ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

UCIrvine

Master Thesis

Upcompiling Legacy Code to Java

Author
Urs Fässler

Supervisor

ETH Zürich	*UC Irvine*
Prof. Dr. Thomas Gross	Prof. Dr. Michael Franz
	Dr. Stefan Brunthaler
	Dr. Per Larsen

September 7, 2012

ABSTRACT

This thesis investigates the process of "upcompilation", the transformation of a binary program back into source code. Unlike a decompiler, the resulting code is in a language with higher abstraction than the original source code was originally written in. Thus, it supports the migration of legacy applications with missing source code to a virtual machine. The result of the thesis is a deeper understanding of the problems occurring in upcompilers.

To identify the problems, we wrote an upcompiler which transforms simple x86 binary programs to Java source code. We recover local variables, function arguments and return values from registers and memory. The expression reduction phase reduces the amount of variables. We detect calls to library functions and translate memory allocation and basic input/output operations to Java constructs. The structuring phase transforms the control flow graph to an abstract syntax tree. We type the variables to integers and pointers to integer. In order to optimize the produced code for readability, we developed a data flow aware coalescing algorithm.

The discovered obstacles include type recovery, structuring, handling of obfuscated code, pointer representation in Java, and optimization for readability, to only name a few. For most of them we refer to related literature.

We show that upcompilation is possible and where the problems are. More investigation and implementation effort is needed to tackle specific problems and to make upcompilation applicable for real world programs.

ACKNOWLEDGEMENTS

I like to thank Professor Michael Franz for the invitation to his group and making this work possible. A special thank goes to Stefan Brunthaler and Per Larsen for the discussions and suggestions. I thank Professor Thomas Gross for enabling the work. Finally, I like to thank the supporters of free software for providing all tools needed for this work.

CONTENTS

1

INTRODUCTION

Legacy applications without source code become a considerable problem when their target processors are no longer available. Cross-compilation or binary translation, i.e., translating the binary application to another instruction set architecture (ISA), are not sustainable solutions, as the target architecture also becomes obsolete eventually.

In contrast, the Java virtual machine (JVM) decouples the source code from any particular processor. The JVM itself runs on a wide range of processors. As the specification of the JVM as also the Java language are publicly available and there exist free and open source implementations of the JVM and Java compiler, it provides a future proof architecture. Moreover, the language is widely used in the industry and academic communities. If we are able to translate binaries into Java source code, they do not only compile for the JVM but also allows programmers to fix, improve or extend the applications.

In this thesis we investigate the process of "upcompilation". An upcompiler translates a binary program into source code of a language of higher abstraction than the program was originally written in. In comparison to decompilation, upcompiler face the additional challenges of bridging a broader semantic gap. The gain of this thesis is a deeper understanding of the problems occurring when building a real world upcompiler.

1.1 PROBLEMS

Disassembly and decompilation are already difficult problems since a compiler destroys a lot of information. As we want to upcompile, we do not only have the same problems but also new ones. The reason is that we want to go to an higher abstraction than the code was before compiling. Figure 1.1 on the following page shows the levels of machine abstraction of different forms of a program.

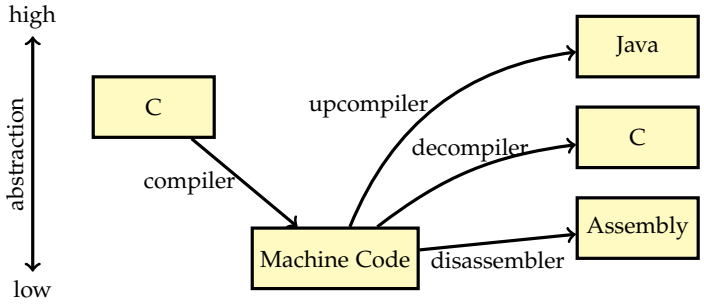


Figure 1.1: Machine code abstraction level. The reconstructed C code on the right side is on a lower abstraction than the original because information was destroyed during compilation.

On the disassembly level, we have to distinguish between code and data. In the general case, it is considered as a hard problem [18]. Since we have the entry point to the binary and follow all paths, we avoid this problem. Disassembly the binary gets difficult if code obfuscation is used or the code is self modifying.

Decompilation is more difficult as we want to recover variables, their types, and high level language constructs. A value can be stored in a register, memory or both. But not every value in a register or memory represents a variable. As the machine needs to store the result of every operation, several registers are used. It is not clear which values belonged to a variable and which ones are for temporary storage. In the absence of debugging information, only a small amount of information about types is available. This includes the size of an integer or if it is a signed value. Moreover, there is no explicit information about composite types like arrays or records.

Optimization frequently moves code around, which makes the recovery of high level language constructs difficult or impossible. Nevertheless, since C has a low machine code abstraction, several problems are not so hard if we do not care about readability of the resulting code. Otherwise, optimizations like function inlining and loop unrolling have to be detected and reverted. Furthermore, the compiler or linker add helper code to the program. Such code has to be detected and removed, too.

Upcompilers have the additional challenge of bridging the semantic gap. This is the problem that a language of higher abstraction has more constraints than the language of the original program. For example, C or Object Pascal allows to access every memory address with a pointer of any type. In Java or Ruby, neither accessing memory nor arbitrary type casts are possible.

As we have limited resources, we restricted the possible set of input programs. This allows us to implement all parts of the upcompiler.

We only worked with 32-bit x86 binaries compiled by Clang, a C compiler with LLVM as back end. All binaries need to be in the Executable and Linkable Format (ELF). We only support dynamically linked libraries, and only a small number of library calls. Those are `printf` and `scanf` with simple static format strings as well as `malloc` and `free`. Multi-threading, signals and function pointers are not supported. Parameters to functions have to be passed on the stack. Supported data types are signed integers and pointers to integers.

Obfuscated code, hand written assembly code, computed jumps, just-in-time compilers, self modifying code and similar techniques are not considered in our implementation.

2

RELATED WORK

First decompilers are mentioned in the 1960's by Halstead [19]. Back then, they had the same motivation as we have for the upcompiler: It was used to guide the migration of programs to a new computer generation.

Modern decompilation is based on the work of Cifuentes [11]. Since then, findings in research and industry changed and improved compilers. Most notably, the static single assignment form (SSA) [15] is used by many modern compilers. Its usefulness for decompilation is described by Mycroft [25] and Emmerik [18].

Upcompiling binaries to Java or similar targets is a new research topic. Nevertheless, it should be possible to reach the same goal with the usage of several existing tools. As there exists a lot of research on decompilers, it seems obvious to first decompile a binary and then use a C to Java converter. There exist decompilers like Boomerang [1] or REC [2] producing C like code, and C to Java converter like C2J [3], but they all seem to be immature or incomplete.

It should be possible to use tools, such as RevGen/s2e [10] to decompile binaries to LLVM IR. From there, LLJVM [26] produces Java bytecode. Many Java decompilers exist; some of them may be able to work as the last step to retrieve Java source code. Unfortunately, both RevGen and LLJVM are still prototypes.

NestedVM [7] is able to translate MIPS binaries to Java Bytecode. The authors choose MIPS because of "the many similarities between the MIPS ISA and the Java Virtual Machine" (Alliet and Megacz [7, page 3]). Since our motivation is to upcompile legacy applications for any processor architecture, we can not rely on this solution.

3

IMPLEMENTATION

The upcompiler is written in Java using additional libraries. ELF parsing is based on the source code of Frida IRE [20]. DiStorm3 [16] disassembles the binary. JGraphT [4] provides the graph infrastructure.

The translation from machine code to Java is split up into several phases. Every phase can perform multiple passes over the internal representation. We use a control flow graph (CFG) for the low level representation and later an abstract syntax tree (AST) for the high level representation.

We describe the phases of the upcompiler in the following sections. For practical reasons, they are not strictly in the same order as they are implemented. In most sections, a listing or figure illustrates code snippets before and after the phase. At the end of every section, we indicate the relevant classes of the upcompiler with an arrow (⇒).

3.1 LOW LEVEL PHASES

The whole program consists of a set of functions. Every function is represented as a CFG where the vertices are basic blocks. There is exactly one vertex with an in-degree of zero, the entry point of the function. A function has one or more exit points. The registers and recovered variables are in SSA form.

Every basic block is constructed in the same way. It is identified by the memory address of its first instruction, even if this instruction is deleted during the process. At the beginning, zero or more phi functions are allowed. They have a key/value pair for every incoming edge of the basic block. The key is the ID of the source basic block of the edge and the value is the expression which is used when the control flow transfers from the mentioned basic block. Then, zero or more statements follow. At the end, there is exactly one `jump` or `return` statement.

The syntax is intuitive, only the jump statement needs a description. It takes two arguments, an expression and a list of basic block identifiers. The expression is evaluated and that value is used as index for the list. Finally, the control flow transfers to the basic block at the mentioned index. This representation models all occurring jumps. Listing 3.1 on the current page shows the use cases.

```
1 jump( 0, [BB_N] )
2 jump( a < 10, [BB_F, BB_T] )
3 jump( a, [BB_0, BB_1, BB_2] )
```

Listing 3.1: The 3 different use cases of the jump statement. Line 1 contains an unconditional jump, line 2 represents a jump depending on a boolean condition and line 3 contains a multi-branch.

3.1.1 *Disassembler*

First, the upcompiler disassembles and splits the binary program into basic blocks. The disassembly library provides the decoded assembly instructions which are directly translated into our intermediate representation.

In addition, we load the dynamic library section for the later resolution of addresses to library function names as discussed in Section 4.1.7 on page 29.

We detect all functions of the call graph with the entry point of the binary as the root vertex. Whenever a call instruction occurs during parsing, we add the destination address to a work queue. Already parsed functions and library functions are excluded.

Basic blocks of a function are parsed similarly. The first basic block starts at the address of the function. For a conditional jump, we put the address of the following instruction and the jump destination to a work queue. If the jump is unconditional, only the destination address is added. Those jump instructions also terminate the basic blocks. In addition, function return or program abort instructions also terminate a basic block. If a jump instruction points into an already parsed basic block, we split that basic block at the destination address. Before parsing an instruction, we check if there is already a basic block starting with this instruction. If so, we add a jump to that basic block. We discuss the handling of jump tables in Section 4.1.4 on page 27.

Listing 3.2 on this page is an example of a basic block after this phase.

⇒ `phases.cfg.Disassembler`

```
0x8048480:
EDX:22 := 'ESI'
ESI:23 := 'ECX'
ESI:24, C:24, P:24, A:24, Z:24, S:24, O:24 := ('ESI' + 'EDX')
EDI:25, P:25, A:25, Z:25, S:25, O:25 := ('EDI' - 0x1)
ECX:26 := 'EDX'
jump( ('Z' == 0x0), [0x804848b, 0x8048480] )
```

Listing 3.2: A basic block after the disassembly phase.

3.1.2 *Variable linking*

In this phase we link all variables by converting them to SSA. To get the positions for the phi functions, we implemented a generic dominator tree algorithm as described by Cooper et al. [14] and a generic dominance frontier algorithm as in Cooper and Torczon [13].

We only consider registers and flags as variables as we have no aliasing information about memory. Registers can also be aliased, for example `EAX` with `AL`, but we know all possible cases and treat them accordingly (see Section 4.1.9 on page 29 for a discussion about aliasing).

⇒ `phases.cfg.VariableLinker`

3.1.3 *Function linking*

After this phase all function calls are linked to the function definition. We also detect the variables used to return values from the function.

Library functions are assumed to have C calling convention (cdecl), therefore we know which variable holds the return value. However, it is not known for private functions, as the compiler can use any calling convention.

We implemented an algorithm to detect the variables used for return values. With the algorithm shown in Listing 3.3 on the following page, we search the variables killed by the function `f`. If a variable `v`

is used after a call of `f` and killed by `f`, we know that `v` is a return value of `f`.

```
1  function dfs( f )
2      kill[f] := gen[f]
3      visited := visited ∪ f
4      for g ∈ callee[f]
5          if g ∉ visited
6              dfs( g )
7          kill[f] := kill[f] ∪ kill[g]
8      kill[f] := kill[f] - save[f]
```

Listing 3.3: Algorithm to detect killed variables. Called on a function `f`, we get the killed variables in `kill[f]`. The variables written by a function `f` are in `gen[f]`, the ones saved on the stack are in `save[f]`.

After we added the variables with the return values, we run the variable linker again. A subsequent step removes all superfluous phi nodes. This ensures that all variable references are linked to a definition.

⇒ `knowledge.KnowKilledByFunction`
`phases.cfg.FunctionVariableLinker`

3.1.4 *Condition code reconstruction*

In this phase we replace the flag tests in conditions with relational operators. Depending on the flags tested and the instruction generating the flags, we know the relation. After this step, all flags should be removable. Listings 3.4 on the current page shows an example.

⇒ `phases.cfg.ConditionReplacer`

```
1  C:27, P:27, A:27, Z:27, S:27, O:27 := (ECX:23 and ECX:23)
2  jump( (Z:27 == 0x0), [0x804850f, 0x8048500] )
```

```
1
2  jump( (ECX:23 != 0x0), [0x804850f, 0x8048500] )
```

Listing 3.4: Before and after the condition replacement phase. The `and` operation is removed since it is no longer used.

3.1.5 *Stack variable recovery*

After this phase we have array accesses instead of memory accesses relative to EBP and ESP. In addition, the caller and the callee use function parameter instead of memory accesses.

Local variables from the original source program are translated to registers or memory accesses relative to the stack pointers. If a memory access is above of EBP, we know that this is an argument. By checking all memory accesses we find the highest offset relative to EBP and therefore the number of arguments. This works because we only have integers and pointers as arguments. Below EBP we find the saved registers and after them the local variables and the arguments for the callees. There is no need to separate these. Figure 3.1 on the following page shows the phases of variable recovery.

In the callees, we replace memory accesses to the argument area by accesses to the newly introduced argument-variables. At every call expression, we insert the actual arguments. The arguments are the stack-array elements at the corresponding positions; the number of arguments is known from the callee.

The area for local variables is seen as an array. Memory accesses are translated into array accesses. If there is an access with a static array offset, we split the array at that position. This separates the different variables of each other. For the limitations, see Section 4.1.6 on page 28.

Listing 3.5 on this page shows the difference before and after this phase.

⇒ `phases.cfg.FunctionRecovery`

```
1 *(ESP:0 + 0x0) := 0x80486d0
2 call 'puts'()
```

```
1 loc0[0x0] := 0x80486d0
2 call 'puts'(loc0[0x0])
```

Listing 3.5: Changes of the stack variable recovery phase. Memory accesses relative to the stack pointer are replaced by introducing local arrays. Function calls get their actual arguments.

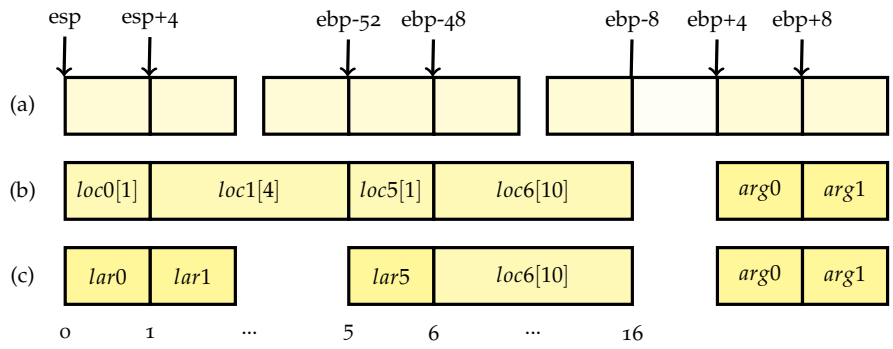


Figure 3.1: Transformation of stack memory to local variables and arguments. The arrows on top illustrate direct pointer accesses, the numbers at the bottom the array indices. Saved registers, stack pointer and the return address are in the area between `ebp-8` and `ebp+4`. They are not used in the recovering process.
First, there are only memory accesses relative to `esp` and `ebp` as on row (a). We consider this part of the stack to be an array. The stack variable recovery phase (see page 11) splits the array at the indices of memory accesses, resulting in row (b). We also replace the stack area above `ebp+4` with function argument variables. After the phase where we replace arrays with variables (see page 15) we get row (c). The variables `loc0`, `loc1` and `loc5` are introduces since there was no index calculation on the corresponding arrays.
The variables relative to `esp` are arguments to functions, the ones relative to `ebp` are local variables. We see that this function has at least one local integer variable, and one array with 10 elements. Moreover, it calls functions with no more than 2 arguments.

3.1.6 *Expression reduction*

This phase reduces expressions by inlining. First, we reduce tautological expressions like an addition with zero or dereferencing of address-of operations. Second, we inline expressions. For every usage of a variable we calculate the cost for inlining the definition of the variable. The cost function counts the number of operation and function calls for an expression. References to variables or memory have no cost. Then we multiply the cost with $\#usages - 1$ of this expression. This ensures a cost of zero if the expression is used only once. If the cost is zero, we replace the reference to the expression with the expression itself. Movement of instructions is only allowed within one basic block. Code in phi functions belongs to the previous basic blocks.

A memory model check can prevent the movement of expressions, too. We need the memory model because we do not have information about aliasing for all memory accesses. Moving arbitrary reads with respect to other reads is allowed, arbitrary writes cannot be reordered with respect to other writes or reads. If we know that the memory locations are different, we allow the movement.

Finally, we clean up by removing all definition of unused variables and all statements without effect. These are statements which do not define a variable, do not write to memory and with no function calls. Listing 3.6 on the current page gives an impression of this phase.

$\Rightarrow$ `phases.cfg.MathReduction`
`phases.cfg.SsaReduce`

```
1 EAX:17, EDX:17, ST0:17 := call 'malloc'(loc0[0x0])
2 ESI:18 := EAX:17
3 ...
4 EAX:30 := loc3[0x0]
5 ECX:31 := *((EAX:30 * 0x4) + (ESI:18 + -0x4))
6 loc2[0x0] := ECX:31
```

```
1 EAX:17 := call 'malloc'(loc0[0x0])
2
3 ...
4 EAX:30 := loc3[0x0]
5
6 loc2[0x0] := *((loc3[0x0] * 0x4) + (EAX:17 + -0x4))
```

Listing 3.6: Effects of expression reduction. On Line 1 it can be seen how the unused variables are removed, line 6 shows the effect of inlining.

3.1.7 *Function replacement*

This phase replaces all calls to known library functions by calls to internal functions. For some functions, like `malloc` and `free`, this is just a replacement of the call destination.

However, functions like `scanf` and `printf` need special handling. They have a variable number of arguments, which we do not support in general. We implemented a parser for static `scanf` and `printf` format strings. The current solution requires that the format string resides in the read-only data section of the program.

For every read of an integer, a call to the function `readInt` is inserted and the return value is assigned to the corresponding variable. For every format string of a `printf` call, we create a new function with the same functionality. Listing 3.7 on this page shows some examples of the translation.

⇒ `phases.cfg.IoFunctionReplacer`
`phases.cfg.MemoryFuncReplacer`

```
1  loc1[0] := @(loc3[0])
2  loc0[0] := 0x80486aa
3  call 'scanf'(loc0[0])
4  EAX:7 := call 'malloc'(4*loc3[0])
5  ...
6  loc1[0] := EAX:12
7  loc0[0] := 0x80485d5
8  call 'printf'(loc0[0])

21 0x80485d5:
22 "result: %i\n"
23 0x80486aa:
24 "%i"
```

```
1  loc1[0] := @(loc3[0])
2  loc0[0] := 0x80486aa
3  loc3[0] := call readInt()
4  EAX:7 := call ptrMalloc(4*loc3[0])
5  ...
6  loc1[0] := EAX:12
7  loc0[0] := 0x80485d5
8  call f80485d5(loc1[0])
9  ...
10 f80485d5:
11 call writeStr("result: ")
12 call writeInt(arg0:0)
13 call writeNl()
14 return
```

Listing 3.7: Replacement of functions. On line 1 to 3 one can see how `scanf` is replaced with information of the format string on line 24. The `malloc` function on line 4 is replaced by `ptrMalloc`. The call of `printf` on line 8 is replaced by a call to a new function with the same semantics as the format string seen on line 22.

3.1.8 *Replacing arrays with variables*

We replace arrays with variables in order to reduce more expressions. Some arrays have only one element, others have no index calculation when accessing them. After an expression analysis does not find an address-of operation on the array, we replace it with a variable. Since variables can not alias each other, another run of the variable reduction phase as in Section 3.1.6 on page 13 simplifies the code again. Listing 3.8 on this page shows the replacement of the arrays and the result of the reduction, Figure 3.1 on page 12 explains the idea.

⇒ `phases.cfg.ArrayReplacer`

```
1 EAX:30 := loc3[0x0]
2 loc2[0x0] := *((loc3[0x0] * 0x4) + (EAX:17 + -0x4))
3 loc1[0x0] := EAX:30
4 call f80486ad(loc1[0x0], loc2[0x0])
5 loc0[0x0] := EAX:17
6 call ptrFree(loc0[0x0])
```

```
1 EAX:30 := lar_loc3_0:0
2 lar_loc2_0:0 := *((lar_loc3_0:0 * 0x4) + (EAX:17 + -0x4))
3 lar_loc1_0:0 := EAX:30
4 call f80486ad(lar_loc1_0:0, lar_loc2_0:0)
5 lar_loc0_0:0 := EAX:17
6 call ptrFree(lar_loc0_0:0)
```

```
4 call f80486ad(lar_loc3_0:0, *((lar_loc3_0:0 * 0x4) + (EAX:17 + -0x4)))
5
6 call ptrFree(EAX:17)
```

Listing 3.8: Effects of array replacement together with reduction. We can replace most arrays with variables. Therefore, a repeated reducing phase can inline and remove a significant amount of copy operations and temporary variables.

3.1.9 *SSA destruction*

In this phase, we go out of SSA by removing all phi functions. It is necessary since SSA is not compatible with our AST. As a prerequisite, we check that all variables are linked to their definition. Then we rename all variables to be sure that no two variables with the same name exist. This is necessary because we do not support different versions of the same variable name later on.

We use the straight forward SSA destruction algorithm "Method I" described by Sreedhar et al. [30]. It is a simple method with the downside that it introduces a lot of new variables. The coalescing phase in Section 3.2.6 on page 21 removes many of them. Listing 3.9 on the current page gives an idea how the algorithm works.

⇒ `phases.cfg.SsaBacktranslator`

```
A:
jump( 0, [C] )



B:
jump( 0, [C] )



C:
tmp2 := phi( A::0; B::*(tmp5-4); )
tmp5 := phi( A::tmp1+8; B::tmp5+4; )
```

```
A:
tmp2_t := 0
tmp5_t := tmp1+8
jump( 0, [C] )

B:
tmp2_t := *(tmp5-4)
tmp5_t := tmp5+4
jump( 0, [C] )

C:
tmp2 := tmp2_t
tmp5 := tmp5_t
```

Listing 3.9: Before and after SSA destruction. The destruction algorithm adds an additional variable for every phi function. Many of them are removed in the coalescing phase (see section 3.2.6 on page 21).

3.2 HIGH LEVEL PHASES

The AST for the second part consists of high-level language constructs, such as `while` and `if` statements. The expressions start by more C like forms with pointer dereferencing. They are successively reshaped into a Java syntax. Finally, we optimize the code and generate the Java files.

We have the following types of statements:

- nop
- block
- call
- return
- variable definition
- assignment
- while
- do-while
- if
- switch-case

They are C and Java compatible, but we added more constraints. Loops have no break or continue and the switch-case construct can not have fall-through cases.

During the phases more and more code is dependent on three helper classes. `Main` in Listing D.1 is used to handle the return value of the main function and the arguments. `IoSupport` in Listing D.2 provides methods to read from and write to the console. `Pointer` in Listing D.3 is needed for pointers since they do not have a corresponding language concept in Java.

3.2.1 *Structuring*

This is the phase where we transform the CFG into the AST. For every function, our algorithm searches for specific patterns of vertices in the CFG. If one is found, the algorithm replaces the vertices by a semantically equivalent vertex. This is done until only one vertex is left. Figure 3.2 on the following page shows the structuring process.

It is possible that we can not find a pattern. Such a CFG result from manual jumps in the source program, unknown program constructs and compiler optimizations. We implemented a fallback mode for such cases. In this mode, we simulate the jumps with a switch-case in an infinite loop. It is similar to an interpreter using a switch-based dispatch loop. Every case entry consists of one basic block where we replace the jump at the end of every basic block with a write to the control variable of the switch.

There is room for improvement as we discuss in Section 4.1.5 on page 28.

⇒ `phases.ast.Structurer`

3.2.2 *Linking*

Here, we insert the definitions for the variables, link the references to them, and link function references to their definition, too. Contrary to the SSA representation, we need a specific definition for every variable. To make it simple we insert all variable definitions in the beginning of the function.

In addition we link and rename the functions. As we focus on stripped binaries, we rename all functions, even if the name of the function is available. The entry-function gets the name “main”, all others have a generic, numbered name.

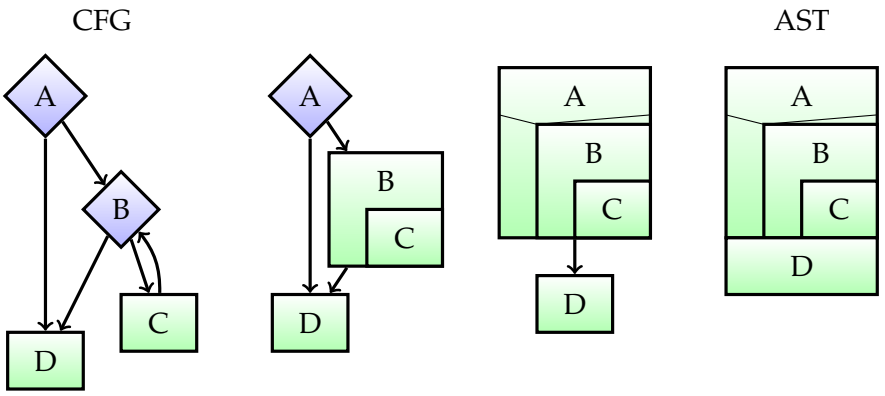


Figure 3.2: From the CFG to AST. The algorithm searches for the pattern shown in Figure 3.3 on this page. If a pattern is found, the corresponding vertices are merged until only one vertex remains. A fallback mode ensures that any control-flow graph can be represented in the abstract syntax tree.

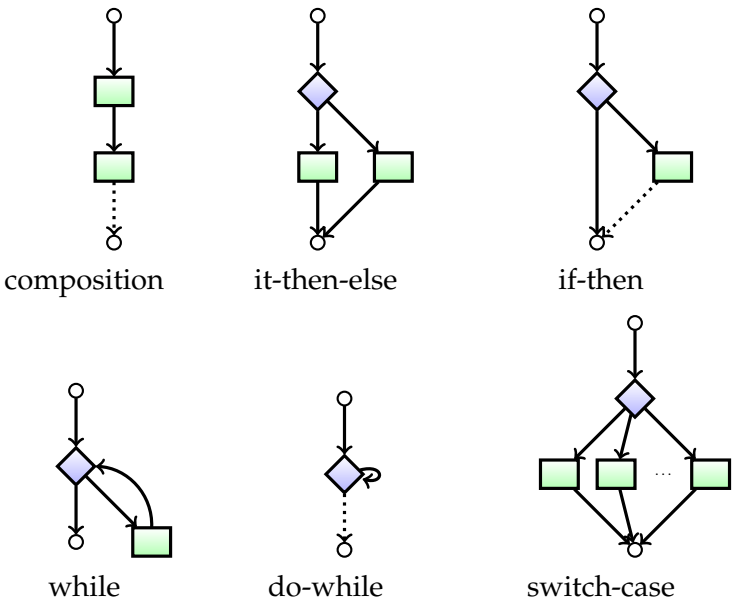


Figure 3.3: Recognized patterns in the CFG.

⇒ `phases.ast.VariableLinker`
`phases.ast.FunctionLinker`

3.2.3 *Statement normalization*

In this phase we clean up the AST. Our structuring algorithm introduces several block statements, most of them have no functional use. We reduce those whenever possible. In addition, we remove dead code.

As a result, we get code which has just as many blocks as necessary. Listing 3.10 on this page shows code after this phase.

⇒ `phases.ast.StmtNormalizer`

```
1  f8440:
2
3  jump( tmp4 < 2, [0x8450, 0x846c] )
4  0x846c:
5  tmp1_t := tmp4
6  jump( 0, [0x846e] )
7  0x8450:
8  tmp2 := call f8440(tmp4 + -1)
9  tmp3 := call f8440(tmp4 + -2)
10 tmp1_t := tmp3 + tmp2
11 jump( 0, [0x846e] )
12
13 0x846e:
14 tmp1 := tmp1_t
15 return tmp1
16
```

```
1  public ? func0( ? tmp4) {
2    ? tmp1, tmp1_t, tmp2, tmp3;
3    if(tmp4 < 2){
4
5      tmp1_t = tmp4;
6
7    } else {
8      tmp2 = func0(tmp4 + -1);
9      tmp3 = func0(tmp4 + -2);
10     tmp1_t = tmp3 + tmp2;
11
12   }
13
14   tmp1 = tmp1_t;
15   return tmp1;
16 }
```

Listing 3.10: A function in low- and high-level representation. The code on the same line has identical semantics on both sides. It is not yet optimized.

3.2.4 *Reconstructing types*

After this phase, every variable, argument and function has a type. In order to keep the reconstructing algorithm simple, we only consider information available from the AST. It is possible due to the fact, that we limited the types of the input program to integer and pointer to integer. More sophisticated methods are discussed in Section 5.2.1 on page 40.

We get most information from expressions like pointer dereferencing. The type of a variable is propagated whenever applicable, i.e., when assigned to another variable, when used as argument or in an expression. This allows the type reconstruction for most variables. If there is no information available, it has to be an integer due to the mentioned constraints. Listings 3.11 on this page shows the first part of a function after type reconstruction.

⇒ `phases.ast.TypeReconstruction`
`knowledge.KnowTypes`

```
1 public void func0( Pointer<Integer> tmp1, int tmp0 ){
2   int tmp4_t = 0;
3   Pointer<Integer> tmp5_t = null;
4   *tmp1 = 0;
5   *(tmp1 + 4) = 1;
6   if( !(tmp0 < 3) ){
7     tmp4_t = tmp0 + -3;
8     tmp5_t = tmp1 + 8;
```

Listing 3.11: Head of a function after type reconstruction. The variables are initialized with zero or `null` to suppress error messages by the Java compiler. Since the code contains pointer dereferencing operations, it is not yet Java compatible.

3.2.5 *Pointer access encapsulation*

We replace all pointers by pointer objects and dereferencing operators with method calls on the pointer object. The pointer class contains a reference to an Java array and an index. Since the array represents only part of the memory, the pointer can operate only on this part, too. The whole class is in Listing D.3 on page 51. We replace every read access to a pointer with a copy of the pointer object as copying the reference results in an wrong behaviour. Listing 3.12 on this page shows an example.

⇒ `phases.ast.PointerReplacer`

```
1 *tmp1 = 0;
2 *(tmp1 + 4) = 1;
3 tmp5_t = tmp1 + 8;
```

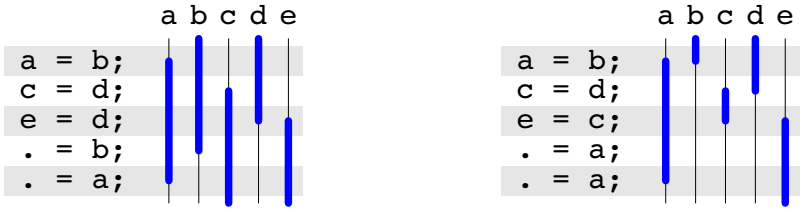
```
1 tmp1.setValue( 0, 0 );
2 tmp1.setValue( 1, 1 );
3 tmp5_t = new Pointer<Integer>(tmp1, 2);
```

Listing 3.12: Pointer access encapsulation. The variables `tmp1` and `tmp5_t` are of the type `Pointer<Integer>` which is shown in listing D.3 on page 51.

3.2.6 *Coalescing*

In this phase, we reduce the number of variables. It is a pure optimization phase. For a discussion about optimization, see Section 5.2.3 on page 41.

As a preparation, we shorten the live range, i.e., the distance between the definition and the last use of variables. This separates some live ranges, leading to better coalescing results. The algorithm replaces reads of a variable x with a different variable y whenever $x = y$ and the value of y was assigned later than the one of x. Listing 3.13 on this page illustrates the idea.



Listing 3.13: Live range shortening. The live range of every variable is shown as bold vertical line. After this phase, the live ranges are shorter, leading to better coalescing results.

For coalescing, we developed an algorithm based on graph coloring as described by Chaitin et al. [8]. Our changes are due to different goals. We are interested in high readability rather then good performance. The requirements are different, too. In brief, we do not need to eliminate as many variables as possible since there is no limitation. Actually, we add more constraints as we do not coalesce unrelated variables. For a full description of the goals as also the algorithm see Section 4.1.11 on page 30.

Finally we do copy propagation. It undoes the work of the preparation, i.e., extending the live ranges of the variables. In addition, we remove copy statements where the destination variable is the same as the source variable.

Listing 3.14 on page 23 shows the changes of a code snippet after the different operations.

$\Rightarrow$

```
phases.ast.LiveRangeShortening
phases.ast.Coalescing
phases.ast.CopyPropagator
phases.ast.VariableReduction
```

3.2.7 *Expression normalization*

This phase cleans up expressions. It simplifies negated compare operations and removes idempotent arithmetic operations. Finally, statements are normalized as described in Section 3.2.3 on page 19.

⇒ `phases.ast.ExprNormalizer`

3.2.8 *Generating Java code*

The final phase renames the variables and generates the Java source code file. In addition, the helper classes as shown in listings D.1, D.2 and D.3, all in the appendix, are written. Listing 3.15 on the facing page gives an idea how the Java code looks like.

⇒ `phases.ast.VariableNamer`
`phases.ast.JavaWriter`

(a)

```
1  x0 = 1;
2  y1 = ...;
3  x3 = 0;
4  do {
5    x2 = x0;
6    y0 = y1;
7    x1 = x3;
8    y2 = y0 - 1;
9    x0 = x1 + x2;
10   y1 = y2;
11   x3 = x2;
12   x4 = x2;
13 } while(y2 ≠ 0);
14 ... = x4;
```

(b)

```
1  x0 = 1;
2  y1 = ...;
3  x3 = 0;
4  do {
5    x2 = x0;
6    y0 = y1;
7    x1 = x3;
8    y2 = y0 - 1;
9    x0 = x1 + x2;
10   y1 = y2;
11   x3 = x2;
12   x4 = x3;
13 } while(y1 ≠ 0);
14 ... = x4;
```

(c)

```
1  x0 = 1;
2  y0 = ...;
3  x1 = 0;
4  do {
5    x2 = x0;
6    y0 = y0;
7    x1 = x1;
8    y0 = y0 - 1;
9    x0 = x1 + x2;
10   y0 = y0;
11   x1 = x2;
12   x2 = x1;
13 } while(y0 ≠ 0);
14 ... = x2;
```

(d)

```
1  x0 = 1;
2  y0 = ...;
3  x1 = 0;
4  do {
5    x2 = x0;
6    ;
7    ;
8    y0 = y0 - 1;
9    x0 = x1 + x0;
10   ;
11   x1 = x2;
12   ;
13 } while(y0 ≠ 0);
14 ... = x2;
```

Listing 3.14: Effects of coalescing. The input code is listed in (a). After live range splitting we get (b). The affected variables are printed bold. Coalescing produces the code in (c). The following variables are coalesced: $\{y_0, y_1, y_2\} \rightarrow y_0$, $\{x_1, x_3\} \rightarrow x_1$ and $\{x_4, x_2\} \rightarrow x_2$. Finally, copy propagation produces the code in (d).

```
1  public int main( ) {
2    int var0 = 0;
3    int var1 = 0;
4    Pointer<Integer> var2 = null;
5    func2();
6    var1 = IoSupport.readInt();
7    var2 = new Pointer<Integer>((( (var1 >= 2) ? (var1 * 4) : 8 ) / 4));
8    if( (var2 != null)) {
9      func0(var2, var1);
10     func3(var1, var2.getValue( (-1 + var1) ));
11     var2.free();
12     var0 = 0;
13   }
14   else {
15     func1();
16     var0 = -1;
17   }
18
19   return var0;
20 }
```

Listing 3.15: Upcompiled Java code.

4

EVALUATION

First, we discuss problems and ideas related to upcompiling. In the second part, we present performance measurements on both, the upcompiled programs and the upcompiler itself.

4.1 DISCUSSION

We explain problems we have had and ideas we came up during implementation. For some problems we did not had the resources to tackle them, for others we refer to the available literature.

4.1.1 *Function pointers*

The upcompiler does not yet handle function pointers. We see three approaches to support them. For all of them, we have to know the functions called via a function pointer. It can be a conservative guess resulting in considering all functions.

The first idea is that we create a dispatcher function for function pointer types. These are functions with the same return type and arguments as the function to be called. An additional argument contains the address of the function we want to call with those arguments. The body of this function consists of one switch-case statement with the function address as case value and the function call as the only code. A call of a function pointer results in a call of the dispatcher function with the function pointer as address.

The second idea is based on single access methods (SAM). We create a class for every function. All those classes implement an interface containing one method declaration with the signature of the original function. One instance for every class is created at startup of the program. Pointers to a function are replaced with a reference to such an instance. A call of a function pointer results in a method call on the referenced instance.

The third idea uses lambda expression. It is similar to the second idea with the difference, that we do not generate an object for each function but a lambda expression. We also create them at startup and use references whenever we had a function pointer in the original program. The Java specification request 335 [5] describes the implementation of lambda expressions. It is not yet included in Java but on the roadmap for JDK 8.

4.1.2 *Function argument reconstruction*

The upcompiler detects only arguments passed on the stack. Arguments passed by registers are not yet handled, they appear as unlinked variables. As there exists literature from Emmerik [18] on that problem, we did not investigate this issue in detail.

The usage of integers as the only data type simplifies the problem considerably. For a real world application, every data type has to be supported. Furthermore, the behavior of the upcompiler is undefined when, for example, pointer arithmetic is done on a composed argument passed by value.

We do not generally consider variable length function arguments. It is a challenging problem as the upcompiler has no information how the number of arguments is calculated. Apart from this, most C code will use `va_start`, `va_arg` and `va_end`. Since they are preprocessor macros and therefore inlined, they are hard to detect.

4.1.3 *Non returning functions*

Some functions, like `exit` from the C library, do not return. The compiler is aware of this and hence it does not produce code for stack-frame cleanup or function epilogue. We only see the function call and do not recognize the end of the function. For library functions, this is not a problem since we know whether they return or not. Unfortunately, it can not be known for user functions. The problem in our implementation is that we consider all following code as part of the function.

A solution might be that we use pattern matching to detect function boundaries. This approach may produce wrong results, since some functions have no specific pattern or some pattern exists inside a function. Another method is a similar approach we use for the detection

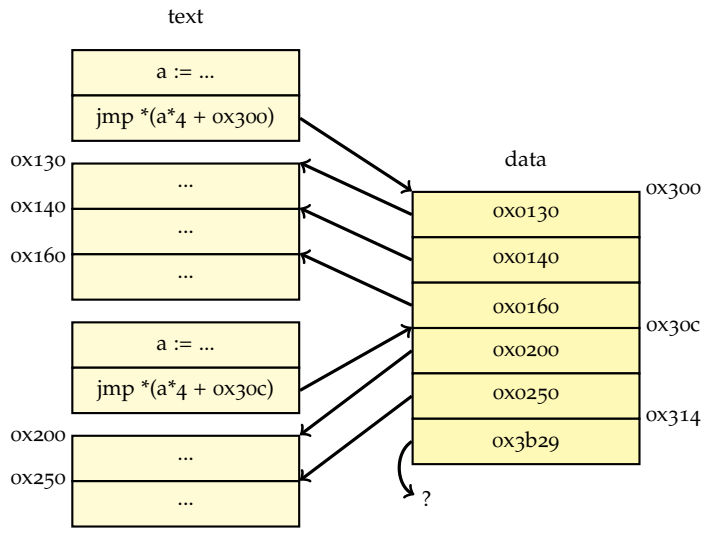


Figure 4.1: Two jump tables in a program. As the value at address 314_{16} does not point into the text section, it does not belong to the jump table.

of basic blocks, i.e., we do nothing and split the functions later. Split points are then the destination addresses of call instructions.

4.1.4 *Jump tables*

Switch-case statement recovery is not a trivial task as pointed out by Cifuentes and Emmerik [12]. As there is already literature available, we only describe how we recover jump tables. Jump tables are generated when a switch-case statement has several successive numbered cases. The value of the control expression is used as index of the jump table. At the indexed position, an address into the code is found. The control flow is directed to this address by a jump instruction. Figure 4.1 on the current page shows how the jump table looks in the compiled program.

While we disassemble the binary, we need the destination address of the jump table to find all basic blocks. To simplify the detection, we interpret every computed jump as a jump table. It is possible due to our constraints on the input programs. In the computation, an address of the read-only data segment is involved; this address marks the start of the jump table.

To find the size of the jump table we need a data flow analysis. However, it is not possible to do this analysis at the current time, as it needs information from a later phase. Therefore, we assume the size

of the table as big as possible. One limitation is the end of the data segment. The other limitation is not obvious. Every entry in the table points into the text segment since it is a jump destination. As soon as an entry does not point into the text segment we know that it does not belong to the table. With this method, we probably read too many entries. The superfluous ones can be removed in a later phase. This is a working algorithm, but there is much room for improvements.

4.1.5 *Structuring*

Our structuring algorithm uses a very narrow set of patterns. Since Java has labeled `break` and `continue`, they can also be used to get better code. In addition, fall through in the switch-case statement is not considered.

The problem of structuring is not new. Lichtblau [22], Cifuentes [11] and Simon [28] provide more information for improvement of structuring.

4.1.6 *Stack variables*

The simple method to detect stack variables, as illustrated in Figure 3.1 on page 12, is error prone. If an array is not only accessed by index 0 but also at other, constant indexes, our algorithm splits the array at those positions, too. We encountered cases where the compiler optimized in such a way that the only direct array access was at index -1. In addition, the array size can be determined to be longer than used if the memory after the array is used for padding.

We expect better results with data flow analysis and range checks for dynamic arrays accesses. In addition, a way to use information provided by the user is necessary, since we do not expect perfect results for every input.

There exists literature about the topic; Chen et al. [9] propose a shadow stack and Mycroft [25] discusses several problems and ideas.

4.1.7 *Library calls*

Detecting library calls is necessary to properly link functions. A later translation of the library calls also depends on correct recognized library functions. Contrary to previous work by Cifuentes [11], it is no longer a difficult problem when using ELF binaries with dynamic linked libraries on Linux.

To find the translation of addresses to function names, we use the same method as the dynamic linker does. A database provides further information about the library functions, such as the names and types of the arguments. Currently, the database entries are hand crafted. To some extent it should be possible to generate the database automatically from the header files of the libraries.

4.1.8 *Tail calls*

Tail call elimination is an optimization where the last function call in a function is replaced by a jump instruction. With this technique, there is no need to allocate an additional stack frame. There is no return instruction after the call statement, as the return of the function we jumped in is used.

A tail call is not a problem as the algorithm just follows the jumps. The called function is seen as part of the original function. If multiple functions "tail call" a function, the code of the callee is duplicated for every caller. It is not a problem with respect to correctness. The readability is higher when we detect such tail calls, but it can be part of the optimization as described in Section 5.2.3 on page 41. Further work could be done to detect cases where code is duplicated, and extract that code into one function.

4.1.9 *Aliasing of registers*

To translate into SSA, we consider only flags and registers as variables as we do not have any information about aliasing of memory accesses. On x86, registers can be aliased, too, such as `AL`, `AH` and `AX` are part of `EAX`.

We implemented special rules to handle situations of register aliasing. Consider the case when `EAX` is written and later `AH` is read. The write

to `EAX` kills the registers `AL`, `AH`, `AX` and defines `EAX`. When reading `AH` we see that it is not defined but `EAX` is. Because we know that `AH := ((EAX shr 8) and 0xff)`, we replace the access to `AH` with the expression `((EAX shr 8) and 0xff)`.

4.1.10 *SSA as intermediate representation*

As described by Mycroft [25] and Emmerik [18], SSA has several benefits in decompilation, such as simplified propagation or implicit use-def information. We see another important aspect of SSA. During compilation, many different variables are mapped to the same register. The variables are originally used because they express different things and moreover, they could have had different types. This information is lost as we only have a narrow set of registers without type information. If we transform the program into SSA, we decouple the various variables from the registers. Therefore, the only dependencies between variables are when they are assigned to each other, but not when they accidentally use the same register.

4.1.11 *Coalescing unrelated variables*

Coalescing has two effects. It reduces the number of used variables and the number of copy statements. The second is a consequence from the first as the merging of variables leads to assignments where the destination is the same variable as the source. So far it is the same as in a compiler. However, we have a different motivation for this phase.

For once, this optimization has non-functional reasons. Unlike in a compiler, we have arbitrary many variables we can use. We implemented it to make the code more readable.

Moreover, we do have additional constraints. Variables of different types are never allowed to be coalesced. In addition, we do not allow coalescing of unrelated variables as we consider it as bad coding style.

For coalescing, we use the simple graph coloring method. Disallowing that two variables u and v are coalesced, means adding an edge between vertex u and v. Since we only allow coalescing of variables with the same type, we add edges from every variable to those variables with different types.

We also want to prevent the coalescing of unrelated variables. To achieve this, we developed an algorithm to find additional constraints. The basic idea is, that we only allow coalescing of variables on the same data flow path. If they are on the same path, they are related.

Our algorithm to build the constraint works as follows. First we build the data flow graph f. Then we build the transitive closure t of graph f and convert it to an undirected graph u. Figure 4.2 on page 34 shows an example for f and u. Finally the complement c of u contains the constraints needed to disallow coalescing of unrelated variables. Therefore, we add the edges from c to the constraint graph used for graph coloring.

With this approach we get the desired results. Unrelated variables are not coalesced, even though it is allowed by the live ranges. Furthermore, it does not influence the number of copy statements. It is not possible as there is never an assignment of one of those variables to another one, since the variables are on separate data flow paths. Listing 4.1 on page 34 shows the results of our algorithm.

Nevertheless, there is still room for optimizations. Thus it can happen, that a variable containing the size of an array, is used to count an index down. If we are able to decide the kind of the variables as described in Section 5.2.3 on page 41, we can add more constraints.

4.1.12 *Semantic gap*

In contrast to Java, C and assembly code gives more control over the code to the programmer. Since this means that Java can not express everything we can find in a binary program, we need to address some issues.

We showed in our upcompiler how pointers can be handled by implementing an abstraction. However, pointers can be used in other ways we do not yet handle, i.e., cast parts of memory to different types. On the other hand, we think a direct translation of such cases does not result in good Java code.

Nevertheless, some specific constructs can be handled. Casting a floating point value to its bit representation can be done by the Java function `Float.floatToIntBits`. There are also functions for the other direction and for doubles. An array of bytes can be converted to and from an integer by basic mathematical operations. Java also provides methods to convert arrays of bytes into strings and back.

For memory blocks where we do not know anything about the content, we should still be able to handle all cases. The memory is represented by a memory stream class. This class has methods to access the data like the Java classes `DataInputStream` and `DataOutputStream`. In addition, seeking has to be possible. Accessing the stream class has to be abstracted by a pointer class, since different pointer variables have different positions in the memory. Unfortunately, this relies on the Unsafe API which is not standardized but widely supported.

Unsigned data types have to be converted to signed types since Java does not yet have unsigned types, they are announced for JDK 8. If the most significant bit is used too, the next bigger data type needs to be used. With the `BigInt` class in Java, there is always a bigger data type available. Additional code may be inserted to ensure correct behavior for operations like subtraction.

It is possible that the upcompiler is not able to translate some programs. We have some ideas to handle such cases. For example, parts or the whole program is executed in a virtual machine like JPC [6]. As it does not solve the upcompile problem, it opens the field for dynamic code analysis.

4.1.13 *Obfuscation and optimization*

Obfuscated binaries are actively changed in order to make it difficult to understand them. There exists tools and methods to obfuscate binaries, but we think the problem is more subtle. The compilation process itself destroys information contained in the source code. While compiling with no optimization and all debug symbols let you reconstruct most of the code, additional comments and other information is lost. Compilation with full optimization and no debug symbols makes the generated binary already hard to understand.

When a binary is actively obfuscated, it is hard to understand or decompile it. Such techniques are used to protect the software against inspection. The reasons vary from enforcing of the copyright, protecting trade secrets, protecting against malware, protection against virus scanner and more.

Nevertheless, there is research going on. Kruegel et al. [21] and Vigna [31] describe how obfuscated binaries can be read and analyzed. But it is not done with the implementation of those methods in the upcompiler. The obfuscation will also affect other phases of the upcompiler.

Upcomiling legacy applications may be less challenging than new ones. Old compiler did less optimization or the optimization methods are now better understood. On the other hand, software written decades ago may use techniques or patterns which are no longer seen as good practice. Such techniques include spaghetti code or extensive inline assembly.

Furthermore, obsolete processor architectures may led to different optimizations than modern ones. As a optimization is often bound to a specific hardware, understanding the hardware may be needed to understand the optimization. Reverting an optimization is more easy, if not crucial, when the optimization is well understood.

4.1.14 *Functional programming languages*

We assume that an upcompiler should be able to handle programs written in a functional programming language. Based on the Church-Turing thesis, for every algorithm written in a functional programing language an imperative version exists. If an upcompiler is complete, i.e., can handle the whole instruction set, it can upcompile every program.

Practically, we were not able to upcompile a Haskell program. A major problem is the huge size of the runtime system included in the compiled program. A "Hello World" program compiles into a 1.1 MiB binary, while the user code compiles into an 2.3 KiB object file. With different tools we got a 199 line assembly file or 1702 line C code. The runtime system consists of around 50,000 lines of highly optimized C code. The tasks includes memory management, a parallel garbage collector, thread management and scheduler, primitive operations for GHC and a bytecode interpreter [23].

Nevertheless, we think it is possible to adjust an upcompiler to functional programs. First, the user application has to be separated from the runtime system. For the runtime system, equivalent functionality has to be provided as Java code. We expect that rewriting is much simpler than upcompiling. For Haskell, this is true as the original code is available. The upcompiler only needs to handle the user code. It is possible that some parts of the user code is in bytecode. In this case, the upcompiler needs a front end for the bytecode or a bytecode interpreter is used during run time.

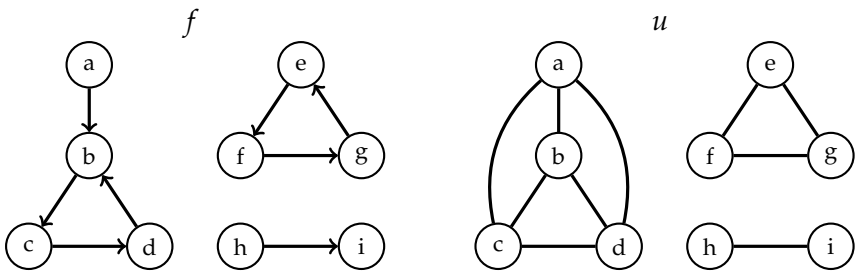


Figure 4.2: Data flow graph f of the program from Listing 4.1 on the current page, and the undirected version of the transitive closure u of it. The components are not always complete. If two variables (vertices) are connected, coalescing those variables is allowed, hence the complement of u is added to the constraint graph.

```
int main(){                          a b c d e f g h i
  a = readInt();                     ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
  if(a > 0){                         ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    b = a;                           ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    do{                              ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
      c = b;                         ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
      d = c - 1;                     ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
      func0(d);                      ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
      b = d;                         ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    }while(b > 0);                   ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    func0(a);                        ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    e = 0;                           ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    do{                              ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
      f = e;                         ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
      func0(f);                      ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
      g = f + 1;                     ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
      e = g;                         ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    }while(e < a);                   ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    h = 0;                           ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
  }else{                             ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
    h = -1;                          ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
  }                                  ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
  i = h;                             ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
  return i;                          ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮ ▮
}                                    a b c d e f g h i
```

```
int main(){                          a b e h
  a = readInt();                     ▮ ▮ ▮ ▮
  if(a > 0){                         ▮ ▮ ▮ ▮
    b = a;                           ▮ ▮ ▮ ▮
    do{                              ▮ ▮ ▮ ▮
      b = b - 1;                     ▮ ▮ ▮ ▮
      func0(b);                      ▮ ▮ ▮ ▮
    }while(b > 0);                   ▮ ▮ ▮ ▮
    func0(a);                        ▮ ▮ ▮ ▮
    e = 0;                           ▮ ▮ ▮ ▮
    do{                              ▮ ▮ ▮ ▮
      func0(e);                      ▮ ▮ ▮ ▮
      e = e + 1;                     ▮ ▮ ▮ ▮
    }while(e < a);                   ▮ ▮ ▮ ▮
    h = 0;                           ▮ ▮ ▮ ▮
  }else{                             ▮ ▮ ▮ ▮
    h = -1;                          ▮ ▮ ▮ ▮
  }                                  ▮ ▮ ▮ ▮
  return h;                          ▮ ▮ ▮ ▮
}                                    a b e h
```

Listing 4.1: Result of the constrained coalescing. Left is the program before, right the same program after coalescing. The lines show the live ranges of the variables. Variable `b`, `e` and `h` have no overlapping live ranges. Due to our constraint for unrelated variables, they are not coalesced.

We measured the performance of the upcompiler as also upcompiled programs. All measurements were done on a computer with the following specifications:

Processor	Intel Core 2 Duo CPU L9600 at 2.13GHz
Memory	1.7 GiB
OS	Ubuntu 12.04 with Linux 3.2, 64-bit
LLVM	2.9
Java	1.6.0_24 Sun Microsystems Inc., 64-Bit Server VM

4.2.1 *Upcompiler*

To measure the time of the phases, we instrumented the upcompiler. The numbers are the medians over 16 measurements on 7 different programs.

The most interesting numbers are the time spent in the different phases and the influence of the optimization level. Figure 4.3 on the following page shows both of them.

We see that the upcompiler performs best on unoptimized programs. Most of the differences in the time compared to the optimized programs are in variable-handling related phases. Inspections of the unoptimized binaries have shown that the variables are mostly written back to memory. With this, the variables are preserved and the upcompiler hardly inserts phi nodes. In consequence, the upcompiler does also not produce so many variables during SSA destruction which makes coalescing fast.

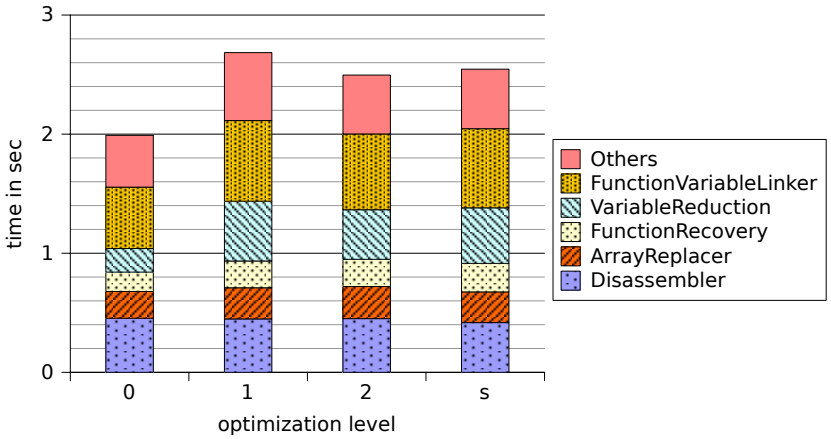


Figure 4.3: Time spent in the different phases of the upcompiler.

4.2.2 *Programs*

To determine the differences between compiled and upcompiled code, we measured the execution time of 4 programs. The first program, `fibrec`, calculates a user specified Fibonacci number in a very inefficient, recursive way. The second program, `fibloop`, has the same functionality but uses a loop to do it. Finally, we have a `quicksort` and a `heapsort` program.

For every program we compare 3 versions. We have a binary version compiled with optimization `-O2`, the upcompiled Java program, and a hand crafted Java version based on the C source. The changes for the last one are in order to make it Java compatible.

Figure 4.4 on the next page shows the median and standard deviation over 10 measurements with same input for every program run. The time is the user-time measured with the `time` utility. We try to explain some striking results.

The C version of the test program `fibrec` is more than 2 times slower than the Java version. This huge difference is due to the fact that the recursive function is public and hence the compiler can optimize it only very conservatively. We declared the function private to test the hypothesis and measured an execution time of 5.5 seconds.

Since the upcompiled code of both Fibonacci programs is nearly the same as the C code, the similar times of the two versions are not surprising.

For both sorting algorithms, the upcompiled version is much slower than the Java version. In both cases the structuring algorithm failed for the compute-intense kernel function. The fallback mode is used there. We assume, this prevents Java from optimizing the code.

The reasons for the difference between the C and Java version is not entirely known. We assume that the range checks on array accesses have a significant impact. Furthermore, the Java version spends a considerable time on tasks not related to sorting. We measured the sorting time of the hand written `quicksort` to be 3.0 seconds. This does not include the start of the JVM, allocating memory, reading input or writing output.

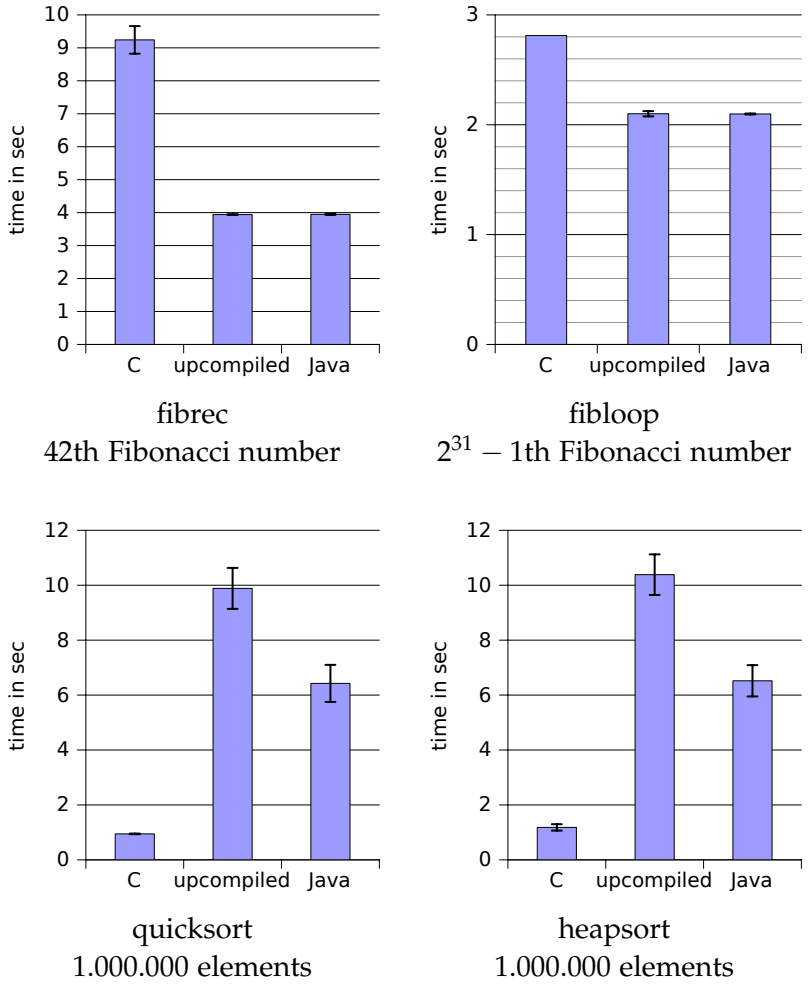


Figure 4.4: Performance comparison of upcompiled programs. We compare an optimized C program, an upcompiled Java program and a hand written Java program based on the C code.

5

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

We have demonstrated that upcompilation is possible by implementing an upcompiler. While it supports only a narrow set of input programs, we are able to fully and automatically translate those into Java source code.

Our upcompiler handles optimized 32-bit x86 binary programs. The binaries doe not need to have debug information or symbols. We detect calls to dynamically linked library functions and convert some of them to Java system calls. For the functions in the binary, we recover the local variables, arguments and return values. Memory accesses are converted to variables and arrays whenever possible.

We convert the control flow graph to an abstract syntax tree, consisting of high-level language constructs. We type the variables to integer and pointer to integer. The upcompiler optimizes the code for readability. This includes expression reduction and code cleanup. Furthermore, we developed a custom coalescing algorithm in order to reduce the number of variables without merging unrelated variables.

Since we implemented all phases, we touched all major areas in upcompilation. Within this thesis we describe the phases of our upcompiler and describe the algorithms. The implementation can be found in the source code. We discuss problems we faced during implementation. For some of them we present solutions, e.g., coalescing in the context of readability, detection of return values or the transformation of stack memory to local variables. For the unsolved challenges, we present ideas or refer to available research literature.

We are aware that our upcompiler is not yet ready for real world applications. Since it is free software, we hope it is useful for others to study and/or extend it.

A closer look at our upcompiler reveals that the binary is translated into a representation which is close to C and then wrapped into Java. It would be interesting to find a way to go directly to Java. However, since C is close to assembly, it may be difficult to do this.

We only analyze the code statically. If the upcompiler uses dynamic feedback, it can improve the code. Some conservative decisions may lead to slow execution or complicated source code. If the upcompiler has information from program runs, it is able to optimize the normal case. Such a dynamic method can be used on the original binary as well as in the produced Java source code.

The following sections describe problems we consider as most important for improving the overall result of the upcompiler.

5.2.1 *Type recovery*

The upcompiler only supports 32 bit signed integers and pointers to them. For real world problems, this is clearly not good enough. Adding support for other primitive data types should not be a difficult task. Already Cifuentes [11] describes how integer types with more bits and unsigned integers can be detected.

Reconstruction of composite data-types like arrays, records or even objects is the the next step. We refer to other literature like Mycroft [25] and Dolgova and Chernov [17]. They describe the reconstruction of composite data types and give further references. Beside the static analysis, Slowinska et al. [29] describes how they use dynamic analysis.

5.2.2 *Generality*

To widen the scope of input programs, the upcompiler needs several improvements.

We focused on binaries produced by LLVM, binaries from different compilers have to be tested. Support for more library functions is necessary and better support for `printf` and `scanf`, too. A broader range of input programs will also use more assembly instructions

than we support at the moment. Unusual composition, the use of inline assembly or code obfuscation tools also needs consideration.

Furthermore, function pointers are not yet supported. They need special handling during upcompilation. The mapping to Java has to be implemented, too.

Improvements of the structuring phase together with SSA destruction could lead to better output. A switch-case reconstruction from jump tables is needed for more readable code. Better structuring leads to less cases where the fallback mode is used. A more sophisticated SSA destruction method may render coalescing obsolete or produces less copy statements in the final code.

Finally, support for more machines will improve the overall robustness of the methods. This is necessary because the goal of the upcomiler is to handle binaries of obsolete machines. Adding support for obsolete machines will be a recurring task.

5.2.3 *Optimization and code metrics*

The upcompiler has optimization phases like compilers have. As the optimizations of compilers can be measured in hard units, like execution time or size of the produced code, there is no obvious metric for source code. Our coalescing phase has shown that reducing the number of variables improves the code quality, but there are more factors which have to be considered.

Obviously, it is difficult to write an optimizer without having a cost function. Literature about code metrics exists, we recommend McConnell [24] as a starting point. In the remaining paragraphs, we will discuss some optimizations.

The expression reduction we implemented is rudimentary, advanced code metrics are not considered. Our only consideration is, that we do not increase the number of expressions by copying them. This may improve the readability in a specific situation. But reducing all expressions as far as possible can reduce the readability if the expression size becomes to big.

Loop recovery also improves the code readability. Transforming a do-while loop surrounded by an if statement into an while loop should reduce the complexity. Replacing while loops with for loops may also increases the readability.

Variable naming can directed by the kind of the variable. Iteration variables can be named as `i`, `j` and `k`, variables containing a size would be named `n`, `m`, `size` or `count`. Furthermore, name inference, e.g., using the names from library function arguments and propagate them, can significantly improve the naming quality.

For numeric values in the code, we propose a constant reconstruction. This means the replacement of numeric values by constants. If the same value is used multiple times, the same constant can be used if it has the same meaning. This can be decided with the same method as we use for coalescing. Naming the constants can be done as variables are named.

Function extraction reduces function sizes. It reverts the inlining from the compiler. The compiler can inline a function at several locations. With code clone detection [27] techniques we should be able to find copied code. Nevertheless, we have to be able to find a good point where a function can be extracted. An extracted function should return at most one variable and the number of arguments should be small.

A

BIBLIOGRAPHY

[1] Boomerang. URL http://boomerang.sourceforge.net/. (Cited on page 5.)

[2] REC. URL http://www.backerstreet.com/rec/rec.htm. (Cited on page 5.)

[3] C2j. URL http://tech.novosoft-us.com/product_c2j.jsp. (Cited on page 5.)

[4] JGraphT. http://jgrapht.org/. (Cited on page 7.)

[5] JSR 335: Lambda Expressions for the Java Programming Language. URL http://jcp.org/en/jsr/detail?id=335. (Cited on page 26.)

[6] JPC - The Pure Java x86 PC Emulator. http://jpc.sourceforge.net, 2009. (Cited on page 32.)

[7] Brian Alliet and Adam Megacz. Complete Translation of Unsafe Native Code to Safe Bytecode. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '04, pages 32–41, New York, NY, USA, 2004. ACM. ISBN 1-58113-909-8. URL http://doi.acm.org/10.1145/1059579.1059589. (Cited on page 5.)

[8] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981. ISSN 0096-0551. URL http://www.sciencedirect.com/science/article/pii/0096055181900485. (Cited on page 21.)

[9] Gengbiao Chen, Zhuo Wang, Ruoyu Zhang, Kan Zhou, Shiqiu Huang, Kangqi Ni, Zhengwei Qi, Kai Chen, and Haibing Guan. A Refined Decompiler to Generate C Code with High Readability. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 150 –154, oct. 2010. (Cited on page 28.)

[10] Vitaly Chipounov and George Candea. Enabling Sophisticated Analysis of x86 Binaries with RevGen. In *Proceedings of the 7th Workshop on Hot Topics in System Dependability (HotDep)*, 2011. (Cited on page 5.)

[11] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994. (Cited on pages 5, 28, 29, and 40.)

[12] Cristina Cifuentes and Mike Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Proceedings of the 1999 International Conference on Program Comprehension*, pages 192–199, 1999. (Cited on page 27.)

[13] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Elsevier Science, 2011. ISBN 978-0-12-088478-0. (Cited on page 9.)

[14] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A Simple, Fast Dominance Algorithm. *Software Practice and Experience*, (4): 1–28, 2001. (Cited on page 9.)

[15] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4): 451–490, October 1991. ISSN 0164-0925. (Cited on page 5.)

[16] Gil Dabah. diStorm, 2011. URL `http://www.ragestorm.net/distorm/`. (Cited on page 7.)

[17] E. Dolgova and A. Chernov. Automatic Reconstruction of Data Types in the Decompilation Problem. *Programming and Computer Software*, 35:105–119, 2009. ISSN 0361-7688. URL `http://dx.doi.org/10.1134/S0361768809020066`. (Cited on page 40.)

[18] Michael James Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, The University of Queensland, 2007. (Cited on pages 2, 5, 26, and 30.)

[19] Maurice H. Halstead. Machine-independence and third-generation computers. In *Proceedings SJCC (Sprint Joint Computer Conference)*, page 587–592, 1967. (Cited on page 5.)

[20] Karl Trygve Kalleberg, Ole André Vadla Ravnås, Johann Prieur, and Haakon Sporsheim. Frida IRE, 2010. URL `http://code.google.com/p/frida-ire/`. (Cited on page 7.)

[21] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium*, SEC'04. USENIX Association, 2004. URL `http://static.usenix.org/event/sec04/tech/full_papers/kruegel/kruegel_html/disassemble.html`. (Cited on page 32.)

[22] Ulrike Lichtblau. Decompilation of Control Structures by Means of Graph Transformations. In *TAPSOFT, Vol.1*, Lecture Notes

in Computer Science, pages 284–297, 1985. ISBN 3-540-15198-2. (Cited on page 28.)

[23] Simon Marlow and Simon Peyton-Jones. *The Architecture of Open Source Applications*, volume 2. URL `http://www.aosabook.org/en/ghc.html`. (Cited on page 33.)

[24] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2nd edition, 2004. ISBN 0735619670. (Cited on page 41.)

[25] Alan Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 208–223. Springer, 1999. ISBN 3-540-65699-5. (Cited on pages 5, 28, 30, and 40.)

[26] David A Roberts. LLJVM. URL `http://da.vidr.cc/projects/lljvm/`. (Cited on page 5.)

[27] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7): 470–495, 2009. (Cited on page 42.)

[28] D. Simon. Structuring Assembly Programs. Honours thesis, The University of Queensland, Department of Computer Science and Electrical Engineering, 1997. (Cited on page 28.)

[29] Asia Slowinska, Traian Stancescu, and Herbert Bos. DDE: dynamic data structure excavation. In *ApSys*, pages 13–18, 2010. (Cited on page 40.)

[30] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating Out of Static Single Assignment Form. In *Static Analysis Symposium/Workshop on Static Analysis*, pages 194–210, 1999. (Cited on page 16.)

[31] Giovanni Vigna. Static Disassembly and Code Analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 19–41. Springer US, 2007. ISBN 978-0-387-44599-1. URL `http://dx.doi.org/10.1007/978-0-387-44599-1_2`. (Cited on page 32.)

B

LISTINGS

C

LIST OF FIGURES

D

APPENDIX

D.1 GLUE CODE

The following three classes are written by the upcompiler to support the generated code.

```
1  public class Main {
2    public static void main(String[] args) {
3      UserPrg user = new UserPrg();
4      System.exit(user.main());
5    }
6  }
```

Listing D.1: The Main class. Used to instantiate the user program and return the exit value.

```
1  import java.util.Scanner;
2
3  public class IoSupport {
4    private static Scanner s = new Scanner(System.in);
5
6    public static int readInt() {
7      return s.nextInt();
8    }
9
10   public static void writeStr(String string) {
11     System.out.print(string);
12   }
13
14   public static void writeNl() {
15     System.out.println();
16   }
17
18   public static void writeInt(int value) {
19     System.out.print(value);
20   }
21 }
```

Listing D.2: The IoSupport class. Provides the functionality of `readInt`, `writeInt`, `writeStr` and `writeNl`.

```
1  import java.util.ArrayList;
2
3  public class Pointer<T> {
4    private ArrayList<T> memory;
```

```
5   private int position = 0;
6
7   public Pointer(int n) {
8     memory = new ArrayList<T>(n);
9     for (int i = 0; i < n; i++) {
10      memory.add(null);
11    }
12  }
13
14  public Pointer(Pointer<T> old, int displacement) {
15    memory = old.memory;
16    position = old.position + displacement;
17  }
18
19  public void free() {
20    memory.clear();
21  }
22
23  public T getValue(int offset) {
24    return memory.get(position + offset);
25  }
26
27  public void setValue(int offset, T value) {
28    memory.set(position + offset, value);
29  }
30
31  public int hashCode() { ... }
32  public boolean equals(Object obj) { ... }
33 }
```

Listing D.3: The Pointer class. It is used to encapsulate pointer accesses.

D.2 TESTCASES

In here, we show the output of some test cases. The code and graphs are generated by the upcompiler without post-processing them. All C programs are compiled with LLVM and optimization level 2. The symbols of the binary are removed before upcompiling.

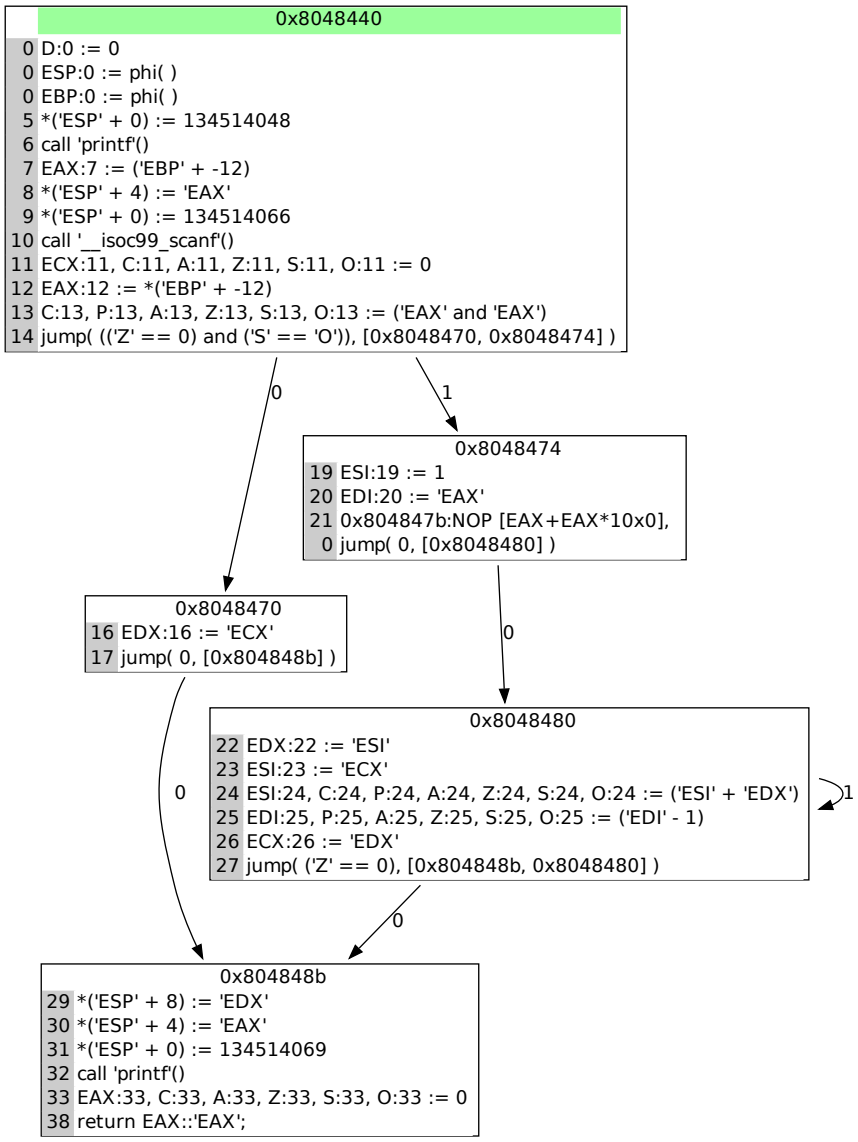
D.2.1 *fibloop*

This program asks the user for an number and calculates the corresponding Fibonacci number within a loop.

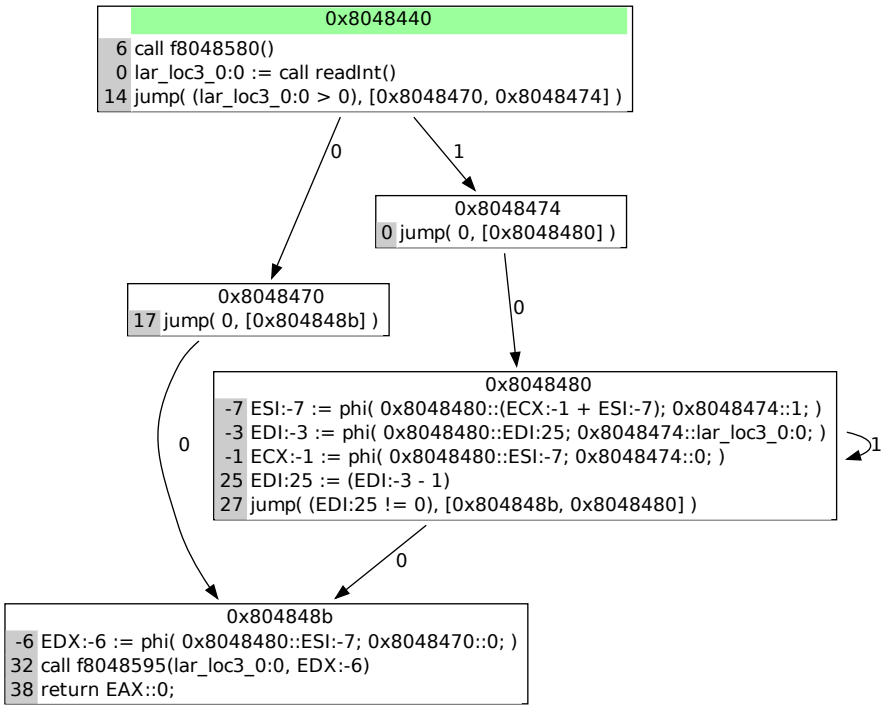
Source

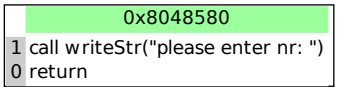
```c
#include <stdio.h>

static int fib( int n ){
  int an1 = 0;
  int an0 = 1;

  for( int i = 0; i < n; i++ ){
    int a = an0 + an1;
    an1 = an0;
    an0 = a;
  }
  return an1;
}

int main(){
  int n, f;
  printf( "please enter nr: " );
  scanf( "%i", &n );
  f = fib(n);
  printf( "the %ith fibonacci number is %i\n", n, f );
  return 0;
}
```

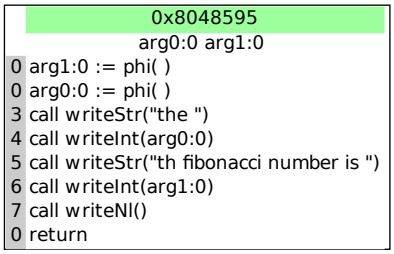
After disassembling



After all low level phases except SSA destruction







After structuring and normalization

```
1  public class UserPrg {
2    public Generic? main( ) {
3      Generic? tmp4 = 0;
4      Generic? tmp4_t = 0;
5      Generic? tmp5 = 0;
6      Generic? tmp5_t = 0;
7      Generic? tmp6 = 0;
8      Generic? tmp6_t = 0;
9      Generic? tmp7 = 0;
10     Generic? tmp8 = 0;
11     Generic? tmp9 = 0;
12     Generic? tmp9_t = 0;
13     func1();
14     tmp8 = IoSupport.readInt();
15     if( (tmp8 > 0)) {
16       tmp4_t = 1;
17       tmp5_t = tmp8;
18       tmp6_t = 0;
19       do {
20         tmp4 = tmp4_t;
21         tmp5 = tmp5_t;
22         tmp6 = tmp6_t;
23         tmp7 = (tmp5 - 1);
24         tmp4_t = (tmp6 + tmp4);
25         tmp5_t = tmp7;
26         tmp6_t = tmp4;
27         tmp9_t = tmp4;
28       }
29       while( (tmp7 != 0) );
30     }
31     else tmp9_t = 0;
32
33     tmp9 = tmp9_t;
34     func0(tmp8, tmp9);
35     return 0;
36   }
37
38   public Generic? func0( Generic? tmp1, Generic? tmp0) {
39     IoSupport.writeStr("the ");
40     IoSupport.writeInt(tmp1);
41     IoSupport.writeStr("th fibonacci number is ");
42     IoSupport.writeInt(tmp0);
43     IoSupport.writeNl();
44     return ;
45   }
46
47   public Generic? func1( ) {
48     IoSupport.writeStr("please enter nr: ");
49     return ;
50   }
51
52 }
```

The final code

```
1  public class UserPrg {
2    public int main( ) {
3      int var0 = 0;
4      int var1 = 0;
5      int var2 = 0;
6      int var3 = 0;
7      int var4 = 0;
8      func1();
9      var4 = IoSupport.readInt();
10     if( (var4 > 0)) {
11       var1 = 1;
12       var2 = var4;
13       var3 = 0;
14       do {
15         var0 = var1;
16         var2 = (var2 - 1);
17         var1 = (var3 + var1);
18         var3 = var0;
19       }
20       while( (var2 != 0) );
21     }
22     else var0 = 0;
23
24     var3 = var0;
25     func0(var4, var0);
26     return 0;
27   }
28
29   public void func0( int var0, int var1) {
30     IoSupport.writeStr("the ");
31     IoSupport.writeInt(var0);
32     IoSupport.writeStr("th fibonacci number is ");
33     IoSupport.writeInt(var1);
34     IoSupport.writeNl();
35     return ;
36   }
37
38   public void func1( ) {
39     IoSupport.writeStr("please enter nr: ");
40     return ;
41   }
42
43 }
```

D.2.2 *fibrec*

This program asks the user for a number and calculates the corresponding Fibonacci number recursively.

Source

```
1  #include <stdio.h>
2
3  int fib( int n ){
4    if( n < 2 ){
5      return n;
6    } else {
7      return fib(n-1) + fib(n-2);
8    }
9  }
10
11 int main(){
12   int n, f;
13   printf( "please enter nr: " );
14   scanf( "%i", &n );
15   f = fib(n);
16   printf( "the %ith fibonacci number is %i\n", n, f );
17   return 0;
18 }
```

Output of the upcompiler

```
1  public class UserPrg {
2    public int main( ) {
3      int var0 = 0;
4      int var1 = 0;
5      func2();
6      var0 = IoSupport.readInt();
7      var1 = func0(var0);
8      func1(var0, var1);
9      return 0;
10   }
11
12   public int func0( int var0) {
13     int var1 = 0;
14     if( (var0 < 2)) ;
15     else {
16       var1 = func0((var0 + -1));
17       var0 = func0((var0 + -2));
18       var0 = (var0 + var1);
19     }
20
21     return var0;
22   }
23
24   public void func1( int var0, int var1) {
25     IoSupport.writeStr("the ");
26     IoSupport.writeInt(var0);
27     IoSupport.writeStr("th fibonacci number is ");
28     IoSupport.writeInt(var1);
29     IoSupport.writeNl();
30     return ;
31   }
32
33   public void func2( ) {
34     IoSupport.writeStr("please enter nr: ");
35     return ;
36   }
37
38 }
```

D.2.3 *fibdyn*

This program asks the user for a number and calculates the corresponding Fibonacci number with a dynamic allocated array.

Source

```
#include <stdio.h>
#include <stdlib.h>

__attribute__ ((noinline))
void fib( int *data, int n ){
  data[0] = 0;
  data[1] = 1;
  for( int i = 2; i <= n; i++ ){
    data[i] = data[i-1] + data[i-2];
  }
}

int main(){
  int n;
  int *data;
  printf( "please enter nr: " );
  scanf( "%i", &n );
  data = (int*)malloc( ((n<2?2:n)+1) * sizeof(*data) );
  if( data == NULL ){
    printf( "error by memory allocation\n" );
    return -1;
  }
  fib(data,n);
  printf( "the %ith fibonacci number is %i\n", n, data[n] );
  free( data );
  return 0;
}
```

Output of the upcompiler

```
1  public class UserPrg {
2    public int main( ) {
3      int var0 = 0;
4      int var1 = 0;
5      Pointer<Integer> var2 = null;
6      func2();
7      var1 = IoSupport.readInt();
8      var2 = new Pointer<Integer>((( (var1 >= 2) ? ((var1 * 4) + 4) : 12
           ) / 4));
9      if( (var2 != null)) {
10       func0(var2, var1);
11       func3(var1, var2.getValue( var1 ));
12       var2.free();
13       var0 = 0;
14     }
15     else {
16       func1();
17       var0 = -1;
18     }
19
20     return var0;
21   }
22
23   public void func0( Pointer<Integer> var0, int var1) {
24     int var2 = 0;
25     int var3 = 0;
26     Pointer<Integer> var4 = null;
27     var0.setValue( 0, 0 );
28     var0.setValue( 1, 1 );
29     if( (var1 >= 2)) {
30       var2 = 0;
31       var3 = 1;
32       var1 = (var1 + -2);
33       var4 = new Pointer<Integer>(var0, 2);
34       var3 = (var3 + var2);
35       var4.setValue( 0, var3 );
36       while( (var1 != 0) ){
37         var2 = var4.getValue( -1 );
38         var1 = (var1 - 1);
39         var4 = new Pointer<Integer>(var4, 1);
40         var3 = (var3 + var2);
41         var4.setValue( 0, var3 );
42       }
43     }
44
45     return ;
46   }
47
48   public void func1( ) {
49     IoSupport.writeStr("error by memory allocation");
50     IoSupport.writeNl();
51     return ;
52   }
53
54   public void func2( ) {
```

```
55      IoSupport.writeStr("please enter nr: ");
56      return ;
57    }
58
59    public void func3( int var0, int var1) {
60      IoSupport.writeStr("the ");
61      IoSupport.writeInt(var0);
62      IoSupport.writeStr("th fibonacci number is ");
63      IoSupport.writeInt(var1);
64      IoSupport.writeNl();
65      return ;
66    }
67
68  }
```

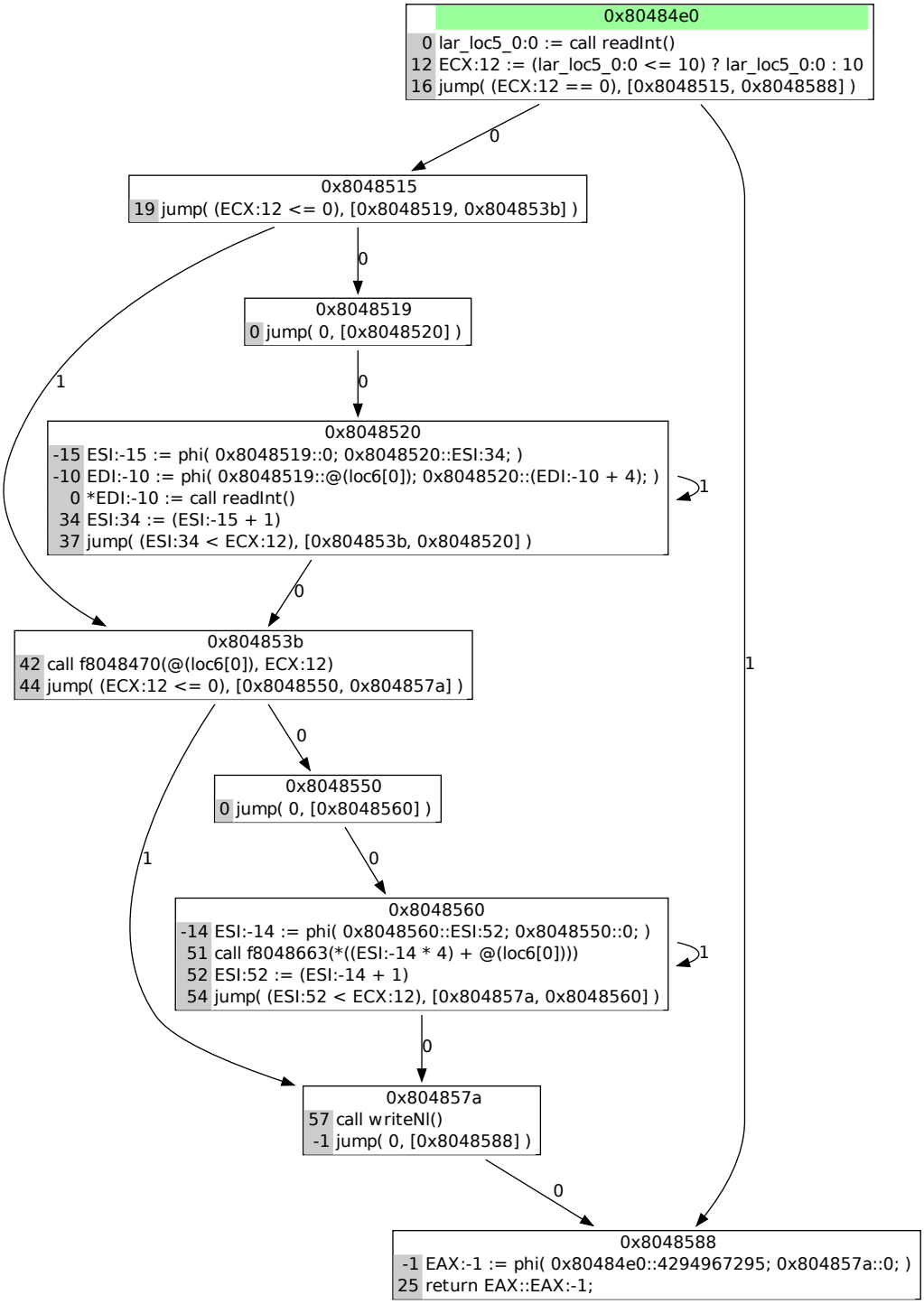
D.2.4 *bubblesort*

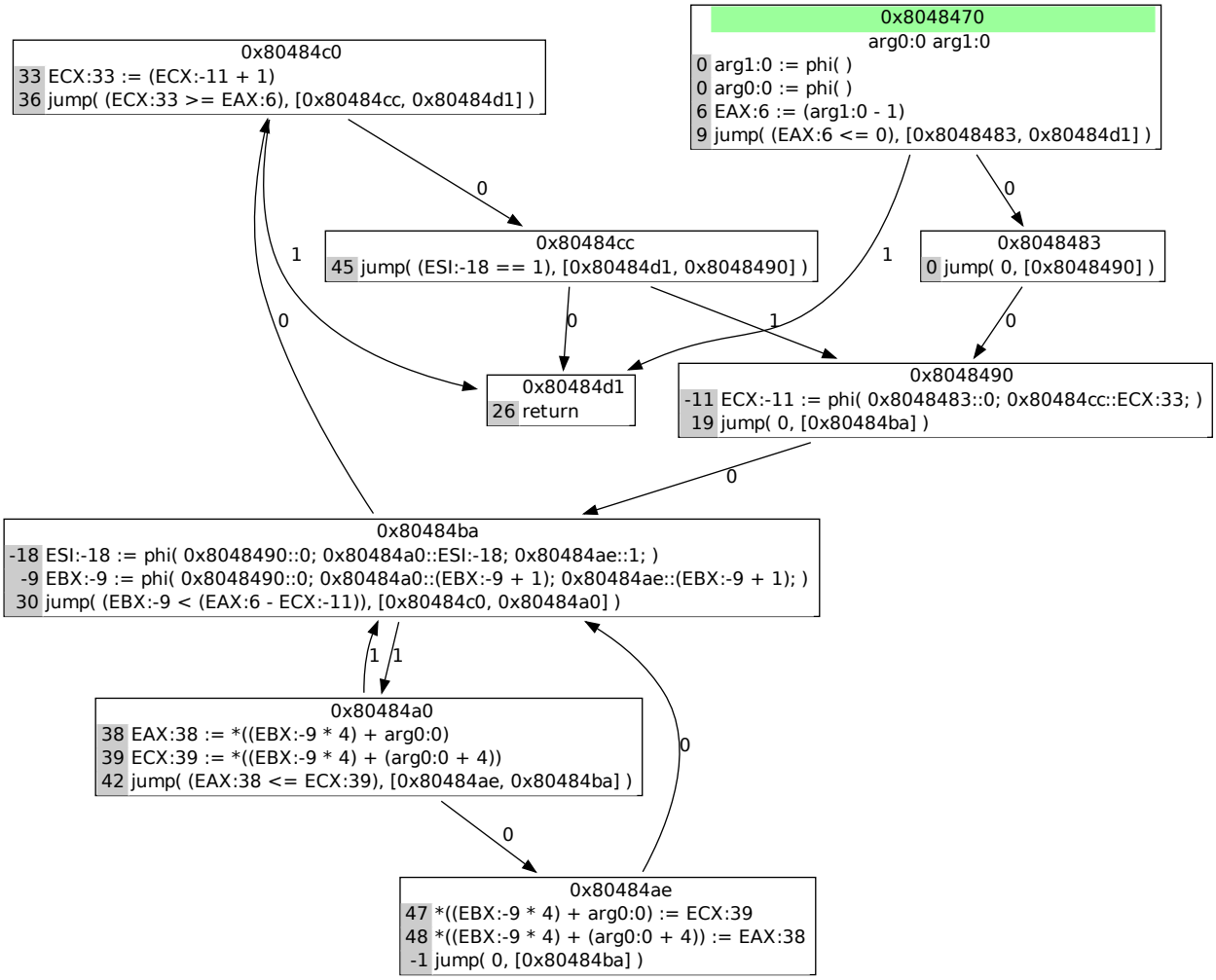
This programs reads values, sorts them and writes them to the output. The examples shows the fallback mode of structuring and an array on the stack.

Source

```
#include <stdio.h>
#define MAX_N 10

__attribute__ ((noinline))
void sort(int x[],int n) {
  int switched = 1;

  for( int i = 0; (i < n-1) && (switched == 1); i++ ) {
    switched = 0;

    for( int j = 0; j < (n-i-1); j++ ){
      if( x[j] > x[j+1] ) {
        int tmp;
        switched = 1;
        tmp      = x[j];
        x[j]     = x[j+1];
        x[j+1]   = tmp;
      }
    }
  }
}

int main() {
  int marks[MAX_N];
  int i, n;

  scanf( "%i", &n );
  n = n>MAX_N?MAX_N:n;
  if( n == 0 ){
    return -1;
  }

  for( i = 0; i < n; i++ ){
    scanf( "%i", &marks[i] );
  }

  sort(marks, n);

  for( i = 0; i < n; i++ ){
    printf( "%i ", marks[i] );
  }
  printf( "\n" );

  return 0;
}
```

After all low level phases except SSA destruction




0x8048470
arg0:0 arg1:0
0 arg1:0 := phi( )
0 arg0:0 := phi( )
6 EAX:6 := (arg1:0 - 1)
9 jump((EAX:6 <= 0), [0x8048483, 0x80484d1])
0x80484c0
33 ECX:33 := (ECX:-11 + 1)
36 jump((ECX:33 >= EAX:6), [0x80484cc, 0x80484d1])
0x80484cc
45 jump((ESI:-18 == 1), [0x80484d1, 0x8048490])
0x8048483
0 jump(0, [0x8048490])
0x80484d1
26 return
0x8048490
-11 ECX:-11 := phi(0x8048483::0; 0x80484cc::ECX:33;)
19 jump(0, [0x80484ba])
0x80484ba
-18 ESI:-18 := phi(0x8048490::0; 0x80484a0::ESI:-18; 0x80484ae::1;)
-9 EBX:-9 := phi(0x8048490::0; 0x80484a0::(EBX:-9 + 1); 0x80484ae::(EBX:-9 + 1);)
30 jump((EBX:-9 < (EAX:6 - ECX:-11)), [0x80484c0, 0x80484a0])
0x80484a0
38 EAX:38 := *((EBX:-9 * 4) + arg0:0)
39 ECX:39 := *((EBX:-9 * 4) + (arg0:0 + 4))
42 jump((EAX:38 <= ECX:39), [0x80484ae, 0x80484ba])
0x80484ae
47 *((EBX:-9 * 4) + arg0:0) := ECX:39
48 *((EBX:-9 * 4) + (arg0:0 + 4)) := EAX:38
-1 jump(0, [0x80484ba])
0
1
0
1
0
1
0
1
0
1
0
1
0


0x80486b3
arg0:0
0 arg0:0 := phi( )
2 call writeInt(arg0:0)
3 call writeStr(" ")
0 return
0x8048663
arg0:0
0 arg0:0 := phi( )
2 call writeInt(arg0:0)
3 call writeStr(" ")
0 return

Output of the upcompiler

```
1  public class UserPrg {
2    public int main( ) {
3      int var0 = 0;
4      int var1 = 0;
5      int var2 = 0;
6      int var3 = 0;
7      Pointer<Integer> var4 = null;
8      Pointer<Integer> var5 = new Pointer<Integer>( 10 );
9      var2 = IoSupport.readInt();
10     var2 = ( (var2 <= 10) ? var2 : 10 );
11     var1 = -1;
12     if( (var2 != 0)) {
13       if( (var2 > 0)) {
14         var0 = 0;
15         var4 = new Pointer<Integer>(var5, 0);
16         do {
17           var4.setValue( 0, IoSupport.readInt() );
18           var0 = (var0 + 1);
19           var4 = new Pointer<Integer>(var4, 1);
20         }
21         while( (var0 < var2) );
22       }
23
24       func0(new Pointer<Integer>(var5, 0), var2);
25       if( (var2 > 0)) {
26         var3 = 0;
27         do {
28           func1(var5.getValue( var3 ));
29           var3 = (var3 + 1);
30         }
31         while( (var3 < var2) );
32       }
33
34       IoSupport.writeNl();
35       var1 = 0;
36     }
37
38     return var1;
39   }
40
41   public void func0( Pointer<Integer> var0, int var1) {
42     int var2 = 0;
43     int var3 = 0;
44     int var4 = 0;
45     int var5 = 0;
46     int var6 = 0;
47     int var7 = 0;
48     int var8 = 0;
49     int var9 = 0;
50     int var10 = 0;
51     var2 = 0;
52     while( true )switch( var2 ){
53       case 0x0: {
54         {
55           var9 = (var1 - 1);
```

```
56          var2 = ( (var9 <= 0) ? 4 : 1 );
57        }
58        break;
59      }
60      case 0x1: {
61        {
62          var6 = 0;
63          var2 = 2;
64        }
65        break;
66      }
67      case 0x2: {
68        {
69          var5 = var6;
70          var10 = 0;
71          var4 = 0;
72          var3 = var4;
73          while( (var4 < (var9 - var5)) ){
74            var7 = var0.getValue( var4 );
75            var8 = var0.getValue( (1 + var4) );
76            var4 = (var4 + 1);
77            if( (var7 > var8)) {
78              var0.setValue( var3, var8 );
79              var0.setValue( (1 + var3), var7 );
80              var10 = 1;
81              var4 = (var3 + 1);
82            }
83
84            var3 = var4;
85          }
86          var5 = (var5 + 1);
87          var2 = ( (var5 >= var9) ? 4 : 3 );
88        }
89        break;
90      }
91      case 0x3: {
92        {
93          var6 = var5;
94          var2 = ( (var10 == 1) ? 2 : 4 );
95        }
96        break;
97      }
98      case 0x4: {
99        return ;
100     }
101   }
102 }
103
104 public void func1( int var0) {
105   IoSupport.writeInt(var0);
106   IoSupport.writeStr(" ");
107   return ;
108 }
109
110 }
```