Delphi object files decompiler

Delphi .NET object files decompiler

Andrey Mikhailov* Alexey Hmelnov*
mikhailov@icc.ru hmelnov@icc.ru

*Matrosov Institute for System Dynamics and Control Theory
Irkutsk

Ivannikov ISPRAS Open Conference 2017, Moscow
30 november, 2017

Tools

- **Partial decompilation**
  1. Executable files («**dcc**», «**REC**», «**Boomerang**», «**HexRays**», «**SmartDec**»)
  2. Delphi («**DeDe** (Delphi Decompiler)», «**IDR**», «**EMS Source Rescuer**»)

     To parse the DCU, use the dcu32int tool
- **Full decompilation**
  1. Java («**DJ Java Decompiler**», «**JD-GUI Java Decompiler**», «**AndroChef Java Decompiler**»)
  2. *.NET* («**ILSpy**», «**NETReflector**»)

Delphi object files

- *«An **object file** is a file containing object code, meaning relocatable format machine code that is usually not directly executable. There are various formats for object files, and the same object code can be packaged in different object files. An object file may also work like a shared library. In addition to the object code itself, object files may contain metadata used for linking or debugging, including: information to resolve symbolic cross-references between different modules, relocation information, stack unwinding information, comments, program symbols, debugging or profiling information.»*
- *DCU* = Delphi Compiled Unit. That is compiled *.pas file for x86
- *DCUIL* – that is compiled *.pas file for .NET
- *DCU32INT*[1] – Delphi unit parser
- DCU $\geq$ OBJ $\geq$ EXE

[1] http://hmelnov.icc.ru/DCU/index.ru.html

The format of DCU files

Delphi object file unlike PE executable file has a more structured program representation, e.g. every procedure has its own memory block. It contains information about all the data types defined in the unit and it may include debugging information. DCUIL has small header file containing common information such as size, compile time, etc. The header is followed by tagged information. Tags are divided into the following groups:

- The list of the used units and dynamic libraries, including information about their definitions (of data types, procedures, etc) used in the unit.
- Information about the data types, procedures, variables, etc defined in the unit.
- The memory block, which contains the memory representation for procedures, constants, etc defined in the unit.
- The linking information for the memory block (where to place the addresses of some objects used when linking).
- Debugging information.

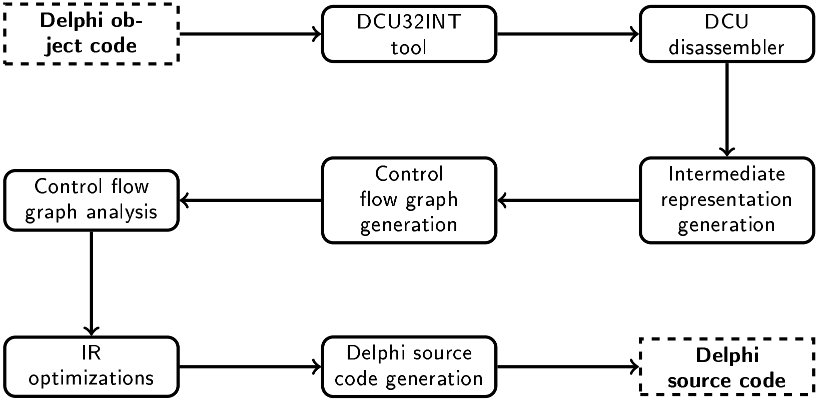
Reached DCU decompilation level

Platform	Source	Version	№	Level
Win 32	x86	2.0 – 7.0, 2005 –	2 – 7,9 –	disassembler+dataflow
Win 64	x64	XE2 –	16 –	disassembler+dataflow
OS X,32	x86	XE2 –	16 –	disassembler
iOS,Simulator	x86	XE4 –	18 –	disassembler
iOS,Device	ARM 32	XE4 –	18 –	no
iOS,Device 64	ARM 64	XE8 –	22 –	no
Android	ARM 32	XE5 –	19 –	no
.NET	CIL	8.0 – 2006	8 – 10	decompiler
*	Inline	2005 –	9 –	decompiler

Decompilation phases

1. Syntax analysis	—	Main task is determine the beginning and end of the «opcode»
2. Semantic analysis	—	We will assume that the object code is always semantically correct
3. Generic intermediate representation	—	For machine-independent Optimization
4. Control flow graph generation	—	Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in some optimization.
5. Data flow analysis	—	Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. The information gathered is often used by decompilers when optimizing a intermidiate representation.
6. Control flow graph analysis	—	Recovering high-level control constructs is essential for decompilation in order to produce structured code that is suitable for human analysts and sourcebased program analysis techniques.
7. Code generation	—	Code generation can be considered as the final phase of decompilation. Source code generating from intermediate representation.

Delphi for .NET object code decompilation scheme



Disassembler phase

The structure of the CIL command:

1. Can consist of one or two bytes
2. After the command, there may be metadata:

Operand	Size	Description
none	0	The operand is empty
int8	1	A signed 8-bit integer
int32	4	A signed 32-bit integer
int64	8	A signed 64-bit integer
unsigned int8	1	Unsigned 8-bit integer
unsigned int16	2	Unsigned 16-bit integer
float32	4	32-bit floating-point number
float64	8	64-bit floating-point number
token	4	FixUp (address binding)
switch	variable	Array of jump addresses

Table – CIL operands

DCU for .NET disassembler implementation

CIL ⟶ *TCILInstr.CILOpCode*

TMethodBody – contain the sequence of TCILInstr

TCILExpr - Abstract language representation

```
TCILOpCode = class
  protected
    Op1 : Byte;
    Op2 : Byte;
    Code : TCILCode;
    FlowControl : TFlowControl;
    OpCodeType : TOpCodeType;
    OperandType : TOperandType;
    StackBehaviorPop : TStackBehaviour;
    StackBehaviorPush : TStackBehaviour;
  public
   ...
  end;
```

Mono — Mono is a software platform designed to allow developers to easily create cross platform applications part of the .NET Foundation

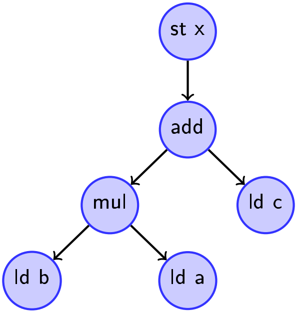
ILSpy — ILSpy is the open-source .NET assembly browser and decompiler

Disassembly algorithm

Data: Procedure memory block containing the CIL bytecode
Result: CIL sequence
while *end of code* **do**
    $B \leftarrow ReadByte()$
    if $b \neq \$FE$ **then**
        $ByteCode \leftarrow OneByteOpCodeTbl(B)$
    end
    else
        $B \leftarrow ReadByte()$
        $ByteCode \leftarrow TwoByteOpCodeTbl(B)$
    end
    $ByteCode.ReadOperand()$
end

Intermediate representation 1. Pseudo register

ldloc a	**MOV** r1, &a
ldloc b	**MOV** r2, &b
mul	**MUL** r1, r1, r2
ldloc c	**MOV** r2, &c
add	**ADD** r1, r1, r2
stloc x	**STORE** r1, &x



TCILInst.Expr $\longleftarrow$ TCILExpr

Expressions

- TCILExpr
  - TCILBinOp
  - TCILUnOp
  - TCILSemOp

All conditional or unconditional branches replace with

- TCILCondGoTo
- TCILUncondGoTo

Algorithm 1 Example. Callvirt method

1. An object reference obj is pushed onto the stack
2. Method arguments arg1 through argN are pushed onto the stack
3. Method arguments arg1 through argN and the object reference obj are popped from the stack; the method call is performed with these arguments and control is transferred to the method in obj referred to by the method metadata token. When complete, a return value is generated by the callee method and sent to the caller
4. The return value is pushed onto the stack

Control flow generation

Basic block *is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. The code in a basic block has:*

1. *One entry point, meaning no code within it is the destination of a jump instruction anywhere in the program*
2. *One exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block*

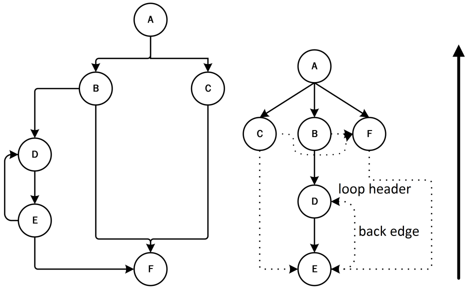
Data: A sequence of instructions
Result: A list of basic blocks with each three-address statement in exactly one block

1. The first instruction is a leader
2. The target of a conditional or an unconditional goto/jump instruction is a leader
3. The instruction that immediately follows a conditional goto/jump instruction is a leader
4. The first instruction of the exception block is the leader

Starting from a leader, the set of all following instructions until and not including the next leader is the basic block corresponding to the starting leader.

Creating edges

1. Calculate jump addresses
2. Create edges for
   1. nodes with branch instructions
   2. exceptions



Structuring Decompiled Graphs [2]:

- Edges are marked as direct, back, oblique
- Structuring Loops
- Structuring 2-way conditions (+ compound conditions)

[2] Cifuentes C. Structuring decompiled graphs //International Conference on Compiler Construction. – Springer Berlin Heidelberg, 1996. – C. 91-105. MLA

Control flow analysis 2

Data: G, D, P
Result: n abstract node containing a hierarchy of folded subgraphs

```
foreach v ∈ D in a backward breadth-first order do
    foreach p ∈ Children(v) do
        if p  pidom  v then
            S ← Children(v) \ p
            if Classify_Region(S) ≠ undeterminated then
                Apply_Template(S)
            end
            else
                Recognize_Undeterminanted_Region(S)
            end
            Modify(G, D, P)
        end
    end
end
```

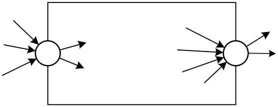


Fig. – TT-region

Templates

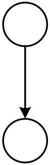


Fig. – line

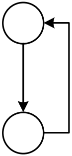


Fig. – loop

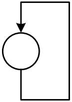


Fig. – self-loop

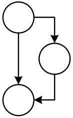


Fig. – if-then

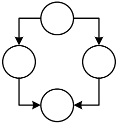


Fig. – if-then-else

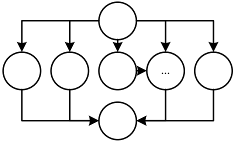


Fig. – switch

IR for regions

Regions

- **TCILExpr**
  1. TCILIfThenBlock
  2. TCILIfThenElseBlock
  3. TCILRepeatSt
  4. TCILWhileSt
  5. TCILCaseSt

```
TCILIfThenElseBlock = class (TCILExpr)
protected
  FTrue, FFalse: TCtrlFlowNode;
  FCond: TCILCondition;
public
  constructor Create(ACond: TCILCondition; ATrue, AFalse: TCtrlFlowNode);
  destructor Destroy;
  function AsString(BrRq: boolean): String; override;
  procedure Show(BrRq: boolean); override;
  property Cond: TCILCondition read FCond;
end;
```

Generation of expressions

Result := a + b + c + D;

...

```
ldarg.0 Pop: Pop0 Push: Push1 Type: InlineNone
ldarg.1 Pop: Pop0 Push: Push1 Type: InlineNone
add Pop: Pop1_pop1 Push: Push1 Type: InlineNone
ldarg.2 Pop: Pop0 Push: Push1 Type: InlineNone
add Pop: Pop1_pop1 Push: Push1 Type: InlineNone
ldloc.0 Pop: Pop0 Push: Push1 Type: InlineNone
add Pop: Pop1_pop1 Push: Push1 Type: InlineNone
stloc.1 Pop: Pop1 Push: Push0 Type: InlineNone
```

CILCtx

- Locals – local variables
- Args – procedure arguments
- Stack – stack state

Features of data flow analysis

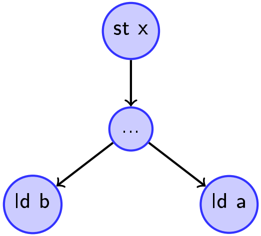


Fig. – The value is determined by one branch, used in different

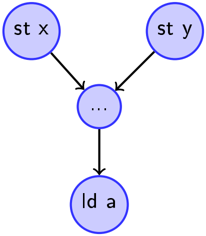


Fig. – The value is defined in different branches, used in one

- st – push value on to the stack
- ld – load values from the stack

IR optimizations

- ***Merging variables.*** Elimination of intermediate calculations
- ***Delete unused code.*** Removing unreachable code, because it is impossible to determine the state of the stack
- ***Copies propagation***
  1. Any instruction loading the address is copied to the "opcode" of its use
  2. Parameters propagation
- ***Removing unused variables***
- ***Simplify the instruction set for the jump instructions.*** Jump commands are given to the general view (TCILCondGoTo, TCILUncondGoTo)
- ***Combining complex logical expressions***

Test. Decompilation quality[3]

TS – set of test programs
$prog$ – source program
$KLOC(prog)$ – number of thousands of significant lines of the prog
K – amount of penalties for the original program
$K^{'}$ – amount of penalties for the decompiled program

$$C_{decom} = \sum_{prog \in TS} \frac{max(0, K^{'} - K)}{KLOC(prog)}$$

(1)

Table – Penalties

Name	Penalty
non-recovery of variable name	1
goto operator	3
break operator	1
continue operator	1
non-recovery of for operator	1

[3] ***Troshina***, *«Issledovanie i razrabotka metodov dekompilyatsii programm»*, 2009 г.

Test. Decompilation quality.

Table – Decompilation quality

Name	DCUIL2PAS	ILSpy
BitWise	62,5	133,3
Compression	18,6	146
LZRW1KHCompressor	75	140
GetMatch	0	166,6

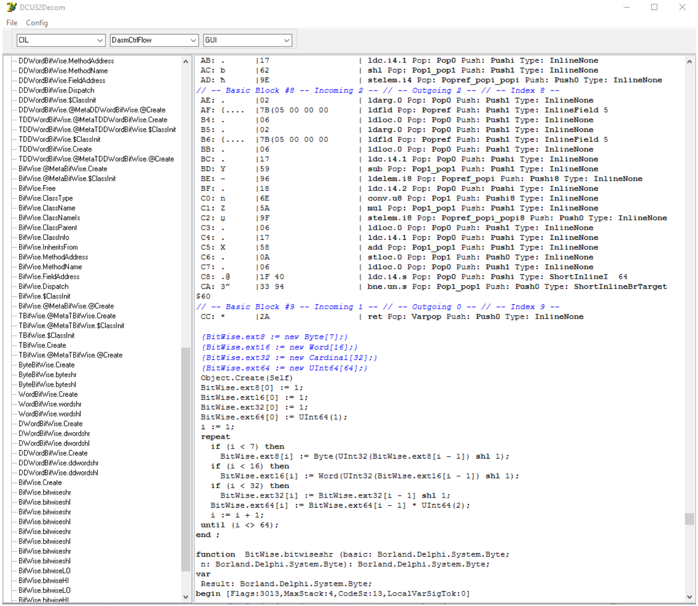
Table – Performance

Name	files count	Size (mb)	Time (s)
Delphi 8 VCL	325	39	396

Table – Quality

Name	procedures count	without goto	with goto	%
Delphi 8 VCL	9003	8879	124	1,3

Example 1


DCU32Decom
File
Config
CIL
DasmCtrlFlow
GUI
DDWordBitWise.MethodAddress
DDWordBitWise.MethodName
DDWordBitWise.FieldAddress
DDWordBitWise.Dispatch
DDWordBitWise.$ClassInit
DDWordBitWise.@MetaDDWordBitWise.@Create
TDDWordBitWise.@MetaTDDWordBitWise.Create
TDDWordBitWise.@MetaTDDWordBitWise.$ClassInit
TDDWordBitWise.$ClassInit
TDDWordBitWise.Create
TDDWordBitWise.@MetaTDDWordBitWise.@Create
BitWise.@MetaBitWise.Create
BitWise.@MetaBitWise.$ClassInit
BitWise.Free
BitWise.ClassType
BitWise.ClassName
BitWise.ClassNameIs
BitWise.ClassParent
BitWise.ClassInfo
BitWise.InheritsFrom
BitWise.MethodAddress
BitWise.MethodName
BitWise.FieldAddress
BitWise.Dispatch
BitWise.$ClassInit
BitWise.@MetaBitWise.@Create
TBitWise.@MetaTBitWise.Create
TBitWise.@MetaTBitWise.$ClassInit
TBitWise.$ClassInit
TBitWise.Create
TBitWise.@MetaTBitWise.@Create
ByteBitWise.Create
ByteBitWise.byteshr
ByteBitWise.byteshl
WordBitWise.Create
WordBitWise.wordshr
WordBitWise.wordshl
DWordBitWise.Create
DWordBitWise.dwordshr
DWordBitWise.dwordshl
DDWordBitWise.Create
DDWordBitWise.ddwordshr
DDWordBitWise.ddwordshl
BitWise.Create
BitWise.bitwiseshr
BitWise.bitwiseshl
BitWise.bitwiseshr
BitWise.bitwiseshl
BitWise.bitwiseshr
BitWise.bitwiseshl
BitWise.bitwiseshr
BitWise.bitwiseshl
BitWise.bitwiseLO
BitWise.bitwiseHI
BitWise.bitwiseLO
BitWise.bitwiseHI
AB: . |17 | ldc.i4.1 Pop: Pop0 Push: Pushi Type: InlineNone
AC: b |62 | shl Pop: Pop1_pop1 Push: Push1 Type: InlineNone
AD: ħ |9E | stelem.i4 Pop: Popref_popi_popi Push: Push0 Type: InlineNone
// -- Basic Block #8 -- Incoming 2 -- // -- Outgoing 2 -- // -- Index 8 --
AE: . |02 | ldarg.0 Pop: Pop0 Push: Push1 Type: InlineNone
AF: {.... |7B(05 00 00 00 | ldfld Pop: Popref Push: Push1 Type: InlineField 5
B4: . |06 | ldloc.0 Pop: Pop0 Push: Push1 Type: InlineNone
B5: . |02 | ldarg.0 Pop: Pop0 Push: Push1 Type: InlineNone
B6: {.... |7B(05 00 00 00 | ldfld Pop: Popref Push: Push1 Type: InlineField 5
BB: . |06 | ldloc.0 Pop: Pop0 Push: Push1 Type: InlineNone
BC: . |17 | ldc.i4.1 Pop: Pop0 Push: Pushi Type: InlineNone
BD: Y |59 | sub Pop: Pop1_pop1 Push: Push1 Type: InlineNone
BE: – |96 | ldelem.i8 Pop: Popref_popi Push: Pushi8 Type: InlineNone
BF: . |18 | ldc.i4.2 Pop: Pop0 Push: Pushi Type: InlineNone
C0: n |6E | conv.u8 Pop: Pop1 Push: Pushi8 Type: InlineNone
C1: Z |5A | mul Pop: Pop1_pop1 Push: Push1 Type: InlineNone
C2: џ |9F | stelem.i8 Pop: Popref_popi_popi8 Push: Push0 Type: InlineNone
C3: . |06 | ldloc.0 Pop: Pop0 Push: Push1 Type: InlineNone
C4: . |17 | ldc.i4.1 Pop: Pop0 Push: Pushi Type: InlineNone
C5: X |58 | add Pop: Pop1_pop1 Push: Push1 Type: InlineNone
C6: . |0A | stloc.0 Pop: Pop1 Push: Push0 Type: InlineNone
C7: . |06 | ldloc.0 Pop: Pop0 Push: Push1 Type: InlineNone
C8: .@ |1F 40 | ldc.i4.s Pop: Pop0 Push: Pushi Type: ShortInlineI 64
CA: 3” |33 94 | bne.un.s Pop: Pop1_pop1 Push: Push0 Type: ShortInlineBrTarget
$60
// -- Basic Block #9 -- Incoming 1 -- // -- Outgoing 0 -- // -- Index 9 --
CC: * |2A | ret Pop: Varpop Push: Push0 Type: InlineNone
{BitWise.ext8 := new Byte[7];}
{BitWise.ext16 := new Word[16];}
{BitWise.ext32 := new Cardinal[32];}
{BitWise.ext64 := new UInt64[64];}
Object.Create(Self)
BitWise.ext8[0] := 1;
BitWise.ext16[0] := 1;
BitWise.ext32[0] := 1;
BitWise.ext64[0] := UInt64(1);
i := 1;
repeat
if (i < 7) then
BitWise.ext8[i] := Byte(UInt32(BitWise.ext8[i - 1]) shl 1);
if (i < 16) then
BitWise.ext16[i] := Word(UInt32(BitWise.ext16[i - 1]) shl 1);
if (i < 32) then
BitWise.ext32[i] := BitWise.ext32[i - 1] shl 1;
BitWise.ext64[i] := BitWise.ext64[i - 1] * UInt64(2);
i := i + 1;
until (i <> 64);
end ;
function BitWise.bitwiseshr (basic: Borland.Delphi.System.Byte;
n: Borland.Delphi.System.Byte): Borland.Delphi.System.Byte;
var
Result: Borland.Delphi.System.Byte;
begin [Flags:3013,MaxStack:4,CodeSz:13,LocalVarSigTok:0]

Example 2

```
procedure TWinForm.btnCompress_Click(sender: System.Object; e: System.EventArgs);
var
  finfo : FileInfo; finput :FileStream; bwriter : BinaryWriter; ms : MemoryStream; fs : FileStream;
begin
  finfo := FileInfo.Create(textInput.Text);
  if (finfo.Exists) then begin
    finput := finfo.OpenRead();
    ms := MemoryStream.Create;
    bwriter := BinaryWriter.Create(ms);
    LZRWCompressFileToStream(finput, bwriter);
    if (bwriter <> nil) then begin
      fs := FileStream.Create(textOutput.Text,FileMode.Create);
      bwriter.BaseStream.Seek(0,SeekOrigin(0));
      (MemoryStream (bwriter.BaseStream)).WriteTo(fs);
      fs.Close();
      bwriter.Close();
    end;
    finput.Close();
   end;
end;

procedure TWinForm.btnCompress_Click (sender: Object; e: EventArgs);
var
 finfo: FileInfo; finput: FileStream; bwriter: BinaryWriter; ms: MemoryStream; fs: FileStream;
begin [Flags:3013,MaxStack:3,CodeSz:80,LocalVarSigTok:0]
 finfo := FileInfo.Create(Control.get_Text(TWinForm.textInput));
 if (FileSystemInfo.get_Exists(finfo) <> 0) then begin
   finput := FileInfo.OpenRead(finfo);
   ms := MemoryStream.Create();
   bwriter := BinaryWriter.Create(ms);
   TWinForm.LZRWCompressFileToStream(Self, finput, bwriter);
   if (bwriter <> 0) then begin
     fs := FileStream.Create(Control.get_Text(TWinForm.textOutput), 2);
     MemoryStream.WriteTo(MemoryStream(BinaryWriter.get_BaseStream(bwriter)), fs);
     FileStream.Close(fs);
     BinaryWriter.Close(bwriter);
   end;
   FileStream.Close(finput);
 end;
```

Delphi object files decompiler

Delphi .NET object files decompiler

Andrey Mikhailov* Alexey Hmelnov*
mikhailov@icc.ru hmelnov@icc.ru

*Matrosov Institute for System Dynamics and Control Theory
Irkutsk

Ivannikov ISPRAS Open Conference 2017, Moscow
30 november, 2017