 RPC specialization

Remote procedure call (RPC) is a widely used abstraction in distributed programming. A significant overhead of a general-purpose RPC call is the cost of copying the call's arguments into a message buffer ("argument marshaling"). Figure 4 presents a MAGIK extension that uses partial evaluation to remove the main contributor to this overhead, the interpretation of argument types, by crafting marshaling code specialized to a particular callsite.

The extensions infrastructure is similar to that used to implement output: it scans for calls to rpc and examines its argument which, syntactically, is a call to a remote function. It decomposes this call into its constituent pieces and then builds marshaling code to copy each argument in the RPC call into a memory vector. It then rewrites the call to rpc to take a pointer to a local copy of the remote procedure along with a pointer to the constructed message buffer and its size. A sample usage is as follows:

```
int k,j,i;
double d;
/* ... */
```

```
/* call remote procedure remote_foo */
rpc(remote_foo(j, i, k, d));
```

Of course, this usage can be made prettier by communicating the names of remote procedures to the extension, thereby eliminating the need for the rpc annotation. For simplicity we do not perform this syntactic cleanup (we also ignore result passing).

```
/* Look for function calls or assignments */
static int HasSideEffect(X_IR c) {
   if(!c)
      return 0;
   else if(Op(c) == ASGN || Op(c) == CALL)
      return 1;
   else
      return HasSideEffect(Left(c))
              || HasSideEffect(Right(c));
}

/* Check that assertions do not contain
      side-effecting optations. */
int AssertCk(X_IR c) {
   for(c = FirstCall(c, "assert");
       c != NULL;
       c = NextCall(c, "assert")) {

/* The assertion expression is the call's
               first argument. */
       if(HasSideEffect(FirstArg(c)))
          warning("assert has a side-effect\n");
   }
   return MAGIK_OK;
}
```

Figure 3: Routine to guarantee that assertions are free of side-effects.

4.6 A memory copy builtin

To show how MAGIK can be used to define builtin procedures for improved performance we have also written an extension that recognizes the ANSI C memcpy ("memory copy") function. The extension exploits information MAGIK provides to specialize to the local characteristics of each callsite. For example, in the general case, memcpy must treat its operands as unaligned. However, using the semantic information MAGIK provides, the extension can determine when a call site's pointer operands are aligned and specialize accordingly. Additionally, it unrolls and inlines the memory copying loop when the number of bytes to copy is a constant, Static specialization removes runtime selection overhead, and shrinks the function's memcpy footprint (due to the fact that the gaps introduced by non-taken cases is eliminated). These optimizations are profitable in the context of operating system device driver and networking code, which can extensively access fixed-sized quantities of partially unaligned memory.

```
/* Find RPC calls and build marshalling code. */
int MarshalGen(X_IR r) {
   /* foreach callsite, rewrite output call */
   for(r = FirstCall(r, "rpc");
       r != NULL; r = NextCall(r, "rpc")) {
      I_IR index, marshalv;
      int offset, sz;
      X_IR a, c;

      /* Allocate marshaling array on stack. */
      marshalv = Array(doubletype, Nargs(r));
      offset = 0;
      /* Remote call is rpc's first argument. */
      c = FirstArg(r);

      /* Store arguments in marshalling vec. */
      for(a = FirstArg(c); a != NULL;
                         a = NextArg(a)) {
         /* ensure correct alignment. */
         offset = roundup(offset, NodeAlign(a));
         sz = NodeSize(a);

         /* Form expression "*(type *)(marshal +
            offset)" where type is typeof(a). */
         index = Index(Cast(Copy(marshalv),
                            Ptr(NodeType(a))),
                      Cnsti(offset/sz));

         /* marshalv[offset] = a */
         PushStmt(c,
                 Asgn(index, ImportExprCopy(a)));
         /* Add size of argument. */
         offset += NodeSize(a);
      }
      /* Replace rpc call with message send; send
         takes a pointer to a local copy of the
         remote function and the marshal vector
         and size as arguments. */
      c = ReplaceExpr(c,
           Call("send", inttype,
                CallName(c), Copy(marshalv),
                       Cnsti(offset), NULL));
    }
   return MAGIK_OK;
}
```

Figure 4: Extension that creates specialized marshaling code based on remote procedure call argument types.

```
/* Used by qsort to compare element sizes. */
static int pack_cmp(void *p, void *q) {
  return FieldSize(*(Field *)p) -
         FieldSize(*(Field *)q);
}

/* Look for structures with "pack_" prefix and
     minimize their storage size by sorting their
     elements by size.  */
void Packer(Symbol p) {
  unsigned n;
  Field *fl;

  if(strncmp(StructName(p), "pack_", 5) != 0)
     return;
  /* Get fields */
  fl = ImportFields(p, &n);
  /* Sort them. */
  qsort(fl, n, sizeof fl[0], pack_cmp);
  /* Write them out. */
  ExportFields(p, fl, n);
}
```

Figure 5: Routine to minimize structure size by sorting elements by alignment requirements.

4.7 Structure packing

Dense structure layout can be used to improve locality. Figure 5 presents a data structure extension that rearranges structure fields to reduce structure size. Using the same capabilities extensions can perform many useful structure transformations: fields can be automatically arranged to be endian-neutral and on machines that lack sub-word operations, shorts and chars can be promoted to ints.

4.8 General code transformations

MAGIK also enables code transoformations that are independent of any interface. Example transformations are software fault isolation, the translation of pointers from one representation to another, or the insertion of checks to ensure a pointer use is not nil.

In MAGIK, code transformations are typically implemented by searching for specific IR trees and (possibly) replacing or augmenting them. To make this style of usage easy, MAGIK provides an interface specialized to this domain. IR navigation can be implemented using MAGIK-provided pattern matching routines that iterate over IR, returning all locations that extension-specified IR trees occur at. Rewriting support includes procedures that insert, delete and augment IR subtrees. These routines isolate the programmer from implementation-specific details of IR modification (e.g., the need to update all pointers to a node that has been used as a CSE).

Using MAGIK's interfaces, applications can implement a vast set of general code transformations such as the insertion of reference counting, software address translation or providing protection via software fault isolation [19].

Similarly ready access to a semantically-rich intermediate language can be used to answer many questions about source-level code. For example, it can be used to verify hypothesis about software engineering by correlating bug reports to how many times an abstraction layer is broken (perhaps by tracking structure accesses) or by correlating ease of modification to the number of intermodule dependencies a source file has. Checks can be inserted to check for the aliasing of pointers to determine what optimizations would be profitable. It can also be used to support graphical performance monitoring in the spirit of Jeffery and Griswold [9] by automatically inserting display calls around interface uses.

5 Discussion

MAGIK attempts to literally make "library design language design." It does this by attacking the three crucial differences between writing a function-level interface and defining an input language and compiler. The first difference is obvious: languages have syntactic sugar, libraries do not. (I.e., other than what can be synthesized from macros, libraries have little syntactic support for their idioms.) By enabling interface designers to include context- and semantic-sensitive code transformers, sugar can be judiciously added to function interfaces (e.g., as done in the output and rpc examples in Section 4). The second difference is more subtle: languages allow semantic checks that can be difficult for a library to replicate in terms of its implementation language. By giving extensions access to both the symbol table and function-level IR this barrier can be eliminated. Finally, languages can be optimized. Encoding their semantics in a compiler allows a ready implementation of both local (e.g., peephole optimization) and global (e.g., CSE) optimizations. Current compilers are blind to interface semantics, precluding analogous optimizations. MAGIK provides mechanisms that can be used to build interface optimizers that optimize interface primitives as aggressively as source language constructs.

5.1 Interface issues

An interesting research question is determining the design rules for building interfaces that are amenable to language-like optimization techniques. Two principles seem relatively safe. First, high-level optimization is aided by the use of declarative, high-level interfaces that can then be "strength-reduced" to the characteristics of local usage. Second, optimization across interface calls is eased if the result of one interface call is immediately used by another: function call nesting is an ideal way of eliminating data-flow ambiguities. Thoroughly codifying practical precepts will be challenging.

Careful (but, unfortunately, iterative) design of the MAGIK system has allowed us build it so that it is integrated with the infrastructure lcc uses to construct its internal IR. An important result of this integration is that we have been able to reuse lcc's front-end routines for constructing abstract syntax trees. Using this code has two significant benefits. First, the interface simplifies code construction by only requiring users to supply types when an operand (such as a variable or constant) is defined. Operations then derive their type from their operands. For example, the addition operator, Add, can determine whether it is an integer or floating-point addition by examining its operands rather than having the type encoded as AddL ("add long"), AddF ("add float"), etc. Eliminating the need to explicitly encode types has dramatically simplified MAGIK's code construction interface. Second, lcc's routines are designed to perform implicit conversions as required by the rules for ANSI C. As a result, they type-check their arguments (providing users with safety) and perform coercions as necessary (providing users with convenience).

There are a few challenges to using the current IR system. The first is dealing with IR tree layouts across compiler versions. Layout of IR trees is a fairly volatile implementation feature. Currently, MAGIK decrees an IR interface and layout. The cost of this solution is that future implementations may require extra mapping code to compile their IR to the standardized MAGIK IR and back. An alternative solution is to specify code using a higher-level representation. The main technical challenge of using IR to specify patterns is that functionally identical language expressions may be compiled to structurally different trees. Fortunately, the lcc IR is spare enough that this problem is not difficult: the number of possibilities tends overwhelmingly to one and, in rare cases, two. In fact, the use of a low-level IR can have a significant benefit over both source-level and machine-code matching in this respect since both, in practice, can contain significant numbers of synonyms (e.g., consider the possible ways to get values to and from memory on the x86, or the different but equivalent methods to reference an array element in C). However, while the IR representation has been sufficient for all examples we've wanted to implement, there are times when a less strenuous mechanism of code specification is preferable. We are currently investigating alternatives.

5.2 System limitations

There are a number of limitations with the current system; most were deliberately chosen in order to allow it to be built quickly so that real programmers could use it in the near future, thereby allowing the wheel of iterative design to begin turning with the least amount of delay. Four main limitations are discussed.

First, constructing large pieces of code is tedious. This would naturally be remedied with language support. A promising avenue is to use the 'C ("tick-C") language [5] (designed to construct code dynamically) as a sugary method of dynamically constructing MAGIK IR. 'C solves most of the semantic issues dealing with variable binding, and code construction, leaving us with the fairly straightforward task of modifying it to dynamically emit MAGIK IR rather than executable code.

Second, the current code specification MAGIK interface — the low-level IR of lcc — while simple, is perhaps not the most natural for mainstream programmers. There are tradeoffs in this representation: a low-level IR can be more precise, however, it can also be more complex than necessary. We are investigating the representation of code templates used for matching via language support: here to a modification of the 'C language seems promising.

Third, the system is manual, even for tasks that could be done automatically (e.g., in the spirit of Vandevoorde and Guttag [18, 17]). As we determine which of these tasks are important and common, automation will be added.

Finally, lcc, while simple and easy to modify, is a poor optimizer. We are examining ways to improve its code quality.

5.3 A simple language extension

Exploitation of application semantics is helped if semantics can be clearly and unambiguously indicated. For example, translating shared memory accesses is eased if every such access can be explicitly labeled as "shared." The clean, clear conveyance of semantic information to extensions is a general problem. Fortunately, it has a simple solution: the addition of a new syntax operation to ANSI C (annotation) that is used to create new, scoped type qualifiers. These qualifiers would be syntactically parsed and internally stored in the symbol table but otherwise ignored by the compiler proper — their semantics provided solely by extensions. An example usage:

```
/* add ''shared'' as a new type qualifier */
annotation shared;
/* Allocate an integer with new type qualifier. */
shared int x;
```

6 Related Work

Examples of including application-level information into compilation are compiler-directed prefetching and management of I/O [14] and ParaSoft's Insure++ [12], which can check for Unix system call errors (similar to the MAGIK checker shown in Figure 2). Using a MAGIK-based approach, systems such as these could be built without compiler modifications.

Concurrently with this work, Tom Lord proposed the construction of application-specific semantic checkers and built a scheme-based system, ctool, for implementing them [13]. There are many specific differences between his system and MAGIK but much of motivation is shared. Outside the topic of semantic checking there are two key differences. First, Lord restricts his attention to read-only semantic checking, while this paper additionally proposes application-specific optimization and general program transformation ("incorporating programmer control into compilation"). Second, this paper proposes the metaphor of treating programming interfaces as languages, and sketches a methodology for interface compilation which can be used by interface builders to compile their interfaces in the same manner as traditional languages such as ANSI C or FORTRAN (see Section 2 for further discussion).

Prior to this work, Shigeru Chiba implemented Open C++, a system that extends C++ with a "meta object protocol" [4, 3]. Meta objects are objects that exist only at compile time and control the compilation of the program. Included in this control is the ability to add new functions, types and data to the translated code [4]. Mechanically, there are many similarities between the abilities given by Open C++ to meta objects and by MAGIK to interface checkers and optimizers. The main difference between the two approaches is one of perspective: Open C++ appears to be driven more by a focus on low-level transformations, while MAGIK is driven by the desire to raise compilation to the level of interfaces. While the results produced from these two desires overlap, the main question of this paper ("how to compile an interface?") is not addressed in this prior work.

Below, we compare MAGIK to macro systems, semantic-based optimizers, extensible compilers, and object code modifiers.

Macro systems are the most venerable instance of user-level code transformers. An advantage of such systems (Lisp is a good example) over MAGIK is their tight integration with the source language — extensions are typically written in the same language and style as the rest of the application. The main advantage MAGIK provides is power. Macro systems such as Weise and Crew's recent work [21] are restricted to fairly localized code transformations, while MAGIK extensions can perform global transformations across many interface calls, using symbol table and flow graph information provided by the compiler.

Mark Vandevoorde and John Guttag [17, 18] describe a system that provides programmers with a safe way to impart some classes of semantic information to the optimizer. User-level specifications for a restricted functional language are consumed by a theorem prover that optimizes based on the specific situation in which function calls are used. While their system is more automatic than MAGIK, it is less powerful. For instance, MAGIK gives programmers the ability to perform optimizations that appear difficult to express as specifications. The cost of this power is that MAGIK more difficult to use. Further practical experience is needed to determine if MAGIK's added power is worth this cost.

MAGIK follows in the footsteps of the Atom object code modification system [16] (foreshadowed by the object code modifiers of Wall [20] and Srivastava and Wall [15]), which provides users with the ability to modify object code in a clean, simple manner. Atom was one of the first tools to give programmers ready access to the transformational abilities encased in compilers. MAGIK complements this work, and trades the practical generality of dealing with object code for improved information and code efficiency gained by working within a high-level source compiler. Since MAGIK has access to all the information available to the source compiler (e.g., symbol table, flow graph information, high-level semantics) it can derive facts lost at the object code level. For instance, it can easily insert reference counts around all accesses to a particular pointer type; an object code modifier, working solely at the level of loads and stores, cannot. Furthermore, since MAGIK extensions are integrated with the optimization done by the compiler, they can be implemented more efficiently: IR added by an extension is optimized no differently than IR produced from source. In contrast, object code have to both work without much source-level information and cannot bootstrap existing compiler optimizers [20]. An important practical difference between MAGIK and object code modifiers is that MAGIK is significantly easier to implement. The system described in this paper took the author less than a month to implement and it runs on all targets that the base compiler supports (x86, Mips, Sparc). In contrast, duplicating the functionality of ATOM for even a single architecture would require significantly more work (especially on an architecture such as the x86).

There are many compilers designed to support easy addition of optimizations (e.g., SUIF [1]). These system could have been used to implement MAGIK; lcc was chosen because of the author's familiarity with it. To the best of our knowledge, none of these compilers have been used explicitly for extending the optimizer with user-level semantics or transformations.

MAGIK can be viewed as an "Open System" in the spirit of Kiczales' work [10].

Of course, programmers have long performed interface optimizations by hand. The advantages of automated optimization are well known.

7 Conclusion

This paper has addressed two problems programmers have historically faced. First, the languages they define via interfaces have not been treated as first-class languages. As a result, these languages have had no language-specific semantic checkers, transformers, or optimizers. Second, their programs are passively consumed with little support for active transformation (such as rewriting of structure fields and the addition of profiling code).

The MAGIK system is a first step towards solving these problems. MAGIK provides a modular interface implementors can use to extend compilation. The main interaction is through a set of interfaces that give extensions access to the IR produced from source. MAGIK thus provides a method that system implementors use both to incorporate domain-specific semantics into compilation (thereby enjoying the obvious advantages of automated optimization and checking) and to perform general transformations on the IR produced from source (thereby having both access to high-level semantic information and the resulting transformation code optimized as aggressively as code produced from source).

This paper can be viewed as an initial step towards a new style of programming that treats interfaces as languages. It has proposed the beginnings of a methodology for interface compilation and provided a system that allows this methodology to be applied.

This paper has presented many example clients of the MAGIK system. Many of these extensions provide capabilities that programmers did not previously have. Future research will involve both extending these capabilities and exploring their consequences.

It is the hope of the author that elevating the languages interfaces define to first-class citizenship (where they are optimized and checked easily and well by compilers) will change programming practice in a non-trivial way. For decades, there has been a clamor for higher and higher-level languages. But, in fact, these languages are already prevalent, as a simple perusal of header files and module definitions will show. Their apparent absence is merely due to lack of compiler support.

8 Acknowlegements

The author would like to thank Saman Amarasinghe, Frans Kasshoek, David Mazieres, Thomas Pinckney, Massimiliano Poletto, and Martin Rinard for interesting discussions of interface compilation and proof reading. Martin Rinard articulated the concept of "language subsetting" discussed in Section 2. Eddie Kohler and Massimiliano Poletto helped enormously with fonts and formatting. Dave Wile and Constantine Sapuntzakis graciously provided last minute proof reading. Chris Ramming and the anonymous referees were particularly helpful in improving the paper's presentation.

Dawson Engler is an assistant professor at Stanford University, where he holds joint appointments in the electrical engineering and computer science departments. He received his PhD degree in computer science from the Massachusetts Institute of Technology, where he co-initiated the exokernel operating system project. He received his BS in math and computer science from the University of Arizona, where he was a bouncer at a blues bar. His research focuses on systems and compilation. Past projects have included dynamic code generation and extensible operating systems. Current projects include interface compilation and novel approaches to verifying code correctness.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[2] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 1996.

[3] S. Chiba. Open c++ programmer's guide. Technical Report TR93-93-3, Dept. of information science, University of Tokyo, 1993.

[4] Shigeru Chiba. A metaobject protocol for c++. In *OOPSLA 1995 Conference Proceedings Object-oriented programming systems, languages, and applications*, pages 285–299, October 1995.

[5] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg, FL, 1995.

[6] C. W. Fraser and D. R. Hanson. *A retargetable C compiler: design and implementation.* Benjamin/Cummings Publishing Co., Redwood City, CA, 1995.

[7] C.W. Fraser and D.R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, September 1991.

[8] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software—Practice and Experience*, 24(4):375–390, April 1991.

[9] Clinton L. Jeffery and Ralph E. Griswold. A framework for execution monitoring in Icon. *Software—Practice and Experience*, 24(11):1025–1049, November 1994.

[10] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[12] Adam Kolawa and Arthur Hicken. Insure++: A tool to support total quality software. `http://www.parasoft.com/insure/papers/tech.htm`.

[13] Tom Lord. Application specific static code checking for c programs: Ctool. In *t̃waddle: A Digital Zine (version 1.0)*, 1997.

[14] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[15] A. Srivastava and D.W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, March 1992.

[16] Amitabh Srivastava and Alan Eustace. Atom - a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[17] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance.* PhD thesis, M.I.T., 1994.

[18] Mark T. Vandevoorde and John V. Guttag. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *Proceedings of the 1994 ACM/SIGSOFT Foundations of Software Engineering Conference*, 1994.

[19] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 1993.

[20] D.W. Wall. Systems for late code modification. *CODE 91 Workshop on Code Generation*, 1991.

[21] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of PLDI '93*, pages 156–165, Albuquerque, NM, June 1993.

Operation	Description
X_IR LeftChild(X_IR n)	Returns n's left child or nil on error.
X_IR RightChild(X_IR n)	Returns n's right child or nil on error.
int OpType(X_IR n)	Returns opcode of n.
int Type(X_IR n)	Returns type of n.
int Align(X_IR n)	Returns alignment of n.
int Size(X_IR n)	Returns size of n.

Table 2: Base IR Interface.

Class	Examples	Prototype
Arithmetic binary operations.	ADD SUB MUL DIV XOR AND OR	I_IR op(I_IR a, I_IR b)
Arithmetic unary operations.	NEG COM	I_IR op(I_IR a)
Conversions ("convert to *type*").	CVTI CVTD CVTUS	I_IR *op*(I_IR a)
Memory operations.	ADDR INDIR	I_IR *op*(I_IR a)

Table 3: Partial IR-construction Interface. Functions determine the type of opcode to use based on operand type. Conversion conventions are those of ANSI C.

Operation	Description
I_IR Local(Type ty)	Creates a local variable of type t and returns its symbol.
I_IR LocalArray(Type ty, int n)	Creates a local array of type t and size n and returns its symbol.
I_IR Global(Type ty)	Creates a global variable of type t and returns its symbol.
I_IR GlobalArray(Type t, int n)	Creates a global array of type t and size n and returns its symbol.
I_IR Cast(I_IR var, Type t)	Creates a copy of symbol var changing its type to t.
I_IR Lookup(char *name)	Lookup symbol for variable name.

Table 4: Symbol construction and manipulation routines (routines to construct new aggregate types are elided).

Operation	Description
X_IR Copy(X_IR n)	Create a copy of node n. This function is typically used when adding a new subtree between a node and its child.
I_IR ImportExprRef(X_IR expr)	Import a reference to expr. This reference can then be used as an argument to functions that require a I_IR type.
I_IR ImportExprCopy(X_IR expr)	Import a copy of expr. This copy can then be used as an argument to functions that require a I_IR type.
X_IR AddStmt(X_IR a, I_IR stmt)	Add stmt after node a. Returns stmt.
X_IR PushStmt(X_IR a, I_IR stmt)	Add stmt before node a. Returns stmt.
X_IR DeleteStmt(X_IR stmt)	Remove stmt, returns its successor.
X_IR DeleteExpr(X_IR expr, I_IR replacement)	Delete node expr; replaces the tree with replacement. If replacement is nil, MAGIK will coalesce the tree expr was part of until it is well-formed.
X_IR AddExpr(X_IR a, I_IR b)	Insert b on top of a.
I_IR If(I_IR bool, I_IR stmt)	If bool is true, execute stmt.
I_IR IfElse(I_IR bool, I_IR stmt1, I_IR stmt2)	If bool is true, execute stmt1 otherwise execute stmt2.
I_IR While(I_IR bool, I_IR stmt)	While bool is true, execute stmt.

Table 5: Partial High-level IR construction Interface

Operation	Description
X_IR FirstCall(char *name)	Returns pointer to first call of name or nil if none is found.
X_IR FirstCallV(char **namelist)	Returns pointer to first call of any function in namelist or nil if none is found.
X_IR NextCall(X_IR c, char *name)	Returns pointer to next call of name or nil if none is found.
X_IR NextCallV(X_IR c, char **namelist)	Returns pointer to next call of any function in namelist or nil if none is found.
X_IR RewriteCall(X_IR call, char *newname)	Replace name of call to be newname.
X_IR FirstArg(X_IR call)	Return first argument (if any) of call.
X_IR NextArg(X_IR arg)	Get next argument (if any) after arg.
X_IR Arg(X_IR call, int n)	Returns the nth argument of call; returns nil on error.
void PushArg(X_IR call, I_IR arg)	Adds arg as the first argument to call.
void AppendArg(X_IR call, I_IR arg)	Adds arg as the last argument to call.
int NArgs(X_IR call)	Return number of arguments to call.
X_IR ReplaceArg(X_IR call, int argno, I_IR arg)	Replace argument argno in call with arg.

Table 6: Partial Function Navigation and Modification Interface

Operation	Description
X_IR Search(X_IR n, I_IR pattern)	Search for the tree pattern starting at location n. If n is nil, the search starts at the beginning of the function. Unspecified subtrees in pattern can be created using the function I_IR Any(Type ty).
X_IR FindStruct(X_IR n, char *StructName)	Search for use of StructName starting at n.
X_IR FindField(X_IR n, char *StructName, char *FieldName)	Search for use of field FieldName of type StructName starting at n.
Fields *ImportFields(Symbol p, unsigned *n)	Returns a pointer to an array of pointers to data structure p's fields. Elements in this vector can be reordered, deleted, and added.
Fields *ExportFields(Symbol p, Fields *fieldlist, unsigned *n)	Export fields (defined by fieldlist) as the layout for data structure p.
Field AddField(Symbol p, Field f1, Field f2)	Add field f2 after field f1 in structure p.
Field PushField(Symbol p, Field f1, Field f2)	Add field f2 before field f1 in structure p.
Field OverrideField(Symbol p, Field f, Type ty)	Change field structure p's field f type to ty.
Field FirstField(Symbol p)	Returns the first field in data structure p.
Field NextField(Symbol p, Field f)	Returns the next field in data structure p.

Table 7: Partial Structure Navigation and Modification Interface