Detecting Shared Representations Using Type Inference

Robert O'Callahan and Daniel Jackson
September 1995
CMU-CS-95-202

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Many questions that arise in the reverse engineering or restructuring of a program can be answered by determining when, because of transfers of values, variables must share a common representation.

We have developed a new method for answering such questions. It is based on the observation that variables being of the same type is a necessary, but not sufficient, condition for them to have common values. We use polymorphic type inference to compute new types for the variables, eliminating cases of incidental type sharing where the variables could safely have different representations. The method is computationally inexpensive, fully automatic, and elegantly handles pointer aliasing and higher-order functions. We describe the use of a prototype tool to answer questions about the C utility program "GNU cmp".

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 ARPA, TRW, Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, or the U.S. government.

Keywords: types, type inference, representation, abstraction, dataflow, C, restructuring, reverse engineering

1 Introduction

During the reverse engineering or restructuring of a program, questions often arise that a tool should be capable of answering. Suppose, for example, a programmer wants to port the GNU "cmp" utility to a different platform and needs to know which parts of the code would be affected by changing the implementation of file I/O. We have implemented a tool that can answer this question efficiently and accurately for C code, using a type inference algorithm.

The file cmp.c contains this declaration of a global array of file descriptors for the two files being compared:

```
static int file_desc[2];
```

Roughly speaking, our method determines where in the program the values in file_desc can be mentioned. It tells us that the values are used in the functions file_position, main, cmp, and block_read, but file _desc and the argument fd of block_read are the only variables that contain the values. (file_position, main, and cmp just refer to the global variable.) See Figure 1 for the relevant code.

This kind of information could have been gathered by standard dataflow techniques such as slicing, but they are unwieldy for this problem. Typically, slicing methods track dataflow from a **site**, a variable at a certain location in the program, so we would have to combine multiple slices to answer the question as it is phrased. In addition, our method accomodates pointer aliasing and higher-order functions in an elegant way (though they do not appear in these examples), and is, in general, potentially much less computationally expensive.

Syntactic tools are extremely efficient but inaccurate. A simple "grep" for file_desc would have shown all its direct references, but would not have followed the call to block_read and searched for references to fd.

2 Overview of the Method

Assuming that the program has been compiled and is thus well-typed, our question can be answered in terms of types and answered by a type analysis. The values in file_desc can only appear in variables that have the same type as the elements of file_desc. Therefore a very conservative answer to the question would be "all the variables that have type int".

```
static size_t
block_read (fd, buf, nchars)
     int fd;
     char *buf;
     size_t nchars;
{
...
      size_t nread = read (fd, bp, nchars);
...
}
...
int
main (argc, argv)
     int argc;
     char *argv[];
{
...
      file_desc[i1] = (strcmp (file[i1], "-") == 0
                       ? STDIN_FILENO
                       : open (file[i1], O_RDONLY, 0));
      if (file_desc[i1] < 0 || fstat (file_desc[i1], &stat_buf[i1]) != 0)
        {
          if (file_desc[i1] < 0 && comparison_type == type_status)
...
    if (close (file_desc[i]) != 0)
...
}

static int
cmp ()
{
...
              size_t r = read (file_desc[i], buf0, (size_t) min (ig, buf_size));
...
      read0 = block_read (file_desc[0], buf0, buf_size);
...
      read1 = block_read (file_desc[1], buf1, buf_size);
...
}
...
static off_t
file_position (i)
     int i;
{
...
      position[i] = lseek (file_desc[i], ignore_initial, SEEK_CUR);
...
}
```

Figure 1: Where file descriptor values are mentioned in GNU “cmp”

However, we will normally have much “incidental” type sharing, such as the ubiquitous use of integers. GNU “cmp” uses integers to represent both file descriptors and buffer sizes. Intuitively, this problem would be mitigated by using (wherever possible) new, distinct abstract types as “aliases” for existing types. We could introduce an abstract type

for file descriptors and another for buffer sizes. Our method does exactly this, using type inference to distinguish the types of variables even when declared to have the same type, wherever this does not violate the semantics of the program (that is, wherever it does not cause type checking to fail).

This type system need not be restricted to the type system of the underlying language(s) used to express the program artefacts. In particular, we introduce polymorphism – not available in C's type system - to separate independent uses of a function. "Utility functions" that implement ADT-like operations should not require that the types of the arguments be the same when called in different contexts; intuitively, such a function could have multiple implementations (one for each different argument type), and the fact that only one function is used is simply a matter of economic code-sharing.

Consider the toy program in Figure 2. Our method yields four distinct types:

- One type for budget, pay_increment, manager_salary, coder_salary, manager_increment and coder_increment, and to which salary is a reference.
- One type for old_performance and new_performance; the pick_raise function is polymorphically generalized on this type.
- One type for manager_old_performance and manager_new_performance.
- One type for coder_old_performance and manager_old_performance.

Here the two salaries are grouped together only because they both contribute to the common budget.

To analyze a C program, we first mechanically translate it into a simpler language – discarding the existing type declarations. Then we apply a type inference procedure to derive new, more specialized, types.

To explain how the method works, we first present the type system for a simplified language, demonstrating its application to a toy program. Some elaborations needed to handle C are then added. Finally, we show how the method was applied to a real program, GNU "cmp", and how the results are interpreted.

```
int budget = 5000;
void pick_raise(int old_performance,
                int new_performance,
                int pay_increment,
                int * salary)
{ if (old_performance < new_performance)
    *salary = *salary + pay_increment;
  budget = budget - *salary;
}

{ int manager_old_performance = 50, manager_new_performance = 40;
  int coder_old_performance = 8000, coder_new_performance = 9000;
  int manager_salary = 100000, coder_salary = 35000;
  int manager_increment = 10000, coder_increment = 5000;

  pick_raise(manager_old_performance, manager_new_performance,
             manager_increment, &manager_salary);
  pick_raise(coder_old_performance, coder_new_performance,
             coder_increment, &coder_salary);
}
```

Figure 2: Toy program

3 Using Polymorphism to Introduce Abstraction

The type system we use was developed to typecheck ML and other type-rich languages. To use it to analyze C programs, only a few small changes are needed. The biggest change is our deviation from standard practice in the way we construct the initial type environment for the "built-ins"– the standard constants, operators and functions that one normally provides in a language. For example, "+" normally has the single type[1] "int $\rightarrow$ int $\rightarrow$ int"[2]. We want to distinguish independent uses, so that "+" could have type "buffersize $\rightarrow$ buffersize $\rightarrow$ buffersize" in some contexts, and type "dollars $\rightarrow$ dollars $\rightarrow$ dollars" in other contexts, and so on.

To realize this, we make the existing constants and operators **polymorphic**. For example, "+" is assigned type "$\forall$t.t $\rightarrow$ t $\rightarrow$ t". Wherever we use "+", we can instantiate "t" to be an arbitrary type. "t" can be instantiated to types that do not already exist, and this allows types such as "buffersize" to be inferred even though they are not explicitly mentioned in the code. These new types are assigned internal names.

We choose types for each constant and operator so that its standard type is one of the possible instantiations of its polymorphic type. For example, if "foo" is of type "int$\rightarrow$(int

---

[1] Of course, in C, "+" is overloaded to have a certain number of different signatures, but this is irrelevant here.

[2] We often use "curried" types for the signatures of built-in operators. Here, "int$\rightarrow$int$\rightarrow$int" means the function that takes two integer parameters and returns an integer.

*)" then we **cannot** give it the polymorphic type "$\forall t.t \rightarrow t$". Obeying that rule guarantees that if a program is well-typed with respect to the standard types of the built-ins, then that typing is a possible typing with the new types of the built-ins. Therefore the type inference procedure will not fail to find specialized types unless the program was not well-typed to start with.

4 Type Inference for the Q Language

We analyze C programs by first translating them into a simpler "Q" language. Then, given the types for the "built-ins", we reconstruct the types for the rest of the program. We have adopted, almost unchanged, Tofte's type discipline for a simple polymorphic language with imperative features. To simplify the presentation, we have omitted some details, which can be found in Tofte's thesis [Tof87].

Programs in Q are expressions:

e	::=	x	(Variable)
	|	$\lambda x.e_1$	(Lambda abstraction)
	|	$e_1\ e_2$	(Function application)
	|	**let** $x = e_1$ **in** e_2	(Let binding)
	|	**if** e_1 **then** e_2 **else** e_3	(If statement/expression)
	|	**while** e_1 **do** e_2	(While statement)
	|	e_1 ; e_2	(Statement sequencing)

Expressions have types of the following form:

σ	::=	τ	(Monomorphic type)
	|	$\forall\alpha.\sigma_1$	(Polymorphic quantification)
τ	::=	α	(Type variable)
	|	$\tau_1 \rightarrow \tau_2$	(Function)
	|	*unit*	(Empty type)
	|	τ_1 *ref*	(Reference (pointer))
	|	$(\tau_1, \tau_2, ..., \tau_n)$	(Tuple)

We have omitted type constructors (that is, the built-in types such as **bool** and **int**), because they are not used in our method.

The "primitive" operations one might expect in the language are provided as values in the initial environment for the program. In particular, these include the "ref" (reference) function that takes a value argument, allocates a new memory cell, stores the value in the cell and returns a reference to the cell; the "assign" (assignment) function that takes two arguments consisting of a reference to a cell and a value, updates the cell with the new value, and returns nothing; and the "deref" (dereference) function that takes a cell reference argument and returns the contents of the cell.

The types of variables, including built-in operators and constants, are recorded in a type environment that maps expressions to their types. In the initial type environment, the primitive store operations are given these standard signatures:

$$\begin{aligned} \text{TE(ref)} &= \forall u.u \to u\ \textit{ref} \\ \text{TE(assign)} &= \forall t.t\ \textit{ref} \to t \to \textit{unit} \\ \text{TE(deref)} &= \forall t.t\ \textit{ref} \to t \end{aligned}$$

The type variables are composed of two distinct subsets: imperative and applicative. Here, "u" is an imperative type variable and "t" is an applicative type variable. Only imperative type variables may appear in store types (that is, the types of the updateable values in the store, accessed through references). This distinction is introduced to preserve the soundness of the type system in the presence of polymorphism[3].

The basic rules for type inference are identical to those presented by Tofte. For example, the rule for variable accesses is

$$\frac{x \in \text{dom TE} \qquad \text{TE}(x) > \tau}{\text{TE} \Vdash x{:}\tau} \qquad \text{(Variable access)}$$

The condition "TE(x)>τ" can be read "τ is a specialization of TE(x)". This means that whenever a variable is used, arbitrary types can be substituted for the polymorphic type variables. For example, the type of the identity function "id" in the type environment might be "$\forall t.t \to t$". According to this rule, one use of "id" might have type "*unit* $\to$ *unit*", another "$v \to v$", another "$(a \to a) \to (a \to a)$", and so on.

The rest of the standard inference rules are:

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

$$\frac{\text{TE} \Vdash e_1{:}\tau' \to \tau \qquad \text{TE} \Vdash e_2{:}\tau'}{\text{TE} \Vdash e_1\ e_2{:}\tau} \qquad \text{(Function application)}$$

$$\frac{e_1 \text{ is non-expansive} \qquad \text{TE} \Vdash e_1{:}\tau_1 \qquad \text{TE} \oplus \{x \to \text{Clos}_{\text{TE}}\,\tau_1\} \Vdash e_2{:}\tau}{\text{TE} \Vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2{:}\tau} \qquad \text{(Let binding)}$$

$$\frac{e_1 \text{ is expansive} \qquad \text{TE} \Vdash e_1{:}\tau_1 \qquad \text{TE} \oplus \{x \to \text{AppClos}_{\text{TE}}\,\tau_1\} \Vdash e_2{:}\tau}{\text{TE} \Vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2{:}\tau} \qquad \text{(Let binding)}$$

---

[3] Substitutions during type inference are required to propagate "imperativeness"; when a type expression is substituted for an imperative type variable, any applicative type variables in the expression are made imperative (by globally substituting an imperative type variable for each applicative one). The imperative variables in the signature of "ref" guarantee that all store types will contain only imperative type variables. The signatures of "assign" and "deref" do not need to specify imperative type variables because the "imperativeness" will be inferred from where the referenced memory cells were allocated.

Here "⊕" adds a variable, and its inferred type, to the type environment.

An "expansive" expression is one whose evaluation might potentially allocate new memory cells in the store (that is, apply the "ref" function). A common and safe approximation is to declare an expression "non-expansive" if it is a syntactic value (a variable or lambda abstraction)[4]. In particular, function declarations are non-expansive. The "Clos" function gives the generalization of τ_1 with respect to the type environment TE, by quantifying over all the free type variables in τ_1 that are not free in TE. The "AppClos" function gives the generalization of τ_1 by quantifying over all the free *applicative* type variables in τ_1 (that are not free in TE).

To see why this complication is needed, imagine a naive system (lacking this distinction between imperative and applicative type variables) being used to analyze this nonsensical program:

```
let idref=ref (λx.x) in (
  idref := λy.y+1;
  (deref idref) true)
```

To give a type to idref, the type inference procedure applies the lambda abstraction rule and picks a fresh type variable τ' for x. After applying the variable access rule it deduces that λx.x has type $\tau' \rightarrow \tau'$. The result after applying ref has type "$(\tau' \rightarrow \tau')$ ref". Now, because τ' was a fresh type variable, it is not in the type environment, and so it is generalized by Clos to give idref the polymorphic type "$\forall \tau'.(\tau' \rightarrow \tau')$ ref". The assignment statement is processed by instantiating τ' to be (for example) *int*. The last statement is processed by instantiating τ' to be *bool*.

Unfortunately, something has gone wrong. The program would fail with a run-time type error if executed, even though type inference would give no warning. In our method, the analysis would incorrectly report that y and true are unrelated, even though y receives the value of true at run-time, just before the error occurs. Imperative type variables and the extra rule for non-expansive **let** declarations are introduced specifically to close this hole in the type system; see Tofte's thesis for details [Tof87].

Some of our rules differ slightly from Tofte's:

$$\frac{TE \Vdash e_1 : \tau' \quad TE \Vdash e_2 : \tau}{TE \Vdash e_1 \; ; \; e_2 : \tau} \qquad \text{(Sequencing)}$$

This is very slightly different from Tofte's rule for his extended language; he requires that e_1 have type *unit* (or *stm* as he calls it). We relax that constraint, to model C's feature of discarding function results in statements.

In contrast, our rule for **while** differs significantly from his:

---

[4] There are similar approaches that work just as well; we actually declare an expression expansive if and only if it contains a function application not inside a lambda abstraction

$$\frac{TE \Vdash e_1:\tau' \quad TE \Vdash e_2:\tau}{TE \Vdash \textbf{while}\ e_1\ \textbf{do}\ e_2:\text{unit}} \qquad \text{(While)}$$

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

Tofte constrains the condition expression, e_1, to be *bool* but we do not. This is because we are not doing type checking, and we wish to admit a family of different boolean types.

The only extensions we have introduced to Tofte's "extended" language are the **if** statement (omitted from his presentation) and tuple types (see below). Because our rules differ only trivially from Tofte's, we can adopt, almost unaltered, his W1 type inference algorithm for automatically inferring types[5].

5 Application to Real Programs

Code is mechanically translated from its source language, C, into our intermediate language Q which is described above.

The translation does not preserve meaning, because Q is a simpler language; a few C features used in our example cannot be represented in Q. Arrays are handled by collapsing them into a single element (normally each element of an array has the same type, so this does not affect the accuracy of the type information). We ignore type-casts, which in our example were all casts between different scalar types, or "casting away const" (other kinds of type-casts are not ignored[6]). We ignore the optional arguments of variable-argument functions. We sometimes ignore the details of control flow when it is difficult to represent in Q (for example, functions with multiple **return** statements, and **switch** statements with cases that fall through to the next case); the actual flow of control does not affect the type inference in any case. Because we only need to extract the type structure, we ignore the actual values of constants, and coalesce standard operators with the same signatures (for example, addition and subtraction both became "generic-binary-op")[6].

---

[5] From a type-theoretic point of view, there is one particularly unusual feature of our method: we include a "NULL pointer" as one of the constants. In C, this value (denoted by "0") represents a memory cell that cannot be assigned to or read from, but it is useful as a sentinel value to store in reference variables. In C it is compatible with all pointer types, and so in our environment we give it type "∀u.u *ref*", where "u" is an imperative type variable. Strictly speaking this introduces an unsoundness into our type system, because a program could dereference NULL to get (for example) an integer and also a function. Therefore we assume that programs never assign to or dereference NULL at run-time.

[6] C has some notoriously hard-to-analyze features that we handle, perhaps with some loss of accuracy. These include type casting between scalars and references, and the use of pointer arithmetic and type casting to access memory in arbitrary ways. In the first case, we must treat the result of such a cast as a fresh type (we can't allow references and

None of these affect accuracy except the omission of the optional arguments passed to variable-argument functions, and the collapsing of arrays into a single element (in the presence of heterogeneous arrays). The former is a rarely-used feature; only theerror function took a variable number of arguments in our example. The interpretation of the extra arguments (including what typeserror expects them to be) depends on the contents of a format string passed to error (as is true of most uses of variable-argument functions in C), and so no static analysis could reliably determine how those arguments are used anyway.

Heterogeneous arrays do not occur in our example, and are not amenable to static analysis in the general case.

We use the following signatures for the standard operators of C that occur in GNU "cmp":

TE(skip)	= *unit*	(so we can translate empty statements)
TE(generic-unary-op)	= $\forall t.t \rightarrow t$	(logical "not", negation, bitwise "not")
TE(generic-binary-op)	= $\forall t.t \rightarrow t \rightarrow t$	(addition, subtraction, multiplication, bitwise "and", bitwise "or")
TE(ptr-add)	= $\forall t_1.\forall t_2.t_1 \rightarrow t_2 \rightarrow t_1$	(pointer addition; p+i has the same type as p)
TE(ptr-sub)	= $\forall t_1.\forall t_2.t_1 \rightarrow t_1 \rightarrow t_2$	(pointer subtraction; i=p1-p2)
TE(generic-cmp-op)	= $\forall t_1.\forall t_2.t_1 \rightarrow t_1 \rightarrow t_2$	(equality, inequality, less-than, greater-than, logical "and", logical "or")
TE(scalar-const)	= $\forall t.t$	(constants for int, char, long, etc)

The tuple types do not appear in the typing rules given in the previous section, but they appear in the signatures of the primitive functions that create tuples and extract their elements:

$TE(\text{mk-tuple}_n) = \forall t_1.\forall t_2. \ldots \forall t_n.(t_1 \rightarrow t_2 \rightarrow \ldots t_n \rightarrow (t_1, t_2, \ldots, t_n))$

$TE(\text{elem-tuple}_{nk}) = \forall t_1.\forall t_2. \ldots \forall t_n.(t_1, t_2, \ldots, t_n) \rightarrow t_k$

The mutable variables of C become reference variables in Q. For example:

```
/* File descriptors of the files.  */
static int file_desc[2];
```

becomes

```
let file_desc=ref scalar-const in
(...)
```

---

scalars to be confused in our type inference system), so a cast from a scalar to a reference and back to a scalar will fail to assign both the scalars to the same type. In the second case, we are not able to determine what the pointer is pointing to, and we will always assume it continues to point to the same object or within the same array.

```
static char * my_index (str, chr)
    const char *str;
    int chr;
{
  while (*str)
    {
      if (*str == chr)
        return (char *) str;
      str++;
    }
  return 0;
}
```

Figure 4: C code for the my_index function

```
let my_index=λargs.
  let str=ref (elem-tuple2,1 args) in
  let chr=ref (elem-tuple2,2 args) in
  let result=ref NULL in
    while deref (deref str) do
      (if generic-cmp-op (deref (deref str)) (deref chr)
        then assign result (deref str)
        else assign str (ptr-add (deref str) scalar-const) );
    assign result NULL;
    deref result
in
(...)
```

Figure 3: Translated Q code for the my_index function

This is read as declaring file_desc to be a name for a fresh memory location initially holding a scalar constant (the array size has been ignored). This has the same semantics as the source declaration in C (the C compiler initializes the global variable file_desc to zero).

Function calls with parameters and results are illustrated by the my_index function. The C code is in Figure 4 and the translated code is in Figure 3.

We model the semantics of C, which treats the actual arguments as mutable copies of the formals. The formals are passed to the function in the tuple args. Because Q-functions can only have a single point of exit, we introduce a variable to hold the result (as one would if rewriting the C function to eliminate multiple "return" statements). We do not actually need to translate the control flow effect of the return statement, as it does not affect the types.

The function call

```
my_index (optstring, *nextchar)
```

becomes simply

```
my_index (mk-tuple2 (deref optstring)
                    (deref (deref nextchar)))
```

We implement C "structs" using tuples. The example below is from the C standard I/O library. _iob holds the three FILE objects for standard input, standard output and standard error.

```
extern struct _iobuf {
  int    _cnt;
  char   *_ptr;
  char   *_base;
  int    _bufsiz;
  short  _flag;
  short  _file;
} _iob[3 ];
typedef struct _iobuf FILE;
```

The declaration of _iob is translated to

```
let _iob=ref (mk-tuple7 struct-tag (ref scalar-const)
                (ref NULL) (ref NULL)
                (ref scalar-const)
                (ref scalar-const) (ref scalar-const))
in
(...)
```

The first element of the tuple, "struct-tag", is a special constant that has type "∀t.t". It does not correspond to any of the fields of the C structure. Because the struct-tag can never be directly accessed by the program, two struct-tags will remain distinct unless the structures they belong to are constrained to have identical types. This allows us to disambiguate the cases of two structures happening to have the same type for each corresponding field, and the structures themselves having a direct relationship.

let my_index=λargs.

"args" is given a fresh type variable t0.

let str=ref (elem-tuple$_{2,1}$ args) **in**

The signature of "elem-tuple$_{2,1}$" is instantiated to (t1,t2)->t1. t0 is bound to (t1,t2). The signature of "ref" is instantiated to u1->u1 ref. t1 is bound to u1. "str" is given the type "u1 ref"; no polymorphic generalization is possible because u1 is imperative and "ref (...)" is an expansive expression.

let chr=ref (elem-tuple$_{2,2}$ args) **in**

This is similar to above. "chr" is given the type "u2 ref". t2 was bound to u2.

let result=ref NULL **in**

The signature of "NULL" is instantiated to u3 ref. "result" is given the type u3 ref ref.

while deref (deref str) **do**

The signature of the inner "deref" is instantiated to t3 ref->t3. t3 is bound to u1. The signature of the outer "deref" is instantiated to t4 ref->t4. t4 is bound to u4, and u1 is bound to u4 ref (t4 becomes u4 so that the latter substitution will preserve imperativeness). The derived type of the condition (u4) is ignored.

(**if** generic-cmp-op (deref (deref str)) (deref chr)

The first argument has type u4. The second has type u2. The signature of "generic-cmp-op" instantiates to t5->t5->t6. We bind t5 to u4, then u2 to u4. The derived type of the condition (t6) is ignored.

then assign result (deref str)

The first argument has type u3 ref ref. The second has type u4 ref. The signature of "assign" instantiates to t7 ref->t7->unit. We bind t7 to u3 ref, then bind u3 to u4.

else assign str (ptr-add (deref str) scalar-const));

The first argument of "ptr-add" has type u4 ref. The signature of "scalar-const" is instantiated to t8. The signature of "ptr-add" is instantiated to "t9->t10->t9". t9 is bound to u4 ref and t10 is bound to t8. Then the first argument of "assign" has type u4 ref ref and the second has type u4 ref; the signature of assign is instantiated to t11 ref->t11->unit and t11 is bound to u4 ref. The types of the results of the "then" and "else" branches are both "unit"; with no further binding required, the type of the overall "if" expression is "unit", as required by the "while" statement.

assign result NULL;

The first argument of has type u4 ref ref, and the signature of NULL is instantiated to u5 ref. u5 is bound to u4.

deref result

This expression has type "u4 ref". The overall lambda expression has type "(u4 ref, u4) -> u4 ref". The declaration of "my_index" is non-expansive, since declaring the function does not actually involve any evaluation, so the type of "my_index" is generalized to ∀t4.(u4 ref, u4) -> u4 ref". "str" is type "u4 ref ref", "chr" is type "u4 ref", and "result" is type "u4 ref ref".

in

(...)

Figure 5: Commentary on the analysis of my_index

We choose appropriate signatures for external functions (those for which we do not have the source), such as the standard C library functions. A listing of the signatures we used for our experiment is given in Appendix A.

6 A Simple Example

Figure 5 shows the type inference procedure applied to the translated my_index function of GNU "cmp" (see Figure 4 and Figure 3). The comments describe the progress of the type inference procedure. The names of applicative type variables start with "t", the names of imperative type variables start with "u".

The results show that str, chr and the function result could all refer to the same character values. Furthermore, because "u4" is polymorphic, my_index can operate on different types of character values in when called in different contexts.

7 Our Example: GNU cmp

We examined GNU "cmp" version 2.7, which incorporates the GNU "getopt" library. There are seven C files containing 1733 lines altogether. These were translated into the Q language. The translated Q syntax is written as Standard ML expressions, and we implemented the type inference procedure in Standard ML.

Instead of simply inferring types for all variables at once, we carry out the method on one function at a time, and recover information showing how the types are related between functions. We construct a graph that shows these relationships. When answering particular questions, we can easily prune the graph so that it shows only the relationships for a certain type.

Each node represents a pair consisting of a type and a function (or more generally a module, such as a C file) that the type appears in. The top label in the node identifies the module. Subsequent labels name variables that have the type, or have a component with the type. (There may not be any such variables if the type was associated with a temporary value.) Solid lines indicate that there is a semantic constraint that requires the types in the different modules to be the same. A dotted arrow indicates that the type in the "pointed-to" module is an instantiation of the polymorphic type in the source module (or a reference to such an instantiation, if the arrow is so labelled).

1. The question in the introduction, rephrased as "show the relationships between objects of the same type as file_desc", yields the graph in Figure 6.

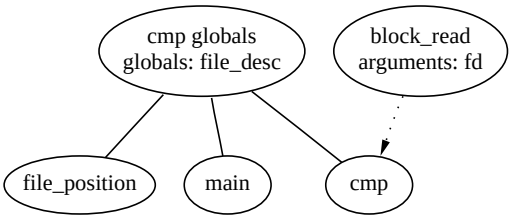


Figure 6: The component of the type graph containing file_desc

   This picture shows the result mentioned in the introduction – that the type is mentioned only in the functions file_position, main, cmp and block_read (cmp_globals is the name of a module that holds the global variables from the file "cmp.c"), and that the only named variables involved are fd and file_desc itself. It also shows that cmp, main and file_position only share the type through cmp_globals, and that cmp passes a value of the type into block_read.

2. Continuing with the theme of changing the file I/O implementation, many other questions arise. We know that currently I/O errors are reported in the C library's global variable errno. Where could those error values propagate to?

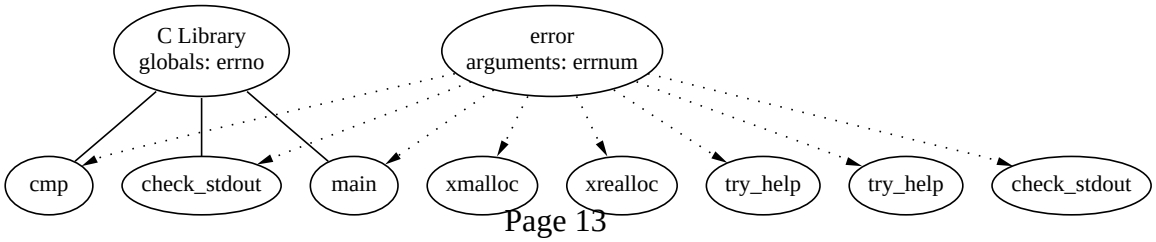


Figure 7: The component of the type graph containing errno

errno is used in main, check_stdout, and cmp. The values are passed into the function error as the errnum parameter. The dotted lines indicate that the error function is polymorphic in that argument; this means that the function could be specialized to handle different types inerrnum, as there is no necessary relationship between the type of errnum at different calls. See Figure 2 for the relevant code.

3. We might wish to change the way we specify positions in a file. A good question might be, "what variables must be the same type as theposition variable of the file_position function?"

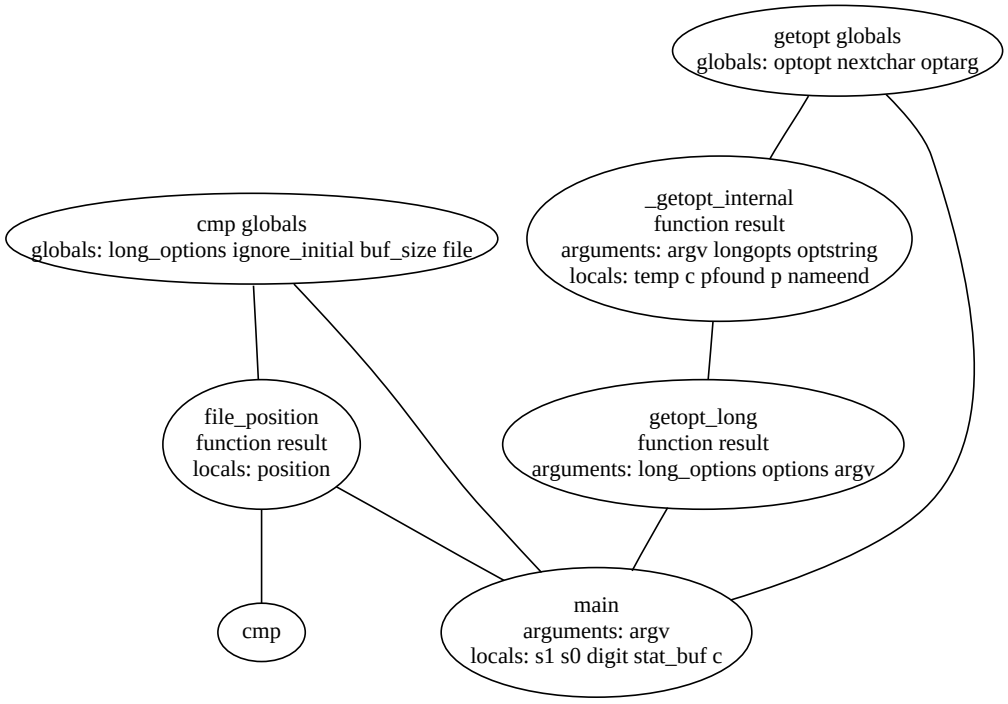


Figure 8: Simplified excerpt from Appendix B

This surprisingly large picture[7] exposes some interesting relationships. Among other things, it shows a connection between the command-line arguments of the program (main:argv) and the position variable! This comes about because one of the command-line arguments is a file offset value; the value is computed directly from the characters of the argument string, and the result is used as a file position. This will obviously cause problems if we alter the meaning of file positions. Conceptually, one way to proceed would be to introduce conversions and abstractions to effectively isolate the file positions as an abstract data type, so that their representation could then be changed in a controlled way. In terms of our diagram, this means introducing

[7] This picture has many nodes and connections omitted; for the complete picture, see Appendix B. That picture was produced with less postprocessing and includes extra information such as the identification of the components of structured variables.

operations to remove connections and break off the file position type into a smaller graph.

8 Current Status and Future Work

The current language Q is deficient in two major ways for expressing general C programs. It does not allow recursive functions, and it does not allow recursive data types, which commonly appear when analyzing pointer-based structures, such as linked lists. Recursive functions are easily added using standard type-theoretic techniques. We plan to exploit existing theory to add recursive types to our type system.

Currently, our prototype tool does not translate C code automatically; it generates a rough translation which must be finished manually. We need to complete the translator so that it translates from C to the enhanced Q language completely automatically. When this is done we can carry out much larger and more interesting case studies.

There is some freedom to choose the new typings for the built-in operators and constants and the external functions; for example, the standard typing is admissible. Of course the more types that can be given to each value with polymorphism, the fewer "incidental" type relationships will appear and the more meaningful the results will be. The appropriate typings may depend on the application. We treat the choice of initial type environment as a parameter of the analysis. The qualitative effects of different typings on the analysis is an open question which must be explored, and may lead to alternative applications for the method.

Currently we are using the "dot" graph-drawing tool to display results broken down by modules. This 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.

Our data and results are available by anonymous FTP in
ftp://ftp.cs.cmu.edu/afs/cs.cmu.edu/user/roc/public/DSRUTI.paper/

9 Related Work

We have addressed the question of determining when, because of transfers of values, variables must share a common representation.

It helps to think of our method in terms of abstract data types. The soundness of the type inference system means that if we had dynamic types, so that each value was tagged with its type *in our inferred scheme*, then each variable would always contain values whose dynamic type tag corresponded to the static type we give the variable. This means that the representation of the type is never propagated beyond the variables with the type; each type we infer is a candidate for being made an abstract data type.

Bowdidge and Griswold introduced a "star diagram" tool to aid in encapsulating abstract data types [BG94]. We do not explicitly deal with issues of encapsulation or abstraction, because we do not attempt to distinguish accesses that are exposures of representation from those that are not. Their tool would be quite complementary to ours. In fact, in their

paper, they discuss extending their method for encapsulating existing data structures to operate on data structures with more than one instance. 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.

Our method is very similar in flavour to a technique used by programming language researchers known as “region inference”. For example, in [TT94] Tofte and Taplin use region inference as part of a method for implementing simple functional languages more space-efficiently. The store is (conceptually) partitioned into distinct “regions”, and each reference value is associated with a region. Reference variables are statically assigned to regions in the same way that we classify variables as belonging to types. A key difference is that we assign types to all values and variables, whereas region inference only applies to references. Of course, we also have quite different applications in mind.

Other researchers have recently been investigating type inference methods for inferring properties of programs in commonly-used languages such as C. In [S96] Steensgaard presents a method based on type inference that yields an almost-linear time “points-to” analysis algorithm. His type system is customized for his application; it lacks polymorphism and is focused on pointer types.

The task of detecting value transfers is commonly addressed by dataflow methods such as slicing [Wei84]. Both methods compute relationships between program objects. Slicing is more fine-grained; it computes a directed “affects” relationship between sites (a site is a variable at a point in the program), but we compute an undirected “same type as” relationship between variables only. This means that our method is potentially much cheaper, because there is much less information to deal with, but the greater accuracy of slicing may be required for many applications. Also, for a slicing method to answer the question as posed, multiple slices must be computed and combined. Slicing tools exist that allow this to be done manually [NEK94].

When used to detect shared representations, standard slicing methods introduce spurious “control dependence” relationships that must be suppressed. This is because the “affects” relationship used in slicing normally means something quite different from our “same type as” relationship. One way of interpreting the “affects” relationship is that a change in a variable when the program is at one location may result in another variable being different when the program arrives at another location. This interpretation requires that the “affects” relationship propagate through control dependencies, so that, for example, if a variable is modified in one arm of an “if” statement, then it is affected by the variables used in the “if” predicate. The “same type as” relationship ignores such control dependencies, but it captures some dataflow relationships, in the sense that if a value is transmitted from one variable to another by copying, or by sharing through references, then the two variables must have the same type. Dataflow relationships induced by basic operators or external functions can be captured by our method, depending on the signatures given to those operators or functions in the type environment.

The granularity of an analysis affects the nature of the results. When results are computed at the procedure level (or higher) instead of the statement level, as with the Chopshop slicing tool [JR94], it is much easier to deal with large systems. We compute information for each variable, so our results are best viewed at the procedure level or above. It is worth noting that our method is fundamentally interprocedural (indeed, it handles higher-order functions with ease), as opposed to interprocedural slicing methods that typically require an extra layer of analysis [HRB90, RHSR94].

Muller et al. [MTOCM92] 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, which examines procedure calls and global variable accesses, but we could certainly benefit from such visualization and manipulation techniques.

Acknowledgements

We would like to thank Robert Harper for his helpful advice.

References

[BG94] 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

[HRB90] Susan Horwitz, Thomas Reps and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1), January 1990, pp. 26-60.

[JR94] Daniel Jackson and Eugene Rollins. Abstractions of program dependences for reverse engineering. *Proc. ACM SIGSOFT Conf. On Foundations of Software Engineering*, New Orleans, December 1994

[MTOCM92] 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 Devleopment Environments* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88-98, December 1992. In *ACM Software Engineering Notes*, 17(5).

[NEK94] Jim Q. Ning, Andre Engberts and Wojtek Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5), May 1994, pp. 50-57.

[RHSR94] Thomas Reps, Susan Horwitz, Mooly Sagiv and Genevieve Rosay. *Speeding up slicing*. Technical report D-214, Datalogisk Institut, University of Copenhagen, 1994.

[S96] Bjarne Steensgaard. Points-to analysis in almost linear time*.Proceedings*

of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 1996 (to appear, draft version sighted).

[Tof87] Mads Tofte. Operational Semantics and Polymorphic Type Inference (PhD thesis), technical report ECS-LFCS-88-54, Department of Computer Science, University of Edinburgh, Edinburgh, May 1988.

[TT94] 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.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4), July 1984, pp.352-357.

Appendix A

These are the signatures we assign to the C library functions used by GNU "cmp".

TE(malloc)	= $\forall t.\forall u.(t) \rightarrow u$ ref	(memory allocation)
TE(realloc)	= $\forall t.\forall u.(u$ ref, $t) \rightarrow u$ ref	(adjust memory block size)
TE(free)	= $\forall t.\forall r.(t$ ref$) \rightarrow r$	(release memory)
TE(__builtin_next_arg)	= $\forall t.(t$ ref$) \rightarrow t$ ref	(GCC internal function for var-arg access)
TE(strerror)	= $\forall t.\forall r.(t) \rightarrow r$ ref	(converts error code to error string)
TE(printf)	= $\forall t.\forall r.(t$ ref$) \rightarrow r$	(formatted output)
TE(exit)	= $\forall t.\forall r.(t) \rightarrow r$	(exit program)
TE(getenv)	= $\forall t.\forall r.(t$ ref$) \rightarrow (r$ ref$)$	(get string from process environment)
TE(open)	= $\forall t.\forall a.\forall b.\forall r.(t$ ref, a, b$) \rightarrow r$	(open file syscall)
TE(write)	= $\forall t.\forall a.\forall b.\forall r.(a, b$ ref, t$) \rightarrow r$	(write file syscall)
TE(read)	= $\forall t.\forall a.\forall b.\forall r.(a, b$ ref, t$) \rightarrow r$	(read file syscall)
TE(lseek)	= $\forall t.\forall a.\forall b.\forall r.(a, t, b) \rightarrow r$	(seek file syscall)
TE(close)	= $\forall t.\forall r.(t) \rightarrow r$	(close file syscall)
TE(strlen)	= $\forall t.\forall r.(t$ ref$) \rightarrow r$	(length of string)
TE(strcmp)	= $\forall t.\forall r.(t$ ref, t ref$) \rightarrow r$	(compare strings)
TE(strncmp)	= $\forall t.\forall a.\forall r.(t$ ref, t ref, a$) \rightarrow r$	(compare strings, restricted length)
TE(fprintf)	= $\forall r.$**F**((**FILE** ref, Fbuf ref$) \rightarrow r$)	(formatted output to file)
TE(vfprintf)	= $\forall a.\forall r.$**F**((**FILE** ref, Fbuf ref, a ref$) \rightarrow r$)	(formatted output to file)
TE(_flsbuf)	= $\forall r.$**F**((Fbuf, **FILE** ref$) \rightarrow r$)	(write char to file and flush)
TE(fflush)	= $\forall r.$**F**(**FILE** ref $\rightarrow r$)	(flush file buffers)
TE(fclose)	= $\forall r.$**F**(**FILE** ref $\rightarrow r$)	(close file)
TE(fstat)	= $\forall t.\forall r.$**S**((t, **STAT** ref$) \rightarrow r$)	(get info on an open file)
TE(stat)	= $\forall t.\forall r.$**S**((t ref, **STAT** ref$) \rightarrow r$)	(get info on a file by name)

where

F(x) = $\forall$Ftrack.$\forall$Fsize.$\forall$Fbuf.$\forall$Fflag.$\forall$Ffile.x (Type vars for FILE structure fields)

S(x) = $\forall$Strack.$\forall$Sdev.$\forall$Sino.$\forall$Smode.$\forall$Snlink.$\forall$Suid.$\forall$Sgid.$\forall$Srdev.$\forall$Ssize.$\forall$Sspare.
$\forall$Stime.$\forall$Sblksize.$\forall$Sblocks.$\forall$Sgennum.x (Type vars for stat_buf structure fields)

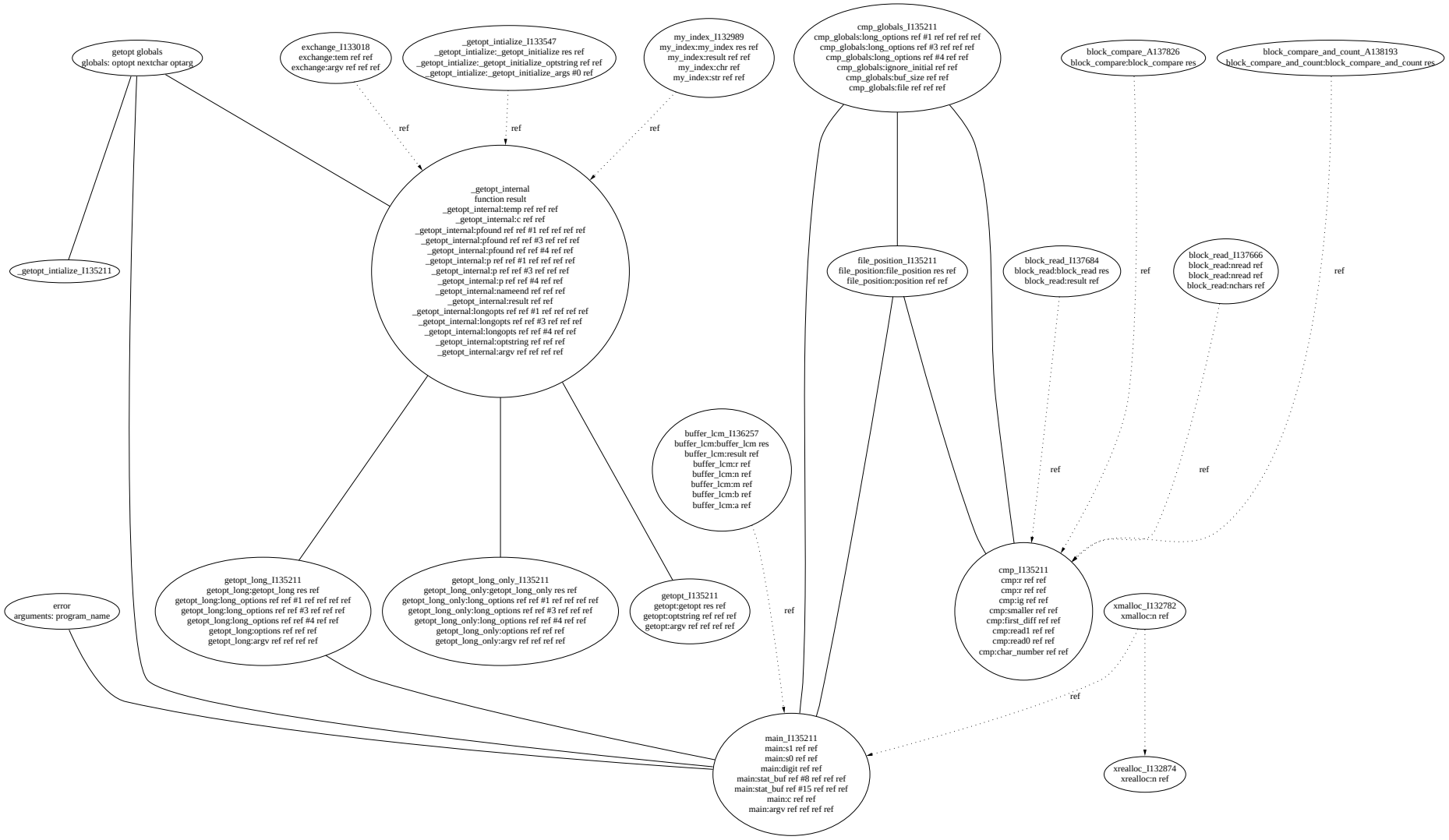
FILE = (Ftrack, Fsize ref, Fbuf ref ref, Fbuf ref ref, Fsize ref, Fflag ref, Ffile ref)

STAT = (Strack, Sdev, Sino, Smode, Snlink, Suid, Sgid, Srdev, Ssize, Stime, Sspare, Sblksize, Sblocks, Sgennum)

We declare the C library's global variables explicitly:

```
let _iob=ref (mk-tuple7 struct-tag (ref scalar-const) (ref NULL)
        (ref NULL) (ref scalar-const) (ref scalar-const)
        (ref scalar-const))
in let _pctype=ref NULL
in let errno=ref scalar-const
in
(...)
```

Appendix B


getopt globals
globals: optopt nextchar optarg
exchange_I133018
exchange:tem ref ref
exchange:argv ref ref ref
_getopt_intialize_I133547
_getopt_intialize:_getopt_initialize res ref
_getopt_intialize:_getopt_initialize_optstring ref ref
_getopt_intialize:_getopt_initialize_args #0 ref
my_index_I132989
my_index:my_index res ref
my_index:result ref ref
my_index:chr ref
my_index:str ref ref
cmp_globals_I135211
cmp_globals:long_options ref #1 ref ref ref ref
cmp_globals:long_options ref #3 ref ref ref
cmp_globals:long_options ref #4 ref ref
cmp_globals:ignore_initial ref ref
cmp_globals:buf_size ref ref
cmp_globals:file ref ref ref
block_compare_A137826
block_compare:block_compare res
block_compare_and_count_A138193
block_compare_and_count:block_compare_and_count res
ref
_getopt_internal
function result
_getopt_internal:temp ref ref ref
_getopt_internal:c ref ref
_getopt_internal:pfound ref ref #1 ref ref ref ref
_getopt_internal:pfound ref ref #3 ref ref ref
_getopt_internal:pfound ref ref #4 ref ref
_getopt_internal:p ref ref #1 ref ref ref ref
_getopt_internal:p ref ref #3 ref ref ref
_getopt_internal:p ref ref #4 ref ref
_getopt_internal:nameend ref ref ref
_getopt_internal:result ref ref
_getopt_internal:longopts ref ref #1 ref ref ref ref
_getopt_internal:longopts ref ref #3 ref ref ref
_getopt_internal:longopts ref ref #4 ref ref
_getopt_internal:optstring ref ref ref
_getopt_internal:argv ref ref ref ref
_getopt_intialize_I135211
file_position_I135211
file_position:file_position res ref
file_position:position ref ref
block_read_I137684
block_read:block_read res
block_read:result ref
block_read_I137666
block_read:nread ref
block_read:nread ref
block_read:nchars ref
buffer_lcm_I136257
buffer_lcm:buffer_lcm res
buffer_lcm:result ref
buffer_lcm:r ref
buffer_lcm:n ref
buffer_lcm:m ref
buffer_lcm:b ref
buffer_lcm:a ref
getopt_long_I135211
getopt_long:getopt_long res ref
getopt_long:long_options ref ref #1 ref ref ref ref
getopt_long:long_options ref ref #3 ref ref ref
getopt_long:long_options ref ref #4 ref ref
getopt_long:options ref ref ref
getopt_long:argv ref ref ref ref
getopt_long_only_I135211
getopt_long_only:getopt_long_only res ref
getopt_long_only:long_options ref ref #1 ref ref ref ref
getopt_long_only:long_options ref ref #3 ref ref ref
getopt_long_only:long_options ref ref #4 ref ref
getopt_long_only:options ref ref ref
getopt_long_only:argv ref ref ref ref
getopt_I135211
getopt:getopt res ref
getopt:optstring ref ref ref
getopt:argv ref ref ref ref
cmp_I135211
cmp:r ref ref
cmp:r ref ref
cmp:ig ref ref
cmp:smaller ref ref
cmp:first_diff ref ref
cmp:read1 ref ref
cmp:read0 ref ref
cmp:char_number ref ref
xmalloc_I132782
xmalloc:n ref
error
arguments: program_name
main_I135211
main:s1 ref ref
main:s0 ref ref
main:digit ref ref
main:stat_buf ref #8 ref ref ref
main:stat_buf ref #15 ref ref ref
main:c ref ref
main:argv ref ref ref ref
xrealloc_I132874
xrealloc:n ref