Determining Interfaces using Type Inference

Stephen A. Edwards* and Chun Li

Columbia University, Computer Science Department
New York, USA
sedwards@cs.columbia.edu

Abstract. Porting software usually requires understanding what library functions the program being ported uses since this functionality must be either found or reproduced in the ported program's new environment. This is usually done manually through code inspections. We propose a type inference algorithm able to infer basic information about the library functions a particular C program uses in the absence of declaration information for the library (e.g., without header files). Based on a simple but efficient inference algorithm, we were able to infer declarations for much of the PalmOS API from the source of a twenty-seven-thousand-line C program. Such a tool will aid in the problem of program understanding when porting programs, especially from poorly-documented or lost legacy environments.

1 Introduction

Good software inevitably outlasts hardware and even operating systems, so there will always be a need to port software between environments. The challenges of porting software, which we define as reproducing the behavior of a program in a new environment[1], are legion, but most boil down to mismatches between assumptions in the original source code and the new environment. These assumptions take two forms. Expectations about the programming language are the most fundamental. For example, the expression "a = 1" is an assignment in C and a comparison in ML. The other expectation is about the existence and behavior of external entities such as libraries and operating systems. This paper proposes an analysis technique that helps to identify these expectations as a prelude to porting.

We propose a technique that infers the interfaces to C library functions from how they are used. It is designed for when header files declaring the functions are not available, although it also performs the useful function of identifying which functions could be called and therefore must be considered during a port. Our goal is to bring a more formal methodology to the porting process by quickly providing an abstract summary of the assumptions made by the source program.

* Edwards and his group are supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State's NYSTAR program.

[1] In particular, we do not consider substantial algorithmic changes such as those required to "port" a compiler to a new processor.

Armed with this information, the person responsible for the port can begin evaluating how to proceed, e.g., by writing new library functions or wrappers for library functions in the new environment.

The information supplied by our technique is most helpful when the program being ported uses only a small fraction of a very large library that does not exist in the new environment and must be recreated. By identifying which parts of the library are actually used, our technique help a programmer avoid having to recreate the entire library. For example, suppose a program that uses the svgalib library (a medium-sized library with over 100 calls) is being ported to a PDA that does not have it. Our tool can quickly report, for example, that the program calls *vga_drawline*, that *vga_drawline* has four integer arguments, and that it does not need *vga_waitretrace*.

```
f(g());          extern void f(int);
h(g());          extern int  g(void);
f(4);            extern void h(int);
```

(a) (b)

Fig. 1. (a) Three function calls. (b) The declarations inferred by our type inference procedure.

Our algorithm performs type inference through deduction. Consider the three function calls in Figure 1a. The first call suggests the function f takes an integer argument, although C's promotion rules would also allow it to take a different numerical type. Assuming this is true, the second call then suggests g returns an integer, so that the third call suggests h takes a single integer argument. Using such reasoning, our procedure produces the declarations shown in Figure 1b.

2 The Type Inference Procedure

Our procedure is based on the ideas of type inference used in, for example, ML [1,2], but deviates from this ideal because of details in C's type system, notably implicit conversions and functions with a variable number of arguments.

C's basic types are fluid and expressions involving these types often imply conversions, typically promotions. For example, a *char* is automatically promoted to an *int* whenever it "interacts" with an *int*, such as when a *char* and an *int* are added. Similarly, adding an *int* to a *float* causes both to be promoted to *double*. Our algorithm addresses this problem by assuming the more general type when faced with two possibilities for a particular argument.

C's somewhat ad-hoc support for variable-length argument lists also presents a challenge. In C, it is legal to call a function with argument lists of completely different types and lengths provided the function is declared with ellipses in its argument list.

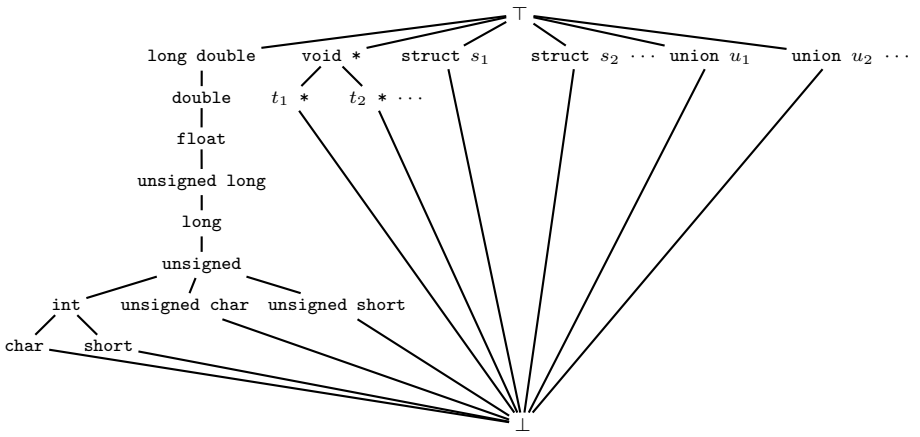

Fig. 2. The Hasse diagram for C's basic types, assuming *char* is signed and *long int* is larger than *unsigned*. The symbols t_1 and t_2 represent arbitrary non-void types (e.g., *int*s, *struct*s, etc.), and the s and u represent arbitrary *struct*s and *union*s.

2.1 Type Lattices

After a careful reading of the C language specification, we adopted the ordering of C's types depicted in Figure 2. This diagram was constructed so the least common ancestor of two nodes is the most restrictive type that can capture them both (i.e., the join operation, written $\sqcup$). For example, if a function f is called in one place with a single *unsigned short* argument and in another with a single *char* argument, the least common ancestor of these two types is *unsigned*, which we use as the guess for the true type of its argument because it is the most restrictive type to which both *char* and *unsigned short* can be converted.

With the exception of *void **, all pointer types are treated as distinct. This follows directly from a statement in the C reference manual, "when both operands [of the = operator] are pointers of any kind, no conversion ever takes place." Similarly, although it is safe to cast a pointer, for example,

from
```
struct foo {
  int a;
  float b;
  int c;
}
```
to
```
struct bar {
  int a;
  float b;
}
```

C considers an assignment from such a pointer type to another to be erroneous.

Two members of this lattice are special. The $\bot$ element represents an unknown type, which we use to represent arguments whose types we have not yet deduced. The $\top$ element represents multiple, conflicting types for an argument, which might arise if an one-argument function is called with a number at one site and a pointer at another. Thus, arguments passed through ellipses ("...") such as the second and later arguments to *printf*, can be represented with $\top$.

The type of a function is simply a vector of basic types taken from the lattice in Figure 2. To perform our type inference procedure, we need to define the join operation on two function types. This operation answers the following question: given two call sites for a function f and the types of its arguments at both sites, we want to find the most conservative guess for types of the arguments of f.

For two argument lists of the same size, the type of each argument is the join of the corresponding argument types in the two argument lists. For example,

$$(\texttt{int}, \texttt{int}, \texttt{char *}) \sqcup (\texttt{int}, \texttt{double}, \texttt{int *}) = (\texttt{int}, \texttt{double}, \texttt{void *})$$
$$(\bot, \texttt{float}, \texttt{char *}) \sqcup (\texttt{long}, \texttt{double}, \texttt{int}) = (\texttt{long}, \texttt{double}, \top).$$

The third argument in the second example is odd: is it ever correct to pass a function a *char ** in one case and *int* in another? It is, provided the function declaration contains an ellipsis (i.e., "..."). In C, the signature resulting from the second example would mean *(long,double,...)*.

One final possibility remains: two calls to the same function with a different number of arguments. Again, this is permissible with variable argument lists. Our solution is to $\top$-extend the shorter argument list to the length of the longer one and then perform a normal component-wise join operation. We write

$$(\texttt{int}, \texttt{int}) \sqcup (\texttt{int}, \texttt{double}, \texttt{int}) = (\texttt{int}, \texttt{double}, \top)$$

This rule always produces signatures of the form

$$(t_1, \ldots, t_k, \top, \ldots, \top).$$

We interpret the first k non-$\top$ arguments as usual C types and the first $\top$ as representing an ellipsis, i.e., as a variable-argument function.

```
f(g());
h('a', g());
h('b', q());
i(q());
f(1);
```

(a)

```
extern void f(int);
extern void h(char, int);
extern void i(int);
extern int  q(void);
extern int  g(void);
```

(b)

Fig. 3. (a) Five function calls with a transitive constraint. (b) The declarations our procedure infers.

2.2 The Unification Algorithm

We employ a straightforward relaxation-based type unification algorithm built around the join operation described earlier. The main challenge is dealing with transitive constraints, such as those presented by the example in Figure 3. We

```
1: procedure resolve-all-types
2:     Clear all type signatures
3:     Clear all constraints
4:     for each call t_0 = f(t_1, ..., t_k) of function f do
5:         if there is no type signature for f then
6:             Set f's signature to t_0 = f(t_1, ..., t_k)
7:         else
8:             Let u_0 = f(u_1, ..., u_j) be the signature for f
9:             if t_0 exists (i.e., the result of f is assigned) then
10:                u_0 = u_0 ⊔ t_0
11:            if k > j then
12:                Set u_{j+1}, ..., u_k = ⊤
13:                Set j = k
14:            else if k < j then
15:                Set t_{k+1}, ..., t_j = ⊤
16:            for i = 1, ..., j do
17:                Set u_i = u_i ⊔ t_i
18:                if u_i = ⊥ and t_i is the return type of g then
19:                    Add a constraint f(g)
20:    for each function f do
21:        resolve-arguments(f)

22: procedure resolve-arguments(f)
23:    Let u_0 = f(u_1, ..., u_j) be the current signature for f
24:    for i = 1, ..., j do
25:        for each g that is ever an ith argument of f do
26:            u_i = u_i ⊔ return-type(g)
27:            Set the return type of g to u_i

28: function return-type(g)
29:    for each constraint of the form f(g) do
30:        Remove the constraint f(g)
31:        resolve-arguments(f)
32:    return the current guess for the return type of g
```

Fig. 4. The unification algorithm.

handle this by maintaining a graph of such dependencies and traversing it as part of the relaxation procedure.

Figure 4 shows our type unification algorithm, which operates in two phases. The first phase, lines 2–19, handles all the simple inferences and prepares the constraints for the second phase, which consists of the loop in lines 20–21, the *resolve-arguments* procedure and the return-type function.

The first pass starts by initializing all type signatures to effectively nothing and clearing all constraints. Such constraints, created in line 19, indicate that the result of a function g is passed as an argument to a function f, thereby constraining the return type of g to match the type of the argument. We write such a constraint $f(g)$, where f and g are function names.

The main loop of the first pass, lines 4–19, iterates over each call of an undeclared function (in the sequel, only undeclared functions will be considered) and guesses the types of its arguments based on the actual parameters passed to it. If the function has not been seen before, the guess is initialized to have the types of arguments to the call (line 6). Otherwise, if the result of the call is assigned to a variable, that type is used to refine the guess of the function's return type (lines 9–10).

Next, if the function has been seen before and the number of arguments passed at the current call (k) exceeds how many were seen before, the function must necessarily have a variable number of arguments so the current guess is extended with $\top$s out to k (lines 11–13). Another possibility—that the current call has fewer arguments than previously seen—is handled in lines 14–15 by $\top$-extending the call arguments.

When control reaches the *for* loop in lines 16–19, the number of arguments in the current signature and the call match. In line 17, the loop performs the join between the types of the actual arguments and those in the signature. Finally, if the type is still unresolved (i.e., the join was $\bot$ and the type is the return type of a function g, a constraint is added that indicates that the return type of g depends on the type of some argument to f.

This completes the first pass, which resolves all simple types but not transitive constraints. Running the procedure to this point on the example in Figure 1a gives the signatures $\bot = f(\text{int})$, $\bot = h(\bot)$ and $\bot = g()$. Similarly, the signatures for Figure 3a would be $\bot = f(\text{int})$, $\bot = g()$, $\bot = h(\text{char}, \bot)$, and $\bot = i(\bot)$.

The second pass resolves type signatures that derive from constraints, such as that for h in Figure 1a. It consists of the loop in lines 20–21, which resolves the arguments for every undeclared function by calling *resolve-arguments*, which calls itself recursively through the *return-type* function to establish the return types of any functions that ever appear as arguments.

The *resolve-arguments* procedure steps through each of the arguments to f (the highest number of arguments ever seen has been resolved by this point) and computes the join of the return types of all the functions that ever appear as arguments (line 26). It calls *return-type* to compute these, which recursively calls *resolve-arguments* to handle complex cases such as the one in Figure 3a.

The *return-type* function calls *resolve-arguments* for each function that ever takes the function g as an argument (i.e., that constraints the return type of g). It removes the constraint first to avoid circular dependencies.

For the example in Figure 1, when *resolve-arguments* is first called on function f, the type signature from the first pass is $\perp = f(\text{int})$. The loop in lines 25–27 only finds one function that is an argument to f, namely g. So *return-type*(g) is called in line 26. *Return-type* finds the constraint $f(g)$, removes it, and calls *resolve-arguments*(f) again, the second invocation of *resolve-arguments*(f). The second one also calls *return-type*(g), but because the $f(g)$ constraint was removed, *return-type* immediately returns $\perp$, a guess for the return type of g.

After *return-type*(g) returns $\perp$, it computes the join of int and $\perp$ in line 26, giving int, and assigns it to the return type of g in line 27. The signature of g is now int $= g()$.

3 Experimental Results

We implemented our system on top of Necula et al.'s CIL [3], a C front end written in OCAML designed for analysis and source-to-source translations.

We tested our tool on two real-world programs: iSSL, an open-source network cryptographic service[2], and Pilot-DB[3], an open-source database system for PalmOS. These were chosen because we had prior experience with iSSL [4] and also wanted something designed for a different operating system (i.e., one for which we did not have the header files). The iSSL distribution consists of roughly 13 000 lines of source code, and Pilot-DB contains roughly 27 000 lines.

The entire analysis, including two invocations of the C preprocessor to identify missing header files (an inefficient technique that should be replaced with a modified version of the preprocessor that does not fail on missing #include files), took less than three seconds for the Pilot-DB program and less than two for iSSL on a 1.7 GHz Pentium 4 machine running Linux.

The iSSL package compiles fine on our Linux machine, so to test our algorithms we removed the multi-precision arithmetic library and headers from the iSSL code (it was taken from GNUPG, which had adapted it from the GNU MP library) and asked our tool to reconstruct the prototypes. Figure 5 illustrates the results. Our procedure was able to deduce most prototypes correctly from context; a few differences came from assuming an integer instead of an unsigned long. Another interesting effect: *mpi_get_nlimbs* is implemented as a macro, but our tool inferred the types of its argument and return type. The few other differences arose from things like *typedef*s (struct _IO_FILE * instead of FILE *) and const char *s.

The Pilot-DB example was more interesting. We did not have the PalmOS header files when we ran our tool, so we did not need to modify its source. Not

[2] The author of iSSL is a German who refers to himself (herself?) as "Mixter." The sources reside on sourceforge.net.

[3] Originally by Tom Dyas, now developed and maintained by Marc Chalain and Scott Wallace. The source is also on sourceforge.net

```
MPI mpi_alloc_set_ui(int);
void mpi_set(MPI, MPI);
void mpi_mul_ui(MPI, MPI, int);
void mpi_mul(MPI, MPI, MPI);
void mpi_add_ui(MPI, MPI, unsigned int);
unsigned int mpi_get_nbits(MPI);
void log_mpidump(char * , MPI);

MPI mpi_copy(MPI);
unsigned int mpi_get_nlimbs(MPI);
void mpi_sub_ui(MPI, MPI, int);
void mpi_set_ui(MPI, int);
void mpi_print(struct _IO_FILE *, MPI, int);
```

(a)

```
MPI mpi_alloc_set_ui(unsigned long u);
void mpi_set(MPI w, MPI u);
void mpi_mul_ui(MPI w, MPI u, ulong v);
void mpi_mul( MPI w, MPI u, MPI v);
void mpi_add_ui(MPI w, MPI u, ulong v );
unsigned mpi_get_nbits( MPI a );
void g10_log_mpidump( const char *text, MPI a );
#define log_mpidump g10_log_mpidump
MPI  mpi_copy( MPI a );
#define mpi_get_nlimbs(a)     ((a)->nlimbs)
void mpi_sub_ui(MPI w, MPI u, ulong v );
void mpi_set_ui( MPI w, ulong u);
int mpi_print( FILE *fp, MPI a, int mode );
```

(b)

Fig. 5. (a) Some of the prototypes inferred by our tool from the iSSL source. (b) The corresponding declarations from the actual header file. A few arguments are actually *unsigned long*s where our tool tought they were *int*s. Also, *mpi_get_nlimbs* is actually a macro—our tool assumed it was a function and produced a prototype for it.

```
UInt16 TblGetRowHeight(void *,
       int);
void   WinDrawLine(Coord, Coord,
                   Coord, Coord);
UInt16 TblGetLastUsableRow(TablePtr);
UInt16 TblGetRowID(...);

void   FrmUpdateScrollers(FormPtr___0,
         UInt16, UInt16,
         Boolean,
         Boolean);
UInt16 TblGetNumberOfRows(TablePtr);
void   TblSetRowUsable(TablePtr, UInt16,
                       int);
void   TblSetRowSelectable(TablePtr,
                           UInt16,
                           int);
void   TblSetRowHeight(TablePtr, UInt16,
                       int);
```

(a)

```
Coord  TblGetRowHeight(const TableType *tableP,
                       Int16 row);
void   WinDrawLine(Coord x1, Coord y1,
                   Coord x2, Coord y2);
Int16  TblGetLastUsableRow(const TableType *tableP);
UInt16 TblGetRowID(const TableType *tableP,
                   Int16 row);
void   FrmUpdateScrollers(FormType *formP,
          UInt16 upIndex, UInt16 downIndex,
          Boolean scrollableUp,
          Boolean scrollableDown);
Int16  TblGetNumberOfRows(const TableType *tableP);
void   TblSetRowUsable(TableType *tableP, Int16 row,
                       Boolean usable);
void   TblSetRowSelectable(TableType *tableP,
                           Int16 row,
                           Boolean selectable);
void   TblSetRowHeight(TableType *tableP, Int16 row,
                       Coord height);
```

(b)

Fig. 6. (a) Some prototypes inferred from the Pilot-DB source. (b) The true declarations from the Palm OS SDK header files. Some inconsistencies stem from sloppily-written code.

surprisingly, the program makes extensive use of PalmOS GUI calls, a few of which are listed in Figure 6. Our program largely succeeded at inferring the right types, but there are a few inconsistencies.

The true first argument of *TblGetRowHeight* is a pointer to TableType, but our tool inferred a void *. The culprit was a function in which the developers were type-sloppy:

```
static void
DrawPopupIndicator(void * table, Int16 row, Int16 col, RectangleType *b)
{
  /* ... */
  y = b->topLeft.y + (TblGetRowHeight(table, row) - (width / 2)) / 2;
```

We also inferred a seemingly incorrect return type for *TblGetRowHeight*, but it turns out that PalmOS.h *typedef*s the Coord type as Int16. It inferred UInt16, though, again because of poorly chosen types in another file. Figure 2 shows that unsigned dominates signed.

```
UInt16 oldHeight;
/* .. */
oldHeight = TblGetRowHeight(table, row);
```

Our procedure inferred a variable number of arguments for *TblGetRowID*, which is clearly a mistake. The source of the problem is an incomplete type, i.e., a function passes a field from an incomplete type to *TblGetRowID*:

```
Boolean SortEditor_HandleEvent(EventPtr event)
{
 /* .. */
 UInt16 entry = TblGetRowID(event->data.tblSelect.pTable,
                            event->data.tblSelect.row);
```

Our tool currently does not infer the type of *EventPtr* (a pointer to a struct containing *data.tblSelect.pTable*), and as such, misinterprets it to the point where no information is returned about the function's arguments.

These results suggest another application of our tool: as a lint-like checker that flags suspicious implicit typecasts.

4 Related Work

Our type inference algorithm follows in the footsteps of Milner's work on ML [1,2]. C's type system, of course, is much less disciplined and as a result, our type inference system is not exact. Nevertheless, our procedure does produce useful information.

The AnnoDomini system [5] uses reasoning similar to ours to both identify and modify COBOL source code that suffers from the Y2K problem (i.e., representing years as two decimal digits).

O'Callahan and Jackson's Lackwit [6] tool bears the closest resemblance to ours, but they have a different objective. They, too, use type inference, but their goal is to catch type-related errors, such as adding a number representing a weight to a height.

Wright and Cartwright [7] similarly use type inference for checking the correctness of a program, but their goal is mainly to eliminate run-time type checks in an otherwise dynamically-typed language—Scheme in their case.

Van Deursen and Moonen [8] apply type inference to the problem of understanding COBOL programs, which have the unique problem of not actually defining types (e.g., to declare the equivalent of two variables of the same structural type, the details of their structure must be repeated). They use observations about how variables are actually used to infer what types the program actually uses to assist people maintaining programs.

Ramalingam et al. [9] also apply type inference to COBOL programs to eliminate unused fields in aggregate structures, but also strive to identify when otherwise unrelated variables are manipulated as a group.

The inspiration for this work came from our earlier experience with porting iSSL to an embedded eight-bit processor [4]. It turned out that understanding exactly which facilities the source code was using was fairly difficult and ultimately surprising. For example, we were surprised to find that an apparently pure networking application wanted to use a filesystem (for an unwanted logging feature, it turned out) and it took a while to enumerate all such dependencies. An automatic tool would have been very useful.

5 Conclusions and Future Work

We presented a practical tool able to infer useful function prototype information from C source code whose environment was missing. As a side effect, our procedure identifies what part of an API a program actually uses, a useful piece of information when a developer is faced with having to reimplement a library in a new environment.

Header files often contain more than just function type declarations. In particular missing type declarations (i.e., *typedef*s) play havoc on most C front ends. Because of C's odd declaration syntax, most front ends, including CIL's, require type names to be known at parsing time. As a result, missing type information usually causes a non-recoverable syntax error that makes the front end terminate prematurely.

We worked around this problem by simply creating a header file that included dummy definitions for all the missing types that were causing syntax errors, but this is not an adequate solution in general. In the future, we intend to modify the C front end to allow missing types to be inferred as well, perhaps using some of the techniques developed by Van Deursen and Moonen [8] and Ramalingam et al. [9] to infer aggregate structures. A very difficult problem would be to distinguish between C structs and unions, since they are syntactically similar and only differ semantically.

Another obvious direction for future work would be applying these same ideas to object-oriented languages such as C++ and Java. C++'s ability to overload functions and operators promises to make this a difficult problem.

Finally, we plan to explore extracting more information about the behavior of the library elements, such as observing the order in which particular functions can be invoked. This would also have applications to program verification—a too-liberal model of invocation order would suggest a bug.

References

1. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
2. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico (1982) 207–212
3. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Proceedings of the International Conference on Compiler Construction (CC). Volume 2304 of Lecture Notes in Computer Science., Grenoble, France (2002) 212–228
4. Jan, S., de Dios, P., Edwards, S.A.: Porting a network cryptographic service to the RMC2000: A case study in embedded software development. In: Designers' Forum: Design Automation and Test in Europe Conference and Exhibition, Munich, Germany (2003) 150–155
5. Eidorff, P.H., Henglein, F., Mossin, C., Niss, H., Sørensen, M.H., Tofte, M.: AnnoDomini: from type theory to Year 2000 conversion tool. In: ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas (1999) 1–14
6. O'Callahan, R., Jackson, D.: Lackwit: A program understanding tool based on type inference. In: Proceedings of the International Conference on Software Engineering (ICSE), Boston, Massachusetts (1997) 338–348
7. Wright, A.K., Cartwright, R.: A practical soft type system for Scheme. ACM Transactions on Programming Languages and Systems **19** (1994) 87–152
8. van Deursen, A., Moonen, L.: Understanding COBOL systems using inferred types. In Woods, S., ed.: Proceedings of the 7th International Workshop on Program Comprehension, Pittsburgh, Pennsylvania, IEEE Computer Society Press (1999)
9. Ramalingam, G., Field, J., Tip, F.: Aggregate structure identification and its application to program analysis. In: ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas (1999) 119–132