A Methodology for Decompilation *

Cristina Cifuentes
cifuente@fitmail.qut.edu.au

K.John Gough
gough@fitmail.qut.edu.au

School of Computing Science
Queensland University of Technology
GPO Box 2434, Brisbane, QLD 4001, Australia

Abstract

A proposed methodology for decompilation of binary programs is presented, along with a description of a particular implementation of this methodology, dcc. dcc *is a decompiler for the Intel 80x86 architecture, which takes as input a binary program from a DOS environment and produces C programs as output.*

The decompiler has been divided into three separate modules which resemble the structure of the compiler. The front-end module is machine dependent and performs the loading and parsing of the program, as well as the generation of an intermediate representation. The universal decompiling machine module is machine and language independent, and performs all the flow analysis of the program. Finally, the back-end module is language dependent and deals with the details of the target high level language.

Even though the problem of decompilation is insoluble in general, a partial solution can be found, which gives information about the binary program. This paper describes some of the results found so far.

*[***Key words:** *decompiler, reverse compiler, halting problem]*

1 Introduction

A compiler is an executable program that takes as input a program written in a high level language and produces as output an executable program for a target machine; in other words, the input is language dependent and the output is machine dependent. A decompiler, or reverse compiler, attempts to perform the inverse process: given an executable program the aim is to produce a high level language program that performs the same function as the executable program. The input in this case is machine dependent, and the output is language dependent.

Compilers have been around since the early 1950s and there are widely known methods for writing compilers for any language. Decompilers, on the other hand, have been around only since the 1960s and there is no accepted methodology for their construction. It would be desirable for decompilers to perform automatic program translation as compilers do, but unfortunately this is not possible as decompilation is insoluble in general.

A naive approach to decompilation attempts to enumerate all valid phrases of an arbitrary attribute grammar, and then to perform a reverse match of these phrases to their original source code. An algorithm to solve this problem has been proved to be halting problem equivalent[1]. A more sensible approach is to try to determine which addresses contain data and which ones contain instructions in the given binary program. Given that in a Von Neumann machine, data and instructions are represented in the same way in the computer memory, an algorithm that solves this data/instruction problem would also solve the halting problem[2], and that is impossible. This means that the decompilation problem belongs to the class of non-computable problems; it is equivalent to the halting problem, and is therefore only partially computable[3]. In other words, we can build a decompiler which produces the right output for some input programs, but not for all input programs in general. The reader might ask, "why are we interested in building a decompiler". There are two reasons: first, to get a solution for the cases for which it is possible (i.e. to

[0] appeared in *XIX Conferencia Latinoamericana de Informática*, Buenos Aires, Argentina, 2-6 August 1993, pp. 257-266.

recover source code), and second, to get some information about the underlying algorithm used by the input program in the cases that do not have a complete solution.

This paper proposes a methodology for decompilation, and describes the current development state of ***dcc***, a decompiler project currently under development at the Queensland University of Technology (QUT). A brief description of the use of decompilers throughout the last decades is given, the proposed methodology and phases are explained, followed by an explanation of the implementation of ***dcc***, a summary and conclusions.

1.1 Previous Work

The first decompilers were used as tools in the translation of software from second to third generation machines in the 1960s. The very first decompiler was developed by Maurice Halstead at the Navy Electronic Labs[4]. This decompiler took machine code for the IBM 7094® as input and produced Neliac code for the Univac 1108®, whenever possible. It flagged ambiguities and produced inline assembler for pieces of code that could not be decompiled.

In the 1970s and 1980s, decompilers were used to port programs, recreate lost source code, modify existing binaries, document, and debug binaries. The Piler system[5] was an attempt at a general decompiler for a large class of source-target language pairs. Because of the large number of languages and operating systems it tried to cover, it was never finished. The ***decomp*** decompiler[6] took as input VAX® BSD 4.2® object files and produced C-like code. Although this decompiler worked correctly, it is not a complete decompiler given that it takes as input an ***object*** file with symbol table information (i.e. the program must have been compiled with the debugging flag on), instead of a binary file without symbol table information. In general, different techniques for decompilation were implemented, including pattern matching of assembler instructions[7]; which does not prove to be ideal, and graph oriented methods[8, 9] which are suitable for this job. Most of these methods limited themselves to the analysis of the underlying graph and took as granted the problem of separation of data from instructions, given that their input programs were ***assembler*** files rather than binary files. This assumption simplifies the problem considerably.

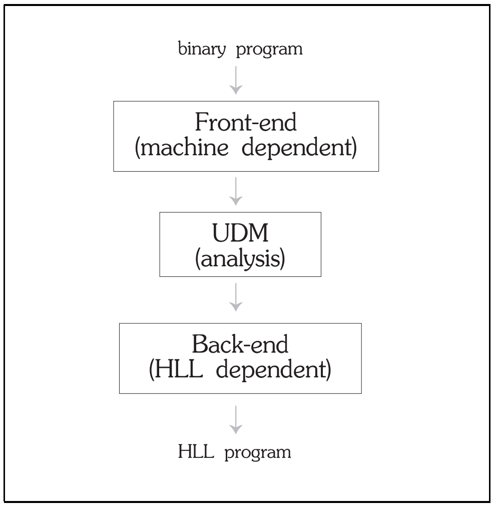

Figure 1: Decompiler Modules

In the 1990s, decompilation has become part of a wider area, reverse engineering. In brief, reverse engineering attempts to produce source code from object code by the use of disassemblers, decompilers, debuggers, and related tools. Some people believe that reverse engineering commercial programs violates an author's copyright and exclusive right to make copies, whereas others debate that this is not the case since fair use permits the copying of programs for the purpose of learning the idea behind the product; the idea not being protected by the copyright law[10]. In Europe, the European Parliament decided to permit reverse engineering of products for the sole purpose of interoperability[11], and a similar law has been proposed in Australia[12]. USA and Japan have laws to permit reverse compilation; in Japan's case there is no provision for fair use, whereas in the USA decompilation is permitted if it qualifies under fair use[13].

2 Proposed Methodology

This section presents a proposed methodology for reverse compilation of programs. We are not concerned at this moment with any particular machine or language, all we are given is a binary program and we need to produce another program in a high level language. The output language is not necessarily the language from which the program was compiled.

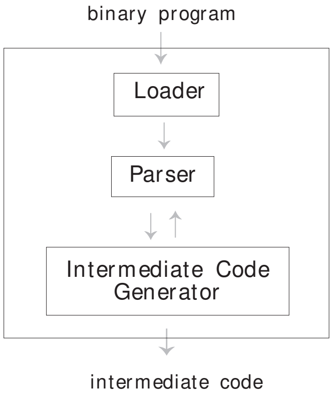

Figure 2: Front-end Phases

A decompiler can be structured in a similar way to a compiler, that is, by a series of modules that deal with machine or language dependent features. It is proposed to have three main modules dedicated to these features; a machine dependent module that reads in the program, loads it into virtual memory and parses it (the front-end), a machine and language independent module that analyses the program in memory (the universal decompiling machine), and a language dependent module that writes formatted output for the target language (the back-end) (refer to ***Figure 1***). In this way, different front-ends can be used for different machines, and different back-ends can be built for different target languages; this makes it easier to write decompilers for different machine/target language pairs.

2.1 The Front-end

The front end module deals with machine dependent features and produces a machine independent representation. It takes as input a binary program for a specific machine, loads it into virtual memory, parses it, and produces an intermediate representation of the program (see ***Figure 2***).

The **loader** is an operating system program that loads an executable program into memory if there is sufficient free memory available, sets up the segment registers and the stack, and transfers control to the program. The decompiler's loader must perform a similar function by allocating dynamic memory (virtual memory) to load the program, loading it, relocating addresses specified in the relocation table, and setting up the initial contents of registers. Note that executable files do not contain much information about which segments are used as data and which ones are used as code, data segments can contain code and/or addresses.

The **parser** decides the type of machine instruction at a given memory location, determines its operands and any offsets involved. The parsing of machine instructions is not as easy as it might appear. First of all, there are addressing modes that depend on the value of variables or registers at runtime. Second, indexed and indirect access to memory locations are difficult to resolve. Third, the complex machine instruction sets in today's machines utilize almost all combination of bytes, and therefore it is very hard to determine if a given byte is an instruction or is data. Fourth, there is no difference as to how data and instructions are stored in memory in a Von Neumann machine. Finally, idioms [1] are used by compiler writers to perform a function in the minimal number of machine cycles, and therefore a group of instructions will make sense only in a logical way, but not individually.

In order to determine which bytes of information are instructions and which ones are data, we start at the unique entry point to the program, given by the loader. This entry point must the first instruction for the program, in order to begin execution. From there on, instructions are parsed sequentially, until the flow of control changes due to a branch, a procedure call, etc. In this case, the target location is like a new entry point to part of the program, and from there onwards, instructions can be parsed in the previous way. Once there are no more instructions to parse, due to an end of procedure or end of program, we return to where the branch of control occurred and continue parsing at that level. This method traverses all possible instruction paths. At the same time, data references are placed in a global or local symbol table, depending on where the data is stored (i.e. as an offset on the stack, or at a definite memory location).

A major problem is introduced by the access of indexed and indirect memory instructions and locations.

[1] An idiom is a sequence of instructions which forms a logical entity and has a meaning that cannot be derived by considering the primary meanings of the individual instructions.

To handle these, heuristic methods need to be implemented to determine as much information as possible; analytic methods, such as emulation, cannot provide the whole range of solutions anyway. In general, it is impossible to solve these types of problems as they are equivalent to solving the halting problem, as previously mentioned.

Different problems are introduced by self-modifying code and virus tricks. A way to tackle these cases is to flag the sections of code involved, and comment them in the final program. Assembler code might be all that can be produced in these cases. Even more, a suggested optimal algorithm for parsing consists in finding the maximum number of trees that contain instructions; this is a combinatorial method that has been proved to be NP-complete[2]. For dense machine instruction sets, this algorithm does not solve the problem of data residing in code segments.

The **intermediate code generator** produces an intermediate representation of the program. It works close together with the parser, invoking it to get the next instruction. Each machine instruction gets translated into an intermediate code instruction, such representation being machine and language independent. Defined/used (du) chains of registers are also attached to the intermediate instruction; these are used later in the data flow analysis phase.

The quality of the intermediate code can be improved by an optimization stage that eliminates any redundant instructions, finds probable idioms, and replaces them by an appropriate intermediate instruction. Many idioms are machine dependent and reveal some of the semantics associated with the program at hand. Such idioms represent low level functions that are normally provided by the compiler at a higher level (e.g. multiplication and division of integers by powers of 2). Other idioms are machine independent and they reflect a shortcut used by the compiler writer in order to get faster code (i.e. fewer machine cycles for a given function), such as the addition and substraction of long numbers. Some of these idioms are widely known in the compiler community, and should be coded into the decompiler.

2.2 The Universal Decompiling Machine

The universal decompiling machine (UDM) is an intermediate module which is totally machine and language independent. It deals with flow graphs and the intermediate representation of the program, and performs all the flow analysis the input program needs (see ***Figure 3***).

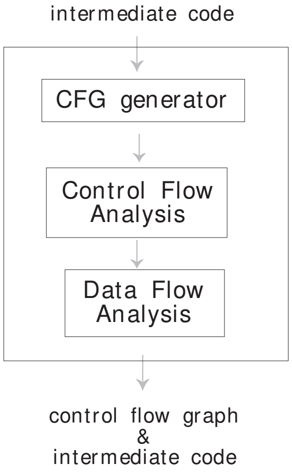

Figure 3: UDM phases

The **control flow graph (cfg) generator** constructs a cfg of basic blocks[2] for each procedure. A cfg is a connected, directed graph with nodes representing basic blocks and directed arcs representing flow of control from one node to another. Each basic block needs to record information such as predecessors, successors, and associated intermediate code. The type of a basic block is determined by the final intermediate instruction that changes the flow of control, such as unconditional or conditional branch, procedure call, procedure return, self loops, n-way branch, and end of program. When any of these instructions is met, the end of a basic block is reached as well. There is one other type of basic block, the one that falls through a labelled[3] basic block. Given that backward branches may split a basic block into two, and hence create a labelled basic block, two passes are needed to generate the graph; one to create a list of basic blocks, and the next to transform this list into a graph of basic blocks.

Due to the nature of machine code instructions, the compiler might need to introduce intermediate branches in an executable program, because there is no machine instruction capable of branching more than a

[2] A basic block is a sequence of instructions that has a single entry point and a single exit point.

[3] A labelled basic block is one whose entry point is the target of a branch.

certain maximum distance in bytes (architecture dependent). An optimization pass over the cfg removes this redundancy, by replacing the target branch location of all conditional or unconditional jumps that branch to an unconditional jump (and any recursive branches in this format) with the final one. While performing this process, some basic blocks are not going to be referenced any more, as they were used only for intermediate branches. These nodes must be eliminated from the graph as well.

The **control flow analysis** phase is concerned with the analysis of the flow of control of the cfg. First of all, this phase needs to determine the type of graph it is dealing with (reducible or irreducible), and then, for reducible graphs only, structure the graph into a set of high level language constructs.

Flow graphs produced by structured languages that do not make use of the **goto** statement are reducible[14, 15]. Structured control constructs such as **while** loops, **for** loops, **if..then..else**s, **case** statements, multiexit loops, and multilevel exits, found in commonly used languages such as C, Pascal, Modula-2 and Ada, will always produce reducible flow graphs. Unstructuredness is introduced by the use of **goto** statements, either available in the language (e.g. C, Pascal) or introduced by the optimizer. Given that we do not know whether the optimizer has unstructured a graph or not, it is not safe to say that even languages that do not implement the **goto** (e.g. Modula-2, Bliss), will produce structured graphs. And given that most languages allow for the use of **goto**s, there is a small probability that the graph at hand is irreducible and needs to be converted into a reducible one.

Graph **reducibility** is a concept introduced by Frances Allen [16, 17] and defined in terms of intervals, a graph construct defined by John Cocke [18]. An interval headed at node h is the maximal single-entry subgraph in which h is the unique entry node and in which all closed paths contain h. By selecting the proper set of header nodes, a flow graph G can be partitioned into a unique set of disjoint intervals. The process of reducibility consists in constructing a series of graphs, $G^1 \ldots G^n$, by collapsing the intervals of the graph into a single node. The limit flow graph G^n determines whether the original graph $G \equiv G^1$ is reducible; if G^n is a trivial graph, G is reducible, otherwise it is irreducible.

Irreducible graphs can always be transformed into functionally equivalent reducible graphs by a method of node replication known as node splitting; different algorithms have been specified in the literature[19, 18, 14]. Although node splitting does not assure the generation of a reducible graph, successive applications of interval reduction and node splitting will always transform an irreducible graph into a reducible one[14]. In practical cases, node splitting needs to be applied only once.

A **structuring algorithm** determines which high level language (hll) structures are present in the graph. We are concerned with control structures that are available in most languages; loops, conditionals, and case statements, and that form the basis for the creation of other control structures. The structuring algorithm determines where hll constructs are, as well as their extent. A predetermined set of hll constructs should be selected from the most commonly used constructs in current high level languages. This set forms the basis for the algorithm. Common constructs should include loops and conditionals. Whenever there is a piece of code that cannot be structured using the selected constructs, a **goto** is used instead and the target node is flagged as needing a label during code generation.

The **data flow analysis** phase makes use of compiler optimization theory to analyse the data and determine its type, temporary variables used for intermediate operations, expressions described in the intermediate code, arguments to procedures and functions, and values returned by functions. Def/use chains have been built for registers during the parsing stage, and they are now used to determine expressions. Aliases and value sets of each variable are tracked in order to generate better and easier to understand high level language code. In order to collect as much information as possible, global data flow analysis would be desirable.

2.3 The Back-end

The back end module is language dependent, as it deals with the target high level language. This module, optionally, restructures the graph into control constructs available only in the particular target language, and then generates code for this language (see ***Figure 4***).

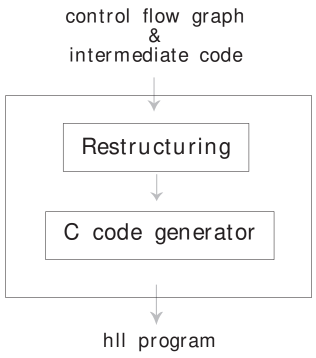

Figure 4: Back-end Phases

The **restructuring** stage is optional and it aims at structuring the graph even further, so that control structures available in the target language but not present in the set of control structures of the structuring algorithm (see section 2.2) are utilized. For instance, if the target language is Ada, multilevel exits are allowed. After the graph has been structured, multilevel exits will look like a loop with abnormal **goto** exits. The restructuring stage can check the target destination of each **goto**, and determine if an **exit(i)** statement is suitable instead. Another example is the **for** loop; such a loop is equivalent to a **while** loop that makes use of an induction variable. In this case, the induction variable needs to be found.

The final stage is the **code generation**, which emits code for the target language based on the cfg and the associated intermediate code. First of all, global variables are defined according to their type, described in the global symbol table. Then, code is emitted on a procedure by procedure basis, following a depth first traversal of the cfg. For each procedure, local variables are defined according to the type specified in the local symbol table. The flow of control is given by the type of basic block in hand, and sequential code is produced for each basic block from its associated intermediate code. Whenever a basic block has been flagged as needing a label, a unique label is emitted and any branches to the entry of this basic block are replaced by **goto**s. All variables get assigned names of the form, var1, var2, etc, given that there is not enough information concerning their use. In the same way, procedures are named proc1, proc2, and so on. Any data types and simple functions not supported by the target language must be placed in a header file which should be imported by the decompiled program.

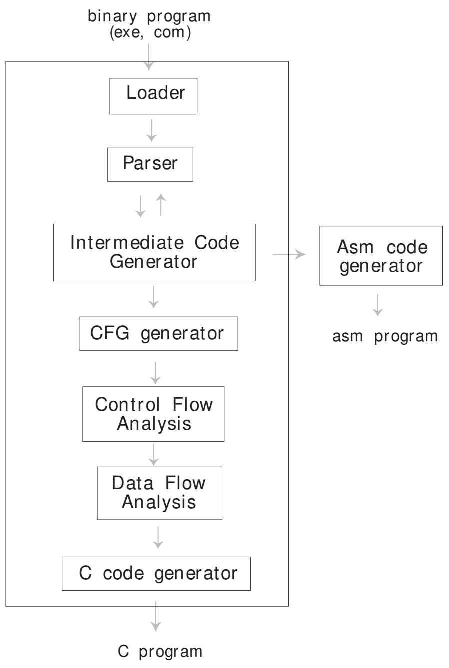

Figure 5: Structure of dcc

3 The RevComp Project

The Reverse Compilation project (RevComp) is currently under development by the School of Computing Science at QUT. Its aim is to produce a decompiler for the Intel 80x86® architecture, ***dcc***, which takes as input .exe or .com files from a DOS environment and produces C programs as output. C was selected as the target output language, given its flexibility, ease of low level manipulation, and portability. This decompiler is currently being implemented on a DECstation 3100® where a virtual 80x86 machine is built.

The ***dcc*** decompiler differs from previous decompilation projects in several ways. First of all, binary programs are analysed instead of assembler or object files. Given that we are dealing with Von Neumann machines, heuristic methods are used to separate instructions from data. Second, reducibility of the underlying program's control flow graph is checked for, as irreducible programs can be produced by the optimizer (even if the language does not use **goto**s, as previously discussed). Third, well known idioms dependent on the computer architecture are checked for and replaced by their logical meaning. Fourth, a data flow analysis phase is to be implemented to determine the type of data being used and related data issues. Finally, restructuring of control structures according to the target high level language constructs has been introduced as an optional stage, in order to eliminate as much as possible the use of **goto**s, and maximize the number of hll constructs used.

The main structure of *dcc* is illustrated in ***Figure 5***. This structure follows the proposed method, and it also integrates a disassembler in the system, given that assembler code can be produced once the program is parsed; no constructs or data need to be analysed in this case. The following paragraphs highlight some of the more important aspects of this project.

3.1 Implementation aspects

One of the heuristic methods implemented in *dcc*is a widely known implementation of **case** statements by the use of indexed tables. ***Figure 6*** is a particular implementation of **case** statements in the 8086. In this case, an assembler statement of the form **jmp word ptr CS:0DE0[bx]** does not provide the necessary information to calculate the target jump address. A known idiom is to check for lower and upper bounds before indexing into a table. In this way, the statements preceding the indexed jump will very likely have the information that we need; the bounds of the **bx** register. This heuristic method works in most cases. In our example, the statements **cmp ax,17h** and **jbe labA** check for the contents of the register **ax** to be between 0 and 23, in which case the offset into the table is calculated in register **bx** and the indexed jump is performed; otherwise, an unconditional jump to the end of the case statement code occurs.

Heuristics are also needed when building the control flow graph, specifically, when trying to determine the extent of a basic block that reaches the end of program. In any DOS executable, there are 7 different possible ways of exiting a program. They are all based on interrupts and some of them depend on the contents of certain registers. Our approach is to simulate the state of the virtual machine for most of the registers, so that when an interrupt is reached, the contents of the required registers can be checked for.

```
        ...
        cmp   ax,17h
        jbe   labA
        jmp   labZ
      labA:
        mov   bx,ax
        shl   bx,1
        jmp   word ptr CS:0DE0[bx]
CS:0DE0 dw    lab1
CS:0DE2 dw    lab2
        ...
CS:0E0E dw    lab24
CS:0E10 lab1:
        ...
CS:11C7 lab24:
        ...
CS:11F4 labZ:         ; end of case
        ...
```

Figure 6: An implementation of indexed tables

A difficult case is presented with indirect procedure calls or indexed branches that do not fit into our **case** statement idiom. No heuristic method has been implemented yet for these cases, and therefore, we flag the corresponding basic block as going nowhere. No more instructions are parsed after such instructions along the current path, and the basic block is finished. In the case of indirect and indexed data accesses, these locations are not placed in the symbol table, and we expect later data flow analysis to provide us with a plausible data type for these variables, based upon their use.

Several idioms are considered, some machine dependent and others more general. Amongst the machine dependent idioms are: procedure entry preamble, procedure exit postamble, and number of local variables defined in the stack by decreasing the contents of register **sp**. General idioms are machine independent, and cover functions like: multiplication and division of integers by powers of 2 by shifting the register left or right, swapping variables, and access to local variables as offsets on the stack.

The intermediate language used in this project is a simple, assembler-like, 3-address code representation where all the operands are made explicit; it has been named Icode. Icode provides an n:1 mapping of assembler to Icode instructions (e.g. all assembler **add** instructions (0x00..0x05) are handled by the one Icode **add** instruction).

Once the control flow graph has been built, the optimization pass that removes redundant intermediate branches has reduced the size of the cfg by up to 50% in different programs tested by ***dcc***. This simplifies the structure of the graphs in hand.

Checking for reducibility has been implemented by constructing the derived sequence of graphs $G^1 \ldots G^n$ and finding their intervals; the implementation makes use of pointers. If the cfg is found to be irreducible, the graph is flagged at this stage; node splitting has not yet been implemented as only a minority of graphs are found to be irreducible.

The set of high level language constructs that was selected as the base set for the structuring algorithm are: **if..then**, **if..then..else**, **case** statement, **while** loop, **repeat** loop, and endless **loop**. These constructs are used in most high level languages (e.g. C, Modula2, Pascal, Ada). These constructs belong to three major groups; loops, cases (n-way branch), and conditionals. The algorithm makes use of the $G^1 \ldots G^n$ graphs to find nested loops, and the immediate dominators to find conditionals and n-way branch (a reverse walk of the underlying tree is performed in these cases). Any abnormal exits from these control structures make use of a **goto** statement. A detailed explanation of this algorithm can be found in [20].

The data flow analysis stage has not yet been implemented. Instead, code generation has been implemented to get a feel for the type of output expected from ***dcc***.

3.2 Results

At present, the output C programs from ***dcc*** reflect the control structures of the program, but the instructions are still low level (i.e. assembler-like). More idioms are currently being coded into ***dcc***. Such idioms are mostly machine independent as they give a semantic interpretation to a group of instructions. For example, the absolute value of a number placed in register **eax** can be calculated in the following way: the sign of the number is moved to the carry bit, a temporary register, **ecx**, is substracted with borrow from itself (in order to get all zeros for positive numbers or all ones for negative numbers), then zero is substracted from the original number in **eax**; resulting in -1 if the number is negative, and finally an **xor** of both registers (**eax**, **ecx**) will negate a negative number, or leave a positive number unmodified. This sequence of instructions should be translated to a **abs(x)**, where **x** is the variable placed in **eax** (refer to ***Figure*** 7).

```
bt  eax, 31      ; sign -> carry
sbb ecx, ecx     ; all 0s or 1s
sbb eax, 0       ; -1 if negative
xor eax, ecx     ; not if negative
```

Figure 7: Idiom for an absolute value

Programs decompiled by ***dcc*** make use of an include header file which defines simple data types (byte, word), has macros to manipulate such data types, defines macros to manipulate the registers (accessed in C with a union REGS structure that is defined to be global for the whole program), defines constants such as true and false, and includes simple functions not supported by C, such as swap.

One of the major problems that we face is the amount of extra code included by the compiler, such as setup procedures and libraries (i.e. not all included library functions are normally invoked by the program), that is indistinguishable from procedures written by a programmer (given that most of the library procedures have a procedure preamble and postamble, just as any other user procedure). This means that for a program that displays 'hello world' on the screen with the use of printf in C, 23 procedures are decompiled. The same program written in Pascal produces more than 40 procedures. An initial solution was to check for compiler signatures[4], but this is not feasible as, even if we know the compiler that compiled the code, not even pattern matching of the decompiled code with the start of each library procedure would be possible given that different memory models in the PC produce different entries for the same piece of code. Our new solution is to let the user decide for himself which procedures he is interested in, after decompiling all

[4] A compiler signature is a string placed by the compiler in a data section of the binary file. Normally it contains such details as the company name, compiler, and release version.

the available procedures and giving some indication as to which procedures appear to be low level (possibly hand-coded in assembler due to the machine instructions that are used, and by not having a procedure preamble or postamble).

There is an obvious need for a data flow analysis stage, which is planned to be implemented next. This stage will be able to determine expressions, temporary variables that can be removed, and provide a more precise data type for the variables (i.e. according to the use of the variable).

The output from *dcc* provides comments at different levels depending on the switch specified by the user when running *dcc*. By default, procedures have comments for most of the DOS interrupts, to reflect the function that has been invoked. Available switches for extra information are, 'v' for verbose and 'V' for very verbose. The 'v' switch displays information about the binary file (file size, file type, number of relocation items, and maximum memory allocated), a tree of the procedures found, along with any flags that were set up during this procedure, and the basic blocks found during the creation of the graph. The 'V' switch displays the relocation table (if any), the control flow graph, and the derived sequence of graphs, along with all their interval information.

Other switches available in *dcc* are: 'a' to produce assembler output, 'm' to display the memory map (i.e. data, instructions, bytes used as data and instructions, and unknown areas), 'p' to print the procedure list, and 's' to print statistics about the graph optimization stage.

4 Summary and Conclusions

This paper has proposed a methodology for decompilation of binary programs, and describes the current development state of *dcc*, a decompiler for the Intel 80x86 architecture, built upon the proposed methodology. The decompiler structure resembles that of a compiler; three main modules are distinguished: the front-end which is machine dependent, the universal decompiling machine (udm) which is machine and language independent, and the back-end which is language dependent.

The front-end deals with the loading of the binary program, parsing it, and producing an intermediate representation of the program (Icode). The udm constructs a graph for each procedure, associates the intermediate representation with each node, checks for graph reducibility, determines which high level structures are used in the program, and performs a data flow analysis in order to learn how data is used in the program and be able to determine probable data types for the existing variables. Finally, the back-end performs the restructuring needed to accommodate the structures found in the program into structures available in the target high level language, and emits global variable information and code for each procedure.

The *dcc* decompiler takes as input DOS executable programs (.exe and .com) and produces C programs as output. It is in its α stage, and it implements most of the stages defined in the proposed methodology. Currently, data flow analysis has not been implemented, therefore the output C programs are still assembler-like. The control structures of the program are well defined, and variables are defined in terms of the types byte, word, or string. Even though the decompilation problem is insoluble in general, partial solutions provide some information about the original program. Future releases are expected.

Acknowledgements

We would like to thank Jeff Lederman for clarifying some of the concepts associated with this project. This research is partly funded by Australian Research Council (ARC) grant no.A49130261.

References

[1] P.T.Breuer and J.P.Bowen, "Decompilation: The enumeration of types and grammars," Tech. Rep. PRG-TR-11-92, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, 1992.

[2] R.N.Horspool and N.Marovac, "An approach to the problem of detranslation of computer programs," *The Computer Journal*, vol. 23, no. 3, pp. 223–229, 1979.

[3] L.Goldschlager and A.Lister, *Computer Science: A modern introduction*. Prentice-Hall International, 1982.

[4] M.H.Halstead, *Machine-independent computer programming*, ch. 11, pp. 143–150. Spartan Books, 1962.

[5] P.Barbe, "The piler system of computer program translation," tech. rep., Probe Consultants Inc., Sept. 1974.

[6] J. Reuter, "decomp.tar.z." Public domain software. Anonymous ftp cs.washington.edu, directory /pub, 1988.

[7] C.R.Hollander, *Decompilation of Object Programs*. PhD dissertation, Stanford University, Computer Science, Jan. 1973.

[8] B.C.Housel, *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages*. PhD dissertation, Purdue University, Computer Science, Aug. 1973.

[9] G.L.Hopwood, *Decompilation*. PhD dissertation, University of California, Irvine, Computer Science, 1978.

[10] H.Swartz, "The case for reverse engineering," *Business Computer Systems*, vol. 3, pp. 22–25, Dec. 1984.

[11] Anon., "Software protection," *Edge: Work-Group Computing Report*, vol. 2, p. 7, 22 Apr 1991.

[12] S.McNamara, "Australia: proposals open reverse engineering debate," *Newsbytes*, p. ??, 18 Oct 1991.

[13] G.Burkill, "Reverse compilation of computer programs and its permissibility under the berne convention," *Computer Law & Practice*, vol. 6, pp. 114–119, mar-apr 1990.

[14] M.S.Hecht, *Flow Analysis of Computer Programs*. 52 Vanderbilt Avenue, New York, New York 10017: Elsevier North-Holland, Inc, 1977.

[15] S.R.Kosaraju, "Analysis of structured programs," *Journal of Computer and System Sciences*, vol. 9, no. 3, pp. 232–255, 1974.

[16] F.E.Allen, "Control flow analysis," *SIGPLAN Notices*, vol. 5, pp. 1–19, July 1970.

[17] F.E.Allen, "A basis for program optimization," in *Proc. IFIP Congress*, (Amsterdam, Holland), pp. 385–390, North-Holland Pub.Co., 1972.

[18] J.Cocke, "Global common subexpression elimination," *SIGPLAN Notices*, vol. 5, pp. 20–25, July 1970.

[19] F.E.Allen and J.Cocke, "Graph theoretic constructs for program control flow analysis," Tech. Rep. RC 3923 (No. 17789), IBM, Thomas J. Watson Research Center, Yorktown Heights, New York, July 1972.

[20] C.Cifuentes, "A structuring algorithm for decompilation," in *XIX Conferencia Latinoamericana de Informática*, (Buenos Aires, Argentina), pp. 267–276, Centro Latinoamericano de Estudios en Informática, 2-6 August 1993.