Decompile

Ximing Yu

May 3, 2011
Decompiler Definition

- *Decompiler* 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.
- **Input:** Machine dependent
- **Output:** Language dependent
3 Main Modules of Decompiler

Binary Program

Front-end (machine dependent)

Universal Decompilation Machine (analysis)

Back-end (language dependent)

High Level Language Program
Front-end

- Deals with *machine-dependent* features and produces a *machine-independent* representation.
- Input: a binary program for a specific machine
- Produces:
  - Intermediate representation of the program
  - The program’s control flow graph
Front-end Phases

Binary Program

Loader

Parser

Semantic analysis

Low-level intermediate code & control flow graph
Front-end Phases

- **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. — generated by front-end

- A higher-level representation that resembles statements from a high-level language — generated by the inter-procedural data flow analysis.
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.
Universal Decompiling Machine Phases

control flow graph &
low-level intermediate code

Data flow Analysis

Control flow Analysis

high-level intermediate code &
structured control flow graph
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.
- Introduce the concept of expressions and parameter passing, as these can be used in any HLL program.
  - `asgn` (assign)
  - `jcond` (conditional jump)
  - `jmp` (unconditional jump)
  - `call` (sub-routine call)
  - `ret` (sub-routine return)
1 cmp ax, bx ; def: SF,ZF,CF
2 jp labZ ; use: SF,ZF ; ud-cc(SF,ZF)={1}

JCOND (ax > bx)
.. ... ; other code here
28 MOV ax, di ; ASGN ax, di ; du(ax) = {30}
29 MOV bx, 0Ah ; ASGN bx, 0Ah ; du(bx) = {32, 33}
30 CWD ; ASGN dx:ax, ax ; du(ax) = {31}, du(dx) = {31}
31 MOV tmp, dx:ax ; ASGN tmp, dx:ax ; du(tmp) = {32, 33}
32 DIV bx ; ASGN ax, tmp / bx ; du(ax) = {}
33 MOD bx ; ASGN dx, tmp % bx ; du(dx) = {34}
34 MOV si, dx ; ASGN si, dx
.. ... ; other code here, no use of ax

\[\text{ASGN si, di} \% 0Ah\]
High-level control structures:

- **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.
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.
Interval Theory
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 \texttt{while()} loop is characterized by a 2-way header node and a 1-way latching node.
- A \texttt{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.
Structuring Loops — Algorithm

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 $G^2$, $I^2$ are checked for loops, and the process is repeated until intervals in $I^n$ have been checked.
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.
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 ($j\text{cond}$) high-level instruction.
4. Nodes $x$ and $y$ must branch to a common $t$ or $e$ node.
Compound Conditional Graphs
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.
Back-end phases

- High-level intermediate code & structured control flow graph
- Restructuring
- HLL Code Generation
- High Level Language Program
The Decompiling System

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

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()`).