Lackwit: Large-scale Analysis Of C Programs Using Type Inference

Robert O'Callahan[1] and Daniel Jackson[2]

Abstract

Many software engineering tasks require information about whole programs, but available tools either do not scale to handle programs of large size and complexity or provide only superficial semantic information. We have developed a fully automatic tool, Lackwit, that can answer many kinds of queries about C programs (for example, "locate all potential assignments to this field"), taking into account the effects of pointer aliasing, calls through function pointers, and type casts. The analysis is based on the kind of type inference developed for functional languages, but uses enriched types as abstractions of program functions and variables. It is simple and modular, and surprisingly accurate in practice. It can answer queries about a Linux 2.0 kernel (157,000 lines) in under ten minutes on a modestly-equipped PC. In this paper we describe the theory of the analysis,

---

[1] Address: Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213; Email: roc@cs.cmu.edu

[2] Address: Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; Email: dnj@lcs.mit.edu (work done while at CMU)

its realization when applied to C programs, how the tool presents the results of the analysis to the user, and the results of our experiments.

Index Terms: program understanding, semantics, alias analysis, type inference, visualization, modular, context-sensitive, incremental, C

1 Introduction

Maintaining large programs is often tedious and error prone. Tools that extract information from the source code itself can speed up the task, and reduce errors of misunderstanding.

Working programmers use a variety of tools, which we can classify broadly into three classes:

Lexical tools such as "grep" treat source code simply as a stream of characters, or more generally, tokens. They can answer queries such as "Which source lines contain the symbol `foo`?" These tools are usually easy to implement and language-independent. They are, however, limited by their very generality: such tools do not exploit knowledge of the source language.

Syntactic tools such as Bell Labs' "CIA" parse source code into an abstract syntax tree and manipulate the structure of the tree. Syntactic tools are needed to answer queries such as "Which procedures declare a local variable called `foo`?" These tools are harder to implement. They exploit knowledge of the source languages' syntax, but not their

Tool	Semantic depth	System visualization	Scalability
Grep[7]	—	—	✓
Rigi[11], RMT[10]+SMT[18]	—	✓	✓
CIA[19], other CBMSs	—	varies	✓
VDG slicer[4]	✓	—	✓
Chopshop[8]	✓	✓	—
LCLint[5]	✓	—	✓
Lackwit	✓	✓	✓

Table 1: A Broad Classification of Tools

semantics.

Semantic tools such as Lackwit not only parse source code, but analyze the code using the semantics of the source languages. Semantic tools are needed to answer queries such as "Can the value of pointer `foo` in function `bar` ever be passed to `free()`?"

Lexical tools are very commonly used. Syntactic tools are less popular, but many "program database" systems exist and are widely used (often found as part of integrated development environments). Semantic tools are not widely used (apart from translation tools such as compilers which do not really address program understanding tasks). Some modest and very specialized semantic analyses are performed by tools designed to address the "Year 2000" problem [20]. (In fact, some of these analyses are actually special cases of the analysis performed by Lackwit, although they appear to have been designed and implemented in an ad-hoc manner).

Three problems with existing general-purpose semantic analysis tools seem to restrict their utility: poor scability of performance, poor summarization of results, and inability to handle full featured, widely used source languages. Table 1 gives a very broad categorization of some existing tools. We designed Lackwit to answer a wide range of queries while overcoming these problems. For example, Lackwit can help answer questions like these:

- "I'm porting this program from Unix to NT; which integer variables contain file handles?"
- "Can pointer `foo` in function `bar` be passed to `free()`? If so, what paths in the call graph are involved?"
- "Field `f` of variable `v` has an incorrect value. Where in the source code might it have been changed?"
- "Let's restructure the 'current vehicle' object, the `cur_veh` field of the `map_manager_global` variable. Which functions modify it, and how might they interact?"

All these questions amount essentially to the same kind of query. The source text of a large program typically contains a huge number of names for storage locations: local and global variables and procedure parameters, and (for those that denote compound objects), their components. During execution of the program, values flow amongst locations, by copying and by references, and can thus be associated with many different names. Lackwit's archetypal query is: given a name, what other names can have values in common with it? Or, put more operationally, which other names identify locations to which a value could flow to or from a location with this given name?

By associating local variables with their procedures, Lackwit can answer questions phrased in terms of procedures too, identifying procedures to which a value may 'flow'. As we shall see later, a refinement of the scheme makes it possible, in addition, to distinguish reads and writes; this is how Lackwit answers questions (such as the latter two) about modification sites.

Lackwit's analysis is, not surprisingly, approximate. On the other hand, it is conservative: although it may identify names that are not relevant, it will not omit any names that are. Since C is not a safe language, however, this claim must be tempered. There are programs that can be compiled and successfully executed for which Lackwit will fail to give a conservative analysis. To close these loopholes would have required such a degree of conservatism that the analysis would be useless. Therefore, in order to give as much accuracy as possible for the programs that are actually encountered in practice, Lackwit makes a number of assumptions of well-formedness which, if obeyed, guarantee soundness. These assumptions — such as "no out-of-bounds memory accesses" — are exactly those typically made by programmers, and they result in unsound analyses only on programs that most C programmers would deem to be invalid anyway.

1.1 Example

As part of a case study described in more detail later (Section 0), a Lackwit user wanted to know which functions would have to be checked to modify the invariants of a data structure. A query about the `cur_veh` field of the variable `map_manager_global` was posed. Lackwit displayed the annotated call graph in Figure 1.

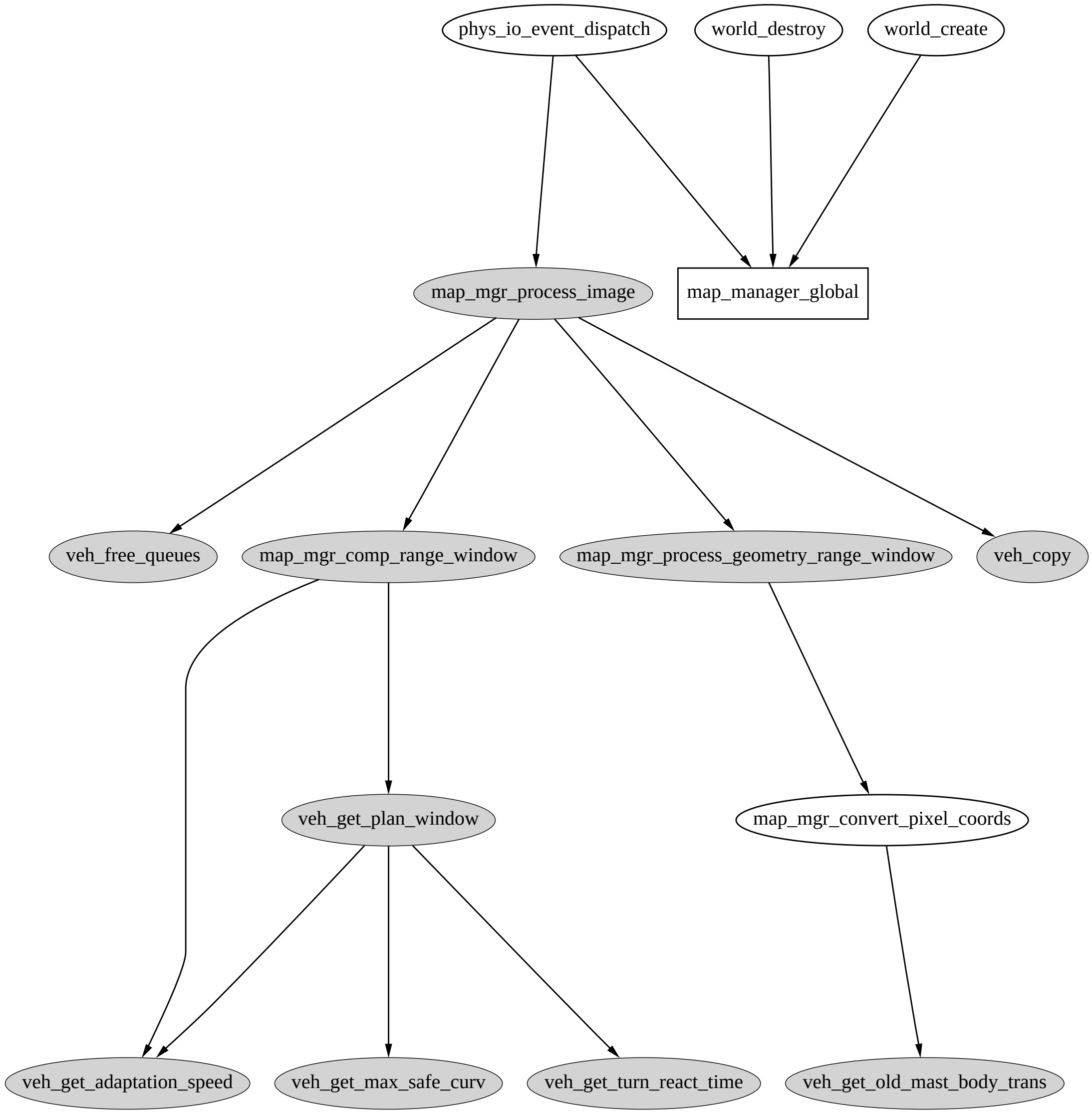

Figure 1: Example Graph View

We shall postpone the details of this graph's construction and interpretation. For now, note that the shaded ovals indicate functions performing operations on the vehicle structure (field extraction), and the unshaded ovals indicate functions passing along a pointer to the structure without actually operating on it. The edges between ovals represent function call relationships. Square boxes represent global variables, and the edge from `phys_io_event_dispatch` to `map_manager_global` represents an access to the variable by the function.

The program analyzed here consisted of 20,000 lines of C and was analyzed from scratch in about five minutes on a modest PC. The graph showed our client which functions would have to be checked when changing the current vehicle object's invariants. This information could not have been obtained by a tool that did not understand the semantics of the program, because many of the connections depend on aliasing induced by pointers. The analysis requires no annotations or other modifications to the program. It is completely static (the program is not executed); in particular, it can be applied to incomplete programs, by making optimistic assumptions about the missing code or by providing stubs that summarize the behavior of missing code. In fact, the program analyzed here was missing some libraries.

2 Using Type Information

Our approach is to exploit type information to answer user queries. Programmers in languages with rich type systems, such as ML, frequently read type signatures to understand program behaviour. This can also be done in C, although as we will see, not much is gained from the declared types alone.

2.1 Using Declared Types

To understand our approach, it helps to consider what information about a program is conveyed by the types of variables and functions. Consider declared types first. From the following prototype alone

```
DollarAmt getSalary(EmployeeNum e);
```

we can deduce – assuming there are no type coercions in the code of the function – that there is no data-flow path from the value of `e` to the result.

In general, if the type system satisfies certain technical conditions described later, and the program type-checks, then given the type of a value we are interested in, *the value can only flow to sites that have the same type*. This is the basic principle of Lackwit's analysis.

2.2 The Need For Analysis

In practice, the types that are declared explicitly are an inadequate source of information. Data abstraction can be used to distinguish types that would otherwise be identical; by introducing more or fewer abstract types, the programmer determines how much information the declared types provide. For example, the function above could have been given the type

```
int getSalary(int e);
```

from which we can no longer deduce that `e` does not flow to the result.

To deal with this problem, we use *type inference* to compute types that are consistent with the code, but in general provide more information than the declared types. We are, in essence, introducing abstract types wherever possible.

Programmers declare types that give less information than possible for three reasons:

- **Convenience**

  Programmers may use a single primitive type even when distinct types are possible, as above, to save the overhead of creating and using many types.

  We address this problem by partioning primitive types into distinct subtypes.

- **Completeness**

  The language may require complete type information to be provided, even if some parts are irrelevant to the code at hand. For example, in a C structure declaration all the fields must be given a type.

  We address this by pruning away any type information not required by the code.

- **Context Insensitivity**

  The language may associate a single type with each variable or function, even if multiple types are possible in different contexts.

  We address this by assigning generic types whenever possible.

These solutions require a richer type system than C's native types. Note that only the first is due to a lack of data abstraction. The second and third arise from a lack of polymorphism.

These problems are not unique to C, nor are they always indicative of poor programming. Programmer convenience will always limit the specificity of explicit type declarations.

The problem arises even for languages with some type inference built in, such as ML, because the programmer must invent new type names or wrappers to distinguish independent instances of the same type. Furthermore, if these annotations are added to encode very precise information about the behaviour of the program, they will need be laboriously changed whenever the program changes; worse, changes to types may not be localized to changes in the code.

3 Enriched Type System

The tool ignores the type declarations provided by the C program and computes new types in our enriched type system for all variables and functions. The types are drawn from the following grammar:

τ	::=	a	(Type variable)
	|	$\tau_1 \rightarrow^{\beta} \tau_2$	(Function)
	|	$\tau_1\ \mathbf{ref}^{\beta}$	(Mutable reference (pointer))
	|	$(\tau_1, \tau_2, ..., \tau_n)^{\beta}$	(Tuple)
	|	$\mathbf{num}^{\beta}$	(Scalar)

The *a* are drawn from an infinite family of type variables. The β are drawn from an infinite family of tag variables.

In our type system, many of the details of C are abstracted away; for example, all the numeric and enumerated types in C are mapped to our single scalar type. More information about the handling of various features of C is given below (Section XX).

Except for the β annotations, this is a standard polymorphic type system for lambda calculi extended with tuples and imperative features. The β annotations on the type constructors allow us to partition the primitive types into distinct subclasses for more precise results; for example, the `getSalary` example above could be given the type $(\mathbf{num}^{\alpha}) \rightarrow^{\phi} \mathbf{num}^{\beta}$.

The **ref** constructor gives pointer types. For example, the `strcpy` library function

```
char * strcpy(char * destination, char * source);
```

is given the type $(\mathbf{num}^{\alpha}\ \mathbf{ref}^{\beta}, \mathbf{num}^{\alpha}\ \mathbf{ref}^{\gamma}) \rightarrow^{\phi} \mathbf{num}^{\alpha}\ \mathbf{ref}^{\beta}$. This implies:

- The result may be aliased with dest; that is, there may be flow between the pointers.
- There may be flow between the characters of the strings.
- There is no flow between the source and destination arguments; the source and destination pointers are not aliased by `strcpy`.

Type variables can appear when some of the type information is incomplete. Consider the function `return1st`:

```
void * return1st(void * x, void * y) { return x; }
```

This can be given the type $(a\ \mathbf{ref}^{\beta}, b) \rightarrow^{\phi} a\ \mathbf{ref}^{\beta}$. The presence of the type variable a indicates that the type of the dereferenced contents of the pointer `x` is unconstrained, except that it must be the same as the type of the contents of `y`.

By distinguishing pointers, Lackwit is able to solve many pointer aliasing problems, such as "can pointer `x` and pointer `y` ever have the same value?". Similarly, by examining the annotations on function types, Lackwit can conservatively determine which functions a function pointer may reference; this is sometimes called "higher-order control flow analysis". The effects of both pointer aliasing and function pointers are accounted for in Lackwit's analysis, even in the presence of recursive data structures and function calls.

3.1 Polymorphism

A function called in different contexts may be presented with arguments of different types. Consider, for example, the following two calls to return1starg:

```
char * ptr1; struct timeval * ptr2; char * * ptr3;
return1st(ptr1, ptr2);
return1st(ptr2, ptr3);
```

The calls are valid because the choice of type variables is arbitrary. If `return1st` is forced to have just one type, then the choices for a, β, b and ϕ must be the same at both call sites, and `ptr1`, `ptr2` and `ptr3` become indistinguishable. However, if `return1st` can have multiple types, then the choices for a, β, b and ϕ can be different at each call site, and `ptr1`, `ptr2` and `ptr3` are distinguishable.

Sets of possible types are represented by polymorphic type schemes in the standard way [9]:

σ	::=	τ	(Monomorphic type)
	|	$\forall a.\sigma_1$	(Polymorphic quantification of a type variable)
	|	$\forall \beta.\sigma_1$	(Polymorphic quantification of a tag variable)

So `return1st` is actually given the type scheme $\forall a.\forall\beta.\forall b.\forall\phi.(a\ \mathbf{ref}^{\beta}, b) \rightarrow^{\phi} a\ \mathbf{ref}^{\beta}$. At each call site, the type scheme is instantiated by substituting for the quantified variables with fresh variables representing a new set of choices. For example,

$$(c\ \mathbf{ref}^{\alpha}, d) \rightarrow^{\chi} c\ \mathbf{ref}^{\alpha}$$

for the first invocation and

$$(e\ \mathbf{ref}^{\gamma}, f) \rightarrow^{\delta} e\ \mathbf{ref}^{\gamma}$$

for the second. Parametric polymorphism thus brings context senstivity to our analysis.

3.2 Type Rules

The tool must determine which types can be assigned to a given program. We therefore define a standard set of type rules to determine whether an assignment of types to program entities is valid. For example, the type rule for assignment expressions says that the source and destination must have the same type.

The type system corresponds to standard type theories which are known to be sound. This means that if an assignment of types to program entities obeys the rules, then the run-time values of variables must match the types given to the variables, and values never flow between entities of different types.

For a further discussion of type rules, see Appendix A.

3.3 Simplifying Type Rules Using An Intermediate Language

The Lackwit tool actually translates C code into a semantically equivalent intermediate form before performing type analysis. This intermediate form is a simple variant of the lambda calculus extended with primitive operations, an “if” statement, and “let” and “letrec” declarations, with a total of only seven different syntactic forms and seven type rules (listed in Appendix B: The Intermediate Language and Type Rules). The implications of these rules at the level of C source code depend on the details of the translation, which is rather complex. These details, and some examples, are given below.

4 Type Inference

The more types are distinguished, the more information they convey. Therefore the analysis needs to compute types for the variables and functions that make as many distinctions as possible while still maintaining type correctness.

4.1 Type Inference

The analysis uses a modified form of the standard Hindley-Milner algorithm “W”, which computes the most general type assignment of all the possible type assignments. These most general types are called ‘principal types’, and have the property that every valid type of an expression can be obtained from the principal type by applying some substitution to the type variables [3].

It is easy to see that principal types are the most general. Suppose that there is a valid type assignment T that gives expressions A and B different types; then any principal type assignment will also give them different types. For, if a type assignment U gives the same type to A and B, then no substitution will map U to T, and therefore U is not principal.

4.2 Handling References Safely

Mutable references are unsafe in the presence of polymorphism unless some special restrictions are imposed [21]. We introduce the standard "value restriction" [16]; in terms of C, this means that functions are polymorphically generalized, but variables are not. Intuitively, there is no relationship between calls to the same polymorphic function, and they can be treated as if they were calls to different copies of the same function (and we can infer different types for them). However, if a global variable is written and then read, those two references are related, and the same type must be used in both contexts.

4.3 Recursive Types

To handle realistic programs, we extend the type system to include recursive types. These arise frequently in C, for example in linked lists and other pointer-based data structures.

Our types are still drawn from the same grammar, but we allow infinite derivations as well as finite derivations. To make the types manageable, the analysis only infers types with the structure of regular trees, which can be finitely represented as graphs with recursion giving rise to cycles. The Hindley-Milner algorithm is altered slightly; instead of using term unification to solve the constraints, it uses regular tree unification. The algorithm still computes principal types, in the sense that every valid type corresponding to a regular tree is isomorphic to the result of applying some substitution to the principal type [2].

For example, consider the declaration

```
let f((a, b)) = if b = 0 then f(a) else b
in …
```

The type rules constrain the arguments of `f`, `a` and the pair `(a, b)`, to be the same type. (The standard rules for non-recursive types would declare that this program is not type-correct, because no finite type can have itself as one of its components; in terms of the unification algorithm, the "occurs check" fails.) As shown in Figure 2, we identify `a` as the type corresponding to the infinite regular tree (left) and the finite graph (right):

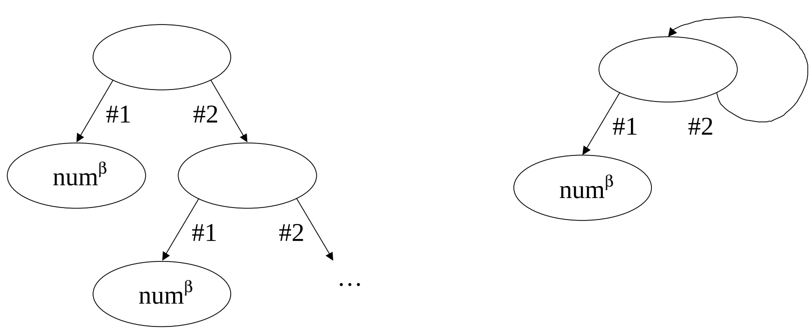

Figure 2: Representations of Recursive Types

The only significant modification to algorithm W is to eliminate the occurs check.

4.4 Type Errors

It is possible to construct programs that do not admit any valid type assignment, and thus cannot be analyzed using our techniques (algorithm W will reject the program). However, we only consider programs accepted by a standard C compiler, which therefore must type-check according to the rules of C. It is possible to translate the C types in such programs into types in our type system which satisfy the type rules, and therefore compilable C programs will never be rejected.

5 Analyzing C Code

The type system we have described was designed for functional languages based on the lambda calculus. In this section, we explain how the type system is applied to programs written in C, which has many features that cannot be handled directly.

Our general approach is to provide a semantics-preserving translation for each such feature, when possible. When that is not possible, the translation preserves the semantics of code that adheres to the ANSI C standard (including unofficial, but ubiquitous, conventions), and ignores any violations of the standard.

5.1 Scalars

All C scalars are translated to a generic "number" type. Thus floats, doubles, ints, chars, shorts, longs, and their variants are treated identically. This simplifies many issues, and in particular obviates the need for the implicit conversions performed by C compilers.

5.2 Structures and Argument Lists

C structures and function argument lists are translated to tuples. The tuples are constructed and projected using functions provided as part of the initial environment.

5.3 Mutable Variables

In C, all variables are mutable. Mutable variables are translated to intermediate code that uses the three primitive operations "ref", "deref" and "assign", with types

ref	$(a) \to^{\alpha} a \text{ ref}^{\beta}$
deref	$(a \text{ ref}^{\beta}) \to^{\alpha} a$
assign	$(a \text{ ref}^{\beta}, a) \to^{\alpha} ()$

These operations are commonly used to model imperative features. "ref V" allocates a fresh memory location initialized to value V, "deref A" returns the contents of a memory location, and "assign A V" stores the value V in memory location A.

C variables are translated by binding each variable to a mutable reference containing the actual data. For example, a declaration "int x = 0;" in C may be translated to "let x = ref 0 in …", which could be given a type such as "$\mathbf{num}^{\phi}\ \mathbf{ref}^{\beta}$" in our system. Accesses to the contents of the variable require an extra dereference. When an expression is used as an l-value (for example, when the "address of" operator (&) is applied to a variable), the extra dereference is simply omitted.

For clarity, these extra references and dereferences have been omitted in the examples in this paper.

The address-of (&x), dereference (*p) and assignment (x = y) operators of C are translated directly to the “ref”, “deref” and “assign” operations. (Other operators are translated to function calls or primitive operations.)

5.4 Null Pointers

The “null” pointer in C is treated as a special value that has type “$\forall a.\ a$ **ref**”. Any assignment to or dereference of “null” is assumed to result in immediate program termination.

5.5 Uninitialized Data

All variables are treated as being initialized to default values. Default values are zero for scalars, “null” for pointers, and for compound types each component is set to its default value. This does not affect the inferred type information. (However, it should be clear that accidental dependencies based on uses of uninitialized data are not tracked.)

5.6 Declaration Order

In C, all global declarations are unordered and potentially mutually recursive. Even though recursion is relatively rarely used in C, it is always implicitly available. In contrast, the intermediate language imposes an explicit ordering on declarations in the form of nested “let x = … in …” and “letrec x = … y = … in …” constructs, where “letrec” defines a set of mutually recursive values. The ordering ensures that every declaration only refers to declarations from an enclosing block, or to declarations in the current “letrec” block. For example, here are some valid programs in the intermediate language:

```
let x = 5
  in let y = x + 3
    in y*y
```

```
letrec f(x) = if x = 0 then 1 else x*g(x-1)
       g(x) = if x = 0 then 1 else f(x-1)/x
in f(100)
```

The natural translation would make a C program one large “letrec” block. Unfortunately this would lead to no useful polymorphism, because the Hindley-Milner algorithm computes the types of all the values declared in a “letrec” before applying polymorphic generalization to any of them. In the example above, type inference computes the types of `f` and `g` assuming that each use of `f` and `g` has exactly the same type as the declaration (no polymorphism allowed). Then they are generalized to type schemes so they can be used polymorphically in the body.

One potential solution to this problem is to use a type system and inference algorithm with “polymorphic recursion” — the ability to polymorphically generalize values declared in “letrec” blocks before all their types are fully computed. These schemes are more complicated and harder to implement; in fact, type checking in the presence of polymorphic recursion is undecidable [22]. Also, when combined with the recursive types described above, principal types may not exist. This makes it harder to be sure that the tool will never reject compilable C programs.

Our solution is based on the observation that although all C functions are potentially mutually recursive, in practice recursion is uncommon. Therefore C declarations are translated into the intermediate language by computing an ordering on the declarations that satisfies their mutual dependencies. Cyclic dependencies due to recursion require the construction of a “letrec” block that declares the mutually recursive entities simultaneously. In terms of the dependence graph, the translator first identifies the cyclic components of the graph and replaces them with single nodes, which will be translated to “letrec” blocks. (The remaining nodes can be translated to simple “let” blocks.) A simple topological sort of the resulting acyclic graph determines the declaration order.

This solution is simple, but it has some disadvantages. Computing this ordering is an inherently global operation, and therefore limits the incrementality of the analysis, as discussed below. Furthermore, some programs exhibit undesirably large cycles of mutually recursive declarations, often in unexpected ways. (One example is when many functions call an error-handling function and the error-handling function itself can call many other functions to clean up state, some of which may encounter errors themselves.) Because this delays the polymorphic generalization of the functions involved, accuracy can be lost.

Here is an example of this part of the translation:

`void main() {`
`  grow(102);`
`}`
 	
`void shrink(int num) {`
`  print(num);`
`  grow(num/2);`
`}`
 	
`void print(int num) {`
`  printf(" %d\n" ,`
`num);`
`}`
 	
`void grow(int num) {`
`  print(num);`
`  shrink(num+42);`
`}`
	let
	  print = λ(num).
	    …
	in letrec
	    shrink = λ(num).
	      … print … grow …
	    grow = λ(num).
	      … print … shrink …
	  in let
	      main = λ(). …
	    in main()

The dependence graph is shown in Figure 3. The boxes indicate the declaration blocks. Type inference will first compute a type for `print`, then for `grow` and `shrink` together, and then for `main`. The practical result of this treatment is that, during the processing of `grow` and `shrink`, calls to `grow` and `shrink` are not treated context sensitively. (In this example, this leads to no loss of information, but in more realistic programs it could be a problem.)

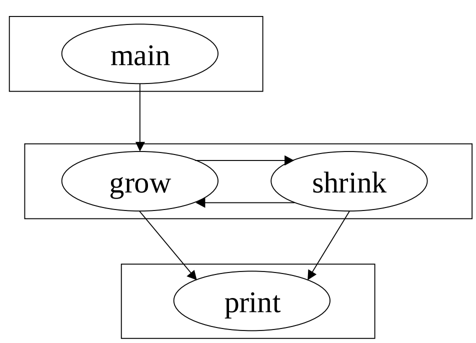

Figure 3: Declaration Dependence Graph

5.7 Arbitrary Control Flow Within A Function

Control flow within C functions can be rather complex, for example due to the "goto" statement. For internal clarity, the translation actually preserves this behaviour. Within a C function, each statement is translated to a single function in the intermediate language, whose specification is to compute "the rest" of the C function. Control transfers from one statement to another are translated to tail calls from one of these functions to another.

Note, however, that the type inference algorithm is actually insensitive to control flow. It only depends on the set of expressions that are evaluated within a function.

To simplify the system, only global functions are polymorphically generalized. In particular, the functions that are generated internally to represent intra-function control flow are treated monomorphically.

5.8 Non-local Control Flow

`setjmp` and `longjmp` are not supported directly. However, since the type inference algorithm is insensitive to control flow, it will not be affected by uses of `setjmp` and `longjmp`, except that the propagation of the result from `longjmp` to `setjmp` will not be tracked.

5.9 Data Lifetimes

In C, storage is usually explicitly deallocated by calling `free` or by returning from a function. This can lead to unsafe behavior due to dangling pointers. Our analysis simply assumes that all data persists until it is inaccessible.

5.10 Type Casts

Perhaps the most challenging aspect of C for the design of our tool is type casts. They cannot be included in the intermediate language, since the validity of the information in the types depends on there being no type casts. However, real programs use type casts and the tool must handle them. Therefore, it translates type casts into our intermediate language while preserving as much of their semantics as possible, especially the semantics that programmers frequently rely on. This is tricky because the semantics expected by programmers is complex, ill-defined and sometimes inconsistent.

The basic idea is to regard a value in the intermediate language (an I-value) as a map from C types to the value (the C-value) that the I-value has when viewed as that C type. For example, a C pointer might be represented as “[`<int>`: 181814; `<char *>`: ref (‘A’)]”. When an I-value is accesssed, the C type used to perform the access is used to select the appropriate C-value.

When an I-value is created or modified, all of the values of the map could be affected. The exact effects across different C types are compiler and machine dependent, so our tool (which is compiler and machine independent) cannot predict them. However, only the inferred types of the data are relevant — the exact values are not. Also, only a limited set of these effects are considered to be “standard”, and thus acceptable to rely on in good programming practice. Because all scalar C types translate to a single “number” type, casts between C types that differ only in the kinds of scalars used can be safely ignored.

Likewise “const”, “volatile” and other storage modifiers are simply dropped. Because the binary representations of pointers are not standardized, the tool assumes that there is no relationship between the C-value of an I-value when viewed as a scalar and its C-value when viewed as a pointer.

In fact, there only needs to be special handling for casts between C types whose memory layouts have a relationship specified by the C standard (and conventions). The primary example is C “structs”. The first field of a struct must be at the same memory location as the struct itself. Two structs of different type must have their fields laid out at the same memory locations, up to and including the first pair of fields that have different types. Thus the following code is valid:

```
struct S { int a; } x;
x.a = 5;
printf(“ %d” , *(int *)(&x)); /* prints 5 */
```

These assumptions about layout are converted to constraints that certain components of the values of a map must be equal. In this example, x would be “[<int>: 5; <struct S>: {5}]”, and a constraint forces the underlined components to always be equal.

Other than structs, similar rules apply to function argument lists and unions. In unions, every field has the same memory location as the union itself.

To infer types for these maps, each map is treated as a record with a field for each C type ever used in the program. For example, for the program above, the value of x would be expanded out to “[<int>: 5; <struct S>: {5}; <int *>: null; <struct S *>:

null]” (unused fields are given default values). The analysis infers a type for this record in the obvious way, as the product of the inferred types of each of the fields. The equality constraints that encode layout assumptions become type equality constraints on the types of the structure fields. In this example, the type is “[<int>: num^{α}; <struct S>: $(num^{\alpha})^{\chi}$; <int *>: $num^{\gamma}\ ref^{\beta}$; <struct S *>: $(num^{\gamma})^{\delta}\ ref^{\beta}$]”.

For non-trivial programs these records quickly become unmanageably large. Therefore the implementation is optimized, so that the types of fields corresponding to C types for which we have no information are not explicitly represented. In particular, there is almost no overhead for the record in the common case in which there is only one field with any information (i.e. the value is never subjected to a type cast).

5.11 Type-unsafe Linkage

C provides no guarantees that a declaration in one module and used in another module will have the same C type at the two sites. (This is often a problem when “function prototypes” are not used and the call of a function fails to match the function’s declaration in some way.) Lackwit detects such situations and automatically inserts an appropriate type cast at each mismatching site.

5.12 Other Unsafe Features

The other major unsafe feature of C that requires special handling is pointer arithmetic (array indexing is just syntactic sugar for pointer arithmetic). The analysis assumes that pointer arithmetic is not used to manipulate memory outside the bounds of an array (treating single values as arrays of size 1). Since every array element has the same type, type inference effectively ignores pointer arithmetic.

Some code violates this assumption. The main examples we have encountered are memory allocators (such as the implementation of "malloc"), functions that take variable numbers of arguments (and use pointer arithmetic to examine the stack directly), and certain pieces of code that use pointer arithmetic to traverse the fields of structs. Most of these examples are low-level functions that occur in libraries. For these, we manually specify the correct type signatures (often simply by providing an alternative "dummy" implementation to the analyzer).

5.13 Example

In this example, we include the extra level of dereferencing.

Calls such as "stmt5(*e*)" are just an idiom for expressing sequencing in this lambda-language; *e* is evaluated (usually for its side effects), and then the result, if any, is passed to stmt5, which simply ignores it and continues with the rest of the computation.

<table>
<tr>
<td>
<pre>
struct list {
  struct list * next;
  int val;
};

int sum(struct list * p) {
  int result = 0;
  while (p != NULL) {
    if (p->val < 0) {
      return result;
    } else {
      result += p->val;
    }
    p = p->next;
  }
  return result;
}
</pre>
</td>
<td>
let
 sum = λ(p).
 letrec
  result = ref(0)
  stmt1 = λ(). if ptr-not-equal(deref(p), NULL) then stmt2() else stmt6()
  stmt2 = λ(). if less-than(deref(#2(deref(p))), 0) then stmt3() else stmt4()
  stmt3 = λ(). deref(result)
  stmt4 = λ(). stmt5(assign(result, plus(deref(result), deref(#2(deref(p))))))
  stmt5 = λ(). stmt1(assign(p, deref(#1(deref(p)))))
  stmt6 = λ(). deref(result)
 in stmt1()
</td>
</tr>
</table>

6 Queries

Lackwit's function, as far as the user is concerned, is very simple. The user presents a C expression denoting a storage location — a global or local variable, a function parameter, perhaps with pointer dereferences and field suffixes — and a context in which it should be interpreted (eg, for a local variable, which function it appears in).

The implied query is then as follows. During execution values stored in this location will appear at other times in locations with different names. Where do these names appear in the program text? The user may choose how the answer is presented: either textually, grep-style, as a series of elementary expressions or statements with their line numbers, or graphically, as a diagram showing the relevant functions and global variables. This diagrammatic view additionally gives information about the relationship between functions and variables. Both forms of output indicate also whether the data in question is accessed or merely passed to another site. The next few sections explain how this information is obtained, and how the diagram is prepared from the results of the type inference.

6.1 Type Compatibility

As indicated by the examples in Section 2, Lackwit determines if two expressions can ever return the same data item by comparing the types it infers for the two expressions.

In the absence of polymorphism, two variables or expressions can return the same data only if they have the same type. In the presence of polymorphism, however, two variables may refer to the same data, but one may have a type that is an instantiation of the type of the other. For example, consider

```
void * f(void * x) { return x; }
void * g(void * y) { return f(y); }
```

`f` could have the type "$\forall a.\forall\beta.\forall\phi.(a\ \mathbf{ref}^{\beta}) \rightarrow^{\phi} a\ \mathbf{ref}^{\beta}$" and `g` the type "$\forall b.\forall\gamma.\forall\alpha.(b\ \mathbf{ref}^{\gamma}) \rightarrow^{\alpha} b\ \mathbf{ref}^{\gamma}$" (since quantified variables can be freely renamed). Thus the types of `x` and `y` are different, but they could actually be the same data item. The solution is to record the fact that when `f` was used in the body of `g`, *a* was instantiated to *b*, and that β was instantiated to γ. In fact, `x` can be a pointer to a data item of any type that is an instantiation of *a*, including values of type *b*.

In general, types T and U may refer to the same data item (are "compatible") if there is a third type V such that V is an instantiation of T and also an instantiation of U. (We define the instantiation relation as the reflexive transitive closure of the individual instantiation steps that occurred.)

Compatibility is a symmetric relation, but not transitive. Thus the structure of the results obtained with polymorphism is significantly more complex than for the monomorphic case (in which the variables and expressions are partitioned into equivalence classes according to their types).

For example, consider the following C program:

```
void copy(char * from, char * to) {
    *to = *from;
}
void copy5(char * fromarray, char * toarray) {
    int i;
    for (i = 0; i < 5; i++) {
        copy(from + i, to + i);
    }
}
void main(void) {
    char from1[5] = { ‘h’, ‘e’, ‘l’, ‘l’, ‘o’ };
    char to1[5];
    char from2[5] = { ‘k’, ‘i', ‘t’, ‘t’, ‘y’ };
    char to2[5];
    copy5(from1, to1);
    copy5(from2, to2);
```

```
}
```

Type inference will give us the following type signatures:[3]

`copy`	$\forall\alpha.\forall\beta.\forall\phi.(\mathrm{num}^{\alpha}\ \mathrm{ref}^{\beta}, \mathrm{num}^{\alpha}\ \mathrm{ref}^{\chi}) \rightarrow^{\phi} ()$
`copy5`	$\forall\delta.\forall\varphi.\forall\sigma.(\mathrm{num}^{\delta}\ \mathrm{ref}^{\varphi}, \mathrm{num}^{\delta}\ \mathrm{ref}^{\kappa}) \rightarrow^{\sigma} ()$
`main:from1`	$\mathrm{num}^{\theta}\ \mathrm{ref}^{\rho}$
`main:to1`	$\mathrm{num}^{\theta}\ \mathrm{ref}^{\eta}$
`main:from2`	$\mathrm{num}^{\mu}\ \mathrm{ref}^{\zeta}$
`main:to2`	$\mathrm{num}^{\mu}\ \mathrm{ref}^{\gamma}$

It also records the fact that α instantiated to δ, β instantiated to φ, χ instantiated to κ, δ instantiated to θ and μ, φ instantiated to ρ and ζ, and κ instantiated to η and γ.

Consider a query on `*from1`. The tool searches for occurrences of types compatible with num^{θ}. In this program, the types compatible with num^{θ} are num^{α} and num^{δ}; in both cases the "witness type" V is num^{θ}. In particular, num^{μ} is not compatible because there is no inferred type V that is an instantiation of both num^{μ} and num^{θ}. The tool will find matches for `*copy:from`, `*copy:to`, `*copy5:from`, `*copy5:to`, `*main:from1`, and `*main:to1`. Likewise, a query on `*from2` will exclude `*from1`, and since both queries include `copy:from`, this illustrates that compatibility is not transitive.

6.2 Components of Data Structures

Frequently, a compatible type may be found in the signature of a function, global variable or local variable not as the type of the entity itself, but as the type of one of its

[3] Here, as elsewhere, we ignore the extra "refs" required to express the mutability of C variables (see Section 5.3).

components. This leads to no special problems. Lackwit presents such results to the user by printing out an expression that denotes the component (in terms of the function, global variable or local variable).

6.3 Local Information

In large programs, there are frequently chains of function calls that pass around data structures as parameters without reading or writing them, just so that data can be accessed where it is needed. When those data structures are the subjects of queries, the functions may show up because they have parameters with component types matching the queried type. It is useful to be able to identify which functions access the data structures directly within their bodies, rather than only through other function calls. In a well-structured program, there will be a small number of these functions, corresponding to the operations defined on abstract data types.

For this reason, the type inference algorithm first solves the constraints arising just from the body of each function, ignoring any function calls it may make. These incomplete types (“local types”) are recorded, then any function calls present are resolved and the completed types (“global types”) are used for the rest of the analysis. The tool can determine the actions of a function body, ignoring the effects of function calls, by examining the local types of the data that it operates on. The important special case is that if the local type of a data item is a type variable, then the function body is oblivious to its type and therefore does not manipulate the data directly (though they may manipulate it indirectly through function calls).

During query processing, when Lackwit finds a type compatible with the type of the query expression, it checks the corresponding local type to see if the data is being directly manipulated, and if not, that particular result is ignored.

In the example above, the local types for `fromarray` and `toarray` are "a ref$^{\varphi}$" and "b ref$^{\kappa}$" respectively, because these pointers are never derefenced in the body of `copy5` and so the local types of their contents are unconstrained. Thus, the tool will not print the matches it finds for `*copy5:from` and `*copy5:to`, since they cannot represent actual accesses to the data.

6.4 "Call Graph" View

Lackwit can display the results of a query as a graph (see Figure 1). The nodes are the global functions (such as `veh_copy`) and variables (such as `map_manager_global`) whose type signatures (or type signatures of local variables) contain types compatible with the type of the query items. Each edge represents the textual use of one declaration by another (almost always a function call or a reference to a global variable). An edge is included if the type inferred for the definition contains a type compatible with the query type, and the type of the corresponding component of the use is also compatible with the query type. This eliminates some edges that cannot propagate the queried value. The shaded nodes are the functions (or global variable initializers) that can manipulate the queried data directly (as indicated by their local types). Thus we can observe that `map_mgr_convert_pixel_coords` passes along a reference to the vehicle object without accessing it directly.

This view is particularly useful for identifying the contexts in which operations may be called for a particular instance of an abstract data type, and how the instance may be passed around between the contexts.

7 Type Extensions

The basic type system allows us to determine where accesses occur, but the user often also wants to know what kind of accesses occur. For example, it is useful to be able to find modifications to part of a data structure (as distinguished from read-only accesses).

The inferred types can encode this information. Consider the problem of detecting modifications. We parameterize reference types to be read-only or read-write, by adding an extra tag as follows:

τ	::=	…	
	|	τ_1 $\mathbf{ref}^{\beta,x}$	
x	::=	read-only	(a read-only reference)
	|	read-write	(a read-write reference)
	|	β	(tag variable)

The type signature of the "assign" operator requires its reference argument to be read-write:

assign : $\tau\ \mathbf{ref}^{\beta,\text{read-write}} \rightarrow \tau \rightarrow \tau\ \mathbf{ref}^{\beta,\text{read-write}}$

There are no other constraints on the *x* tags. In particular, nothing constrains them to be "read-only", so a valid type assignment for the basic type system can be extended to a valid type assignment in the extended type system by setting all the *x* tags to "read-write". Thus, every compilable C program continues to be typable with the extended type system.

Because nothing constrains the *x* tags to be read-only, "read-only" will never show up in an inferred type. The *x* tags of inferred types will either be "read-write" or unbound tag variables. The presence of an unbound tag variable in the inferred type of a variable indicates that the data is never modified through that variable. The presence of an unbound tag variable in the inferred type of a function parameter indicates that the function does not modify the data through that parameter. If a "read-write" tag shows up, in general one can only conclude that the data *might* be modified.

Other properties can be tracked in the same way as "writability". For example, we can add a tag for "readability" and change the signature of the "deref" operator to require the readability tag to be "readable".

7.1 Query Extensions

An extension of the basic query allows the user to specify a predicate in propositional logic, where the literals are the names of different tags (e.g. "readable", "writable"). The output is filtered to include only the accesses whose local types have tags satisfying the predicate. Thus, if the user specifies "writeable", Lackwit will suppress accesses to the data that it determines are read-only.

7.2 “Tag Search” Queries

We support another kind of query that, given a predicate, will find all global types with tags satisfying the predicate. The tool will output the names and contexts of the variables that have the matching types. One application of this query is to detect memory leaks, as described in the Results section below.

If the predicate contains negations, complications may arise due to interactions with polymorphism. For example, suppose that we want to search for data that is never read (perhaps so it can be optimized away). The predicate to search for is simply “¬readable”. But consider the following program:

```
void f(char * ptr) {
    *ptr = ’x’;
}
void g(void) {
    char * buf = (char *)malloc(1);
    f(buf);
    printf(“ %c” , *buf);
}
```

The contents of `ptr` are not read in `f`, and therefore the type of `ptr` has no “readable” tag. However, the contents are read in `g`, so the type of `buf` does have the “readable” tag; this tag is not propagated to `ptr` because the type of `ptr` is polymorphic, and so the type of `buf` is not the same as it, but an instantiation of it. Thus we can have a false match for `ptr`. The same problem arises if we are dealing with an incomplete program, and `g` does not exist but may possibly be added later.

The problem can be avoided by marking types as “escaping” whenever they appear in the signatures of globally visible declarations such as `f`. When the query predicate contains negations, only the non-escaping types are checked against the predicate.

8 Tool Implementation

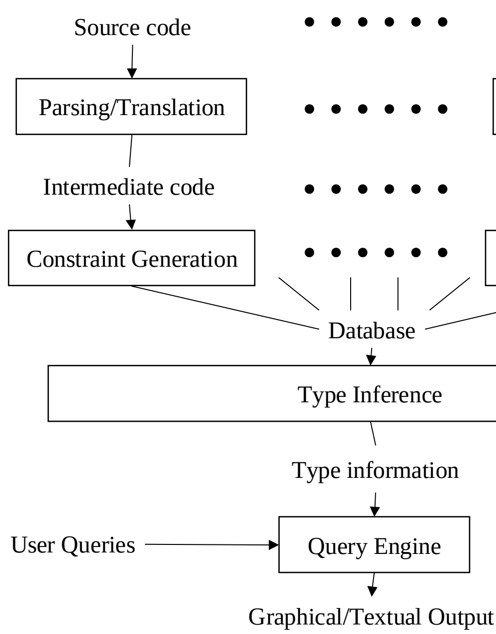

Figure 4: Tool Architecture

Each preprocessed C source file is fed into a tool that parses the code, generates intermediate code, and generates type constraints based on the type rules. The constraints are stored in a database, which is updated incrementally as new source files are added or existing source files change. To perform the analysis, another tool gathers up all the constraints and solves them, producing results in the form of inferred type information. It can then process a series of queries from the user.

8.1 Program Database

The database contains information about global declarations and about C types, organised by source file.

The C type information is a direct encoding of the definition of each C type used in each source file. Intra-file dependencies are resolved, but inter-file dependencies (such as struct tags used by multiple source files) are left unresolved, to simplify incremental updates.

With each global declaration is recorded its name, its C type, the names of its local variables, their C types, the names of the global declarations that it depends on, the C types that it expects for them, whether it is “static”, and which source file it came from. There is also a set of type constraints that must hold between the inferred types of the global declaration, its local variables, and the global declarations that it depends on. With each type constraint is the file and line number of the original source code that gave rise to the constraint. To simplify incremental updates, each global declaration record is entirely independent, even for global declarations from the same source file.

8.2 Source Processing

Each source file is parsed by an LL(k) parser generated using the PCCTS toolkit. The tool accepts a superset of ANSI C, including almost all GNU language extensions. The tool performs a second pass to generate intermediate code, and then a third pass, applying the type rules to generate type constraints. Before the constraints are written to the database, a constraint solver simplifies them as much as possible; the constraints for each global declaration are simplified independently. The simplification step reduced the size of our databases by approximately a factor of four.

Source files are processed completely independently (by design, updating the database incrementally is trivial), and therefore the performance of this phase scales linearly in the number of source files. Replacing the LL(k) parser with a conventional LALR(1) parser would provide a significant speedup, especially for small programs.

Along with the source code for the program to be analyzed, the tool processes a file of dummy declarations for the standard C library functions. The dummy declarations do not implement all the behavior of the C library, but they implement enough of it to generate the same inferred types.

Type information for the standard C operators is also loaded into the database before analysis begins.

8.3 Analysis

The first step of the analysis tool is to combine the C type declarations from all source files and partition them into equivalence classes according to structural equivalence. This enables the tool to efficiently determine the structural equivalence of C types, which is useful for handling C type casts in subsequent stages. In this step, the tool also resolves any inter-file dependencies of one type declaration on another.

The second step is to scan the database to build the set of global declarations and the graph of dependencies between them. In this pass, the tool produces warnings for any duplicate or missing declarations. For duplicate declarations, the one most recently added is used. If there is a missing function, then the tool assumes it has an empty body. Missing global variables are assumed to be initialized to default values.

Then, as described above (in Section 5.6), the dependence graph is traversed to compute a global ordering of the declarations and to identify mutually recursive declarations. The type inference itself is also carried out during this pass.

8.4 Processing Queries

The user enters queries at the keyboard and they are processed as described in the previous section. One significant optimization is introduced to avoid having to scan the entire program for compatible types. Suppose that type T is found in the definitions of two globals X and Y. Then T must appear in the type of X, or it must appear in the type of some global that X depends on — otherwise there is no way that the occurrence of T in X could have been constrained to be the same as the occurrence of T in Y. Thus, one can search for occurrences of T by traversing the global dependence graph (treated as an undirected graph), traversing an edge only if the type of the depended-on global contains an occurrence of T. When searching for “compatible” types, an edge is traversed whenever the type of the definition and the type of the use have matching components with types compatible with T. The context specified in the query determines the global definition at which the search starts.

8.5 Producing Graphs

The tool creates files describing graphs that can be read by the graph visualization tools *daVinci* and *dot*. The queries are processed as normal, but as global declarations are traversed, nodes and edges are added to the graph. When the query is complete, the graph is written out in the desired format. *daVinci* actually runs as a subprocess to display the graphs interactively, but we have not yet added any special interactive features.

9 Results

In this section we discuss the costs of actually running the tool, some measures of its accuracy, and then give a more detailed analysis of the use of Lackwit for a real-world task. We also discuss less obvious applications of Lackwit, such as memory leak detection.

9.1 Performance

Table 2 shows how long it takes to analyze four programs:

Program	Source (kloc)	Database (MB)	Build (s)	Analysis (s)
Lislook	13	6	11	4
Morphin	17	4	17	3
Marshall	48	19	36	8
Linux	157	194	339	359

- **Morphin**, a real-time robot navigation system
- **Lislook**, a command-line utility for accessing remote library databases
- **Marshall**, an interprocess communications package
- **Linux**, the Linux 2.0 kernel (Red Hat 4.0, configured with the full set of network protocols and filesystem drivers, but no hardware device drivers)

These results were gathered on a 266MHz Pentium II PC with 64MB of RAM, running Windows NT. [Note to reviewers: this machine has more memory than the test machine we used for the ICSE paper, which is why the results have improved so dramatically.] The size of the source code was measured before preprocessing. The analysis time does not include the time spent running the C preprocessor. The query time is for one query, chosen to produce graphs with many nodes. The times were measured over five runs, and the standard deviation of each time was less than 8% of the corresponding mean (except for the means of less than ten seconds).

Overall, the results are excellent. The Linux kernel can be analyzed from source in less than fifteen minutes on a reasonable PC.

Recall that when building the database, each source file is processed completely independently. Therefore, the super-linear increase in the size of the database (and the time required to build it) indicates that the average amount of information stored per line of code is increasing — that is, the complexity of the code is increasing. This is largely due to increases in the average number of modules imported by a module (i.e. number of

“.h” files included), and increases in the complexity of those interfaces, especially in the complexity of the data structures exposed through them.

The worst-case time complexity of the Hindley-Milner algorithm applied to the lambda-calculus intermediate language is known to be doubly exponential in the size of the program. In practice, however, ML programmers have found it to be nearly linear. More precisely, the number of unification and generalization operations carried out is linear in the size of the program, and the time required for each such operation is (very nearly) linear in the size of the types being operated on (in our implementation, no occurs check is required). The empirical observation that explains the efficiency of ML type inference, and the efficiency and scalability of our analysis too, is that *the types are small in practice*. When the size of the types is bounded, the analysis is, in fact, almost *linear* in the size of the program.

Some of the query times are long compared to the basic analysis times. These are queries that touch many declarations and return large graphs. These queries can require a lot of processing, partly just to generate and format the output, but also because many large data structures representing types need to be scanned, often multiple times. This part of the tool requires further work.

9.2 Accuracy

It is difficult to measure the general accuracy of the tool, because the space of useful input queries is difficult to characterize. We constructed a large set of sample input queries for Morphin using the following approach:

Consider each local variable, parameter and global variable:

- If the variable is a scalar or union, query on its value.
- If the variable is a struct, query on its value and also recursively consider all of its fields.
- If the variable is a pointer or array, query on the value, and query on the value of its contents. If the contents are a struct, also query on the values of all of the fields (if a field is itself a struct, all its fields are queried, and so on).

The counts of different kinds of queries in the set are shown in Table 3.

Pointers and arrays	1187 queries
Structs and unions	557 queries
Scalars	2559 queries
Function pointers	1 query

Table 3: Counts of Queries By Kind

The single query on a function pointer yields a graph with 164 nodes (global variables and functions), of which only 3 are highlighted as “accessor” nodes; for function pointers, an access is either a function call, or a function definition that may be referenced by the function pointer. Morphin has 305 function definitions and 111 global variables, so (excluding library functions) a graph can have no more than 416 nodes.

Figure 5 shows how large the graphs are for pointer queries. Over 70% of queries on pointers yield graphs with more than 150 nodes. However, more than 90% of the queries yield graphs with less than 30 highlighted “accessor” nodes, so most displays should be manageable.

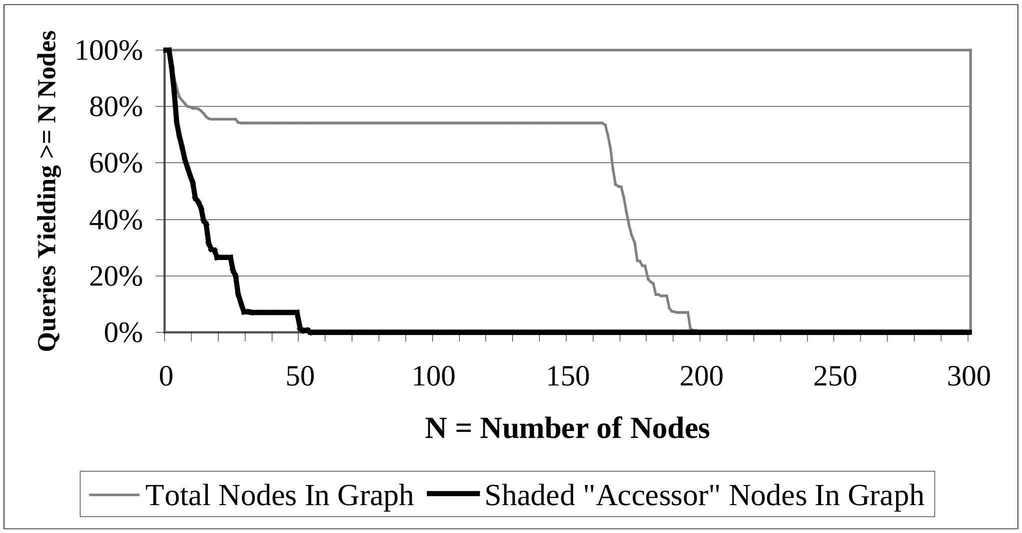

Figure 5: Graph Sizes for Pointer Queries

The high fraction of queries giving large graphs reflects the fact that almost any data is reachable from any point in the program by traversing the right links of pointer-based data structures. This presents no particular problems as long as the accessor nodes are highlighted or otherwise easily locatable.

Figure 6 shows the analogous information for the queries on C structs and unions. The results are very similar to the results for pointers.

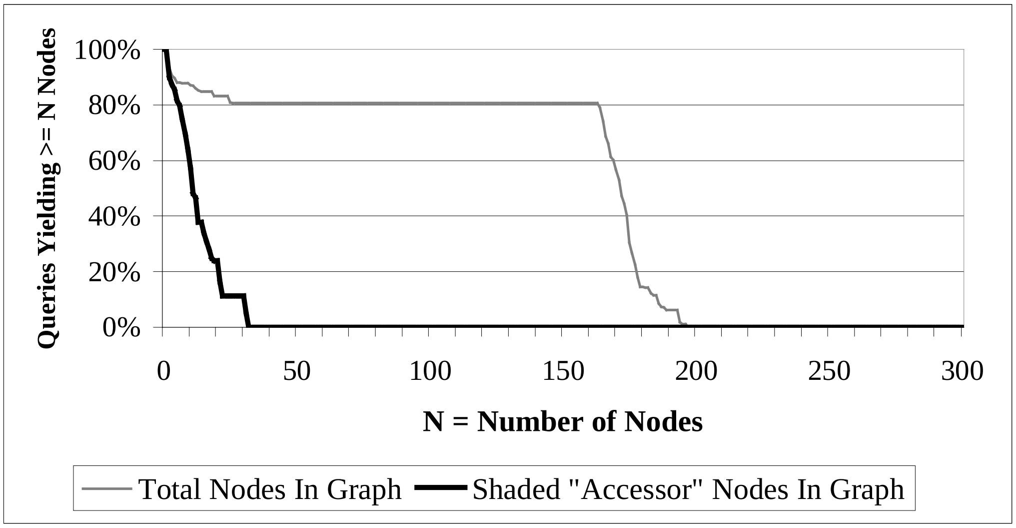

Figure 6: Graph Sizes for Struct Queries

Figure 7 shows the results for queries on scalar values. Clearly, most of the queries will not return useful information. The main problem here is that arithmetic operations on scalar values usually force the operands and results to have the same types, and therefore not many type distinctions can be made. However, queries on values that do not participate in arithmetic, such as Unix file descriptors, may still be effective.

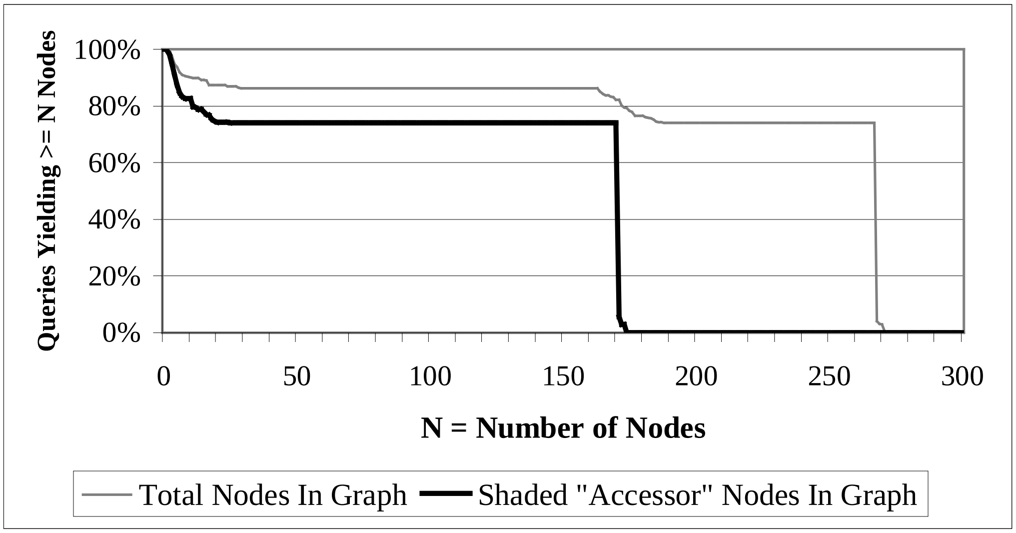

Figure 7: Graph Sizes for Scalar Queries

9.3 Case Study

A client asked for information about Morphin, a robot control program of moderate size (17,000 lines of code). In particular, there is a vehicle object which contains two queue objects, `sque` and `cque`, and our client was investigating the possibility of combining the two queue objects into one.

We performed a query on each queue object to discover which operations were being performed, and in what contexts. The two graphs both contained 171 nodes, but in each graph only five nodes were highlighted as “accessor” nodes — the `queue_` functions for the queue abstract data type. In particular:

- `cque` and `sque` could be accessed by `queue_create`, called by `veh_create` or `queue_copy`

- `cque` and `sque` could be accessed by `queue_copy`, called by `veh_copy`
- `cque` and `sque` could be accessed by `queue_destroy`, called by `veh_destroy` or `veh_free_queues`
- `cque` could be accessed by `queue_put_element`, called by `veh_put_command`
- `cque` could be accessed by `queue_get_prev_element`, called by `veh_get_old_command`
- `sque` could be accessed by `queue_put_element`, called by `veh_put_state`
- `sque` could be accessed by `queue_get_element`, called by `veh_get_old_mast_body_trans`

With this information, it was relatively easy for our client to determine the purpose of the two queues and to identify the code that needed to be changed.

The graphs themselves are illegible in this format, but fr interested readers they are reproduced in Appendix C.

In this example, a simple grep for `sque` or `cque` would have returned false matches with the queue fields of other objects, and would have missed references to the queues through parameters when they are passed into functions. Also, a semantic analysis that lacked context-sensitivity would have been unable to distinguish `sque` and `cque`, because they both are passed as arguments to the same queue functions. In fact, because `queue_create` calls `malloc` once and is called once for each queue, even analyses that are context sensitive, but distinguish heap objects by labelling them with the call site at which they were malloc'ed, would also fail [23] [24].

9.4 Memory Leak Detection

We experimented with using Lackwit to detect memory leaks, using the type system extensions presented above. The basic idea is to add two tags to every pointer type: "alloc", for pointers to dynamically allocated memory, and "dealloc", for pointers to dynamically freed memory. The signatures for `malloc` and `free` (and other memory management functions) are adjusted so that `malloc` returns "alloc"-tagged values and `free` takes "dealloc"-tagged parameters. Then we perform a query to locate all pointers with that are tagged with "alloc" but not tagged with "dealloc". Lackwit guarantees that any such pointer is not passed to `free`, and is probably obtained as a result of `malloc` (since mixing pointers to dynamic and static memory is usually avoided in well-written C programs), and therefore probably indicates a memory leak.

Applying this query to Morphin, Lackwit reported that six global variables referred to dynamic data structures that were never freed. (These did not cause problems in practice, since they were freed by the operating system when the process terminated, but they were a symptom of poor programming style.) It also discovered two fields of data structures that were pointers to memory that was never freed. These were a genuine problem, because these pointers were updated in a frequently executed loop.

Suspect pointers identified by this query are not necessarily memory leaks. [Note to reviewers: this corrects an error in our conference paper, where we asserted that suspect pointers were necessarily memory leaks.] Also, because of the control-insensitive nature of the analysis, many memory leaks will not be identified by the query. In our example all of the warnings indicated real errors, but if the number of false alarms becomes a problem, they can be eliminated by performing further queries to check the results. In particular, for each suspected leak, we perform a basic “value flow” query on the return value from the `malloc` that allocates the memory. If the value is shown to never reach `free`, then we can be sure that if the malloc is ever executed, its result can never be freed. Here is an example of a program that will give a spurious warning that can be eliminated using this extra query:

```
char static_buf[1];
char * my_malloc(int really_alloc) {
    if (really_alloc) {
        return (char *)malloc(1);
    } else {
        return static_buf;
    }
}
void g(void) {
    char * ptr = my_malloc(1);
    free(ptr);
}
void f(void) {
    char * ptr = my_malloc(0);
}
```

9.5 Unread Data Detection

Using a similar approach, we searched for data that was never read. Lackwit reported nine global variables and ten local variables that were never read. In addition, it reported that six local variables were structures containing some fields that were never read.

10 Related Work

Our basic analysis technique is similar to “region inference”, used by Tofte and Taplin [15] to improve the space efficiency of implementations of functional languages. The store is partitioned into distinct “regions”, and each value is associated with a region, in the same way that our analysis associates values with representation types; however, we have no analogue to their approximation of the side effects of functions. To our knowledge, we are the first to use these techniques for program understanding.

Other researchers have been investigating type inference methods for inferring properties of C programs. In [14] Steensgaard presents a method based on type inference that yields an almost-linear time “points-to” analysis. That algorithm is monomorphic (context-insensitive) and does not distinguish e/lements of compound structures, but variants have been constructed that overcome these limitations.

Wright and Cartwright [17] use polymorphic type inference to analyze Scheme programs. Their system is similar to ours because they infer all type information, without relying on any declarations, and they infer types in a richer type system than the language itself provides. They were also among the first to apply type inference with the explicit goal of addressing program understanding tasks. Our extension with vectors of attributes is a simple case of their unions of “prime types”. Unlike us, they do not try to distinguish different occurrences of the same type constructor.

Bowdidge and Griswold’s “star diagram” tool aids in encapsulating abstract data types [1]. They assume that there is a single global variable to be abstracted, but they discuss extending their method to operate on data structures with multiple instances. They consider operating on all data structures of a certain type, but comment “The potential shortcoming of this approach is that two data structures of the same representation type, particularly two arrays, might be used for sufficiently different purposes that they are not really instances of the same type abstraction”. Our method provides an answer to this problem.

Muller et al. [11] have proposed a reverse engineering technique in which first a static analysis is performed, and then the graphical output is visualized and manipulated by the user with the help of various automatic tools, to reveal and impose structure. Our analysis is more powerful than that incorporated in their Rigi tool, but we would certainly benefit greatly from such visualization and manipulation techniques.

LCLint [5] is a tool that finds inconsistencies between C programs and simple specifications. There is some overlap between the properties they are able to check and ours (for example, some abstraction violations and unused data), but their methods cannot simultaneously distinguish different instances of the same C type and handle complex data structures. On the other hand, their checks incorporate more information, such as flow-sensitive dataflow analysis, so they will catch many errors that our analysis cannot. The two tools are complementary.

Murphy and Notkin have developed some lexical analyses that are particularly efficient and easy to customize [10]. Due to its lexical nature, their tool can be more flexible (for example, it can analyze non-compilable programs, and non-C programs), and will be more efficient in most cases. Its strength is also its weakness. By operating purely at the lexical level, it cannot address semantic queries with the precision or soundness of Lackwit.

The same researchers' Reflection Model Tool (RMT) [18] allows the results of a static analysis to be presented at a more abstract level than the code, such as an architecture diagram, and to be compared to the expectations that the user has for that level. It assumes that the result of the source code analysis is a graph, and produces diagrams to show how the abstracted graph differs from that expected. RMT is independent of the tool used to analyze the source code, and Lackwit could be used in this role.

11 Conclusions And Future Work

Our tool is able to handle programs with hundreds of thousands of lines of code while consuming only modest computational resources. We have demonstrated its utility for several different tasks. Based on these results, we conclude that, for many purposes, semantics-based whole-program analysis is feasible even for large software systems.

To achieve true scalability, we had to address potential bottlenecks at every stage of the system. We were surprised to discover that in some cases, even the extraction of query results from inferred types could be the critical bottleneck. In future work, we would like to use database techniques such as indexing to control this problem.

The tool uses an intermediate language, mainly to make it easier to implement and explain. However, the translation step can make the processing of source code much slower than it could be. In a production system, this translation step would be eliminated and the type constraints would be derived directly from the C parse tree.

The graph visualization tools had trouble displaying graphs with large numbers of nodes (more than 100). Rather than just improving layout techniques, we would like to investigate different ways to display the information, such as using 3D graphics or providing interactive navigation or filtering.

Some applications may require more accuracy than our type inference scheme can provide. The type inference itself can be enhanced to improve accuracy. For example, if we allow the internally generated functions corresponding to individual statements to be polymorphically generalized, then certain kinds of inaccuracy due to control-flow merges can be avoided.

Other authors have shown how to use inferred type information to partition a program in preparation for further analysis [reference]. Likewise, one could layer a more expensive analysis, such as a traditional dataflow analysis, on top of Lackwit to obtain more precise information for selected variables. Previous work has shown how using this approach to limit the scope of the expensive analysis can improve overall performance. In fact, if polymorphic type inference is used, we believe that this approach could actually improve the **accuracy** of many of the expensive analyses as well. Many of them use arbitrary "k-limits" to bound context information or alias paths, and since polymorphic type inference does not have any such limits, it could sometimes produce information that k-limited analyses alone would not.

It is challenging to extend our kind of polymorphic type system to object-oriented languages that include subtyping. Seamless, efficient handling of pointer aliasing and higher-order control flow would be especially useful in such settings, and we intend to make this problem the focus of our future research.

The Lackwit prototype is publically available; visit the URL `http://www.cs.cmu.edu/~roc/Lackwit.html` for details.

12 Acknowledgements

We would like to thank Robert Harper , Lars Birkedal and Matthias Felleisen for their helpful advice and comments on previous drafts of this paper. We are very grateful to Reid Simmons for taking the time to explain his needs to us, and for allowing us to use his code. We would also like to thank the members of the CMU Software Group for their help and feedback.

This research was sponsored in part by a Research Initiation Award from the National Science Foundation (NSF), under grant CCR-9308726, by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force

Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA), under grant F33615-93-1-1330.

The views and conclusions contained in this document are those of the author[s] and should not be interpreted as representing the official policies, either expressed or implied, of NSF, TRW, Wright Laboratory, ARPA, or the U.S. government.

13 References

1. R.W. Bowdidge and W.G. Griswold, “Automated Support for Encapsulating Abstract Data Types,”. *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, New Orleans, December 1994.

2. F. Cardone and M. Coppo, “Type Inference with Recursive Types: Syntax and Semantics,” *Information and Computation*, 1992, number 1, pp. 48-80.

3. L. Damas and R. Milner, “Principal Type Schemes for Functional Programs,” *Proceedings of the Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1982, pp. 207-212.

4. M.D. Ernst, “Practical Fine-Grained Static Slicing of Optimized Code,” Technical Report MSR-TR-94-14, Microsoft Research, Microsoft Corporation, Redmond, July 1994. <ftp://ftp.research.microsoft.com/pub/tech-reports/Summer94/TR-94-14.ps>

5. D. Evans et al., “LCLint: a Tool for Using Specifications to Check Code,” *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, New Orleans,

December 1994.

6. E. Gansner, S. North and K. Vo, “DAG — a Graph Drawing Program,” *Software Practice and Experience,* volume 18, number 11, November 1988, pp. 1055-1063.

7. A. Hume, “A Tale of Two Greps,” *Software Practice and Experience*, volume 18, number 11, November 1988, pp. 1063-1072.

8. D. Jackson and E. Rollins, “Abstractions of Program Dependencies for Reverse Engineering,” *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering,* New Orleans, December 1994.

9. R. Milner, “A Theory of Type Polymorphism in Programming,” *Journal of Computer and System Sciences*, 17:348-375, 1978.

10. G. Murphy and D. Notkin, “Lightweight Source Model Extraction,” *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, 1995.

11. H. Müller et al., “A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models,” SIGSOFT '92*: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88-98, December 1992. In *ACM Software Engineering Notes*, 17(5).

12. R. O’Callahan and D. Jackson, “Practical Program Understanding with Type Inference,” Technical Report CMU-CS-96-130, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1996.

<ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-130.ps>

13. T. Parr, H. Dietz, and W. Cohen, “PCCTS Reference Manual (version 1.00)”, *ACM SIGPLAN Notices*, February 1992, pp. 88-165.

14. B. Steensgaard, “Points-to Analysis in Almost Linear Time,” *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

15. M. Tofte and J.-P. Taplin, “Implementation of the Typed Call-By-Value λ-Calculus Using a Stack of Regions,” *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1994, pp. 188-201.

16. A. Wright, “Simple Imperative Polymorphism,” *Lisp and Symbolic Computation*, volume 8, number 4, December 1995, pp. 343-356.

17. A. Wright and R. Cartwright, “A Practical Soft Type System for Scheme,” *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, June 1994.

18. G. Murphy and D. Notkin, “Software Reflexion Models: Bridging the Gap between Source and High-Level Models,” *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, 1995.

19. Y.-F. Chen, M. Nishimoto, and C. Ramamoorthy, "The C Information Abstraction System,” *IEEE Transactions on Software Engineering*, 16(3), pages 325-334, March

1990.

20. “Automated Scalable Correction of Code and Data for Year 2000 Compatability,” Technology Whitepaper, Peritus Software Services Inc. <http://www.peritus.com/pdf/wp-say2k.pdf>

21. R. Harper, “A Simplified Account of Polymorphic References,” Technical Report CMU-CS-93-169, , School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1996.

22. F. Henglein, “Type Inference with Polymorphic Recursion,” ACM Transactions on Programming Languages and Systems, Vol 15, No. 2, pages 253-289, April 1993.

23. R. Wilson and M. Lam, “Efficient Context-Sensitive Pointer Analysis for C Programs,” *Proceedings of the ACM SIGPLAN’95 Conference on Programming Language Design and Implementation*, June, 1995.

24. P. Stocks, B. Ryder, and W. Landi, "Comparing Flow- and Context-sensitivity on the Modification-side-effects Problem", Department of Computer Science, Rutgers University, Number DCS-TR-335, August, 1997.

Appendix A: A Primer on Type Inference

Although type inference is based on a sophisticated scheme (originally developed for functional languages, and carefully polished by type theorists), the basic intuition is extremely simple. Suppose we wish to determine which variables have the same type as some variable *x*. Certainly if we assign *x* to another variable *y*, then *y* must be included. Equally, if *a* is the first parameter of a function, and a call to that function appears with *y* as first argument, *a* must also be included. By giving constants and operators known types, we may also be able to determine something about the type of *x*, such as that it is a number.

This analysis is of course only approximate; if the assignment and the function call happen to be in dead code, there will be no execution in which a value flows from *x* to *y* and *a*. But it is conservative: if there is no sharing of types, then there is no such assignment or function call, and therefore no flow.

The assumption that an argument of a function must share the type of the corresponding parameter can be expressed as an inference rule:

$$\frac{x : A,\ y : B,\ f : (A, B) \rightarrow C}{f\,(x, y) : C}$$

This rule may be read as follows: if *f* is a function that takes an argument of type *A* and an argument of type *B*, and returns a result of type *C*, and *x* has type *A* and *y* has type *B*, then *f(x, y)* is well typed and has type *C*.

A similar rule can be used to express the type constraints for an assignment statement:

$$\frac{x : t,\ y : t}{x = y : \mathrm{unit}}$$

In a functional language, every expression has a type, and there is thus no need to separate the assertion that an expression is well typed from the assertion that it has a particular type. Adapting this formalism to imperative languages requires the slightly artificial introduction of a type for an assignment statement, so that we can retain uniformity and avoid a special assertion that a statement is well typed, but has no type.

A set of inference rules such as these constitute a proof system. Proofs, in this context, are demonstrations that certain expressions have certain types. As in any logical inference system, the rules in themselves imply no method by which the proof is constructed. In the simplest interpretation, all expressions are explicitly typed. The inference rules are then checks that are performed to determine whether the program contains type errors. The explicit declarations here play the role of a detailed proof, and checking amounts to finding an appropriate inference rule with which to check each step.

A different and more interesting interpretation is also possible, however, in which the rules play a more operational role. Rather than viewing a rule as a procedure for checking a program construct's well-formedness, we view it as a guide for generating constraints from which types, undeclared, will be determined. From the assignment rule, for example, we generate the constraint that x and y have the same type t, whatever that might be. The result of analyzing an entire program in this fashion is a set of constraints with many solutions. Our notion of type has nothing corresponding to negation, so we can satisfy an inference rule by giving almost every entity the same type: the function application rule, for example, is trivially satisfied by setting $A = B = C$. On the other hand, we can make as many distinctions as possible, and only equate types when necessary. This is usually the solution we want. In conventional compiler analysis, an algorithm based on unification known as Algorithm W [cite] has been widely applied; it is used, for example, in the analysis of Standard ML programs. Lackwit uses W with only minor modification.

This type inference method can be viewed as performing a kind of flow-insensitive dataflow analysis. The dataflow problem is, roughly, to partition the variables of the program as finely as possible so that two variables belong to the same equivalence class only if there is a dataflow path from one to the other. A dataflow algorithm would join two classes of the partition when a flow has been found between members of the two classes; likewise, algorithm W will merge two classes implicitly by unifying the variables representing their types.

Nevertheless, dataflow analysis and type inference should not be confused. Type inference is not flow-sensitive, and, unlike dataflow analysis, is not open to a collection of ad hoc refinements. On the other hand, it extends naturally to component-wise analysis of structures, and handles higher-order features, pointers and recursion with elegance and power. With the addition of polymorphism, type inference becomes context sensitive. Most importantly, however, type inference is inherently modular, and is thus an ideal foundation for the analysis of large systems in which the ready availability of all source code cannot reasonably be assumed.

More About Type Rules

In the rules below, we make use of "type environments" such as "TE". A type environment is a (partial) map from variables to their types. For example, consider the rule for function declaration:

$$\frac{\mathrm{TE} \oplus \{x \to \tau'\} \bullet e_1{:}\tau}{\mathrm{TE} \bullet \lambda x.e_1{:}\tau' \to^{\beta} \tau} \qquad \text{(Lambda abstraction)}$$

This can be read as follows:

If TE, and the assumption that x has type τ', together justify assigning the type τ to the expression "e_1", then the type environment TE justifies assigning the type "$\tau' \to^{\beta} \tau$" to the expression "$\lambda x.e_1$".

Free variables such as β are implicitly universally quantified.

Appendix B: The Intermediate Language and Type Rules

Here is the syntax of the intermediate language. x stands for variable names.

e	:=	x	(variable reference)
	|	$\lambda x.e_1$	(function introduction)
	|	$e_1 e_2$	(function call)
	|	$primop(e_1, e_2, \ldots, e_n)$	(primitive operation)
	|	if e_1 then e_2 else e_3	(conditional)
	|	let x = e_1 in e_2	(non-recursive declaration)
	|	letrec $x_1 = e_1$, $x_2 = e_2$, … , $x_n = e_n$ in e	(mutually recursive declarations)

primop	:=	ref | deref | := | NULL
	|	mk-tuple_n | $\text{elem-tuple}_{n,k}$
	|	n | + | * | - | / | to-int | < | =

$| \quad <_{ptr} \mid =_{ptr} \mid +_{ptr} \mid -_{ptr}$

Here are the type rules that must be satisfied. They are given as inference rules.

$$\frac{x \in \text{dom TE}}{\text{TE} \bullet x\text{:Inst(TE(x))}} \qquad \text{(Variable access)}$$

"Inst" maps type schemes to types by substituting fresh variables for the universally quantified variables.

$$\frac{\text{TE} \oplus \{x \rightarrow \tau'\} \bullet e_1\text{:}\tau}{\text{TE} \bullet \lambda x.e_1\text{:}\tau' \rightarrow^{\beta} \tau} \qquad \text{(Lambda abstraction)}$$

$$\frac{\text{TE} \bullet e_1\text{:}\tau' \rightarrow^{\beta} \tau \qquad \text{TE} \bullet e_2\text{:}\tau'}{\text{TE} \bullet e_1\, e_2\text{:}\tau} \qquad \text{(Function application)}$$

$$\frac{\text{TE} \bullet e_1\text{:}\tau' \qquad \text{TE} \bullet e_2\text{:}\tau \qquad \text{TE} \bullet e_3\text{:}\tau}{\text{TE} \bullet \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3\text{:}\tau} \qquad \text{(If)}$$

$$\frac{\text{TE} \bullet e_1\text{:}\tau_1 \qquad \text{TE} \oplus \{x \rightarrow \text{Close(TE}, e_1, \tau_1)\} \bullet e_2\text{:}\tau}{\text{TE} \bullet \textbf{let } x = e_1 \textbf{ in } e_2\text{:}\tau} \qquad \text{(Let binding)}$$

"Close" is the generalization function that converts types to type schemes. Close(TE, e, τ) $= \tau$ if e is not of the form $\lambda x.e_1$. Otherwise, it returns $\forall a_1.\forall a_2. \ldots \forall a_n.\tau$, where the a_i are the type or tag variables of τ that are not free in TE.

$$\frac{\begin{array}{c}\text{TE'} = \text{TE} \oplus \{x_1 \rightarrow \tau_1\} \oplus \ldots \oplus \{x_n \rightarrow \tau_n\} \\ \text{TE'} \bullet e_1\text{:}\tau_1 \quad \ldots \quad \text{TE'} \bullet e_n\text{:}\tau_n \\ \text{TE} \oplus \{x_1 \rightarrow \text{Close(TE}, e_1, \tau_1)\} \oplus \ldots \oplus \{x_n \rightarrow \text{Close(TE}, e_n, \tau_n)\} \bullet e\text{:}\tau\end{array}}{\text{TE} \bullet \textbf{letrec } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } \text{e:}\tau} \qquad \text{(Recursive binding)}$$

$$\frac{\begin{array}{c}\text{TE} \bullet e_1\text{:}\tau_1 \quad \ldots \quad \text{TE} \bullet e_n\text{:}\tau_n \\ \text{Inst(Primtype(}primop\text{))} = (\tau_1, \ldots, \tau_n) \rightarrow^{\beta} \tau\end{array}}{\text{TE} \bullet primop(\text{e}_1, \ldots, \text{e}_n)\text{:}\tau} \qquad \text{(Primitive operation)}$$

"Primtype" assigns types to the primitive operations as follows: (all variables are universally quantified)

primop	Primtype(*primop*)
ref	$(a) \rightarrow^{\alpha} a\ \text{ref}^{\beta}$
deref	$(a\ \text{ref}^{\beta}) \rightarrow^{\alpha} a$
:=	$(a\ \text{ref}^{\beta}, a) \rightarrow^{\alpha} ()$
NULL	$() \rightarrow^{\alpha} a\ \text{ref}^{\beta}$
mk-tuple$_n$	$(a_1, \ldots, a_n) \rightarrow^{\alpha} (a_1, \ldots, a_n)$

elem-tuple$_{n,k}$	$((a_1, \ldots, a_n)) \to^{\alpha} a_k$
n	$() \to^{\alpha} \text{num}^{\beta}$
+, *, -, /	$(\text{num}^{\beta}, \text{num}^{\beta}) \to^{\alpha} \text{num}^{\beta}$
to-int	$\text{num}^{\beta} \to^{\alpha} \text{num}^{\beta}$
<, =	$(\text{num}^{\beta}, \text{num}^{\beta}) \to^{\alpha} \text{num}^{\gamma}$
$<_{ptr}$, $=_{ptr}$	$(a \text{ ref}, a \text{ ref}) \to^{\alpha} \text{num}^{\beta}$
$+_{ptr}$	$(a \text{ ref}, \text{num}^{\beta}) \to^{\alpha} a \text{ ref}$
$-_{ptr}$	$(a \text{ ref}, a \text{ ref}) \to^{\alpha} \text{num}^{\beta}$

Appendix C: Case Study Graphs

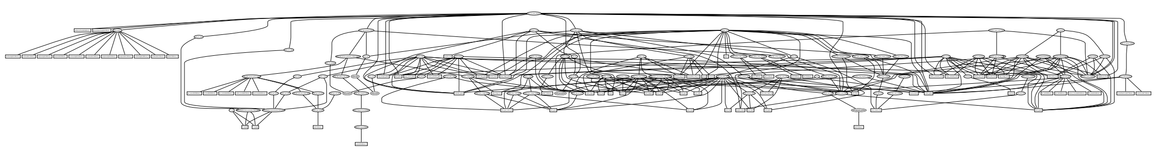

Figure 8: Results of query on *(`vehicle_global->cque`)

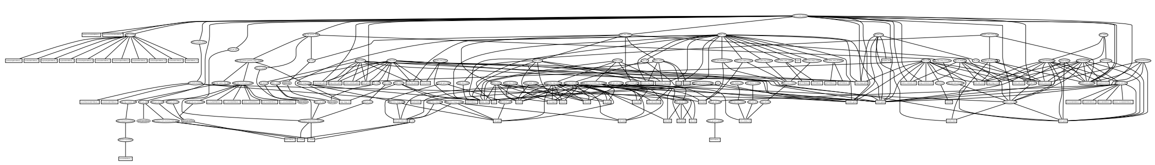

Figure 9: Results of query on `*(vehicle-global->sque)`