Automatic Generation of Library Bindings Using Static Analysis *

Tristan Ravitch Steve Jackson Eric Aderhold Ben Liblit

Computer Sciences Department, University of Wisconsin–Madison

{travitch, sjackso, aderhold, liblit}@cs.wisc.edu

Abstract

High-level languages are growing in popularity. However, decades of C software development have produced large libraries of fast, time-tested, meritorious code that are impractical to recreate from scratch. Cross-language bindings can expose low-level C code to high-level languages. Unfortunately, writing bindings by hand is tedious and error-prone, while mainstream binding generators require extensive manual annotation or fail to offer the language features that users of modern languages have come to expect.

We present an improved binding-generation strategy based on static analysis of unannotated library source code. We characterize three high-level idioms that are not uniquely expressible in C's low-level type system: array parameters, resource managers, and multiple return values. We describe a suite of interprocedural analyses that recover this high-level information, and we show how the results can be used in a binding generator for the Python programming language. In experiments with four large C libraries, we find that our approach avoids the mistakes characteristic of hand-written bindings while offering a level of Python integration unmatched by prior automated approaches. Among the thousands of functions in the public interfaces of these libraries, roughly 40% exhibit the behaviors detected by our static analyses.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Languages; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.3.2 [*Programming Languages*]: Language Classifications—C, Python; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management, Procedures, functions, and subroutines; D.3.4 [*Programming Languages*]: Processors—Code generation, Memory management; E.1 [*Data Structures*]: Arrays; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Reliability, Experiments, Human Factors

* Supported in part by AFOSR grant FA9550-07-1-0210; LLNL contract B580360; and NSF grants CCF-0621487, CCF-0701957, and CNS-0720565. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

Keywords FFI, foreign function interfaces, bindings, libraries, dataflow analysis, modular static program analysis, multi-language code reuse

1. Introduction

In recent years, high-level languages have made inroads into areas formerly reserved for low-level languages, such as scientific computing and non-kernel systems programming. A high-level language can speed development by offering helpful features like automatic memory management.

However, even with increasingly-capable high-level languages, there remains a need to access code written in low-level languages. Some important reasons include:

Direct access: By their very nature, high-level languages attempt to hide low-level details from programmers. Sometimes, these details become important, either for performance in critical loops or for interaction with hardware, and high-level languages do not always provide means to operate "on the metal."

Code reuse: Well-tested code has great value: its bugs are at least known, if not fixed. Rewriting code risks introducing new and unknown bugs. In many situations, there may not be time to port a large code-base to a new language, even if such a port is desirable.

Additionally, sharing a single library among several languages promotes interoperability and frees designers from the burden of reimplementing the same specification in every language.

Many high-level languages facilitate the reuse of low-level libraries through *foreign function interfaces*, which permit high-level languages to make calls to native code. When discussing such calls, we refer to the high-level language as the *host language* and the native code as a *guest library*; the low-level language that produced the guest library is thus the *guest language*. We refer to a set of host language functions which make an entire guest library available through some form of function interface as a *library binding*. We only consider C as the guest language in this paper, but the principles can be applied to other languages.

When a small number of guest functions are needed in a host language, it may be convenient for a developer to use foreign function interfaces "by hand." However, the task of building a complete binding for a large library is nontrivial. In this paper, we argue that this task ought to be automated, and show that existing techniques for such automation may be greatly improved by static analysis of a guest library's source code.

The remainder of this paper is organized as follows. Section 2 motivates our work. Section 3 identifies desirable features for library bindings that cannot be represented in C function declarations alone. Section 4 describes a suite of interprocedural static program analyses that recover this missing information from C library source code. In

Section 5 we present a concrete example client for this information: an automated binding generator for the Python scripting language. We have used this generator to create bindings to four large C libraries, and we evaluate the results in Section 6. Section 7 discusses related research, and Section 8 concludes.

2. Motivation

Creating and maintaining library bindings can be a costly and error-prone process, especially when the interfaces of guest libraries are subject to change. Numerous teams of developers exist to maintain major binding projects. Examples include gtk2-perl, PyQt, gtkmm, and java-gnome. They represent an under-served programmer community whose work will become more critical as mixed-language development becomes more prevalent.

For large libraries, it is impractical to hand-code host language wrappers for all guest library functions. Tools such as SWIG [3] and `ctypeslib` [18] partially automate this process by reading library headers and generating library bindings automatically. Several library binding teams, including PyGTK and PyQt, have created and maintain their own code generators based upon C or C++ headers to partially automate the binding generation process.

However, the interfaces generated by header-scanning systems do not take advantage of higher-level features offered by host languages unless manual annotations to the guest library are provided. This need for annotations arises because there is no one-to-one mapping of programmer intent to the set of C language constructs that declare a library's public interface. Key aspects of high-level design are lost, and no amount of header scanning will bring them back. We assert that these higher-level properties can be recovered, without annotations, by moving beyond headers to deeper static analysis of library implementation code. This, in turn, supports automated generation of library bindings which are less prone to error and more natural to use.

Additionally, the analysis techniques shown in this paper can support library understanding. The analyses produce summaries of function interfaces; the summaries can help developers verify their understanding of these interfaces. If a summary exhibits an unexpected feature, or lacks an expected one, a developer may investigate the cause. Such information may be useful to both the creators and users of a library.

The results of our analysis may also be useful for tracking API evolution. Given function summaries from two different versions of a library, a simple differencing tool can show changes in the prototypes of library functions, as well as changes in their higher-level behavior.

3. Interface Specification in C

We would like to generate library bindings for low-level code that are both safe and natural. A safe binding introduces no more memory unsafety than is inherent to the guest library being used. A natural binding supports host-language programmers by integrating the guest library into the high-level services of the host language.

Unfortunately, the C type system itself stands as an obstacle to both of these goals. In one regard, the type system is highly unambiguous: every parameter in C is passed by value, and pointers are simply addresses. Considered more closely, however, several higher-level idioms are present; programmers simply lack the syntax to distinguish them.

3.1 Pointer Parameter Ambiguities

Modern host languages typically offer some tightly-integrated sequence type, such as a list or dynamic array type. Programmers in a host language will prefer these containers over containers defined by a guest library. Thus, it would be natural to allow programmers to pass host-language sequence types across bindings to guest libraries, with appropriate data transformations applied automatically.

In C, the only first-class sequence type is the array. Unfortunately, C offers no enforced syntax for declaring array-typed function parameters: arrays are passed by address, and therefore a function which processes an array must declare that it takes a pointer to the array's element type as a parameter. C's type system simply cannot distinguish array parameters from other pointer parameters. Any interface generator based only on parsing C headers, then, requires an annotation to recognize opportunities to allow users to pass host-language sequence types in place of raw C arrays.

Pointer-typed parameters are likewise overloaded to simulate call-by-reference, such as for multiple return values. The C language provides at most one return value for any function, and offers no call-by-reference parameters. A function that produces more than one value must either return a wrapper structure or accept pointers to locations where additional output values should be stored. To avoid cumbersome use of custom `struct` types, many libraries choose the latter approach.

For example, the standard C library function `frexp` takes two parameters: a floating point number `x` and an integer pointer `exp`.

```
double frexp(double x, int *exp);
```

This function returns a value `r` and stores a second value `e` through the pointer `exp` such that $r = x \times 2^{e}$. Here, `exp` is used as an output parameter, but is syntactically indistinguishable from a generic pointer (or indeed an array).

Many host languages have better support for multiple return values. Library bindings can perform a useful service by transforming instances of this common pattern into real multiple-return-value constructs in the host language. A wrapper function can automatically allocate space for these output parameters, pass their addresses to the underlying library function, and return the results through the host language's native multiple return value mechanism. Again, C headers do not provide enough information to distinguish this construction from a standard pointer parameter, and a binding generator that relies only on headers would need annotations to detect it.

3.2 Object Ownership

One significant benefit of working in a high-level language is a garbage collector's assistance with memory management. Such a facility can also help to clean up other program resources, like file handles. Since users of most potential host languages are accustomed to automatic resource management, it is desirable to integrate foreign library resources with this infrastructure.

C libraries often contain a set of *constructor* functions that return handles (typically pointers) to newly allocated and initialized objects.[1] A natural high-level language binding for a constructor function would extend automatic resource management to cover these newly-allocated objects. While registering these objects with the host language garbage collector is important, it is equally important to ensure that they are cleaned up, or *finalized*, correctly. No one general-purpose finalizing function is appropriate for all objects allocated in C, since the host language may not know their composition and semantics. Properly reclaiming resources held by complex objects, such as file handles or database connections, requires more than simple memory deallocation.

Constructor functions are syntactically indistinguishable from any other C function that happens to return a pointer. Likewise, finalizers are indistinguishable from any other function that takes a pointer-typed argument. A binding generator that relies on header

[1] C has no formal object system. We use *object* here to describe structured regions of memory that occupy resources when constructed and release them when finalized.

files alone would require annotations to detect constructors and finalizers. However, identifying these functions helps us to create library bindings that are safe and natural. Since header files are insufficient, we must look deeper. By detecting functions that create and destroy resources, static analysis of library source code can help us build the bindings we desire.

The goal of a library binding is to follow the resource management conventions of the guest library without needlessly burdening the host language programmer. Garbage collector integration is only one broadly-applicable example of how our analysis can help with cross-language resource management; other management policies could be applied. Regardless of the strategy employed, cross-language resource management is a serious issue. A search of the GNOME Project Bugzilla database [14] reveals numerous bugs in library bindings relating to memory management[2].

4. Analyses

We have implemented a series of analyses to infer enough information to provide higher-level interfaces. All of our analyses make conservative assumptions because we do not have an entire program to analyze.

Our analyses are all interprocedural, context-insensitive, and path-insensitive. In each analysis, information propagates along the call graph from called functions (callees) to their callers, but not vice versa. This ensures that imprecision incurred from our lack of path-sensitivity does not lead to false inferences about callees.

A library may rely on other libraries, and most C libraries rely on the standard C library. In order to analyze dependent libraries separately, our suite of analyses take a set of *interface descriptions* as an additional input. The interface description for a library contains all the facts about its functions (and their parameters) that our analysis requires. These descriptions are stored as an immutable *base set* of facts, which our analysis consults when it encounters a call to an external function. The domain of facts provided in these descriptions is exactly the domain of facts produced by the analysis itself. Thus, the analysis is modular per-library: the output of the analysis on one library can be used as an input interface description for another library that relies upon it. To bootstrap our analysis, we have hand-written interface descriptions for the C standard library and the Linux kernel system call interface. These descriptions serve as a base set for all libraries.

We have also implemented a lightweight, optional annotation system that can improve the analysis output by putting extra information into the base set of facts before analysis begins. We discuss these annotations with their associated analyses. This annotation system is separate from the source code of the library, and so requires no modification to the library itself.

In the following discussion, we assume that the guest language is C, and that the guest library source code has undergone the following preparatory transformations:

1. Array operations have been replaced with equivalent pointer-arithmetic. For example, `buffer[offset]` is rewritten as `*(buffer + offset)`.
2. Each function has a unique exit node.
3. The entire program has been placed in static single assignment (SSA) form with global value numbering (GVN) per Alpern et al. [1].
4. A global alias analysis has been performed.

[2] A few examples are GNOME bugs 129754, 133681, 313861, 358294, 482795, and 498334. Our approach could have prevented at least five of these.

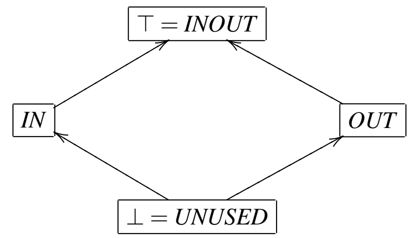


Figure 1: Hasse diagram of output parameter lattice. An edge $a \rightarrow b$ represents $a \prec b$ in the lattice partial order.

4.1 Output Parameters

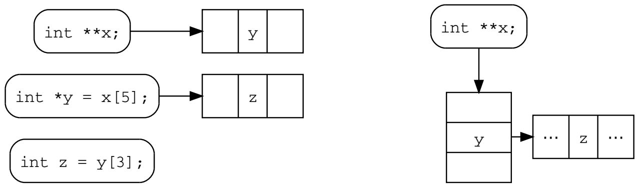
Our first analysis facilitates the multiple return value transformation discussed in Section 3.1. To perform this transformation, we need to know which parameters to any given function are output parameters. An *output parameter* is a pointer-typed parameter that is always written through before being read from. Other pointer-typed parameters fall into one of three remaining categories. An *input parameter* is always read from but never written through. An *input/output parameter* is read from and later written through. An *unused parameter* is neither read from nor written through at all.

We classify pointer-typed parameters using a forward dataflow analysis. The analysis operates over single program statements and considers each pointer-typed parameter in isolation. The analysis for a given formal parameter `p` in a function `f` is constructed as follows. The **state** of `p` at each statement is an element of the bounded lattice given in Figure 1. The **initial state** on entry to `f` is $\bot$. The **join operation** at any statement is the least upper bound (lattice join) of the exit states from all predecessor statements. The **transfer function** at a given statement depends on the syntactic form of that statement:

- For statements that write through `q` where `q` must alias `p`, the exit state is the least upper bound of the entry state and *OUT*. For example, following `*p = v`, a pointer that had never previously been used at all ($\bot$) will acquire the state *OUT*, while a pointer that had previously been read from (*IN*) will acquire the state *INOUT*.
- For statements that read from `q` where `q` may alias `p`, the exit state is *OUT* if the entry state was *OUT*, or is the least upper bound of the entry state and *IN* otherwise. Thus, following `v = *p`, a pointer that was previously unused ($\bot$) will acquire the state *IN*, while a pointer that had previously only been written through (*OUT*) will retain the state *OUT*.
- For function calls that pass `q` as a parameter to function `f'` where `q` may alias `p`, the writing and reading rules discussed above apply and are determined by the states of the appropriate parameter of `f'`. Calls through function pointers are treated conservatively: if `q` is used as a parameter to a function pointer, it acquires the state *INOUT* unless it has already been proved to be *OUT*.
- For all other statements, the exit state is the same as the entry state.

At any given statement, the least solution to these dataflow equations associates *OUT* with each pointer-typed parameter that *must* be written through at least once before it *may* be read from. Parameters in the *OUT* state at a function's unique exit node, then, are the output parameters for that function.

The above algorithm applies only to pointers to primitive types (including other pointers). A straightforward change is required to extend it to handle pointers to aggregates, as follows. At the



(a) Step 1 (b) Step 2

Figure 2: Array reconstruction example

beginning of the analysis of a function, break each input structure down field-wise and perform the analysis as usual. At the end of the analysis, if every field of a pointer-to-structure parameter is in the *OUT* state, that parameter is an output parameter.

We analyze the functions in a library in callees-before-callers order so that the final states of the parameters of all called functions in the current function are known. We resolve cycles in the call graph by iterating over the functions forming the cycle to find a fixed point.

4.2 Arrays

The C type system does not distinguish pointers used to denote reference parameters from those used as a pointer to the first element of an array. (This is a fundamental aspect of C, not merely an artifact of our preparatory replacement of array operators with pointer arithmetic.) Our array analysis recovers this lost information by identifying which pointer parameters are used in *array contexts*. A pointer is used in an array context if it is an operand of a pointer arithmetic expression and the result of that expression is dereferenced (either read from or written through).

The array analysis proceeds in two phases to analyze a function `f`: (1) interprocedural information propagation and (2) local array use identification. In the first phase, the analysis examines each callee `f'` of `f`. Consider a value `q` which *must* alias a parameter `p` of `f`, and which is passed as the *n*th parameter of `f'`. If the *n*th parameter of `f'` is known to be used as an array (either because `f'` was analyzed before `f` or was introduced as part of the base set of the analysis), then `q` and therefore `p` *must* also be arrays.

The second phase of the analysis begins by collecting all SSA values which are instances of pointer arithmetic in a given function, discarding any for which the result is never used. The pointer operand of each of these values is considered to be a one-dimensional array and the value itself is an array element.

Considering only this set of one-dimensional arrays, the initial *array set*, the analysis iteratively reconstructs array types in the function as follows:

1. Consider a pair of values in the current array set $v_1 = (p + \texttt{offset})$ and $v_2 = (v_1 + \texttt{offset})$. Note that the pointer operand of v_2 is v_1.
2. For each such pair v_1 and v_2, extend the array associated with v_1, and any arrays by which v_1 is contained, by one dimension to create the next array set.
3. Continue this process until no arrays are extended.

This process is illustrated in Figure 2. In this example, `x` is a variable of type `int**`, really an integer matrix. Assume that `x` is not passed as a parameter to any other calls. The first pass of the analysis over all pointer arithmetic operations reveals two arrays of one dimension each: `x` (which is known to be at least an array of `int*`) and `y` (which is known to be an array of `int`s). The second pass identifies `y` as being a member of the array pointed to by `x`; the analysis concludes, therefore, that `x` is a two-dimensional array of `int`.

This analysis is flow-insensitive. The full function is only scanned once, to collect the initial array set which serves as a work list. The number of iterations over the instances of pointer arithmetic in a given function is bounded by the number of dimensions of any array in the function. Note that we do not infer which parameters correspond to the lengths of arrays. Our system could be augmented with a separate analysis to infer array-length parameters, such as the one proposed by Cousot and Halbwachs [6].

One important class of array parameter are not identified by the analysis described above: parameters that are assigned to fields of a non-local `struct`, but which are not used in array contexts within the function being analyzed. To handle these case, we make an initial pass over all the statements in the library, identifying `struct` fields that are used in array contexts. Then, when a function is analyzed, any parameter that is assigned to an array field of a `struct` is marked as an array.

4.3 Resource Management

We group several analyses together under the aegis of resource management; their goal is to infer enough information about resource management in a library to allow us to interact intelligently with, and utilize the services of, a host-language memory management system.

We are primarily interested in two types of functions with which we can build a model of resource management in C:

Constructors create and return references to new objects. Two examples from the C standard library are `malloc` and `calloc`. Following the convention of the standard library, we allow constructors to return `NULL` as a means of indicating an error.

Finalizers take references to objects returned by constructors and release the associated resources. The example from the C standard library is `free`. As with constructors, we allow finalizers to return early if they are passed `NULL`.

In order for foreign objects obtained from these constructors to be of any use, they must generally be passed back from the host language to the guest library. This means that two memory management systems interact with foreign objects: one manual and the other automatic. Ensuring that only one of these systems finalizes any given object, and only does so after the other memory management system is done with it, requires an ownership model.

We use a model much like one proposed by Heine and Lam [17], in which

- there is exactly one owning pointer for every object allocated by the guest library,
- ownership may be transferred from one pointer to another, and
- for each member field of a given object class (e.g., a `struct` type), the member is either always owned or never owned.

Unlike Heine and Lam, we do not infer the ownership properties of fields; rather, we assume that objects own each of their member fields unless given explicit annotations to the contrary. Our analysis determines which functions cause ownership transfers. Then, at run time, the wrapper functions created by our code generator provide this information to the host memory-management system. We use an escape analysis to infer where these ownership transfers occur.

4.3.1 Escaping Values

Consider a value `v` pointed to by a pointer `p` in a function `f`. If `v` may become reachable from a new non-local scope due to `f` or one of its callees, we say that `v` *escapes from* `f`. Furthermore, assume that `v` escapes from `f` and that `p` is a parameter of `f`. If this `f` is invoked from a host language, we consider the ownership of v (pointed to by parameter p) to be *transferred*, as per our ownership model.

Our escape analysis is based on the may-alias augmented SSA form of Cytron and Gershbein [8], extended interprocedurally as by Harrold and Soffa [16]. This formulation is well-suited to our problem; it allows us to analyze libraries sparsely, calculating only the alias information for function arguments and their necessary dependencies. If a function pointer is provided from the host language as a callback, there is nothing to analyze at compile time and we conservatively assume that all of its pointer parameters escape. Functions provided by library dependencies may be similarly unavailable at analysis time; however, the interface descriptions for these dependencies include escape information for their functions.

We introduce an optional annotation to model *weak references* in objects. The annotation is of the form:

```
(weak-reference struct-name field-index)
```

where field-index is the zero-based index of a field in a C struct which always holds weak references. A weak reference has no ownership over its contents, and callers from host languages need take no special action to inform the run-time system of their presence. If the escape analysis determines that the value pointed to by some parameter escapes only into one of these weak references, that parameter is recorded was *weakly* escaping. This information is propagated to callers as the analysis proceeds bottom-up.

4.3.2 Constructors

The constructor analysis identifies functions which always return a reference to a new object that can safely be deallocated by an appropriate finalizer. Candidate functions must also relinquish ownership of the returned value; this means that a function is not a constructor if the return value escapes from the function by any means other than the return value. Further, this implies that if a guest library constructor is called from the host language, the run-time system of the host language becomes the new owner of the returned object. Thus, the host language's run-time system is empowered to manage resources given to it by guest-language constructors. Note that this is a "must" analysis: a function which sometimes creates new objects and sometimes reuses existing global objects is not a constructor.

For each value-returning function, the constructor analysis considers three cases:

1. If the return value is the result of calling some other function which is already known to be a constructor, then the function being analyzed is a constructor as well.
2. If the return value is an SSA Φ node and all incoming values can be safely deallocated with some available finalization function, then the function being analyzed is a constructor.
3. Otherwise, the function being analyzed is not a constructor.

Note that the preparatory GVN transformation obviates the need for any explicit copy propagation during this stage.

Developers may steer the constructor analysis using a second optional annotation, of the following form:

```
(assign-finalizer ctor-name finalizer-name)
```

This directs the analysis to assume that ctor-name is a constructor whose returned resources are reclaimed by passing them to finalizer-name. This is useful for forcing the analysis to ignore any caching of returned values that a constructor might perform by way of internal bookkeeping. Internal bookkeeping often manifests at analysis time as an escaping return value. Our escape analysis, described in the previous section, allows our implementation to inform the user of functions which would be constructors if not for an escaping value. This feedback can provide some guidance on where the assign-finalizer annotation would be profitable.

```
void *xmalloc(int size) {
  LIBENV *env = lib_link_env();
  LIBMEM *desc;
  int sz = align(sizeof(LIBMEM));
  desc = malloc(size);

  desc->next = env->mem_ptr;
  env->mem_ptr = desc;

  return (void *)((char *)desc + sz);
}

glp_prob *create_prob() {
  glp_prob *lp = xmalloc(sizeof(glp_prob));
  lp->row = xcalloc(10, sizeof(GLPROW *));
  lp->col = xcalloc(10, sizeof(GLPCOL *));
  return lp;
}

LPX *lpx_create_prob() {
      return create_prob();
}
```

Listing 1: Transitive constructor example, adapted from GLPK [24]

As a base case, our implementation considers the standard C library functions malloc and calloc to be constructors. The free function is their finalizer, as it deallocates the results of those functions. To facilitate error checking in finalizers, the analysis additionally considers the NULL pointer to be an acceptable return value from a constructor. This is both safe and prudent: safe because calling free(NULL) is always valid in C, and prudent because malloc and calloc return NULL on allocation failure.

Starting with this base set of constructors, the analysis transitively identifies additional constructors whose return value is always either NULL or the result of a call to an already-known constructor. Constructor candidates are visited bottom-up, callees before callers. We handle recursion by iterating until a fixed point is reached.

Listing 1 offers an example. The first function, xmalloc, acts as a wrapper around the standard malloc function and performs some additional bookkeeping. Since the escape analysis determines that the value returned by xmalloc escapes into the global environment, a manual annotation is required to inform the analysis that xmalloc is really a constructor. With this annotation in place, the analysis determines that create_prob is also a constructor since the value it returns is itself the result of a constructor. Likewise, lpx_create_prob is a constructor by transitivity, since it returns an object allocated by create_prob.

It is straightforward to extend this analysis to cover values constructed and returned through output parameters (discussed in Section 4.1). Our implementation forgoes this generalization as we find that it is uncommon in practice.

4.3.3 Finalized Parameters

The finalizer analysis is closely related to the constructor analysis. It identifies *finalized* function parameters, defined as those representing resources that are always deallocated on every call to the given function. Functions that finalize parameters assume ownership of those finalized parameters; this means that a host language run-time system must relinquish ownership when calling such functions. As in the constructor analysis, we allow a special exception for NULL pointers: a parameter need not be deallocated if it is known to be `NULL`.

We identify finalized parameters using a forward dataflow analysis. The analysis operates over single program statements (not whole basic blocks) and considers each pointer-typed parameter in isolation. The analysis for a given parameter `p` in a function `f` is constructed as follows. The **state** of `p` at each statement is either *finalized-or-`NULL`* or *other*. The **initial state** on entry to `f` is *other*.

The **join operation** for a statement *s* considers several cases:

- If *s* is the true-edge successor of a conditional of the form `if (q == NULL)` and `q` must alias `p`, then the entry state for `p` at *s* is *finalized-or-`NULL`*. Pre-analysis normalization of conditionals extends this to include negated and non-negated equivalents such as `if (q)` and `if (!q)`.
- Otherwise, if exit states for all predecessors of *s* are *finalized-or-`NULL`*, then the entry state for *s* is *finalized-or-`NULL`* as well.
- For all other statements, the entry state is *other*.

The **transfer function** for a statement *s* considers two cases:

- If *s* is a call, `q` is an actual parameter in that call, `q` must alias `p`, and the argument position of `q` corresponds to one known to be finalized by the callee, then the exit state of *s* is *finalized-or-`NULL`*.
- For all other statements, the exit state is the same as the entry state.

At any given statement, the least solution to these dataflow equations associates *finalized-or-`NULL`* with each pointer-typed parameter that must be finalized or `NULL` at the given statement. Parameters in the *finalized-or-`NULL`* state at a function's unique exit node, then, are finalized parameters for that function.

There are two annotations available to introduce a particular parameter of a function into the base set of known finalizing parameters. These annotations augment the interface descriptions provided as input to the analysis. The interface description of the standard C library provides only one finalizing parameter: the only argument of the `free` function.

The first annotation, introduced in the constructor analysis (Section 4.3.2), is a useful shorthand when the finalizer function has exactly one parameter. The analysis raises an error if this annotation names a destructor with no parameters or more than one parameter. The second annotation is of the form:

```
(declare-finalizer function-name arg-index)
```

This annotation adds the argument at position `arg-index` in the named function to the base set of finalizing parameters.

Like the constructor analysis, the finalized-parameter analysis operates in callees-before-callers order. The analysis resolves cycles in the call graph by iterating over the functions in the cycle until reaching a fixed point; when calculating this fixed point, the analysis is optimistic and initially assumes that all pointer parameters are finalized.

Consider Listing 2 as an example. The `xfree` function passes an alias of its only argument, `ptr`, to a function known to finalize its argument, `free`, on line 11. The state of `ptr` at the end of `xfree` is, then, *finalized-or-`NULL`* and it is therefore a finalized parameter.

The second function, `delete_prob`, builds on `xfree`. On line 15 `delete_prob` checks if it was passed a valid pointer. At line 16, then, we know that `lp` is `NULL` and so its state is *finalized-or-`NULL`*. At the end of the other branch of the `if` statement, the code passes `lp` to a known finalizer, so the state of `lp` after this branch is also *finalized-or-`NULL`*. Applying the join function to these two exit states in the unified return node allows the analysis to conclude that `lp` is a finalized parameter.

```
1  void xfree(void *ptr) {
2    LIBENV *env = lib_link_env();
3    int sz = align(sizeof(LIBMEM));
4    LIBMEM *desc = (void *)((char *)ptr - sz);
5    if (desc->prev == NULL)
6      env->mem_ptr = desc->next;
7    else
8      desc->prev->next = desc->next;
9
10   env->mem_total -= desc->size;
11   free(desc);
12 }
13
14 void delete_prob(glp_prob *lp) {
15   if(!lp)
16     return;
17   xfree(lp->row);
18   xfree(lp->col);
19   xfree(lp);
20 }
```

Listing 2: Transitive finalizer example, adapted from GLPK

5. Generating Library Bindings

Using the information inferred about a library by these analyses, we can automatically generate host language bindings. We have written a binding generator targeting the Python programming language.

Our bindings use Python's standard `ctypes` module, which allows C functions to be described and then called at run time using pure Python code. `ctypes` provides primitives for creating and manipulating C types, as well as support for passing Python functions to C for use as callbacks. Optionally, `ctypes` allows the types of return values and function parameters to be specified, enabling run-time type checks in Python. Our binding generator utilizes such type declarations.

5.1 Multiple Return Values

When creating a Python wrapper for a C function that uses output parameters, our code generator transforms the interface into a more idiomatic Python form. This transformation removes output parameters from the Python argument list and instead returns all function outputs in a single Python tuple.

Specifically, for a C function $\mathtt{f}_c$ of arity n with m output parameters, our generated Python wrapper function $\mathtt{f}_{py}$ has $n-m$ parameters. Wrapper $\mathtt{f}_{py}$ is implemented as follows:

1. For each output parameter, allocate a variable of the appropriate type. The variable's initial value does not matter, since output parameters are always written before being read.
2. Call the original C function $\mathtt{f}_c$ by passing the $n-m$ normal arguments by value, and the m newly-allocated output parameters by reference.
3. Return a Python tuple containing the return value of $\mathtt{f}_c$ (if it exists) followed by the values of the output parameters.

The proper ordering of parameters is tracked and maintained at each step so that any interleaving of output and normal parameters work as expected.

Consider the `frexp` example from Section 3.1. Our analysis creates a Python function of one argument that returns a 2-tuple, with the function interface `(x, exp) = frexp(value)`.

This transformation benefits the user of the Python library wrapper in two ways: (1) it brings the function interface closer to the standard idioms of the Python language, and (2) it encapsulates the `ctypes` boilerplate required to allocate and pass output parameters by reference.

Parameters detected by our analysis to be input parameters or input/output parameters may be interesting for other purposes, but are not currently treated specially by our code generator.

5.2 Sequence Mapping

If a library function $\texttt{f}_c$ has one or more array parameters, our generated Python wrapper $\texttt{f}_{py}$ accepts both `ctypes`-managed arrays and standard Python lists for those parameters. When called, $\texttt{f}_{py}$ checks the run-time types of actual parameters expected to be arrays. If any such parameter is actually a Python list, the wrapper function:

1. Allocates a C array of the same length as the list,
2. Performs a shallow copy of each element from the list to the array, and
3. Passes the array to $\texttt{f}_c$ in lieu of the original list.

This procedure is recursively applied to nested lists, producing an appropriately nested multidimensional array.

When $\texttt{f}_c$ returns, our $\texttt{f}_{py}$ wrapper copies the contents of the array back into the list it received as an argument. As an optimization, this copy-back step is skipped if C array elements are `const`-qualified. It would also be possible for a static analysis to detect functions that never modify their array parameters, for which this copy-back step would be redundant. Such an analysis would be similar to our output parameter analysis, as well as to prior work on `const` qualifier inference by Foster et al. [10].

Like the multiple-return-value transformation described in the previous section, this change brings the wrapper function closer to the standard idioms of Python, as it allows the programmer to use Python's natural list type instead of directly manipulating `ctypes` objects. We choose to copy elements shallowly to minimize performance surprises. The run-time type checks applied by `ctypes` will raise an error if the contents of a passed-in list are incompatible with the array's expected element type.

5.3 Garbage Collector Interaction

C values are never directly exposed in Python. Rather, each C value is encapsulated within a Python wrapper object. This wrapper object provides all of the standard behaviors expected of any Python runtime value, such as method dispatch and proper integration with the Python garbage collector. Figure 3 shows the relationship between an allocated C object and its Python-wrapped pointer. Note that there is exactly one Python wrapper object for every C pointer obtained from a constructor. This strong coupling offers generated library bindings a single point of control over object lifetimes. In particular, we can arrange for C finalization functions to be called on constructed objects whenever the corresponding Python pointer wrapper is garbage-collected.

We subclass the standard `ctypes` pointer-wrapper class to add a private field containing a reference to the appropriate C finalizing function. This subclass also defines a `__del__` method; the garbage collector will call this immediately before reclaiming an unused object. Our `__del__` method retrieves the stored C finalizing function and calls it, thereby using automated Python garbage-collection actions to drive explicit C resource management.

Three scenarios involving calls from Python to C require special handling in wrapper functions with regard to resource management. In each of the following cases, let $\texttt{f}_{py}$ be a Python wrapper function around a C function $\texttt{f}_c$:

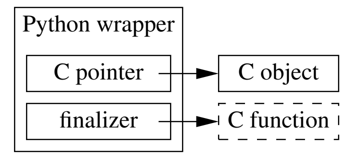


Figure 3: `ctypes` wrapper object structure

- If $\texttt{f}_c$ is a **constructor**, $\texttt{f}_{py}$ calls $\texttt{f}_c$ and temporarily stores its return value, `v`. $\texttt{f}_{py}$ then wraps `v` in our derived pointer wrapper, assigning `v` a finalizing function, producing `v'`. $\texttt{f}_{py}$ returns `v'`.

  This process serves to register objects constructed in the guest library with the host Python memory manager.

- If $\texttt{f}_c$ has any **finalizing** parameters, $\texttt{f}_{py}$ disables the `__del__` method for each one before passing them to $\texttt{f}_c$.

  Disabling the finalizers registered with the Python garbage collector essentially transfers ownership from Python back to C. This allows the Python memory manager to finalize objects in the common case while still giving programmers the power to explicitly (but safely) finalize objects.

  By definition, using an object after it is finalized results in undefined behavior. To prevent finalized objects from being used in our generated Python bindings, we disable the Python wrapper object before passing the pointer that it wraps to a function which finalizes it. This step ensures that future uses result in well-behaved Python exceptions, not the undefined behavior that similar C code would yield.

- Analogously, if $\texttt{f}_c$ has any parameters that **escape**, $\texttt{f}_{py}$ disables their corresponding `__del__` methods. Lifetime management for escaped objects is the responsibility of the new owner, not the Python garbage collector. As discussed in Section 4.3.1, the Python wrappers take no special actions for parameters that only escape into weak references.

  The analysis can falsely detect that a given parameter escapes, subject to the precision of the escape analysis. Disabling the `__del__` method of an object passed through such a parameter will cause that object to leak unless it is explicitly finalized.

To see the importance of following an ownership model, consider Listing 3. The `add_property` function transfers ownership of a property object, `p` to the component `c`. Our analysis correctly infers that `p` escapes. The finalizer for components, `component_free`, later deallocates all properties of which it has assumed ownership. If host language calls to `add_property` did not disable the finalizer on `p`, it would be freed once when it went out of scope in the host language, and again when `c` was deallocated. This could lead to memory corruption or crashes.

When constructors are not explicitly paired to finalizers with the `assign-finalizer` annotation, we apply a naïve but effective matching heuristic: if the return type of a constructor exactly matches the argument type of a single-argument function that finalizes its only argument, then the two are considered a pair. If two finalizers could potentially be applied to an object, the heuristic selects neither and emits a warning. Constructed objects for which no finalizer is provided through annotation, and for which the matching heuristic fails, are not assigned any finalizer. The single-argument restriction allows us to generate calls automatically: if multiple arguments were present, our analysis could not guess what values should be passed for the others. Finalizers that take more than one parameter are still manually callable and can be used in higher-level constructions, such as context managers in Python or `dynamic-wind` in Scheme [9].

```
void add_property(component* c,
                  property* p) {
  pvl_push(c->properties,p);
}

void component_free(component* c) {
  property* p;
  while((p=pvl_pop(c->properties)) != 0) {
    property_free(p);
  }
  pvl_free(c->properties);
  free(c);
}
```

Listing 3: Object ownership transfer, adapted from libical [5]

5.4 Alternatives

The information gathered by our analysis applies equally to any foreign function interface system. Instead of targeting the native high-level foreign function interface of a language, one could target low-level extension modules (which themselves tend to be in C).

Another alternative would be to target a project that includes multiple code generation backends already, such as SWIG. In order to target SWIG, we could apply our inferred annotations to C headers in the format that SWIG expects. While this would give us access to the many code generation backends that SWIG already supports, we chose not to do this because pure Python code is easier to evaluate in an automated way.

6. Experimental Evaluation

We have implemented the analyses of Section 4 for C using version 2.4 of the LLVM compiler infrastructure [23] and the Clang front end [22]. LLVM provides an SSA-based intermediate form and preparatory transformations and analyses, such as GVN. We use Andersen's algorithm for alias analysis, which is flow-, field-, and context-insensitive [2].

We evaluate our analysis and code generator by creating Python bindings to four open-source C libraries:

- The GNU Linear Programming Kit (GLPK), a library for defining and solving certain kinds of optimization problems [24].
- The libarchive project, a library for reading and writing files of various disk storage formats, such as tar [20].
- libical, a library for handling calendar- and schedule-related data in a standard format [5].
- The GNU Scientific Library (GSL), a collection of mathematical routines for scientific computing [13].

GLPK, GSL, and libical rely only on the standard C library, for which we have written manual annotations. In contrast, libarchive relies on four other open-source libraries: libacl [29], libattr [30], zlib [12], and bzip2 [28]. Thus, we build up analysis results for each of libarchive's dependencies before analyzing the library itself. Our analysis can operate on a library with any number of external dependencies, provided we have source code or manual annotations for all of them.

We seeded our analysis with two additional manual annotations to GLPK. These annotations explicitly map its `xmalloc` and `xcalloc` functions to their associated finalizer, `xfree`, as discussed in Section 4.3.2. No other manual annotations were necessary in our experiments.

The running time of our analyses is reasonable. Of our experiments, GLPK takes the longest to process: analyzing it took 17 minutes on a machine with 3 GHz Intel Pentium 4 processor and 1 GB of RAM running Red Hat Enterprise Linux 5. All other experiments ran in 5 minutes or less on the same machine.

Our analysis exports two kinds of information to our Python code generator: *type signatures*, allowing the generated Python code to call C library functions with type checking, and *inferred annotations* indicating higher-level function behavior, as detected by the analyses described in Section 4. We validate the type bindings by comparing them with those found by simpler analyses. Two of our four libraries, GLPK and GSL, also have independently developed, hand-coded `ctypes` bindings. In these cases, we compare the results of our analysis with these human-generated artifacts. This allows us to evaluate the utility of our annotations.

6.1 Correctness of Type Signatures

To validate the results of our type signatures, we use as reference a binding generator that operates solely on header files. Such generators can ensure type safety for parameters and return types, but cannot detect the more detailed information given by our inferred annotations.

Using version 0.5.4a of the `ctypeslib` code generator [18], we produced bindings for each of our four target libraries, and verified that our analysis finds equivalent types for function parameters and returns.[3] We also use these header-based results as an approximation of the "public" interface of each library. That is, we assume that a function declared in a library's public header is meant to be a public function.

The hand-coded wrapper for GLPK [25] provides Python bindings to the public functions of the library without changing any of the interfaces to be more "Pythonic" in style. Thus, it is comparable to the output of a header-based analysis. In the course of our experiments, we discovered several type bugs in the hand-coded wrapper. We have reported these bugs to the developer, who has since released a new version fixing them. Though finding code defects is not the goal of our analysis, this experience argues in favor of automating the creation of foreign function interfaces: manually declaring the low-level type signatures of hundreds of functions is both tedious and highly error prone.

6.2 Usefulness of Inferred Annotations

Higher-level annotations inferred by our analysis let our code generator modify the interfaces and behavior of wrapper functions, as discussed in Section 5. Table 1 shows the number and type of annotations discovered for each of our target libraries. We find that annotations are common: roughly 40% of library functions receive some form of annotation from our analysis. This affirms our hypothesis that richer information can be derived from library implementation code than is visible in library headers alone. It is important to note that inferring annotations for 40% of functions does not mean that the problem is only 40% solved. Rather, it means that bindings generated without this analysis would be substandard for four functions out of ten; the remaining functions simply have no remarkable aspects to their interfaces.

Unlike header-based analyses, our analysis covers all the source code of a library, analyzing every function. Thus, the annotations discovered by our analysis exist on library-internal functions as well as functions in a library's public interface. Table 1 results for both public library interfaces and complete libraries.

[3] In the few cases where our analysis results do not match the `ctypeslib` results, we find the difference is caused by an obscure bug in `ctypeslib` relating to typedefs of function pointers. We have verified the correctness of our results by hand in these cases.

Library	# funcs	constructors	finalizers	out		inout		array		escape		annot funcs
				funcs	params	funcs	params	funcs	params	funcs	params	
GLPK total	861	43	30	16	33	127	157	181	316	257	351	420
GLPK public	239	5	3	6	17	32	37	59	80	43	52	87
libarchive total	216	3	2	6	14	39	50	50	51	55	89	97
libarchive public	193	3	2	5	12	38	49	41	41	42	70	80
dependencies	183	8	1	17	25	18	27	77	98	28	50	99
ical total	1,018	117	11	7	9	16	22	318	329	177	198	513
ical public	917	107	7	3	3	12	17	281	290	168	186	461
GSL total	3,911	250	104	180	380	54	112	879	1,134	240	340	1,428
GSL public	3,863	250	104	168	364	52	110	879	1,134	240	340	1,416

Table 1: Counts of annotations inferred by our analysis. Where applicable, we show both the number of functions that have at least one parameter with a given annotation ("funcs"), as well as the total number of parameters with that annotation ("params"). The final column gives the total number of functions with any annotation. We have summed the numbers for libarchive's four dependencies.

Independently, Jaroszewicz [19] has created a hand-coded `ctypes` binding for the GSL library. Unlike the GLPK binding discussed above, this binding modifies function interfaces to be more compatible with Python idioms. For example, GSL has its own implementation of the `frexp` function discussed in Section 5.1; we derive the same binding as Jaroszewicz's hand-coded wrapper. An equivalent function exists in the Python math library, and also shares this interface.

As a second example, the `gsl_vector_minmax` function takes a `gsl_vector` structure and returns its minimum and maximum elements in two output parameters:

```
void gsl_vector_minmax(const gsl_vector *v,
    double *min_out, double *max_out);
```

The hand-coded wrapper takes only one argument, a vector, and returns the 2-tuple (minimum, maximum). Our binding generator automatically performs the very same transformation.

In some other cases, our transformed interface is similar, but not identical to, the hand-coded wrapper. One function in GSL has two output parameters which it may or may not write to. The return value of this function is the number of output parameters that were actually written: either 0, 1, or 2. The wrapper function produced by our code generator always returns a 3-tuple, with the first value indicating the number of subsequent values that are valid. The hand-coded wrapper function instead checks the number of valid values and returns either a 0-tuple, a 1-tuple, or a 2-tuple. We have not attempted to automate discovery of this rather atypical interface pattern.

One potential combination of annotations deserves special consideration. For a parameter marked as an *array*, our generated wrapper function can accept a Python list, passing a temporary copy as a C array to the library function as described in Section 5.2. However, if this parameter also *escapes* from the library function, then the library may store a reference to the temporary array. To ensure memory safety, our code generator simply disables the array transformation for such parameters; the Python programmer must pass a reference to an actual C array. We do not find this situation to be widespread. In our four experimental libraries, we find 141 array parameters that appear to escape, of which all but 26 are in GSL. Library-specific strategies for handling this combination of annotations could be provided as plug-ins to the code generator if the library consistently follows a convention.

6.3 Alternative Experiments

We perform two experiments that vary the details of our analysis. In the first, we change the definition of a constructor to be more strict: a constructor must return a constructed value, and never return `NULL`. (We continue to define the standard library functions `malloc` and `calloc` as constructors, though they do not meet this stricter standard. We likewise carry over our manual annotations to GLPK.) With this definition, our analysis finds very few constructors: just 38 in GLPK (4 in the public interface), 3 in libical (2 in the public interface), 1 in zlib, and none in any other analyzed library. This suggests that our initial definition of constructors, which allows their return value to be `NULL`, is more useful than the stricter definition.

Our second analysis variant concerns function pointers. All our analyses conservatively assume that the behavior of functions called through pointers is unknown. In particular, our escape analysis assumes that pointers passed as parameters to a function pointer may escape. Since the host-language garbage collector must give up ownership of a parameter whose owner may change (via escaping), objects presumed to escape in calls of this form are not automatically finalized. If neither the guest library nor the programmer finalize such an object explicitly, it becomes a memory leak.

In practice, we find values rarely escape during calls through function pointers; most of the escape annotations thus inferred are false positives. To examine the impact of function pointers on the precision of our analysis, we ran our experiments with the conservative assumption relaxed: parameters to function pointers are assumed *not* to escape. Table 2 shows the changes that result. Function pointers are commonly used in our libraries, and this change reduces the number of escape annotations inferred by our analysis.

Relaxing this assumption may lead our analysis to detect more constructor functions. Recall that we consider a function to be a constructor when it returns a reference to a new object that could not have escaped the function by any means other than the return value. If we conservatively assume more function parameters are escaping, fewer constructors may be detected. However, we find the difference to be minimal in practice. With the conservative assumption relaxed, our analysis detects 50 constructors in GLPK, with 8 in the public interface (compare with the 43 total constructors, 5 public, that are given in Table 1). No other libraries show any change in their constructor counts.

One might use a points-to analysis to compute the set of possible targets for each function pointer. Our analyses could be extended with this technique, possibly yielding more accurate results. However, a library analysis can never be a whole-program analysis. Libraries that expose a callback interface cannot make assumptions about the behavior of callback functions provided by unknown future library users, so conservative assumptions will still be necessary.

Library	original		relaxed	
	funcs	params	funcs	params
GLPK total	257	351	165	205
GLPK public	43	52	3	5
libarchive total	55	89	17	23
libarchive public	42	70	5	5
dependencies	28	50	15	19
ical total	177	198	172	184
ical public	168	186	163	173
GSL total	240	340	213	255
GSL public	240	340	213	255

Table 2: Number of escape annotations inferred under our original analysis rules and under the relaxed analysis rules described in Section 6.3.

6.4 Analysis Limitations

Our analysis, coupled with our Python code generator, can automatically produce a Python interface to a C library. The resulting interface may not be ideal. Even with our transformations, there may be idioms in the higher-level language that are preferable for expressing the lower-level semantics of the library.

For example, several GSL functions store a result into an output parameter, then return an integer error code that indicates the validity of this parameter. The function wrapper made by our binding generator stores the error code and the output parameter into a return tuple. In contrast, Jaroszewicz's hand-coded wrapper checks the returned code and raises a Python run-time exception if an error occurred. Automatically distinguishing integer error codes from ordinary integers, and modifying bindings accordingly, is beyond the scope of the present work. However, recent work in tracking integer error codes through operating systems suggests that this sort of inference is possible [15, 27]. Even without such an analysis, library developers wishing to offer rich, high-level interfaces can save considerable time and effort by building upon our automatically-generated bindings rather than raw C interfaces.

In Section 6.3, we discussed the (potentially unhelpful) effects of our conservative treatment of function pointers. A user of our tool may choose to relax the conservative assumption when creating bindings to a library for which she knows the assumption is overcautious. The user may also use our optional annotations to improve the accuracy of the generated binding. The `weak-reference` annotation (Section 4.3.1) is useful for removing needless escape annotations, and the `assign-finalizer` annotation (Section 4.3.2) can be used to recover missed constructors. We have not tried to "perfect" our results with annotations; our focus has been on developing the analyses themselves.

Our analysis requires either the entire source code of a target library, or manual annotations for each function for which code is unavailable. This is not a difficult restriction for open-source libraries such as those we analyze; however, it may be impractical for proprietary libraries or plug-in architectures that import code at run time.

We do not attempt to handle certain troublesome C idioms. A library that makes extensive use of `setjmp` and `longjmp` may invalidate the result of our interprocedural analyses, which do not model these functions' behavior. Code with inline assembly also falls beyond the scope of our analyses.

We currently do not include functions whose interfaces include unions in our interface description output, thus producing basic bindings which are at least correct, if not ideal. In practice, no library we have examined uses unions in its public interface.

7. Related Work

The SWIG project [3, 4] is a widely used foreign function interface binding generator which generates bindings for C and C++ libraries to many different host languages. It takes C or C++ header files as input, along with a *typemap*. Typemaps allow conversions between host and guest data types, as well as some of the higher-level transformations that we propose (such as converting output parameters into multiple return values). This approach is very flexible, but requires one annotation for every argument with special properties. As previously mentioned, our analyses could be used to supply typemaps to SWIG in order to take advantage of the large number of host languages supported by SWIG.

We are aware of two binding generators for Python's `ctypes` system that operate directly on unannotated C headers: the Python `ctypeslib` code generator [18], which we use to validate our experiments, and the `ctypesgen` project [7]. Each offers equivalent automation to SWIG, and is commensurately as unable to recover or exploit interface information absent from C headers.

Reppy and Song [26] take a similar approach to foreign function interface generation by analyzing header files and taking a set of per-function annotations (again in the form of typemaps) as input. They primarily focus on policy-driven interface generation, where interfaces are tailored to the conventions of the library being wrapped. This could, for example, allow various error return codes to be mapped to host language exceptions. However, it also requires some per-library specification and per-function annotations. Our analysis could provide partial inputs to augment these typemaps.

Instead of generating bindings, Furr and Foster [11] focus on verifying the correctness of existing bindings by performing a deep analysis of both bindings and library source code. They primarily check safety properties, including validating garbage collector interactions. This work is complementary to our own: Furr and Foster verify correctness while we offer correctness by construction.

Our analysis can be seen as a form of specification inference. Kremenek et al. [21] take a probabilistic approach to identifying ownership rules (including constructor and finalization functions). This approach introduces Annotation Factor Graphs, which can incorporate information on functions from any number of sources, including static analysis. Compared with our approach, Kremenek et al. use a more robust ownership model that also characterizes improper uses of APIs. Additionally, Kremenek et al.'s probabilistic approach does not require analysis of the source code of all dependencies of a program.

8. Conclusion

Mixed-language programming is an important practice, but programming languages can differ significantly in their type systems and resource-management models. These differences pose challenges to creators of cross-language bindings. We have characterized several idioms, common in C libraries, that map well to constructs typically provided by high-level languages. Our suite of static analyses over library source code can detect and describe these implicit patterns, freeing programmers from the burden of writing annotations. We use the output of our analyses in a prototype binding generator for Python. Applying this generator to several large, complex libraries yields bindings that integrate well with the high-level language while preserving safety in the low-level library, all with a minimum of programmer effort.

Acknowledgments

We would like to thank Evan Driscoll for early comments on this paper. We also thank the anonymous reviewers for their helpful feedback.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: http://doi.acm.org/10.1145/73560.73561.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Department of Computer Science, University of Cophenhagen, May 1994.

[3] D. M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[4] D. M. Beazley. Simplified wrapper and interface generator. `http://www.swig.org`, Nov. 2008.

[5] E. Busboom, A. Cancro, and W. Goesgens. libical. `http://freeassociation.sourceforge.net/`, Nov. 2008.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1978. ACM. doi: http://doi.acm.org/10.1145/512760.512770.

[7] Ctypesgen Developers. ctypesgen. `http://code.google.com/p/ctypesgen/`, Nov. 2008.

[8] R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 36–45, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: http://doi.acm.org/10.1145/155090.155094.

[9] M. Elder, S. Jackson, and B. Liblit. Code sandwiches. Technical Report 1647, University of Wisconsin–Madison, Oct. 2008.

[10] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006.

[11] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 62–72, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065019.

[12] J. Gailly and M. Adler. zlib home site. `http://zlib.net/`, Nov. 2008.

[13] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. Network Theory Ltd., Bristol, United Kingdom, revised second edition, Aug. 2006.

[14] The GNOME Project. GNOME Bug Tracking System. `http://bugzilla.gnome.org`, Jan. 2009.

[15] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In M. Baker and E. Riedel, editors, *FAST*, pages 207–222. USENIX, 2008. ISBN 978-1-931971-56-0.

[16] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.*, 16(2):175–204, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/174662.174663.

[17] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 168–181, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: http://doi.acm.org/10.1145/781131.781150.

[18] T. Heller. ctypeslib – useful additions to the ctypes FFI library. `http://pypi.python.org/pypi/ctypeslib/`, Nov. 2008.

[19] S. Jaroszewicz. ctypesGSL. `http://www.cs.umb.edu/~sj/ctypesGsl/`, Aug. 2008.

[20] T. Kientzle. libarchive. `http://people.freebsd.org/~kientzle/libarchive/`, Nov. 2008.

[21] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

[22] C. Lattner. LLVM and Clang: Next generation compiler technology. In *BSDCan 2008: The BSD Conference*, Ottawa, Canada, May 2008.

[23] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004. ISBN 0-7695-2102-9.

[24] A. Makhorin. GLPK (GNU linear programming kit). `http://www.gnu.org/software/glpk/`, Nov. 2008.

[25] M.-T. Pham. ctypes-glpk: A Python wrapper for GLPK using ctypes. `http://code.google.com/p/ctypes-glpk`, Nov. 2008.

[26] J. Reppy and C. Song. Application-specific foreign-interface generation. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 49–58, New York, NY, USA, 2006. ACM. ISBN 1-59593-237-2. doi: http://doi.acm.org/10.1145/1173706.1173714.

[27] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 15–20 2009.

[28] J. Seward. bzip2. `http://www.bzip.org/`, Nov. 2008.

[29] Silicon Graphics, Inc. libacl. `http://oss.sgi.com/projects/xfs/`, Feb. 2008.

[30] Silicon Graphics, Inc. libattr. `http://oss.sgi.com/projects/xfs/`, Feb. 2008.