High-Level Composite Type Reconstruction During Decompilation from Assembly Programs

Katerina Troshina[1] and Alexander Chernov[1]

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

Abstract. This paper presents a method for automatic reconstruction of high-level composite types during decompilation of C programs from assembly code. The proposed method is based on expressing memory access operations as pairs (*base*+*offset*), then building sets of equivalence for all memory access bases used in the program and accumulating sets of offsets for all classes of equivalent bases. Experimental results obtained for a number of open-source programs are also presented. The method is an essential part of the tool for program decompilation being developed by authors.

1 Introduction

Decompilation is a reconstruction of a program in a high-level language from a program in a low-level language.

There are many reasons why program decompilation might be necessary. For example, aiding comprehension of programs with unavailable source code for finding bugs and vulnerabilities. Also, it might be necessary to discover how a particular feature or piece of software is implemented or it might be necessary to reverse engineer some binary code for purposes of interoperability. Quality reconstruction of composite types is crucial for recovery of module interfaces. Also, the decompiled code may be optimized for a particular platform, or ported to other platforms.

This paper presents a method for automatic reconstruction of high-level composite types from assembly code. The C programming language is used as the target high-level language. We assume that the original program is a strictly conforming C program. The proposed method is applicable to a wide class of modern processor architectures, however a specification of machine-dependent rules defining constraints for available assembly instructions, registers, processor status flags, calling convention, etc. is required. The ia32 assembly language in AT&T syntax is used throughout the article for the low-level language.

A process of decompilation may be divided into the following stages: 1)recovery of subroutines (functions, procedures) from instruction flow, 2)recovery of parameters and return values of subroutines, 3)reconstruction of high-level control-flow structure, 4)reconstruction of high-level data types.

The first three stages of decompilation are rather well covered both in theory[1] and in practice (plugin HexRays for dissasembly IdaPro). Unfortunately the last stage has drawn little attention.

Reconstruction of high-level data types requires into basic type reconstruction and composite type reconstruction. Our basic type reconstruction algorithm is available in [2].

The remainder of the paper is organized as follows: section 2 discusses related work, section 3 outlines informal description of composite type reconstruction algorithm, section 4 presents its detailed description. Experimental results are presented in section 5 and the last section gives our conclusions and outlines directions for future work.

2 Related Work

There exist few works on type reconstruction during decompilation. One of the most closely related to our algorithms is algorithm VSA presented in [3]. However, the goal of VSA is value propagation, so certain attributes such as size and memory layout of structure types are recovered as side effects. The intuition behind this approach is that data access patterns in the program provide clues about how data is laid out in memory. This method is based on interval analysis and does not join separated uses of the same structure type justified that the main purpose of the VSA is security vulnerability analysis, not data type reconstruction.

Mycroft [4] gives a method of recovering types of variables during program decompilation. He presents a unification-based algorithm for performing type reconstruction. The type system uses disjunctive constraints when several type reconstructions from a single usage pattern are possible.

Past works on decompiling assembly code to high-level languages are also related to our goals [1]. These works have done much for reconstruction of control-flow structures. However, little attention is paid to reconstruction of variable types.

Also there exist several decompilers to the C language such as Boomerang, and REC, however they do not support reconstruction of composite types. Another example is the Hex-Rays plugin to the Ida Pro disassembler, which is probably the most advanced decompiler-like tool currently in existence, although it typically reconstructs arrays as generic pointers to the first element and structures as variables. Generally, existing machine code decompilers have significant deficiencies. These include poor recovery of parameters and returns, poor handling of indirect jumps and calls, and poor to nonexistent type analysis [5].

3 Informal description of composite type reconstruction

Before proceeding to the algorithm, let us introduce some terminology.

Let us define *objects* for type analysis. There are three classes of entities forming objects for type analysis: 1) registers, 2) direct memory locations, and 3) indirect memory locations.

- **Registers.** Certain platforms (especially, ia32) have a very limited set of the general purpose CPU registers and they are often reused in programs. We use register objects corresponding to connected components of the DU-chain (also called *webs* [6]) graph for the CPU registers. For simplicity in this paper they are denoted just as register names, e.g. `%eax`.
- **Direct memory locations** are 1) memory locations at absolute addresses corresponding to global variables, 2) memory locations at fixed offsets from the stack frame pointer register, e.g. `-2(%ebp)` corresponding to local variables, spill slots, etc, and 3) memory locations at fixed offsets from the stack pointer register `%esp` or pushed on stack by corresponding instructions. Originally they were parameters passed at call sites. Objects for them are also built based on DU-chains.
- **Indirect objects** are dereferences of other objects, built at the points of memory access operations in the assembly code.

A memory access operation is a memory load or memory store operation. The following three exceptions do not qualify as memory access operations because they correspond to other types of objects.

- Direct address loads and stores (global objects):
  `movl *0x12345678, %eax`
- Stack frame loads and stores (local objects):
  `movw %cx, -8(%ebp)`
- Stack pointer loads and stores (function argument objects):
  `pushl %ecx`

A local address expression is an expression computing the memory address in a memory access instruction. For example, address expression in the instruction `movl 12(%ebx), %ecx` is `%ebx + 12`.

An address expression is an expression computing the memory address in a memory access operation. Such a computation may require several assembly instructions and may even be split across basic block boundaries. In the example

```
movl    -16(%ebp), %esi
lea     (%esi,%eax,4), %ebx
movl    12(%ebx), %ecx
```

the address expression is `-16(%ebp) + %eax*4 + 12`.

An untraceable value in an assembly program is a result of a load from memory or a value in a register, which cannot be proved to be known.

The purpose of the algorithm for composite type reconstruction is to extract maximum possible information about possible composite types from the assembly listing. Consider the following composite types:

Structures. A structure is a collection of possibly elements of different types located in the memory sequentially. A structure is characterized by its size and occupies a memory region of this size. A compiler may insert padding bytes between the structure fields or at the end of the structure in order to align the fields according to the architecture requirements. Symbolic field names are eliminated during compilation and replaced with numeric offsets. Fields of different structure types may have the same numeric offset, thus some non-local information is required to map numeric offsets back to some new symbolic names.

Structure fields may be reconstructed during decompilation by collecting numeric offsets used for accessing memory regions of supposedly the same type. The types of base pointers are tracked through the program, and for any two memory accesses it is computed if these two memory accesses have provable the same type. This problem is solved by a *type propagation* algorithm, described later, similar to a data-flow constant propagation algorithm.

This algorithm is conservative, and for some memory access pairs type equivalence could not be proved, although these memory accesses do have the same types in the original program. Because of this one structure type in the original program can be reconstructed into several structure types in the decompiled program. The reconstructed types may even have different recovered fields.

It is impossible to guarantee recovery of all fields of all structure types, just because the decompiler does not have information about fields, which are not accessed in the program fragment being decompiled. The undetected fields are reconstructed into arrays of `char` as well as padding bytes.

In the C language structure fields may have arbitrary types: built-in, pointer, array and structure. If one structure type is nested in another structure type it is generally indisguishable from the case when fields of the nested structure type are placed directly in the enclosing structure type, as identical assembly code is generated in both cases. Because of this nested structure types cannot be automatically reconstructed as such. Several heuristics may be proposed allow resolving this case in some situations.

It is worth mentioning that the function parameter area and the local variable area can also be treated as a structure located on the stack with `%ebp` as the base pointer. Thus automatic reconstruction of structure type values located on stack is also problematic. However, as the local variables and function parameters provide initial information for type reconstruction during decompilation, they are treated not as structure fields anyway.

Arrays. An array is a collection of elements of the same type located sequentially in memory. As opposed to structures, which are accessed using constant numeric offsets from the base address, array elements are accessed using index expressions. For example, given an array `int m[32];` and expression `m[j]`, the corresponding address expression may look like `m + j * 4`. Multiplication in address expressions is a "fingerprint" of arrays.

On the other side, since the array elements have the same type, the type of elements `m[0]` and `m[j]` is the same, and the index expression provides no additional information for type reconstruction and thus can be omitted. When

the index part of the address expression is dropped, only its base address and constant offset are considered similar to structure address expressions. The pair *base* + *expression* is a subject of further type recovery analysis.

Multiplicative components of address expressions provide information about the size of array elements. For the example above we can deduce, that 4 is the size of the array elements. Address expressions may have more complicated form in case of multi-dimensional arrays, however, the size of the array elements is the minimal constant in the address expression.

Things complicate a bit if multiplication is used in the index expression in the high-level C program. Consider example `m[2*j]`. The constant 2 transforms the corresponding low-level address expression to `m + j * 8`. Using only one address expression one may incorrectly decide that the array element size is 8. This issue can be resolved if all address expressions pointing into the same array are considered. We may estimate the element size as *GCD* of the multiplicative constants of the address expression.

The estimated element size is not necessarly the true array element size. If all array index expressions in the original program use the same constant multiplier, for example `m[2*j]`, the low-level program will contain no clue to recover the true size of the array elements. A structure of size 8 will be created instead:

```
struct s1
{
  int f1;         // or other 32-bit type depending on its usage
  char unused[4]; // never used in program
};
```

which however produces the same assembly code and thus is correct.

Array access operations with constant index expressions, for example, `m[5]`, are compiled into address expressions with constant offsets from the base addresses. Such address expressions contain no multiplicative part and are indistinguishable from structure field accesses. In certain cases ambiguity may be resolved if all address expressions into the same memory region are considered together.

Determining the number of elements of arrays is a harder task. Almost all C compilers provide no array bounds checking for the generated code, although the C standard does not deny such possibility. Thus information about array sizes is not preserved explicitly in the generated code. One possible approach to array size recovery would be computation of value ranges of index variables using some sort of partial evaluation technique. However, this approach rarely works due to insufficient precision of static analysis. As generally, we'd like to avoid manipulations with values in favor of manipulations with types, out method attempts to infer array sizes by analyzing structure layouts. The array base address gives a natural lower limit of the array memory region. The upper limit of the array memory region in some cases is also known. It would be the memory region boundary, or the lower boundary of another structure field or local variable. By establishing the lower and upper boundaries of the array memory region it is possible to estimate the number of its elements.

4 Algorithm description

This section describes our algorithm in detail.

Address expression recovery. Let us define *an address expression term* t as follows:

$$\begin{aligned} t &::= \text{addr}(b, o, m) \mid \text{nil}, \\ b &::= \text{label} \mid 0, \\ o &::= \text{integer}, \\ m &::= 0 \mid f \mid \text{sum}(f, m), \\ f &::= \text{mul}(\text{integer}), \end{aligned}$$

where integer is an integer number, label is chosen from an enumerable set of labels as described in detail below.

Assume that the reaching definitions data-flow analysis is performed for the registers in the input program. For each register usage u the set of assembly instructions defining (i.e. assigning a value to) this register is known. Denote the set of definitions as $\mathit{Defs}(u) = \{r_1 \dots r_m\}$ for register usage u. For each register definition let $\mathit{AE}(r)$ be the corresponding address expression term. Initially $\mathit{AE}(r) = \text{nil}$.

Term $\text{addr}(0, 0, \text{mul}(1))$ denotes arbitrary (untraceable) value and is abbreviated as *ANY*. Term nil denotes yet unknown value.

Assume that lists of mul terms are always sorted in the ascending order, and each multiplier is greater than 0. Let t_1 and t_2 be two lists of mul terms. The *Merge* function $\mathit{Merge}(t_1, t_2)$ accepts two lists and returns a merged list of the GCD of terms in the ascending order without repetitions.

Let us define the *Join* function over a set of address expression terms $t_1 \dots t_m$: $\mathit{Join}(t_1 \dots t_m)$ as follows.

- if for each $i, j = 1 \dots m$ $t_i \neq \text{nil}$ and $t_i = t_j$, then $\mathit{Join}(t_1 \dots t_m) = t_1$;
- if there exist i, such as $t_i = \mathit{ANY}$, then $\mathit{Join}(t_1 \dots t_m) = \mathit{ANY}$;
- if there exist indices i and j, such as $t_i \neq \text{nil}$ and $t_i \neq t_j$, then $\mathit{Join}(t_1 \dots t_m) = \mathit{ANY}$;
- otherwise $\mathit{Join}(t_1 \dots t_m) = \text{nil}$.

Let us define *Join* function over a set of register definitions $r_1 \dots r_m$: $\mathit{Join}(r_1 \dots r_m) = \mathit{Join}(\mathit{AE}(r_1) \dots \mathit{AE}(r_m))$.

The address expression term propagation rules are defined as follows.

- For instruction: `movl` R_1, R_2
  (which means copying register R_1 to register R_2) $\mathit{AE}(R_2) = \mathit{Join}(\mathit{Defs}(R_1))$.
- For instruction: `movl` \$$C$, R
  (which means loading a numeric constant into register R) $\mathit{AE}(R) = \text{addr}(0, C, 0)$.
- For instruction: `movl` *mem*, R
  (which means loading a value from memory into register R) $\mathit{AE}(R) = \text{addr}(\mathit{newlabel}(), 0, 0)$.
- For instruction: `addl` R_1, R_2, R_3
  let $t_1 = \mathit{Join}(\mathit{Defs}(R_1))$, $t_2 = \mathit{Join}(\mathit{Defs}(R_2))$.
  - if $t_1 = \mathit{ANY}$ or $t_2 = \mathit{ANY}$, then $\mathit{AE}(R_3) = \mathit{ANY}$;

  - if $t_1 = \text{nil}$ or $t_2 = \text{nil}$, then $AE(R_3) = \text{nil}$;
  - if $t_1 = \text{addr}(l_1, 0, 0)$, $t_2 = \text{addr}(l_2, 0, 0)$, then two outcomes are possible: $AE(R_3) = \text{addr}(l_1, 0, \text{mul}(1))$ or $AE(R_3) = \text{addr}(l_2, 0, \text{mul}(1))$; user intervention may be requested to resolve the conflict;
  - if $t_1 = \text{addr}(l_1, c_1, m_1)$, $t_2 = \text{addr}(l_2, c_2, m_2)$, $l_1 \neq 0$, and $l_2 \neq 0$, then $AE(R_3) = ANY$;
  - if $t_1 = \text{addr}(l_1, c_1, m_1)$, $t_2 = \text{addr}(0, c_2, m_2)$, then $AE(R_3) = \text{addr}(l_1, c_1 + c_2, \mathit{Merge}(m_1, m_2))$;
- For instruction: `addl` R_1, C_1, R_2
  where $R_1 = R_2$, that means adding number C_1 to register R_1 and storing the result back to $R_2 = R_1$. If $\mathit{Defs}(R_1) = \{R_3, R_2\}$, and definition point of R_3 dominates definition of R_2, and $AE(R_3) = \text{addr}(0, C_2, 0)$, then $AE(R_2) = \text{addr}(0, 0, \text{mul}(C_1))$. This case corresponds to strength-reduced inductive loop variables.
- For instruction: `mull` R_1, C, R_2
  let $t_1 = \mathit{Join}(\mathit{Defs}(R_1))$
  - if $t_1 = ANY$, then $AE(R_2) = ANY$;
  - if $t_1 = \mathit{nil}$, then $AE(R_2) = \mathit{nil}$;
  - if $t_1 = \text{addr}(l_i, c_1, m_1)$, then $AE(R_2) = \text{addr}(0, 0, \text{sum}(\text{mul}(C), C * m_1))$, where $C * m_1$ denotes multiplication of each element of the m_1 list by C.
- For instruction: `call` *Proc*
  assume that return value of the subroutine is located in the `%eax` register. So this instruction is considered as definition of `%eax`, and the address expression is $\text{addr}(\mathit{newlabel}(), 0, 0)$.

Most other instructions give ANY address expression term. The rules for them are omitted for brevity. The address expression terms are calculated until a fixed point is reached, i. e. all the terms $AE(R_i) \neq \text{nil}$ and there is no applicable rule, which would change the address term value.

Finally, for each memory access the corresponding address expression term is computed in the form $\text{addr}(B, O, \text{sum}(\text{mul}(C_1), \text{sum}(\text{mul}(C_2) \ldots)))$, which corresponds to the form: $(B + O + \sum_{j=1}^{n} C_j x_j)$, where B is the base address, O — the constant (fixed) offset, and the rest is the multiplicative component.

For strictly conforming C programs B will be some label l_i. If the original program uses some low-level manipulations with pointers, the address expression could as well be ANY.

Finally, only those address expression terms, which are used as the addresses for memory loads and stores are preserved in the set of address expression terms. All other (temporary) address expression terms are discarded.

Label propagation. A unique label is assigned to each memory load and subroutine return value on the address expression recovery step as described above. A label expresses a possibility, that the labeled value has some yet unknown pointer type. If the value has actually a non-pointer type, this possibility

is not realized. Of course, several memory load may have the same resulting pointer type, thus we want to build sets of labels, which correspond to the same high-level type.

To find the label equivalence classes efficiently a disjoint-set data structure is maintained for labels. Initially each label is placed into its own disjoint set.

To establish label equivalence relation we use interference graph for registers and primary memory locations. Let $LS_{in}(obj, n)$ be the disjoint set of labels corresponding to the object *obj* immediately preceding position n in the assembly listing. Let $LS_{out}(obj, n)$ be the disjoint set of labels corresponding to the object *obj* immediately following position n in the assembly listing. For simplicity we omit $_{in}$ qualifier from now on. The object *obj* may be a register or a primary memory location. Currently we ignore aliasing issues for memory locations, however a missed aliasing between two objects may result in generation of two separate C types instead of one in the decompiled C program.

The *label-set analysis* is a forward iterative data-flow analysis performed until a fixed point is reached. The disjoint-set *Union* operation is the join function for label sets. If one object is copied to another, *Union* operation on the corresponding disjoint sets is performed. Other instructions (including address arithmetic) do not change label sets.

Aggregation. We say, that two address expressions e_1 and e_2 have *the common base*, if $e_1 = \text{addr}(l_1, o_1, m_1)$, $e_2 = \text{addr}(l_2, o_2, m_2)$, and $l_1 \equiv l_2$.

Let $t_i = \mathit{Find}(l_i)$ be the representative of the equivalence class of label l_i. For each address expression term $\text{addr}(l_i, o_i, m_i)$, if l_i is a label, replace l_i with t_i.

Let us define *array aggregation* AA relation between two address expression terms e_1 and e_2 as follows:

$$\frac{e_1 = \text{addr}(t_1, o_1, \text{add}(\text{mul}(C), m_1)), e_2 = \text{addr}(t_1, o_2, \text{add}(\text{mul}(C), m_2)), |o_1 - o_2| < C}{AA(e_1, e_2)}$$

Unfortunately, the AA relation is not transitive. Consider the following example:

```
struct s1 {
  struct s2 { int f1; int f2; } a[1];
  struct s3 { int f3; int f4; } b[1];
};
```

An access to the `f1` field produces address expression term $\text{addr}(b, 0, \text{mul}(8))$, where b is a base address of the structure, 0 is the offset of the `f1` field from the beginning of the structure, 8 is the size of `struct s2`, because the field `f1` is located in the array of such structures. Similarly, an access to the `f2` field produces address expression term $\text{addr}(b, 4, \text{mul}(8))$, an access to `f3` — $\text{addr}(b, 8, \text{mul}(8))$, and an access to `f4` — $\text{addr}(b, 12, \text{mul}(8))$. All the terms belong to the transitive closure of the AA relation, but, for example, `f1` is not related to `f3`.

The AA relation is called *conflicting*, if the transitivity property does not hold for it, e. g. there exist two address expression terms e_1 and e_2 such as both belong to the transitive closure of the AA relation, but $|o_1 - o_2| \geq C$.

To resolve the conflict divide the conflicting class of the transitive closure into several subclasses as follows:

- a subclass contains only address expression terms with $|o_1 - o_2| < C$,
- for any two subclasses K_1 and K_2: $|\min_{K_1}(o) - \min_{K_2}(o)| \geq C$, e. g. difference between minimal offsets in the subclasses is greater then or equal to the array size C inducing the partition.

The AA relation with conflict resolution fix separates the set of address expression terms into *aggregation sets*. Let $K = \{e_1, e_2, \ldots, e_m\}$ be an aggregation set. Let $o = \min_{j=1}^{m} o_j$ be the minimal offset in the given aggregation set, let C be the first multiplier in the multiplier list of the terms. Denote $e' = \text{addr}(b, o, \text{mul}(C))$ as the address expression term for an array of nested structures. o is considered as the start offset for the nested structure type. Then the address expression terms in the set K are modified as follows: $\text{addr}(b, o_i, \text{add}(\text{mul}(C), m_i)) \rightarrow \text{addr}(e', o_i - o, m_i)$. Note, that the offset is modified, and the first multiplier in the multiplier list is removed.

The aggregation operation changes the set of all address expression terms, particularly because the least multiplier is removed. For the new set of all address expression terms the new aggregation relation AA is constructed, conflicts are resolved, and the next step of aggregation is performed. The process iterates until a fixed point is reached.

```
struct t { int f1; int f2; };     [1]  _main:  pushl   %ebp                 [21]         call    _readint
struct s {                        [2]          movl    %esp, %ebp           [22]         movl    %eax, 16(%esi,%ebx,8)
  struct s *a; int b; char c;     [3]          subl    $12, %esp            [23]         incl    %ebx
  struct t d[4]; char e[4];       [4]          andl    $-16, %esp           [24]         cmpl    $3, %ebx
};                                [5]          xorl    %edi, %edi           [25]         jle     Q7
int main(void) {                  [6]          call    _readint             [26]         xorl    %ebx, %ebx
  struct s *p = 0, *q;            [7]          testl   %eax, %eax           [27] Q11:    call    _readchar
  int bb, cc, dd, ee, i;          [8]          movl    %eax, %ebx           [28]         movb    %al, 44(%ebx,%esi)
  while ((bb = readint()) >= 0) { [9]          js      Q17                  [29]         incl    %ebx
    q = malloc(sizeof(*q));       [10] Q12:    movl    $48, (%esp)          [30]         cmpl    $3, %ebx
    q->a = p;                     [11]         call    _malloc              [31]         jle     Q11
    p = q;                        [12]         movl    %edi, (%eax)         [32]         call    _readint
    q->b = bb;                    [13]         movl    %eax, %esi           [33]         testl   %eax, %eax
    q->c = readchar();            [14]         movl    %eax, %edi           [34]         movl    %eax, %ebx
    for (i = 0; i < 4; ++i) {     [15]         movl    %ebx, 4(%eax)        [35]         jns     Q12
      q->d[i].f1 = readint();     [16]         xorl    %ebx, %ebx           [36] Q17:    movl    %edi, (%esp)
      q->d[i].f2 = readint();     [17]         call    _readchar            [37]         call    _dowork
    }                             [18]         movb    %al, 8(%esi)         [38]         movl    %ebp, %esp
    for (i = 0; i < 4; ++i) {     [19] Q7:     call    _readint             [39]         popl    %ebp
      q->e[i] = readchar();       [20]         movl    %eax, 12(%esi,%ebx,8) [40]        ret
    }
  }
  return dowork(p);
}
```

Fig. 1. A C program and the corresponding assembly code

Structure reconstruction. Let S be the set of address expression terms. The set is separated into equivalence classes $S_1, S_2, \ldots, S_k$ induced by the *common base* relation. Let $S_i = \{\text{addr}(b_{i,1}, o_{i,1}, m_{i,1}) \ldots \text{addr}(b_{i,n}, o_{i,n}, m_{i,n})\}$. $b_{i,j}$ after the aggregation step is not necessary a label, but may be an address expression term. Let $O_i = \{o_{i,1} \ldots o_{i,n}\}$ be the set of offsets in S_i.

– if $O_i = 0$, i. e. only offset 0 is used to access memory from this base address, the address expression term corresponds to a dereference of a plain pointer or access to a plain array in the C program;
– otherwise a new structure type T_i is created with fields corresponding to the set of offsets O_i. Types of newly created fields are not yet known, but the objects are created and added to the set of object for further basic data type reconstruction.

The proposed method for composite type reconstruction has a number of drawbacks as summarized below:

– unions and explicit pointer type casts are not properly handled;
– byte access to objects (permitted by the C standard) is not properly handled;
– array sizes are recovered in rare cases;
– nested structures are reconstructed only if they form arrays.

Our experiments with proposed algorithm on a number of open-source and specially crafted test programs show that the assembly program typically had enough information to merge all uses of the same pointer type correctly.

Some difficult cases may be resolved by observing the run-time behavior of the program and gathering profiling information. This is one of the directions of further research.

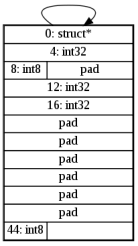
Example. To illustrate our approach to reconstruction of composite types consider a C program shown in Fig. 1. Functions `readint`, `readchar`, `dowork` are supplemental functions irrelevant to the essence of the example and used to suppress compiler optimizations. On the right side in Fig. 1 the corresponding assembly listing is shown. The original C program was compiled by MinGW GCC 3.4.5 compiler. Some irrelevant assembly listing details are omitted for brevity.

We do not write down all address expressions used in this program for brevity. For example, the address expression for memory store instruction in the line 22 is `(%esi + 16 + 8 * %ebx)`. Recovery of address expression term gives label L_{11} in place of `%esi` register, where L_{11} is a label assigned to the result of call to `malloc` function. `%ebx` corresponds to address expression term $\mathrm{addr}(0, 0, \mathrm{mul}(1))$. Finally, address expression term for the instruction in line 22 is $\mathrm{addr}(L_{11}, 16, \mathrm{mul}(8))$.

An aggregation set $\{e_{20} = \mathrm{addr}(L_{11}, 12, \mathrm{mul}(8)), e_{22} = \mathrm{addr}(L_{11}, 16, \mathrm{mul}(8))\}$, is detected by the aggregation step, where e_{20} corresponds to address expression in line 20, e_{22} corresponds to address expression in line 22. A new address expression term $S_1 = \mathrm{addr}(L_{11}, 12, \mathrm{mul}(8))$ is created corresponding to a nested structure type, and the address expression terms e_{20} and e_{22} are rewritten as $e_{20} = \mathrm{addr}(S_1, 0, 0)$, $e_{22} = \mathrm{addr}(S_1, 4, 0)$. Note, that expression terms e_{20} and e_{22} still have the common base with address expression terms $e_{12} = \mathrm{addr}(L_{11}, 0, 0)$, $e_{15} = \mathrm{addr}(L_{11}, 4, 0)$, $e_{18} = \mathrm{addr}(L_{11}, 8, 0)$, and $e_{28} = \mathrm{addr}(L_{11}, 44, \mathrm{mul}(1))$ corresponding to the memory access instructions in the lines 12, 15, 18, and 28 respectively.

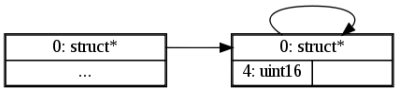
On the structure reconstruction step the address expression terms S_1, e_{12}, e_{15}, e_{18}, and e_{28} are considered as having the same base label L_{11}, that gives set $\{0, 4, 8, 12, 44\}$ of structure field offsets. Offset 16 was removed at the aggregation stage. Also the address expression terms e_{20} and e_{22} are considered as having the same base term S_1, that gives set $\{0, 4\}$ of offsets. Address expression terms S_1 and e_{28} correspond to arrays, the former having the element size 8, and the latter having the element size 1.

Using this information the structure type skeleton shown in Fig.2is generated. Types `t1`, `t2`, `t3`, `t4`, `t5`, `t6` are some basic or pointer types that have to be reconstructed by the algorithm for basic type reconstruction as described above. New objects are created for fields `f1`, `f2`, `f3`, `f4.f1`, `f4.f2`, and `f5`. The objects are added to the working set of the algorithm.




```
struct s2 {
  t1 f1; /* at offset 0 */
  t2 f2; /* at offset 4 */
  t3 f3; /* at offset 8 */
  struct s1 {/* sizeof(struct s1) == 8 */
    t4 f1; /* at offset 0 */
    t5 f2; /* at offset 4 */
  }  f4[]; /* at offset 12 */
  t6 f5[]; /* at offset 44 */
};
```

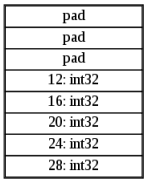
Fig. 2. The structure skeleton for a sample program




```
typedef struct shorts {
  struct shorts *next;
  short value;
} shorts;
short *lookaheads;
```

Fig. 3. Example 1: lalr.c

Note, that in this particular example the array sizes may also be recovered. For array `f4` its size is obtained as $(44 - 12)/8 = 4$.




```
struct tm {
int tm_sec, tm_min, tm_hour;
int tm_mday, tm_mon, tm_year;
int tm_wday, tm_yday, tm_isdst;
long tm_gmtoff;
char *tm_zone;
};
```

Fig. 4. Example 2: day.c

Name	CLOC	ALOC	Description
day.s	503	1383	day.c from calendar utility
lalr.s	711	1664	lalr.c from yacc utility

Table 1. Characteristics of the sample programs

5 Experimental results

This section discusses results of evaluation of our approach on a number of open-source programs.

A graphical representation of a structure skeleton for one of types in the lalr.c file of the FreeBSD yacc implementation is shown to the left in Fig. 3, and the corresponding source code is shown to the right.

Another example is shown the Fig. 4. The calendar program uses `struct tm` from `<time.h>`. The reconstructed view of this structure is shown. Note, that several fields (`tm_sec`, etc) are never used in the code so they are recovered as pad fields.

The size metrics for the sample programs are shown in the Table 1. The 'CLOC' column shows the number of C lines of code, and the 'ALOC' column shows the lines of code number of the corresponding assembly programs being decompiled.

6 Conclusion

The paper presents a method for automatic reconstruction of composite C types from assembly code generated by C compilers. The method calculates address expression terms for each memory access instruction in the assembly program, builds equivalence classes for base addresses and collects sets of offsets for all classes of equivalent bases.

The proposed method is an important part of a tool for program decompilation being developed by the authors.

The directions of further research include enhancing the static analysis with profiling results obtained from sample runs of the program being decompilaed.

the results obtained from tracing of the program being decompiled.

References

1. Cifuentes, C., Simon, D., Fraboulet, A.: Assembly to high-level language translation. Int. Conf. on Softw. Maint.
2. Dolgova, K., Chernov, A.: Automatic type reconstruction in disassembled c programs. In proceedings of the WCRE 2008 (October 2008) 202–206
3. Balakrishnan, G., Reps, T.: Divine: Discovering variables in executables. Verification, Model Checking, and Abstract Interpretation (November 2007)
4. Mycroft, A.: Type-based decompilation. European Symp. on Programming (1999)
5. Emmerik, M.: Static single assignment for decompilation. thesis
6. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers (1997)