Decompilation of Binary Programs & Structuring Decompiled Graphs

Ximing Yu

May 3, 2011

1 Introduction

A *decompiler*, or *reverse compiler*, is a program that attempts to perform the inverse process of the compiler: given an executable program compiled in any high-level language, the aim is to produce a high-level language program that performs the same function as the executable program. Therefore

- Input: machine dependent
- Output: language dependent

1.1 Probllems

Several practical problems are faced when writing a decompiler:

- Representation of data and instructions in the Von Neumann architecture: they are indistinguishable.
- The great number of subroutines introduced by the compiler and the linker, such as start-up subroutines that set up its environment.

1.2 Uses of Decompiler

There are several uses for a decompiler, including two major software areas: maintenance of code and software security.

- Maintenance:
  - Recovery of lost source code
  - Migration of applications to a new hardware platform
  - Translation of code written in an obsolete language into a newer language
  - Structuring of old code written in an unstructured way
  - Debugger tool that helps in finding and correcting bugs in an existing binary program.
- Software security: a binary program can be checked for the existence of malicious code (e.g. viruses) before it is run for the first time on a computer.

2 The Decompiler Structure

There are three main modules:

- *Front-end*: a machine-dependent module that reads in the program, loads it into virtual memory and parses it
- *Universal Decompiling Machine*: a machine- and language-independent module that analyses the program in memory
- *Back-end*: a language-dependent module that writes formatted output for the target language

2.1 The front-end

The front-end module deals with machine-dependent features and produces a machine-independent representation.

- Input: a binary program for a specific machine
- Output: an intermediate representation of the program, and the programs control flow graph.

The front-end module consists of:

- Loader: loads a binary program into virtual memory
- Parser:
  - Disassembles code starting at the entry point given by the loader.
  - Follows the instructions sequentially until a change in flow of control is met. All instruction paths are followed in a recursive manner.
  - The intermediate code is generated and the control flow graph is built.
- Semantic analysis: performs idiom analysis and type propagation.

Two levels of intermediate code are required:

- A low-level representation that resembles the assembler from the machine: mapping of machine instructions to assembler mnemonics.
- A higher-level representation that resembles statements from a high-level language:generated by the inter-procedural data flow analysis.

The front-end generates a low-level representation.

2.2 The universal decompiling machine

The universal decompiling machine (UDM) is an intermediate module that 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.

2.2.1 Data flow analysis

Transform the low-level intermediate representation into a higher-level representation that resembles a HLL statement. Eliminate the concept of condition codes (or flags) and registers, as these concepts do not exist in high-level languages.

- Condition codes
  - Condition codes are classified in two groups: HLCC which is the set of condition codes that are likely to have been generated by a compiler (e.g. overflow, carry), and NHLCC which is the set of condition codes that are likely to have been generated by assembly code (e.g. trap, interrupt).
  - A use/definition, or reaching definition, analysis is performed on condition codes. Once the set of defined/used instructions is known, these low-level instructions can be replaced by a unique conditional high-level instruction that is semantically equivalent to the given instructions.
- Registers
  - Low-level instructions are classified into two sets: HLI which is the set of instructions that are representable in a high-level language (e.g. `mov`, `add`), and NHLI which is the set of instructions that are likely to be generated only by assembly code (e.g. `cli`, `ins`).
  - The elimination of *temporary registers* leads to the elimination of intermediate instructions by replacing several low-level instructions by one high-level instruction that does not make use of the intermediate register.
  - Two preliminary analysis are required: a definition/use analysis on registers, and, an interprocedural live register analysis. The method to eliminate registers is *forward substitution*.

Data flow analysis also introduces the concept of expressions and parameter passing, as these can be used in any HLL program. The high-level instructions that are generated by this analysis are:

- `asgn` (assign)
- `jcond` (conditional jump)
- `jmp` (unconditional jump)
- `call` (sub-routine call)
- `ret` (sub-routine return)

No control structure instructions are restored by this phase.

2.2.2 Control flow analysis

The *control flow analyzer* structures the control flow graph into generic high-level control structures that are available in most languages, including:

- Loops
  - pre-test loop: `while()`

  - post-test loop: `repeat ...until()`
  - infinite loop: `loop`

- Conditionals
  - 2-way conditionals: `if ...then` and `if ...then ...else`
  - n-way conditionals: `case`

There are three types of nodes of subgraphs that represent high-level loops and 2-way structures:

- Header node: entry node of a structure.
- Follow node: the first node that is executed after a possibly nested structure has finished.
- Latching node: the last node in a loop; the one that takes as immediate successor the header of a loop.

Structuring Loops

> By Interval theory, an interval $I(h)$ is the maximal, single-entry subgraph in which h is the only entry node and in which all closed paths contain h. The unique interval node h is called the header node. By selecting the proper set of header nodes, graph G can be partitioned into a unique set of disjoint interval $\mathcal{I} = \{I(h_1), I(h_2), \ldots, I(h_n)\}$
>
> The derived sequence of graphs, $G^1 \ldots g^n$ is based on the intervals of graph G. The first order graph, G^1, is G. The second order graph, G^2, is derived from G^1, by collapsing each interval in G^1 into a node.

Given an interval $I(h_j)$ with header h_j, there is a loop rooted at h_j if there is a back-edge to the header node h_j from a latching node $n_k \in I(h_j)$.

Once a loop has been found, the type of loop is determined by the type of header and latching nodes of the loop.

- A `while()` loop is characterized by a 2-way header node and a 1-way latching node.
- A `repeat ...until()` is characterized by a 2-way latching node a non-conditional header node.
- A endless `loop` loop is characterized by a 1-way latching node and a non-conditional header node.

The algorithms used are:

1. Each header of an interval in G^1 is checked for having a back-edge from a latching node that belongs to the same interval.
2. If this happens, a loop has been found, so its type is determined, and the nodes that belong to it are **marked**.
3. Next, the intervals of G^2, $\mathcal{I}^2$ are checked for loops, and the process is repeated until intervals in $\mathcal{I}^n$ have been checked.

Structuring 2-Way Conditionals Both a single branch conditional (i.e. `if ...then`) and an `if ...then ...else` conditional subgraph have a common follow node that has the property of being immediately dominated by the 2-way header node.

When these subgraphs are nested, they can have different follow nodes or **share** the same common follow node.

During loop structuring, a 2-way node that belongs to either the header or the latching node of a loop is marked as being part of the loop, and must therefore not be processed during 2-way conditional structuring.

Compound Conditions Whenever a subgraph of the form of the short-circuit evaluated graphs is found, it is checked for the following properties:

1. Nodes `x` and `y` are 2-way nodes.
2. Node `y` has only 1 in-edge.
3. Node `y` has a unique instruction; a conditional jump (`jcond`) high-level instruction.
4. Nodes `x` and `y` must branch to a common `t` or `e` node.

2.3 The back-end

The back-end module is language dependent, as it deals with the target high-level language. It has the following phases:

- Restructuring (optional): structuring the graph even further, so that control structures available in the target language but not present in the generic set of control structures of the structuring algorithm, previously described, are utilized.
- HLL code generation: generates code for the target HLL based on the control flow graph and the associated high-level intermediate code. Involves:
  - Defines global variables.
  - Emits code for each procedure/function following a depth first traversal of the call graph of the program.
  - If a goto instruction is required, a unique label identifier is created and placed before the instruction that takes the label.
  - Variables and procedures are given names of the form `loc1`, `proc2`.

3 The Decompiling System

The decompiling system that integrates a decompiler, *dcc*, and an automatic signature generator, *dccSign*.

3.0.1 Signature generator

A signature generator is a front-end module that generates signatures for compilers and library functions of those compilers. Such signatures are stored in a database, and are accessed by *dcc* to check whether a subroutine is a library function or not, in which case, the function is not analyzed by *dcc*, but replaced by its library name (e.g. `printf()`).