Lackwit:
A Program Understanding Tool
Based on Type Inference

Robert O'Callahan
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
+1 412 268 5728
roc@cs.cmu.edu

Daniel Jackson
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
+1 412 268 5143
dnj@cs.cmu.edu

ABSTRACT

By determining, statically, where the structure of a program requires sets of variables to share a common representation, we can identify abstract data types, detect abstraction violations, find unused variables, functions, and fields of data structures, detect simple errors in operations on abstract datatypes, and locate sites of possible references to a value. We compute representation sharing with type inference, using types to encode representations. The method is efficient, fully automatic, and smoothly integrates pointer aliasing and higher-order functions. We show how we used a prototype tool to answer a user's questions about a 17,000 line program written in C.

Keywords

restructuring, abstraction, C, representation

INTRODUCTION

Many interesting properties of programs can be described in terms of constraints on the underlying representation of data values. For example, if a value is an instance of an abstract data type, then the client code must not constrain its representation. If a program requires that the representations of two supposedly abstract types be the same, there is an abstraction violation. Less obviously, the value of a variable is never used if the program places no constraints on its representation. Furthermore, communication induces representation constraints: a necessary condition for a value defined at one site to be used at another is that the two values must have the same representation[1].

By extending the notion of representation we can encode other kinds of useful information. We shall see, for example, how a refinement of the treatment of pointers allows reads and writes to be distinguished, and storage leaks to be exposed.

We show how to generate and solve representation constraints, and how to use the solutions to find abstract data types, detect abstraction violations, identify unused variables, functions, and fields of data structures, detect simple errors of operations on abstract datatypes (such as failure to close after open), locate sites of possible references to a value, and display communication relationships between program modules.

Using a new type system in which representation information is encoded in the types, we express representation constraints as type constraints, and obtain and solve them by type inference. The type inference algorithm computes new types for the variables and textual expressions of a program, based on their actual usage—

[1] Some languages may allow a single value to be viewed as having different types at different points in a program, for example by implicit coercions. However these are just views of a single underlying representation; any meaningful transmission of data must use an agreed common representation.

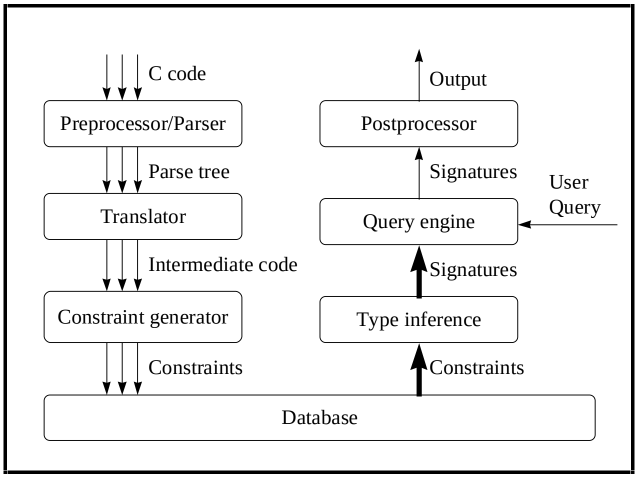

Figure 1: Tool Architecture

that is, their role in primitive operations—and ignores explicit type declarations.

The type system underlying this analysis can be very different from the type system for the source language[2]. This allows us to make finer distinctions than the language's type system would allow, and to handle potential sources of imprecision (such as type casts) in a flexible way.

The inferred types amount to a set of representation constraints. Given a variable or textual expression of interest, we examine its inferred type and draw conclusions about the variable's usage and behavior, and by finding other occurrences of the type (that is, other values that share the representation), we can track propagation and aliasing relationships in the global program.

Type inference is an attractive basis for an analysis for many reasons. It is fully automatic. It elegantly handles complex features of rich source languages like C, such as recursive pointer-based data structures and function pointers. Because our type system is a straightforward elaboration of a standard type system [9], we can employ the standard inference algorithm with only minor adaptation, and can be confident in the soundness of our scheme. Although the algorithm is known to have doubly-exponential time complexity in the worst case, in practice its performance is excellent. It has been implemented in compilers for Standard ML, where it has not proved to be a bottleneck. Our application is actually less demanding, since in contrast to their ML counterparts, C programs rarely use recursion or pass functions around. Our tool usually consumes space and time little more than linear in the size of the program being analyzed.

We have built a tool called Lackwit to demonstrate the feasibility of applying type inference analyses to C programs for program understanding tasks, and to experiment with the kind and quality of information available. The general architecture of the tool is shown in Figure 1. The multiple downward arrows indicate that C modules can be processed individually and fed into the database; the fat upward arrows show where information about the entire system is being passed. In the remaining sections of this paper, we will describe the type inference system, present the results of applying our tool to a real-life C program, and discuss some user interface issues.

Our system has advantages over all other tools we know of for analyzing source code for program understanding; see Figure 2. Lexical tools such as grep [7], Rigi [11] and the Reflection Model Tool [11] lack semantic depth in that they fail to capture effects such as aliasing that are vital for understanding the manipulation of data in large programs. Dataflow-based tools such as the VDG slicer [4] and Chopshop [8] do not scale to handle very large programs. Code checking tools such as LCLint [5] do not try to present high-level views of a large system. (Furthermore, although LCLint does compute some semantic information, it does not have a framework for accurate global analysis.)

Tool	**Semantic Depth**	**Global Display**	**Scalability**
grep	—	—	✓
Rigi, RMT	—	✓	✓
VDG slicer	✓	—	—
Chopshop	✓	✓	—
LCLint	✓	—	✓
Lackwit	✓	✓	✓

Figure 2: Tool feature comparison

EXAMPLE

Consider the trivial program in Figure 4. Suppose that a programmer is interested in the value pointed to by *r*. Our analysis will determine that the contents of *s* and *r* may be the same value, but that the value pointed to by *r* cannot be

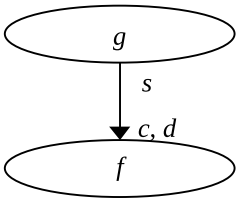

Figure 3: Result of query on *g*'s argument *r*

[2] It is very important not to get the two notions of "type" confused. In this paper we will normally be referring to types in our specialized type system.

```
int x;
int p1;
void f(int a, int b, int * c, int * d)
{ x = a;
  *c = *d;
}
void g(int p, int * q, int * r, int * s)
{ int t1 = 2;
  int c1 = 3, c2 = 4;

  p = p1;
  x++;
  f(c1, p, &t1, q);
  f(c2, 4, r, s);
}
```

Figure 4: Trivial program

copied to or referenced by any of the other variables in this program. The analysis will also determine that the relationship between *r* and *s* occurs because of a call from *g* to *f*; the tool can display the information graphically, as in Figure 3.

ANALYSIS OF THE EXAMPLE

The analysis assigns extended types to variables; the types encode information about representations. In C, the declared type of a variable usually determines its representation, but that is simply a matter of convenience. The program would still operate correctly if we had several different representations of *int*, provided that each version of *int* had the same semantics and that they were used consistently[3]. Intuitively, we could determine consistency by augmenting the declared type with the representation, and type-checking the program.

Figure 5 shows a consistent augmentation of the trivial program of Figure 4. The names superscripted with *capital* letters indicate *polymorphic* type variables: for any choice of representations int^A and int^B, the function *f* would type-check; we might just as well have different versions of *f* for each choice of int^A and int^B. We are thus free to assign *q* and *s* different representations. Note that because of the reference to the global variable *x*, the first argument of *f* is not polymorphic; its representation is fixed across different calls to *f*. These observations introduce context sensitivity into our analysis, since a polymorphic type variable may be instantiated differently at different call sites.

```
int^n x;
int^z p1;
void f(int^n a, int^A b, int^B * c, int^B *
d)
{ x = a;
  *c = *d;
}
void g(int^Y * q, int^X * r, int^X * s)
{ int^Y t1 = 2;
  int^n c1 = 3, c2 = 4;
  int^z p;
  p = p1;
  x++;
  f(c1, p, &t1, q);
  f(c2, c2, r, s);
}
```

Figure 5: Trivial program annotated with representations

From the type signatures of *f* and *g* in Figure 5, we can extract information about the behavior of the functions and how they are connected. For example, because *g* allows different representations for the values pointed to by *q* and *r*, we can conclude that *g* does not cause the two locations to be aliased, nor does it copy any values from one to the other.

Examining the calls to *f* in the body of *g* shows us how data flows between the two functions. If we are interested in tracking the contents of *q* passed into *g*, we can simply look for all the occurrences of its type int^Y; this includes occurrences of int^B, since one of the occurrences of int^B is instantiated to int^Y (in the first call to *f*). We find that values may be propagated to *t1*, and to the contents of arguments *c* and *d* of *f*.

Furthermore, suppose that we specify that the arguments of all primitive arithmetic operations have representation int^n. Then Figure 5 is still consistent. If *p* is intended to be an abstract file descriptor, we can see from its annotated type alone that no arithmetic operations can be performed on it (because it does not have to have type int^n); we can conclude that abstraction is preserved.

THE MOTIVATION FOR TYPE INFERENCE

To reduce the incidental sharing of representations, which lead to spurious connections, we would like to compute the most general assignment of representations to variables that is consistent with the communication patterns of the program. Viewing the representations themselves as types, this is the problem of type inference [9].

In the program above, type inference would proceed as follows. Let *x* have type int^J, and *p1* have type int^K. In *f*, because of the assignment of *a* to *x*, *x* and *a* must have the same type, so *a* has type int^J. **c* and **d* must have the same type, so let their types be “pointer to int^B”. *b* is unconstrained so let it have type int^A. Now we observe that

[3] Of course we do not propose implementing such a scheme. This is merely a pedagogical device.

the choices of int^A and int^B are arbitrary, and therefore *f* is polymorphic in int^A and int^B.

Now, in *g* we perform arithmetic on *x*, so *x* must be of type int^n and we have $int^J = int^n$. We find that *p* has the same type as *p1*, which is int^K. In analyzing the function calls, we obtain types for the function's actual arguments by instantiating the polymorphic type variables of its formal arguments. In the first call to *f* we instantiate int^A to be some fresh variable int^W and int^B to be some fresh int^Y; we find that *c1* must have type int^n, *p* is int^W, and *t1* and *q* are int^Y and "pointer to int^Y" respectively. Since *p* is simultaneously int^K and int^W, we must set $int^W = int^K$. In the second call to *f* we instantiate int^A to be some fresh variable int^V and int^B to be some fresh int^X; then we find that *c2* has type int^n, int^V must be int^n, and *r* and *s* are both "pointer to int^X". Because int^X and int^Y were arbitrary choices, the previous derivation is valid for all possible values of int^X and int^Y, and therefore *g* is polymorphic in all its arguments. Finally, we note that *p1* is of some type int^K where int^K is arbitrary but fixed (intuitively, we cannot have different versions of *p1* with different representations, because that would destroy the sharing implied by global data). Therefore we set int^K to be some arbitrary representation int^z; for maximum generality, we must make each such choice unique.

TYPE INFERENCE

The inference procedure is formalized in a simple polymorphic type system. Monotypes τ and polytypes σ obey the following productions:

$$\begin{array}{lll} \sigma ::= & \tau & \text{(Monomorphic type)} \\ \quad | & \forall\alpha.\sigma_1 & \text{(Polymorphic quantification)} \\ \tau ::= & \alpha & \text{(Type variable)} \\ \quad | & \tau_1 \rightarrow^{\beta} \tau_2 & \text{(Function)} \\ \quad | & \tau_1\ \mathbf{ref}^{\beta} & \text{(Reference (pointer))} \\ \quad | & (\tau_1, \tau_2, ..., \tau_n)^{\beta} & \text{(Tuple)} \\ \quad | & \mathbf{int}^{\beta} & \text{(Scalar)}^4 \end{array}$$

α is a metavariable that ranges over an infinite set of type variables. Ignoring the β tags (described below), this is a completely standard polymorphic type system. We use the standard inference algorithm W [9, 2] to compute the types of all variables of a source program (with no initial type declarations). Roughly speaking, W works by assigning variables to types and unifying type expressions as the program structure requires, in the manner described informally above. W is defined for a simple functional language; we provide a translation from C into such a language (see our technical report [12] for details).

W computes a type for each language construct, proceeding by structural induction over the abstract syntax tree. (Constructs such as statements that have no associated value are assigned a special type.) A type is assigned to a construct by an *inference rule*. We shall illustrate some contrived rules for C to give the flavor of the analysis, even though our tool does not work directly with C constructs.

Here is the rule for assignment:

$$\frac{e_1{:}\tau \qquad e_2{:}\tau}{e_1 = e_2{:}\tau} \qquad \text{(Assignment)}$$

This means that in order to compute a type for the assignment operator, we must first satisfy the constraint of the rule antecedent: e_1 and e_2 must have some common type τ. Then we can conclude that the result of the assignment has the same type τ (C allows chained assignments). Thus when we detect an assignment operator, we introduce the constraint that e_1, e_2 and "$e_1 = e_2$" must all have the same type. The rule for equality tests is similar:

$$\frac{e_1{:}\tau \qquad e_2{:}\tau}{e_1 == e_2{:}\tau'} \qquad \text{(Equality test)}$$

The result of the comparison need not be the same type as the comparison arguments, so the only constraint is that e_1 and e_2 must be the same type. (We do not require the result to be a Boolean, because we are interested in relationships between variables, not type checking *per se*.)

Dereferencing a pointer converts an expression whose type is "reference to type τ" to an expression of type τ *:*

$$\frac{e_1{:}\tau\ \mathbf{ref}}{{*e_1}{:}\tau} \qquad \text{(Dereference)}$$

An if-statement imposes no special constraints; the rule is necessary simply to invoke the recursive application of inference rules to its constituent statements:

$$\frac{e_1{:}\tau \qquad s_2{:}\tau' \qquad s_3{:}\tau'}{\mathbf{if}\ (e_1)\ \mathbf{then}\ s_2\ \mathbf{else}\ s_3{:}\tau'} \qquad \text{(If)}^5$$

Finally, the rule for function call is as expected, with the types of actual and formal arguments matching:

[4] In fact, we use a *number* type that includes all integer and floating-point values, but *int* simplifies the presentation.

[5] Actually, τ' is constrained to be a *continuation*, a function type that takes a dummy argument and returns the same type as the result of the enclosing C function. See our technical report [8] for details.

$$\frac{e_1{:}(\tau', \tau'') \to \tau \quad e_2{:}\tau' \quad e_3{:}\tau''}{e_1(e_2, e_3){:}\tau} \quad \text{(Binary Function Call)}$$

THE TAGS

Note that the type expressions given in the productions above include superscripted tags marked β. These are used to track the identity of type constructors. A fresh tag is generated whenever a type constructor is introduced in a new constraint. Whenever we find that two types are constrained to be identical, we unify their two type constructors and merge their tags; thus the tags partition the set of occurrences of type constructors. For example, if two variables have **ref** constructors with different tags, then they are not aliases[6,7]. If two variables are tuples with the same tag, then they must be structures of the same abstract type (or abstraction violations have occurred). The tags are simple to implement on top of algorithm W: they can be treated merely as an extra parameter of the type constructor, and then no change to the algorithm is required.

We supply type signatures for the built-in primitives and any library functions that are called; these signatures are the only information about external code that we require. Since such signatures are also the result of the analysis, this makes our techniques modular (and contributes to scalability). By adjusting the signatures, we can customize the analysis to compute different kinds of information.

The basic signatures are listed in Appendix 1. The only interesting signature is for the cast operator. We treat "cast" as the identity function, in the hope that casts are merely being used to work around the lack of polymorphism in C's type system (and so the program will still be typable in our type system). If this is not true, we still handle the program, albeit with some loss of accuracy: we report any type errors, display their context, and continue the analysis. The user is able to check whether the results they are interested in have been compromised. In our Morphin example below (17,000 lines), there are just two "bad" type casts, neither of which affected the results for any of the queries the user requested.

The reader familiar with type theory may be interested in the following details. Recursive types are treated as infinite regular trees (see Cardone and Coppo [2] for details). We do not use polymorphic recursion; that is, **let** and **letrec** bindings are the only places where we perform polymorphic generalization. We use a value restriction on polymorphic **let**s to make side-effects safe [16]. Our representation tags correspond closely to the region variables of region inference [15].

OBTAINING AND DISPLAYING GLOBAL INFORMATION

The result of the type inference phase is a mapping from source variables and functions to type signatures in our extended type system. We provide ways to extract interesting information and display it graphically in a way that has a clear relationship to the source program.

Our basic approach is to produce a graph summarizing the information about a single component of a variable (see Figure 6 below). The nodes of the graph represent global declarations, and the edges represent the use of one declaration by another in the text of the program. Arrows point from the using declarations to the used declarations. When the tool has determined, by inspecting signatures (see below), that a use cannot transmit the value of the queried component, then we omit the corresponding edge. By eliminating unreachable nodes, we show just the part of the program that is able to access the value. A value is transmitted by passing a data structure containing it as a parameter in a function call, or by returning such a data structure from a function call, or by referencing global data (containment includes reachability through pointers).

The inspection of signatures used to filter the edges works as follows. The user has specified a global variable, function result, function parameter or local variable, or some component thereof (a chain of pointer dereferences and/or structure fields). Because any accesses to run-time values of the component must agree on the representation, the types of the components through which accesses are made must have tags that are "compatible" with the tag on the type of the queried component. In a monomorphic type system, the compatibility relation is just tag equality, but with polymorphism we have to consider that a polymorphic tag may be instantiated to some other tag, which means the same run-time value could be accessible through different tags. (For example, in Figure 5, the value of *s* is accessed with tags *B* and *X*.) Therefore we define two tags *S* and *T* to be compatible if there is some tag *U* such that the program exhibits a chain of instantiations from *S* to *U* and a chain of instantiations from *T* to *U* (the chains may be empty). Note that this relation is symmetric but not transitive. In Figure 5, for example, *B* is compatible with *X* and *Y*, but *X* is not compatible with *Y*,

[6] See Steensgaard's work on points-to analysis [14]. Similarly, comparing the tags on function types with the tags on declared functions gives us an analysis of higher-order control flow.

[7] Actually in the polymorphic type system we must use a more complicated relation than just tag equality; see below.

and as a result, we can infer that values may be passed from *q* or *r* to *c* or *d*, but not between *q* and *r*.

We determine the set of “tags of interest” that are compatible with the tag of the queried component, and locate the declarations whose signatures contain occurrences of tags of interest. Declarations whose signatures do not contain any occurrences of tags of interest do not transmit any interesting values when they are used, so those uses are omitted from the graph. Since polymorphic signatures can be instantiated to different types at different usage sites, we also omit uses when the type at the usage site does not contain any tags of interest.

There is a difficult tradeoff between detail and clarity in choosing what kinds of information to display for each node and edge. We have found it useful to visually distinguish functions from global data, and to highlight nodes that are “interesting” (for example, functions that directly access the representation of some variable). Clearly it would be beneficial to have an interactive display, which we plan to add in future work.

OBTAINING LOCAL INFORMATION

We can use type inference to compute local information. For example, we often wish to determine whether or not a function directly accesses a piece of data, rather than just passing it to another function. To do this, we apply type inference to a single global function, ignoring constraints induced by other global declarations that it references. For example, in Figure 6, we determine that the definition of *map_mgr_convert_pixel_coords* does not directly access the representation of the vehicle object, even though it calls a function that does.

RESULTS (1)

We have used our tool to analyze Morphin, a robot vehicle control program consisting of over 17,000 lines of C code, with 252 functions and 73 global variables. Morphin is to be restructured and adapted to support new features. The developer responsible for this work asked us to determine where and how certain structures were used, if at all. In the course of answering his questions, Lackwit also highlighted some representation exposures.

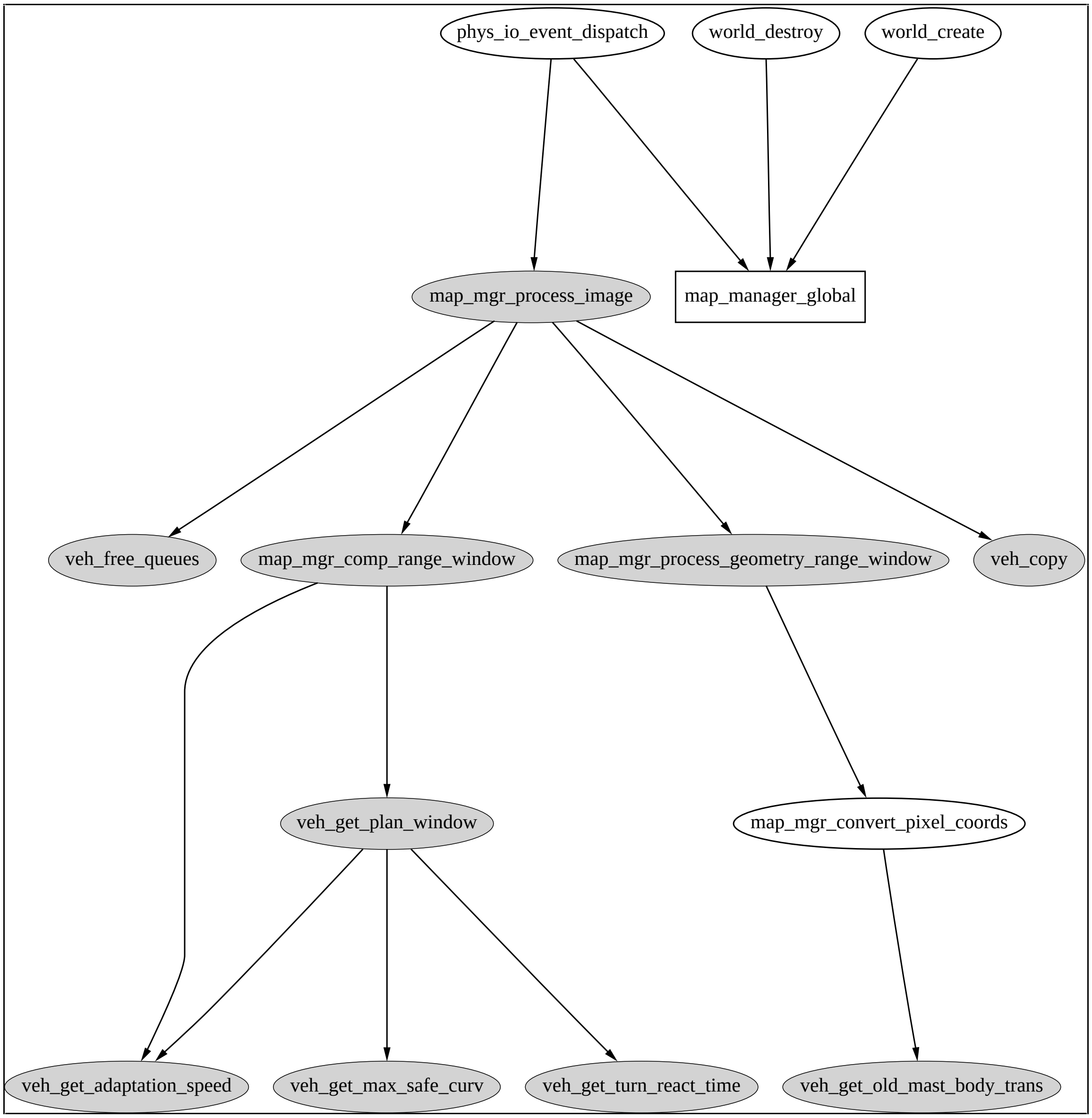

Figure 6: *(map_manager_global->cur_veh)

We computed results for queries of the form "Which functions in the program could directly access the representation of component X of variable Y?" Figure 6 shows the results of a query on the "current vehicle" field of the *map_manager_global* global variable (we also allow queries involving local variables and function parameters).

The shaded nodes are the definitions that directly access representations; that is, whose code constrains the representation of the value in question. In this case, the value in question is a structure, and the shaded nodes constrain the type by accessing fields of the structure.

Given that the "veh_" functions are operations on the vehicle abstract data type, it is easy to see that abstraction may be violated in the functions *map_mgr_process_image*, *map_mgr_process_geometry_range_window*, and *map_mgr_comp_range_window*, but nowhere else.

Lackwit built the constraint database from the 17,000 lines of source in 274 seconds (wall-clock time). The database is about 15MB. Type inference took 78 seconds, about 23 seconds of which was user-level CPU time. Individual queries are then answered almost instantaneously. These numbers are for a 90Mhz Pentium with 32MB of RAM running Windows NT. As we will discuss in below, certain optimizations could dramatically improve performance. The process of building the database can be carried out

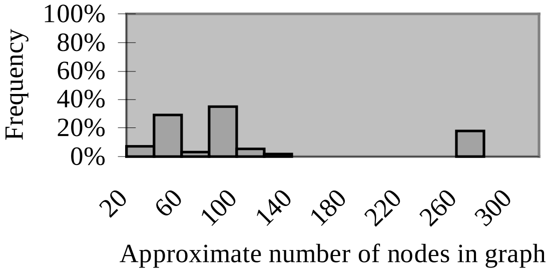

Figure 7: Distribution of Graph Sizes For All Queries

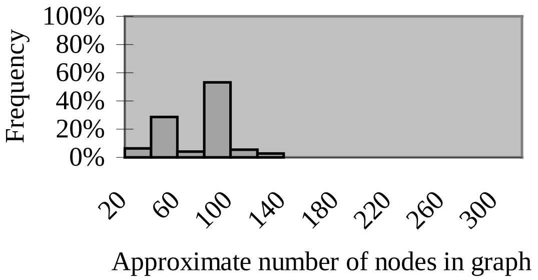

Figure 8: Distribution of Graph Sizes For Queries on Pointers

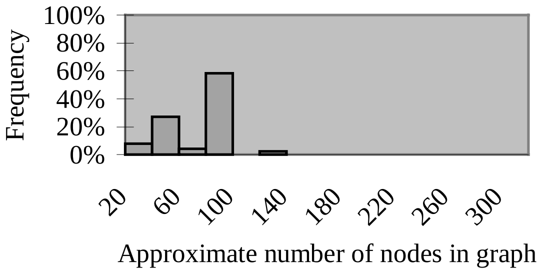

Figure 9: Distribution of Graph Sizes For Queries on Structures

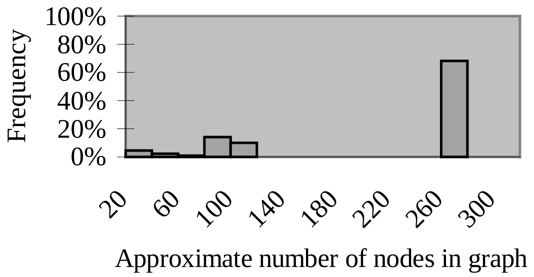

Figure 10: Distribution of Graph Sizes For Queries on Integers

independently for each source module; thus the potential performance bottlenecks are all in the solver.

The type inference algorithm processes functions one at a time, iteratively computing a signature for each function and adding it to a "type environment". While processing a function, the main operation is unification of types, the cost of which is proportional to the smaller of the sizes of the descriptions of the types[8]. The number of unifications performed is proportional to the size of the code for the body of the function, which is preserved (to within a constant factor) by the translator. Therefore, if the size of the inferred types is bounded, the solver takes time and space little more than linear in the size of the program. The types are small in practice.

To attempt to characterize the sizes of query results, we computed for every possible query an upper bound approximation of the number of nodes of the graph we would produce (see Figure 7). Figure 8, Figure 9, and Figure 10 show the sizes of results of queries on all pointers, structures and scalars respectively. The large spikes at the right-hand sides of Figure 7 and Figure 10 are due to a set of integer variables that are grouped together by chains of arithmetic operations; queries that hit this set (about 65% of all queries on integer-valued components) will probably produce too much information to be useful. However, queries on structures and pointers will produce results of manageable size.

EXTENSIONS TO THE TYPE SYSTEM

We can analyze a broader range of program properties by extending the type system in simple ways. For example, it is useful to know whether a memory location is never read or never written, or whether a piece of dynamically-allocated storage is never allocated or never deallocated, or where these effects might occur. We encode such properties by introducing specialized type constructors and a subtype relation (see Figure 11). Primitives such as "assign" and "deref" now constrain their arguments to be "written" refs and "read" refs respectively.

This is easy to implement when the properties are a vector of boolean values and all constraints are of the form

[8] Actually, because we must maintain equivalence classes of type variables using union-find with path compression, there is a superlinear component of $\alpha(n)$, where α is the inverse Ackermann function. For all practical purposes, $\alpha(n) < 6$.

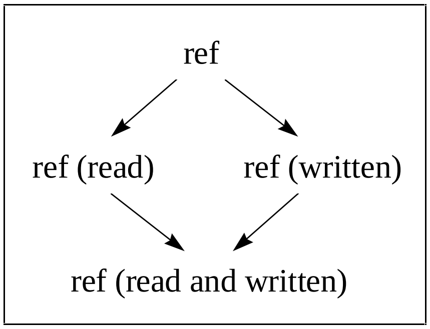

Figure 11: Read/write attributes for memory locations (arrows point from supertypes to subtypes)

"property p is true"[9]. We simply use our existing type inference procedure and attach additional parameters to the type constructors, one for each boolean. Operations that constrain a property of their arguments or results use the special type *yes* as the appropriate parameter of the type constructor, otherwise they have a type variable. For example, if we give a pointer to α the type "(α, *read*, *written*) ref", then the signature of "deref" becomes "$\forall\alpha.\forall\beta.(\alpha, yes, \beta)$ ref $\rightarrow \alpha$".

When type inference is finished and we have the signatures of all the functions and local and global variables, we can easily inspect them to discover anomalous types, such as memory locations that are never read (the "read" parameter is a type variable), or dynamic storage that may be allocated but is never deallocated (the "allocated" parameter is *yes* but the "deallocated" parameter is a type variable).

If the condition of interest can be written in the form "P and not Q" (this includes all the examples we have presented), then we do not report an anomaly unless one actually exists (assuming that there is no "dead code" – for every call to a primitive operation in the program, there is some input that will cause it to be executed). For if a type has property P, then there must be some call to a primitive operation that constrains the type to have that property, and since there is no dead code, there is some execution that invokes this operation (so there will be a run-time value V with property P). If the type does not have property Q, then there is no operation that constrains the type to be Q, and so the run-time value V cannot obtain property Q.

Although we do not report spurious anomalies, we can only give an upper bound approximation to the actual site(s) of any problems, based on the occurrences of the type tag as described above. And of course, we may miss some anomalies if they depend on which control paths are taken in the program.

RESULTS (2)

We used these techniques to perform two specialized analyses: detection of data that is never read, and detection of memory leaks. These correspond to checking, for each type that is a memory location, whether the location is created (by a variable entering scope or by dynamic allocation) and not read, or whether the location is dynamically allocated and not dynamically deallocated. All reported candidates were shown to be non-spurious by manual inspection of the source code.

Lackwit reported nine global variables and ten local variables that are never read. In addition, it reported that six local variables are structures containing some fields that are never read.

Checking for memory leaks, Lackwit reported that six global variables refer to dynamic data structures that are never freed. (These do not cause problems in practice, since they are freed by the operating system when the process terminates, but they are a symptom of poor programming style.) It also discovered two fields of data structures that are pointers to memory that is never freed. These are a genuine problem, because these pointers are updated in a frequently-executed loop.

CURRENT STATUS AND FUTURE WORK

The Lackwit front end is currently written in C and C++, and is based on the PCCTS toolkit [13]. The database is simply a sequential binary file, implemented by hand in C. The solver and query engine are also written in C; currently the query language is very simple and a bit unwieldy. The query engine outputs relational tables in a text format. A Perl script converts these tables to a graph, in the process performing the postprocessing analyses described above. We use the "dot" graph-drawing tool [6] to produce the graph.

Performance is currently good, but could be greatly improved. In particular, the recursive-descent parser should be rewritten as an LALR parser for speed. We would gain a lot of performance at the cost of flexibility and simplicity by eliminating our intermediate language and generating constraints directly from the C abstract syntax (as Steensgaard does [14]). Most importantly,

[9] Although "all properties are true" will always be a solution, we will recover the most general solution. For example, we may discover that a program is consistent with a memory location being either "read" or "unread", in which case we may legitimately treat it as "unread" (and eliminate the offending variable or structure element).

simplifying constraints in the front end would produce at least an order of magnitude saving in the size of the database and the processing time of the solver. This is important because the solver is the potential performance bottleneck as the number of source files increases.

To make the tool easy to use, we need a better query interface, preferably providing the program source code as context. The tables output by the query engine can be very large, suggesting that most of the postprocessing should be folded into the query engine (to reduce output and parse time). Perhaps our biggest problem is the graph-drawing program, “dot”. It does not scale well to large, highly-connected systems. We would like to explore the use of interactive visualizations or other techniques to present the data in a more useful way.

We intend to experiment with alternative type systems. We may be able to incorporate recent work on type inference of ill-behaved programs. We would like to encode more information in types to increase the scope and accuracy of these techniques. Another interesting problem is to enrich the type system to handle a wider class of source languages, for example by adding subtyping for object-oriented languages.

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 we associate 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 elements 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. 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 we cannot. The two tools are complementary.

ACKNOWLEDGEMENTS

We would like to thank Robert Harper and Lars Birkedal for their helpful advice. 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.

REFERENCES

1. Robert W. Bowdidge and William 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. Michael 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, J. Guttag, J. Horning, and Y. Tan. 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. Daniel Jackson and Eugene Rollins. Abstractions of program dependencies for reverse engineering. *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, New Orleans, December 1994.

9. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348-375, 1978.

10. Gail Murphy and David Notkin. Lightweight source model extraction. *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, 1995.

11. H. Müller, S. Tilley, M. Orgun, B. Corrie and N. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In 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. Bjarne 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. Mads Tofte and Jean-Pierre 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. Andrew Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, volume 8, number 4, December 1995, pp. 343-356.

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

APPENDIX 1: BUILT-IN SIGNATURES

The following primitives are used by the translator.

Primitive	**Signature**
ref	$t \rightarrow t\ \mathbf{ref}^{\alpha}$
assign	$t\ \mathbf{ref}^{\alpha} \rightarrow t \rightarrow ()$
deref	$t\ \mathbf{ref}^{\alpha} \rightarrow t$
mk-tuple$_n$	$t_1 \rightarrow t_2 \rightarrow \ldots\ t_n \rightarrow (t_1, t_2, \ldots , t_n)^{\alpha}$
elem-tuple$_{n,k}$	$(t_1, t_2, \ldots , t_n)^{\alpha} \rightarrow t_k$
ref-array$_n$[10]	$(t, t, \ldots , t)^{\alpha} \rightarrow t\ \mathbf{ref}^{\beta}$
copy-array[11]	$(t\ \mathbf{ref}^{\alpha} \rightarrow t) \rightarrow t\ \mathbf{ref}^{\alpha} \rightarrow t\ \mathbf{ref}^{\beta}$
undefined-scalar[12]	T
NULL	$t\ \mathbf{ref}^{\alpha}$
cast	$t \rightarrow t$
scalar$_n$	$\mathbf{int}^{\alpha}$
pointer-arith$_{op}$	$t\ \mathbf{ref}^{\beta} \rightarrow \mathbf{int}^{\alpha} \rightarrow t\ \mathbf{ref}^{\beta}$

[10] This is used to translate array initializers.

[11] This is used when we assign a structure value that contains an array.

[12] This is used to initialize scalars that aren't initialized in C. This allows us to store pointers in them, allowing us to handle programs that use integers polymorphically as integers or pointers.

unary-arith$_{\mathrm{op}}$	$\mathbf{int}^{\alpha} \rightarrow \mathbf{int}^{\alpha}$
binary-arith$_{\mathrm{op}}$	$\mathbf{int}^{\alpha} \rightarrow \mathbf{int}^{\alpha} \rightarrow \mathbf{int}^{\alpha}$
binary-relational$_{\mathrm{op}}$	$\mathbf{int}^{\alpha} \rightarrow \mathbf{int}^{\alpha} \rightarrow \mathbf{int}^{\beta}$
pointer-relational$_{\mathrm{op}}$	$t\ \mathbf{ref}^{\beta} \rightarrow t\ \mathbf{ref}^{\beta} \rightarrow \mathbf{int}^{\alpha}$
pointer-unary$_{\mathrm{op}}$	$t\ \mathbf{ref}^{\beta} \rightarrow \mathbf{int}^{\alpha}$

Pointer arithmetic operators: pointer add, pointer subtract

Unary arithmetic operators: negate, bitwise not, logical not, unary plus

Binary arithmetic operators: add, subtract, multiply, divide, modulus, left shift, right shift, bitwise and, bitwise or, bitwise xor

Binary relational operators: less than, greater than, equal, less than or equal, greater than or equal, not equal

Pointer relational operators: less than, greater than, equal, less than or equal, greater than or equal, not equal

Pointer unary operators: logical not