C Decompilation: Is It Possible?

Katerina Troshina[1], Alexander Chernov[1] and Yegor Derevenets[1]

Institute for System Programming, Russian Academy of Sciences,
25, A. Solzhenitsyn st, 109004, Moscow, Russia,
katerina@ispras.ru, cher@ispras.ru, yegord@ispras.ru

Abstract. Decompilation is reconstruction of a program in a high-level language from a program in a low-level language. Possibility and feasibility of decompilation is a subject of controversy over last years. We present several arguments supporting the idea that in spite of impossibility of full automatic decompilation there exist methods and techniques that cover most of decompilation process for wide class of programs. The proposed methods and techniques are implemented in the *TyDec* decompiler being developed by the authors.

1 Introduction

A typical software development uses many third-party software components, which are often provided without source code. On the other side, it is a common situation when some legacy application runs for years and no source code is available and there are no means of obtaining it. In both cases it is sometimes necessary to fix bugs or adapt the components without source code to changing requirements.

Such problems are addressed by reverse engineering. Reverse engineering is typically an effort on leveraging level of abstraction of an existing software system. A typical reverse engineering track starts from executable files and goes through an assembly program, high-level program (for example, in C programming language) and ends on specification level. The task of obtaining an assembly program from an executable is solved by disassembling, then a decompiler reconstructs a high-level program from an assembly program.

There exist many automated source code analysis tools for high-level languages like C or C++ for bug-fighting [1] and program understanding [2] automating some tasks and making manual review of code easier and faster. There also exist similar tools for low-level programs, however the toolkit arsenal is poorer.

Possibility and feasibility of decompilation is a subject of controversy over last years. Some argue that compilation is like mincing a pig for sausages, so decompilation is hardly possible as we can never recollect a pig from a pack of sausages [3]. We should take into account, however, that compilation preserves program behavior, which would mean that a sausage could grunt and run. From this point of view possibility of decompilation remains open.

Certainly, full automatic decompilation of arbitrary machine-code programs is not possible, it is an undecidable problem like most problems of program transformation. So, from the point of view of theory no algorithm exists for decompilation of all possible programs. However, this (obvious) answer has no practical importance — many problems are undecidable or NP-complete yet there exist algorithms giving precise or approximate solutions for practical important cases. Thus the challenges of decompilation are in identification of subclasses of low-level programs and developing methods and algorithms targeted for these particular classes. The methods may even ask a human expert for help at key points.

Another challenge of decompilation is in quality of decompiler output. If a decompiler outputs a program containing goto's instead if's or while's or type casts in expressions rather than correct types at variable definition sites — such a decompiler is of no practical value. The assembly listing might be much easier to read as it is not cluttered with garbage. An ideal decompiler should output a C program close to what could be written manually: with all structured constructions in place and variables properly typed. This is a crucial requirement, although it cannot be formalized.

Often it is not possible to reconstruct all data types or all control structures precisely using only static decompilation (i.e. by analyzing the low-level program code alone). For example, it is often impossible to choose between an integer and a pointer type for a 32-bit variable (on a 32-bit platform, such as ia32). Fig. 1 illustrates such an example. The type of the `s` variable cannot be unambiguously reconstructed. At least `int *` and `int` are acceptable and produce identical assembly code after compilation. One possible approach in such situations is to ask a user, and another approach is to use information from run-time of the program to provide hints for the static analysis in cases of ambiguity.

```
int f(int *a, int n) {
 int s = 0;
 int *p = a;
 for (; p < a + n; ++p) {
   s += *p;
 }
 return s;
}
```

Fig. 1. A C program with ambiguous type of `s`

The remainder of the paper is organized as follows. Section 2 discusses related work, class of C programs that could be automatically developed presented in section 3, section 4 is devoted to structural analysis, data type analysis presented in section 4 and section 5 presents decompiler TyDec. In conclusion the primary results of the paper are stated and the directions of future work are outlined.

2 Related work

Cifuentes' [4–6] were pioneering works in C decompilation. These works described methods of decompilation that she implemented in the DCC decompiler [7]. The primary topic of the works was structural analysis (i. e. recovery of high-level control structure of C functions). However, structural analysis for compilation is well covered in [8].

Mycroft [9] first gave an unification-based algorithm for recovery of types of variables during static program decompilation.

One of the closely related to our static type reconstruction algorithms is algorithm VSA presented in [10]. However, the goal of VSA is value propagation; certain attributes such as size and memory layout of structure types are recovered as a side effect.

Other from DCC there exist decompilers Boomerang [11], REC [12] and Hex-Rays plug-in [13] for interactive disassembler IDA [14]. However, all of these are not capable of reconstructing a strictly conforming program in many cases, even if the source program is a strictly conforming program. Non of them reconstruct the complete set of C control structures. For example, only Boomerang reconstructs `switch` operator, but misses `for` operator. Moreover, all decompilers often fail to reconstruct data types correctly and produce C programs with explicit type casts and unions, even if the source program does not have such.

3 Decompilation

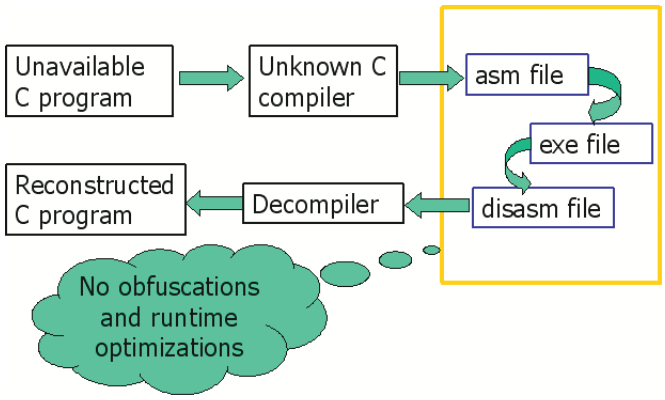

Fig. 2. Program transformation scheme

Generally, we can make no assumptions about nature of an original program, which executable (or other low-level form) is being analyzed. The original

program may be written in C, C++, Delphi, Fortran, or in a mixture of programming languages, or even in scripting languages with the corresponding virtual machine embedded into the executable. Given such a broad spectrum of source languages it is no hope of a decompiler giving good results in all cases.

We assume that no obfuscation and run-time code transformations are performed on the track from the source high-language program to an assembly program to an executable program and then to the corresponding disassembled program. Thus the assembly program resulting from compilation of the source program and the disassembled program are basically the same.

For a start we should limit the universe of the source languages and their combinations to something compact yet widely used and powerful. As such the C programming language is one of the best candidates. From now on we assume that the original program is written in C. This does not imply that the decompiler is unable to decompile non-C program, still any low-level program can be fed to the decompiler, but with uncertain results. Some non-C program may be decompiled into good C, but some may not. The overall operation scheme is shown in Fig. 2.

From the practical point of view, decompilation into C++ would be of greater practical importance. However, C++ decompilation throws down challenges such as recovery of class hierarchy, recovery of exception handling structure etc, recovery of template instantiations, etc. We consider C decompilation as base for passing on to C++ decompilation.

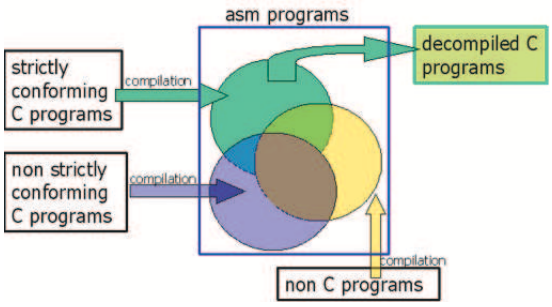

Fig. 3. Classes of original programs

The C language allows writing highly-structured and strictly-typed programs yet it allows writing unstructured and untyped programs as well, which hardly can be decompiled into anything reasonable. Fortunately, suitable language constraints are defined in the C standard itself. A *strictly conforming program* is a program, which operation does not depend on any unspecified, undefined, or implementation-defined behavior. For example, result of assignment of a pointer value to an integer variable is implementation-defined, as well as result of accessing a floating-point value as integer through a union. A strictly conforming

subset of the C language is a sufficiently strictly typed. Typical real-world C programs are almost strictly conforming with non-conforming parts isolated to OS-specific or processor-specific layers. However, the C standard allows bypassing strict typing in a number of ways.

- It is allowed to assign a pointer value of some type to a pointer variable of different type, and then assign it back to a pointer variable of the original type. For example, `void*` pointers are typically used for pointer value passing, but any other pointer type suits as well.
- It is allowed to cast any pointer value to a `char*` pointer type and use it for byte-by-byte access to the pointed value. For example, naive implementations of `memcpy` or `memcmp` standard functions might be implemented this way and remain strictly conforming.

These are handicaps for static analysis and static reconstruction of types for decompilation. However, recent research gives hope that they can be overcome by dynamic or profile-based analysis.

Again, this does not imply that the decompiler is unable to decompile non strictly conforming C programs. Some non-SC C programs may be decompiled into SC C programs, but some may not as illustrated in Fig. 3.

If the source program is a strictly conforming C program, a result of decompilation should also be a strictly conforming C program. If the source program is not a strictly conforming program or is not even written in C, it might accidentally get a strictly conforming counter-image that should be the result of decompilation.

3.1 Correct decompilation

An *original program* is a program in the C language, which assembly code is being reconstructed by the decompiler. Typically, the original program is not available, and the compiler, which was used to compile it, is not known. which source code is not available and which is being reconstructed by decompilation. It is compiled by some (unknown to us) C compiler.

An *input program* for the decompiler is an assembly program that may be obtained by disassembling an executable file. We assume that all program transformations starting from compilation of the original program and up to disassembling do not include obfuscation or dynamic code transformations. Thus the assembly program resulting from compilation of the original program and the disassembled program match each other.

Decompilation is called *correct*, if the following holds. Let us introduce a formal definition of *correct decompilation*. Let K be a compiler from a high-level language to an assembly language. Typically, the output of the compiler depends on compilation options, so let p be a set of compilation options. K_p is the compiler K with the set of options p. Let A be an original program. So $K_p(A) = C$, if C is the assembly program that is the result of compilation of the program A by the compiler K with the options p. Let DK be a decompiler.

Let program $A' = DK(C)$ be the result of decompilation. So let us say that decompilation is *correct* if there exist a compiler K' and an option set p', such as programs $C' = K'_{p'}(A')$ and C are equivalent.

Let C_1 and C_2 be some assembly programs. Let f be an *object renaming* transformation of assembly programs. $f(C_1) = C_2$ if there exists a renaming of named objects such as registers, labels, etc, preserving program semantics and making the programs $f(C_1)$ and C_2 syntactically indistinguishable. Existence of such an f transformation is a sufficient condition of program equivalence.

Consider control flow graphs (CFG) G_1 and G_2 of the programs C_1 and C_2. If G_1 and G_2 are isomorphic and the corresponding basic blocks are equivalent, the programs C_1 and C_2 are equivalent too. This is also a sufficient condition of program equivalence.

A basic block is a sequence of assembly instructions with one entry and one exit. Equivalence of basic blocks may be checked by reducing them to a RISC-like representation. Let h be a basic block reduction transformation. Let B_1 and B_2 be basic blocks, let $B'_1 = h(B_1)$ and $B'_2 = h(B_2)$ be the corresponding basic blocks in the RISC-like representation. Let g be an instruction reordering transformation preserving the program semantics. If for the basic blocks B_1 and B_2 there exist transformations f, g and h making $g(f(h(B_1))$ and $h(B_2)$ syntactically indistinguishable, basic blocks B_1 and B_2 are equivalent.

4 Structural analysis

Decompilation can be performed in the following steps:

- Function interface reconstruction.
- Structural analysis (reconstruction of high-level language structures such as conditions, loop, etc).
- Expression reconstruction.
- Data type analysis.

This section presents methods for structural analysis, other parts are discussed later.

Structural analysis is based on control flow graph (CFG) analysis. A dominator tree is built for the nodes of the CFG, then the edges of CFG are labeled as forward (directed from a dominating node to a dominated node), backward (directed from a dominated node to a dominating node) and cross (all other). Note, that the control-flow graph might not be completely recovered by static analysis if indirect jumps appear. Indirect jumps are typically generated for `switch` operators.

Reconstruction of high-level control structures is performed iteratively when the CFG is labeled. The CFG is transformed to a hierarchical graph of regions. On the first iteration a region is created for each basis block. On each iteration templates corresponding to structural constructs (`if`, `while`, etc) are tried on the current graph of regions. If match is found, the matched set of nodes of the

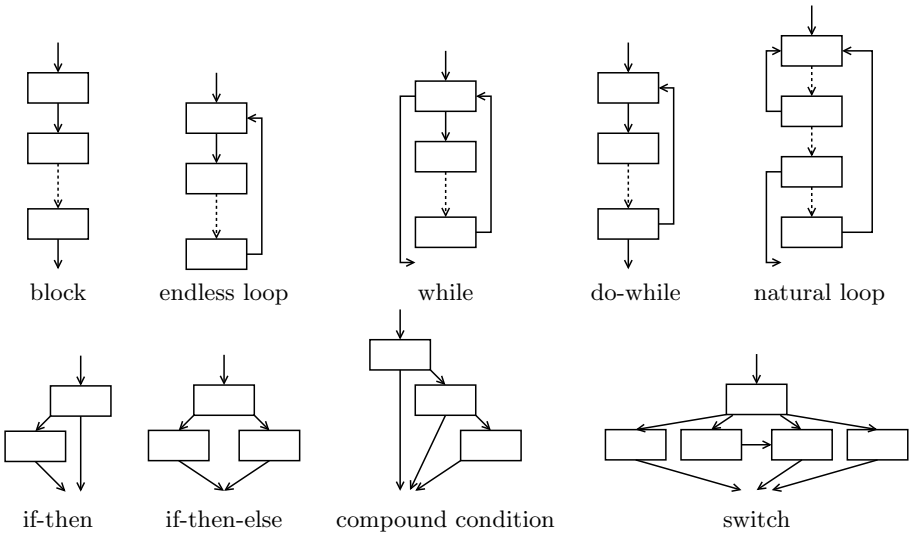

Fig. 4. Control structure templates

graph of regions is folded into one new region node. Algorithm terminates if either the graph of regions contains the only node, or no template matched.

The templates for structural constructs are shown in Fig. 4. The control structure of the decompiled program depends on the order in which templates are matched. Empirically, the following order turns out to be the best: loops, operator `switch`, conditions. A backward edge form node m to node n indicates a loop, which could be one of the following types:

- endless loop — no exit node,
- loop with initial condition — node n has two outgoing edges,
- loop with tail condition — node m has two outgoing edges.

When all loops are marked out the template of the `switch` operator is tried on. During compilation an operator `switch` is translated either to a sequence of `if`'s or to an indirect jump to the address from a jump table as shown in Fig. 5. An example of reconstruction of `switch` operator is presented in Fig. 6.

5 Datatype reconstruction

Data type reconstruction is presented in this section.

Objects for our type analysis algorithm are connected components of the DU-chains (also called *webs* [8]). It allows handling reuse of the limited number of the hardware registers. We prefer DU-chains over SSA representation to avoid excessive copying in join sites. DU-chains are created for 1) CPU registers, 2) memory locations at fixed offsets by the stack frame pointer corresponding

```
        cmpl    $6, -8(%ebp)
        ja      L10
        movl    -8(%ebp), %edx
        movl    L11(,%edx,4), %eax
        jmp     *%eax
        .section .rdata,"dr"
        .align 4
L11:
        .long   L3
        .long   L4
        .long   L5
        .long   L6
        .long   L7
        .long   L8
        .long   L9
        .text
```

Fig. 5. Compilation of `switch`

to function parameters, local variables, spill slots, etc, 3) memory locations at fixed offsets by the stack pointer corresponding to parameter passing at call sites, 4) memory locations at fixed addresses corresponding to global variables, and 5) object dereferences corresponding to pointer dereferences in C programs, and 6) offseted object dereferences corresponding to array or field accesses in C programs.

Our static type reconstruction algorithm consists of two phases:

1. Basic type reconstruction.
2. Composite type reconstruction.

5.1 Basic Type Reconstruction

The primary goals of the basic type reconstruction algorithm are as follows:

1. Determine if an object holds a pointer, floating-point or integral value.
2. Determine the size of the integral and floating-point type, stored in an object.
3. For integral type determine if an object holds a signed or unsigned value.

The set of the types being reconstructed is the set of the basic C types and the generic pointer type `void *`. Each type is identified by three attributes: *core*, *size*, and *sign*. A correspondence between the C types and the triples of attributes is established. For example, triple $\langle \text{int}, 4, \text{unsigned} \rangle$ corresponds to `unsigned int` type on the 32-bit ia32 architecture.

A system of equations is written for an assembly program as follows [1]. The instruction *add* r_1, r_2, r_3 is mapped to an equation $obj_3 : T_3 = obj_1 : T_1 +$

[1] We use RISC-like instructions for the sake of simplicity in the examples.

Original program	Decompiled program

`switch (getch()) {`
`  case 'a':`
`  result = 1;`
`  break;`
`  case 'b':`
`  result = 2;`
`  break;`
`  case 'c':`
`  result = 3;`
`  break;`
`  case 'd':`
`  result = 4;`
`  break;`
`  case 'e':`
`  result = 5;`
`  break;`
`  case 'f':`
`  result = 6;`
`  break;`
`  case 'g':`
`  result = 7;`
`  break;`
`  default:`
`  result = 10;`
`  break;`
`}`
	`eax6 = getch ( );`
	`var8 = eax6 - 97;`
	`if(!((unsigned)var8>6)){`
	`  switch (var8) {`
	`  case 0:`
	`  var9 = 1;`
	`  break;`
	`  case 1:`
	`  var9 = 2;`
	`  break;`
	`  case 2:`
	`  var9 = 3;`
	`  break;`
	`  case 3:`
	`  var9 = 4;`
	`  break;`
	`  case 4:`
	`  var9 = 5;`
	`  break;`
	`  case 5:`
	`  var9 = 6;`
	`  break;`
	`  case 6:`
	`  var9 = 7;`
	`  break;`
	`  }`
	`} else {`
	`  var9 = ( int ) 10;`
	`}`

Fig. 6. Decompilation of `switch`

$obj_2 : T_2$, where objects obj_3, obj_1, obj_2 were formed for registers r_1, r_2, r_3 and types T_1, T_2, and T_3 are being reconstructed. An equation for each attribute is written as shown below. For example, there is the equation for the attribute *core* $T_3(core) = T_1(core) + T_2(core)$. The types of the objects are computed by an iterative algorithm for each attribute. The initial values for type attributes are taken from constraints, which depend on used hardware registers, instruction code operation, etc.

There exist four types of constraints: instruction, flag, sign, and environment. A *register constraint* affects the *core* and the *size* attributes of the type variable. An *instruction constraint* affects the *core*, the *size*, and the *sign* attributes of the involved type variables. A *flag constraint* affects the *sign* attribute of the involved type variables. An *environment constraint* affects all type variable attributes. The environment constraints are imposed based on usage of the

standard library functions. The prototypes of the standard library functions and their corresponding constraints are considered to be known during decompilation. And a *profile constraint* that is imposed based on profile-based analysis that is omitted in this paper. These constraints may indicate that type *is not* pointer or *is not* unsigned, and are optional.

For computing object types we use a subset lattice over types attributes. The type attribute values are joined by the set intersection operation, so the join operation is monotone. An example for the attribute *sign* is shown below.

$$
\begin{array}{l}
(1)\ T_1(unsigned) => \{1,0\}+ \\
(1)\ T_2(noinformation) => \{1,1\} = \\
(1)\ T_3(unsigned) <= \{1,0\}
\end{array}
$$

$$
\begin{array}{l}
(2)\ T_1(unsigned) => \{1,0\}+ \\
(2)\ T_3(unsigned) => \{1,0\} = \\
(2)\ T_2(unsigned) <= \{1,0\}
\end{array}
$$

Two equations (1) and (2) are shown. Type T_1 is `unsigned`. There is no information for type T_2, but according to the C language standard [**?**] type T_3 should be `unsigned`. According to another equation type T_1 is `unsigned`, type T_3 is also `unsigned`, so type T_2 is `unsigned` as well according to the C language standard. Then the join operation is performed for $T_2(1)$ and $T_2(2)$, giving that type T_2 is `unsigned`.

If the attribute set for some type become empty, a conflicting object usage is detected. The resulting C type would be a `union`, or a user intervention might be requested. Also the resulting sets may correspond to several C types which means that the program being decompiled does not have enough information to reconstruct the types unambiguously. An arbitrary C type might be chosen in this case, or a user intervention might be requested.

5.2 Composite Type Reconstruction

The composite type reconstruction algorithm was presented at PSI 2009. Thus, it is omitted from this paper.

6 Decompiler TyDec

An experimental decompiler "TyDec" from Intel x86 architecture to C is being developed at Institute for System Programming Russian Academy of Sciences. The decompiler is able of reconstructing all high-level structure constructs of C. It is intended to supports a number of input formats: assembly programs either in AT&T or Intel syntax, executable files, execution traces collected on a CPU simulator. The decompiler provides a GUI for analyst to control and steer the process of decompilation. The overall decompiler structure is shown in Fig. 7.

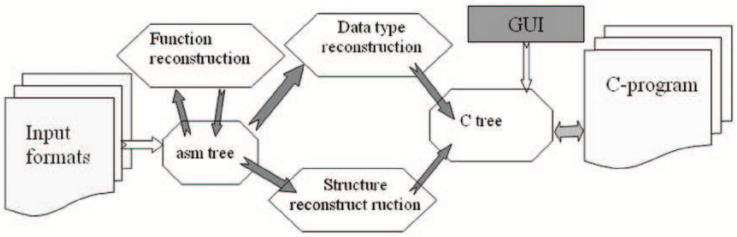

Fig. 7. TyDec Decompiler

ASMTree is an internal representation of the input program as a vector of instructions. Each instruction contains the opcode and its arguments. Arguments may be of the following types: register, constant, string (used for labels in jump instructions), memory address, sum of two arguments, product of two arguments. The ASMTree provides an interface to the internal representation other components of the decompiler.

Function reconstruction module transforms the ASMTree by recovering functions from the instruction stream.

Datatype reconstruction module is described in more detail later.

Structure reconstruction module builds the control flow graph of functions, builds dominator trees and performs structural analysis as described above. The module also provides means for visualization of the control-flow graph and the related structures.

CTree is an internal representation of the output program. The CTree representation is used for storing the output C program in a form convenient for analysis and transformation. CTree provides means of controlling the output text generation, synchronized navigation in the assembly and C windows of the GUI.

GUI provides a graphical interface to the decompiler and supports dialog mode of decompilation.

6.1 Type Reconstruction Module

The overall structure of static basic and composite type reconstruction algorithm is shown in the Fig. 8.

The initial object set is obtained from the assembly program. Than this initial object set is passed to the module that implements basic type reconstruction algorithm. As the result of its executing we get 1) reconstructed basic C types, 2) pointers (indirect objects), and 3) not reconstructed types. The set of indirect objects (pointers) is passed to the composite type reconstruction module, which builds skeletons of composite types and makes new objects for structure fields and array elements. This new object set is merged with set of objects with not reconstructed types and is passed back to the basic type reconstruction

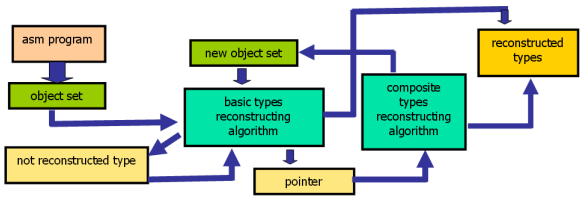

Fig. 8. The scheme of the static type reconstruction algorithm

module. Objects that correspond to structure fields or array elements may be also indirect. A composite type is completely reconstructed if all its fields' types are reconstructed. Algorithm terminates when the set of unrecognized objects and pointers set become empty. Algorithm converges as pointers have finite levels of indirection and basic type reconstruction algorithm operates with finite lattice (type attribute set) with monotone join function (set intersection).

Static type reconstruction algorithm will be modified for using profile-based information as shown in the Fig. 9. Heap and value profiles are collected during one or several profiling runs and is stored in the database. They are used by the basic type reconstruction algorithm as additional type constraints affecting the *core* and the *sign* attributes of involved type variables. Composite type reconstruction algorithm use profile-based information for building type skeletons and for more precise sets of equivalent bases.

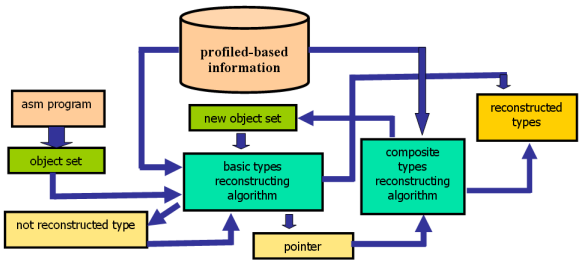

Fig. 9. Modified type reconstruction algorithm

7 Experimental results

This section discusses results of application of the described algorithm to a number of sample C programs.

All source code is taken from FreeBSD-4.0 distribution and compiled with gcc-4.3.2 compiler at the default optimization level on a Linux system. The `execute` function is compiled at -O2 optimization level, so all variables except one were put on the registers. Only one variables was stored in the stack frame. The summary of hand-checked code is shown in the Table 1. The 'CLOC' column shows the number of LOC in the C source, and the 'ALOC' column shows the number of LOC in the corresponding assembly listing.

The type reconstruction statistics for all examples in shown in the Table 2, and the detailed listing of the reconstructed types for the `isnow` function is shown in the Table 3. Note, that the `char *endp` argument of the `isnow` function is only used as an argument to `getfield` function and never dereferenced. By the means of only static analysis it should be reported as a 4-byte integer. However, its *PC* metrics is 1, thus it is assumed to be a `void*` pointer.

The 'BT' column in the Table 2 shows the total number of stack variables of basic types. Variables taken on the registers are now counted. The 'Exact BT' column shows the number of variables, which types were reconstructed exactly, and the 'Corr BT' column shows the number of variables, which types were reconstructed correctly. The 'Fail BT' column shows the number of incorrectly reconstructed basic types. The 'PTRs' column shows the total number of stack variables of pointers to basic types. The 'Exact PTR', 'Corr PTR', 'Fail PTR' columns show the numbers of exactly, correctly and incorrectly reconstructed types respectively.

Example	CLOC	ALOC	Description
`35_wc.s`	107	245	`cnt()` function from wc utility (file 'wc.c')
`36_cat.s`	27	104	`raw_cat()` function from cat utility (file 'cat.c')
`37_execute.s`	486	1167	`execute()` function from bc utility (file 'execute.c')
`38_day.s`	173	346	`isnow()` function from calendar utility (file 'day.c')

Table 1. Hand-checked sample code

	BT	Exact BT	Corr BT	Fail BT	PTRs	Exact PTRs	Corr PTRs	Fail PTRs
35	7	1	6	0	2	1	1	0
36	7	4	3	0	1	0	0	0
37	1	0	1	0	0	0	0	0
38	9	8	1	0	4	2	1	1*

Table 2. Hand-checked reconstructed type statistics

[ebp+8]	[unsigned] int, void* (profile-based)	char *endp
[ebp+12]	int *	int *monthp
[ebp+16]	int *	int *dayp
[ebp+20]	[unsigned] int*	int *varp
[ebp-28]	[unsigned] int	int flags
[ebp-24]	int	int day
[ebp-20]	int	int month
[ebp-16]	int	int v1
[ebp-12]	int	int v2

Table 3. Detailed type reconstruction for the `isnow` function

8 Conclusion

The paper discusses possibility and feasibility of decompilation. Three challenges of decompilation are identified:

- identification of classes of low-level programs suitable for decompilation,
- development of algorithms and methods of decompilation for identified classes,
- quality maintenance of decompilation.

We identified a class of strictly conforming C programs as a candidate class of suitable for automatic decompilation programs. Definition of correct decompilation is proposed for this class. Also, for this class of programs we described algorithms and methods for the most important stages of decompilation.

An experimental decompiler *TyDec* being developed at Institute for System Programming by the authors is described.

Directions of future research include development and integration of profile-based techniques that assisting static type reconstruction into TyDec.

References

1. CodeSonar: http://www.grammatech.com/products/codesonar/overview.html/
2. CodeSurfer: http://www.grammatech.com/products/codesurfer/
3. report, T.: http://citeseer.nj.nec.com/weide94reverse.html

4. Cifuentes, C., Fraboulet, A.: Interprocedural static data flow recovery of high-level language code from assembly. Technical Report
5. Cifuentes, C., Simon, D., Fraboulet, A.: Assembly to high-level language translation. Int. Conf. on Softw. Maint.
6. Cifuentes, C., Emmerik, M., Lewis, B., Ramsey, N.: Experience in the design, implementation and use of a retargetable static binary translation framework. Technical Report
7. DCC: The dcc decompiler, http://www.itee.uq.edu.au/ cristina/dcc.html
8. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers (1997)
9. Mycroft, A.: Type-based decompilation. European Symp. on Programming **1576** (1999) 208 – 223
10. Balakrishnan, G., Reps, T.: Divine: Discovering variables in executables. Verification, Model Checking, and Abstract Interpretation **4349** (November 2007) 1–28
11. Boomerang: Boomerang decompiler sdk, http://boomerang.sourceforge.net/
12. REC: Reverce ingeneering compiler, http://www.backerstreet.com/rec/rec.htm
13. Hex-Rays: Hex-rays decompiler sdk, http://www.hex-rays.com/
14. IDAPRO: Interactive disassembler, http://www.idapro.com/