成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.50 No.2 (2013)pp.63-72

Field-Sensitive Pointer Analysis for C Programs with Integer/Pointer Conversions

Eiichiro CHISHIRO*[1]

ABSTRACT : We present a field-sensitive pointer analysis algorithm for C in the presence of type conversion between integer and pointer. While field-sensitive analysis can give precise solution, it is notoriously difficult to design a correct analysis which handles all low-level dirty features of C. Most difficulties stem from arbitrary integer/pointer conversions allowed as an implementation-defined feature. To incorporate this feature into pointer analysis is not so easy as expected, and previous approaches are either unsound or greatly imprecise. In this paper, we first define the formal semantics which incorporates all low-level features of C and show that it is hardly to have precise analysis if arbitrary integer/pointer conversions are allowed. To address this, we identify the language restriction which many compiler developers implicitly assume and derive a precise analysis algorithm as an approximation of the semantics. Our analysis is shown to be sound under the restriction.

Keywords : compiler, pointer analysis

(Received September 20, 2013)

1. Introduction

Pointer analysis is a generic term for program analysis concerning various properties of pointers. Among a variety of pointer analysis, here we deal with the class of may points-to analysis problem, i.e., given a program, calculating all memory regions to which each pointer may point. This kind of information is known to be useful for compiler optimization, program verification, program understanding and so on.

Many algorithms have been proposed over the last two decades[10]. They can be classified by several aspects such as flow-sensitivity, field-sensitivity, context-sensitivity, object-sensitivity.

In this paper, we concentrate on the issues of field-sensitivity. Informally, analysis is called field-sensitive if it can distinguish each field of structure instead of treating a whole structure atomically. Field-sensitive analysis can give more precise solution than the field-insensitive one, and can provide more opportunities for compiler optimizations.

For type-safe languages such as Java or a type-safe subset of C, field-sensitive analysis is as easy as the field-insensitive one. However, in full C, field-sensitivity brings many complicating-factors, and makes it quite difficult to design a correct analysis[6].

The main challenge is to model correctly the layout of object, or memory block. In C, we can make an object of complex structure which is a combination of structures, unions, and arrays, and make an internal pointer to a field in the middle of a structure object. Moreover, we can access an object through more than one types by using a cast operator or union type. Thus, we cannot simply model an object as a mapping from field identifiers to values as in type-safe languages.

Since the layout of object is implementation-defined, programs which depend on it are not strictly-conforming. Thus algorithms which only treat ANSI C programs can ignore some of these intricate issues. However, when we develop a pointer analysis for a compiler, we should handle all implementation-defined behaviors correctly with respect to its specification.

Another Challenge, which has been overlooked in many works is that most compilers allow the conversion between integer and pointer if they are the same size. Here we use an integer type as a representative of all non-pointer types convertible to/from pointer types. The case for e.g., floating type is similar.

For example, the following code is legal if the size of `int` and pointer are both four bytes:

*[1] : Associate Professor/Dept. of Computer and Information Science（chishiro@st.seikei.ac.jp）

```
struct S { int m, n; } s; int i, *p;
i = (int)&s;
p = (int *)(i + 4);
```

We can use `p` to access `s.n`. This code may seem artificial, but this kind of conversion is widely used, e.g., in `va_arg` macro of the standard library.

Integer/pointer conversion is also used in embedded programs which interact with auxiliary devices through memory-mapped I/O ports. To access the memory region for the I/O ports, it must create a pointer to the I/O port from the system-specific integer value which represents the address of the I/O ports.

A naive but popular approach to this problem is to approximate an integer value as the set of arbitrary addresses. We call it *unknown pointer approach*[1]. This approach is obviously sound as it assumes that an integer value may point to any locations. Moreover, if no such conversion exists in the target program, there is no loss of precision and performance of analysis. However, if such conversion exists, and if there is an assignment `*p = e` where `p` has been assigned some integer value and e is an expression of integer type, the impact on the precision is disastrous. As both sides of the assignment can represent arbitrary locations (we use the term location and address interchangeably), every pointer is considered to point to any location by the assignment! Embedded programs using memory-mapped I/O ports match this case. This is serious in particular for an interprocedural analysis, since it makes such analysis completely useless.

Another approach is to track integer values as well as pointer values[14]. When a pointer value is created from an integer value, we can recover the original pointer value from this information. However, it is costly and difficult to perform precise integer-tracking. For example, an integer value can be copied indirectly through control flow. This is serious in particular for field-sensitive analysis, since we need to calculate the corresponding field from an integer value which may be the result of arbitrary arithmetic operations.

As we will show in Section 3, the integer/pointer conversion problem interacts with the object layout problem in a complicating way, and to design correct field-sensitive analysis, we need to consider all possible interactions exhaustively. It is very difficult to do in an ad-hoc way.

Contributions In this paper, we present an analysis which is field-sensitive and can handle programs with integer/pointer conversions safely and precisely .

To this end, we identify the language restriction which is necessary to do such analysis. The restriction is very weak. It only excludes programs which exploit the accidental coincidence between an integer value and the address of variables.

We give a formal model of memory which incorporates the restriction. The model can represent the meaning of non-well-typed memory accesses without going down to low-level, implementation-dependent details (e.g., byte representation of values). The meaning is the upper-bound of all specific compiler implementations, and thus our analysis based on it can be applicable to any compilers which accept the restriction above. This also incorporates the memory region denoted by system-specific integer values, which is ignored in previous works.

The model enables us to reduce the integer/pointer conversion problem by a simple reachability problem. The soundness of the analysis can be proved in a straightforward way. As far as we know, this is the first work to prove the soundness of pointer analysis for C in the presence of arbitrary type conversions.

Our analysis keeps field-sensitivity inside an array, while it does not distinguish each element of an array. This level of precision is desirable for many compiler optimizations. We do not know any works which achieve this in the presence of arbitrary type conversions.

2. Target Language Syntax and Semantics

First, we introduce a target language called RML (Register-Memory Language), which captures all low-level features of C and also is amenable to formal treatment.

Basically, RML is similar to low-level intermediate representations used in compilers. In particular, RML instructions are very similar to the one used in LLVM[12], except that `field` and `index` are combined into a single complex instruction in LLVM.

2. 1 Syntax

An RML Program P is a tuple $(T_N, T_R, I, N, entry)$. T_N and T_R are type environments for type names and registers, All operations in a program are performed through (virtual)

Table 1.　Instructions of RML.

Instruction	informal meaning
$alloc\ r$	$r_1 = (T_R\ r_1)malloc(\ldots)$
$copy\ r_1\ r_2$	$r_1 = r_2$
$load\ r_1\ r_2$	$r_1\ = *\ r_2$
$store\ r_1\ r_2$	$*\ r_1 = r_2$
$field\ r_1\ n\ r_2$	$r_1 = \&r_2 \rightarrow f$
$index\ r_1\ r_2\ r_3$	$r_1 = \&r_2[r_3]$
$conv\ r_1\ r_2$	$r_1 = (T_R\ r_1)r_2$

registers r. Registers are strongly typed, and their types are given by T_R. I assigns an instruction to each code location in $CLoc$. Representative instructions of RML are shown in Table 1. N represents control flow between instructions. $entry$ is the code location of program entry.

Informally, an execution of a program P starts at $entry$, and follows the control flow defined in N. At each step, an instruction is fetched from I and interpreted.

The meanings of most instructions are intuitive. There is no variable declaration. All variables are created by $alloc\ r$ where r holds the address of a created memory block (object). The address operator & in C is replaced with a use of the register. All address calculations in C (field references, array subscripting, and pointer arithmetic) are realized by field and index. Each field of structure or union is identified by a field index n: the first field is 0, the second is 1, and so on. Elements of array are designated by $index\ r_1\ r_2\ r_3$. r_2 is a pointer to an element of an array, and r_3 is an index relative to r_2. There are only two instructions which interact with memory. $load\ r_1\ r_2$ reads the content of the address r_2 into r_1, and store $r_1\ r_2$ writes the value of r_2 into the address r_1.

To concentrate on the subject of this work, we omit several language constructs such as the free operation, function calls, conditional branches and numeric operations. To support these constructs causes no theoretical difficulty.

Types of RML are inductively defined by the following syntax:

$$t ::= int\ n \mid ptr\ t \mid str\ ts \mid uni\ ts \mid arr\ t\ n \mid name\ x$$

where $n \in Nat$, $x \in TypeName$, and $ts \in List\ t$.

Types of RML are quite similar to C. An integer type $int\ n$ is parameterized with the byte size n. As each field is identified by a field index, fields of structure and union are represented just by a list of type of each field. $TypeName$ is introduced to represent recursive types. We assume that all type names used in a program P are defined in the type name environment T_N of P.

2. 2　Semantics

An execution state s of a program is a tuple of (pc, R, M, T_M, L, A, B). For state s, we represent the component X of s by $X\ s$. pc is the current code location. R and M are the contents of registers and memory. We will explain M in Section 2.3. L is the set of allocated locations. A associates a location with the code location which allocates it. For each location l of a block b, B assigns the top location of b to l. For a program P, we write the set of initial states as $StateInit(P)$, all states in the execution of P as $State(P)$, and a transition from s_1 to s_2 as $s_1 \rightarrow s_2$.

We assume there are two disjoint sets of data locations, $DLoc$ and $ELoc$. $DLoc$is used for allocation by $alloc$. $ELoc$ corresponds to the memory region which is designated by a system-specific constant address. We assume $ELoc$ is contiguous.

2. 3　Formal Model of Memory

The dynamic semantics of RML is defined as a state transition system. For the lack of space, we omit the full detail of the semantics and concentrate on a formal model of memory, which plays the central role in the design of pointer analysis.

It is simple for type-safe programs. Every value loaded from a location l has the preceding store to l of the same type. We can formalize this by simple axioms.

However, in the presence of casts or unions, a value of type t loaded from a location l may not have the preceding store of type t to l. A program may load a pointer value which was stored as an integer value (thus an implicit casting occurs at the load). Or worse, a program may load a pointer value which is composed of parts of two integer values at successive locations.

A common approach in the area of verification is to model a memory as a global array of byte (known as a RAM model[2)]). A store of a value is modeled by a decomposition of the value into a sequence of bytes and updates of the global array. A load of a value is modeled by a load of the global array and a composition of these bytes.

The problem of such byte-level model is that we need to formalize the details of all conversions and arithmetic operations to specify a location which is created from an integer. This is very tedious. It also makes the model heavily depend on a specific implementation policy.

To address these problems, we propose the following approach:

model a memory as a function M from locations and types to values, and a type map T_M which maintains the type of the value stored at each location;

if a load from a location l is well-typed, that is, the type of the load matches the type of l in T_M, the loaded value is the value stored at l;

if a load from a location l is not well-typed, the loaded value should is one of values satisfying a certain condition (which will be given later).

At each store of type t to l, we update the type map T_M by first removing all entries which overlap with stored locations $[l, \ldots, l + |t| - 1]$ and adding the new entry (l, t) to T_M.

Implementation-dependent feature is parameterized in the condition used at step 3. For *type-safe* C, it is false; this means such non-well-typed load causes undefined behavior. For *byte-level* C, it is the value which is composed of a sequence of bytes in $[l, \ldots, l + |t| - 1]$ in a specific way.

The decision of the condition has great impact on the applicability and precision of the analysis based on it. Roughly, the weaker the condition, the analysis based on it is more applicable to many compilers. False is the strongest, and the analysis based on this condition is unsound for almost all C compilers which allow non-well-typed load under certain conditions.

On the other hand, the stronger the condition, the analysis based on it *can* be more precise since the stronger condition decreases the possible values of a non-well-typed load. The condition used in byte-level C is strong in the sense that it determines the value of a non-well-typed load uniquely. Thus if we can analyze the value flow of all bytes, we can make the very precise analysis with respect to non-well-typed load.

However, as we said in the introduction, it is very difficult to track integer (non-pointer) values precisely. If we give up such integer-tracking, we need to consider arbitrary locations as the result of non-well-typed load. We have shown that this makes the analysis greatly imprecise.

In this paper, we propose the following condition: *A pointer value that can be the result of non-well-typed load should be one of the locations which have been accessed as integer, that is, satisfy the language restriction defined below.*

Definition 1 (Language Restriction). If a location l is created from an integer, l should satisfy one of the conditions below prior to the creation:

A location l was converted to integer type by a cast operator. All locations belonging to the same block of l are also regarded as converted.

A location l was stored at some location l', and then the content of l' was loaded as integer type by any means.

A location l is in the set of system-specific locations which is disjoint from any blocks allocated by a program.

This condition just excludes the case that a pointer to a variable x is created from an integer value which is accidentally the same as the location of x. On the other hand, this does not exclude any way of propagation of integer values. Thus we believe that this condition is acceptable for most compilers.

We present how to approximate the set of locations which satisfy the condition in Section 3.

3. Analysis Algorithm

Our analysis is based on the following observations which is derived by considering the effect of all kinds of conversions between pointers and integers:

1. A pointer-to-pointer conversion on a location l may cause the change of layout of the block which l belongs to;
2. The change of layout of a block b may cause all kinds of conversions on contents of locations which belong to b;
3. An integer-to-pointer conversion on a location l may cause a pointer-to-pointer conversion on l;
4. An pointer-to-integer conversion on a location l adds all locations which belong to the same block as l to the location set from which the result of integer-to-pointer

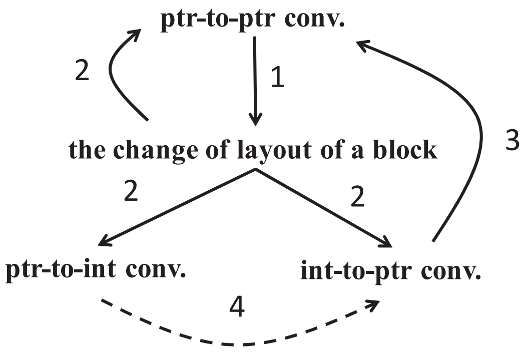

Figure 1. Interactions of Conversions. Solid arrow from *A* to *B* means that *A* causes *B*. Dashed arrow from *A* to *B* means that *A* changes the result of *B*. The number on an edge shows the corresponding observation.

conversions is determined.

We summarize these observations in Figure 1.

Let us consider the following code:

```
1: struct S { struct S *p1, *p2; int j; }
   s1, s2, s3;
2: struct T { int i; struct T *q; } *tp;
3: s1.p1 = &s2;
4: s1.p2 = &s3;
5: tp = (struct T*)&s1;
6: int n = tp->i + 4;
7: struct T *tp2 = (struct T*)n;
```

A pointer-to-pointer conversion on `s1` at Line 5 causes the change of layout of `s1` (Observation 1). This causes a pointer-to-integer conversion on the content of `s1.p1` and a pointer-to-pointer conversion on the content of `s1.p2` (Observation 2). The former adds all locations of `s2` to the set from which the result of integer-to-pointer conversions is determined (Observation 3). The latter causes the change of layout of `s3`, which causes further conversions through the diagram in Figure 1. The integer-to-pointer conversion at Line 7 creates the location of `s2.p2` from an integer `n`, which is justified by the above pointer-to-integer conversion on `s2` (Observation 4).

Now we explain the strategy of our analysis based on these observations. At each execution state s, we call the set of locations which belong to a block whose layout may be changed as $VLoc\,s$, and the set of locations which may be allowed to be the result of integer-to-pointer conversion as $ULoc\,s$. They may change during execution and thus are parameterized by a state. We usually omit a state parameter and just write $VLoc$ and $ULoc$.

Our basic idea is simply to keep field-sensitivity for blocks which do not belong to $VLoc$. If a block does not belong to $VLoc$, we can precisely model the layout of the block by its declared type. We also approximate all integer values by $ULoc$.

As $VLoc$, $ULoc$ and points-to information are mutually dependent (See Figure 1), our analysis does all calculations simultaneously. The calculations of $VLoc$ and $ULoc$ can be reduced to the reachability problem, as we will show in the next section.

3. 1 Formal Definitions of $VLoc$ and $ULoc$

To construct the analysis of $VLoc$ and $ULoc$, we characterize these sets by a kind of reachability predicates. The seeds of the reachability are the sets of locations on which certain type conversions are performed directly. Let C_{pp} (C_{ip}) be the set of locations on which pointer-to-pointer (pointer-to-integer) conversions are performed directly,

From the observations in Section 3.1, we can see:

- If l belongs to C_{pp}, then l and all locations of the same block of l belong to $VLoc$ (Observation 1);
- If l_1 belongs to $VLoc$, and the content of l_1 is l_2, then l_2 belongs to $VLoc$ and $ULoc$ (Observation 2);
- If l belongs to C_{ip}, then l_2 belongs to $VLoc$ (Observation 3);
- If l belongs to C_{ip}, then l and all locations of the same block of l belong to $ULoc$ (Observation 4)

We use three predicates to formalize $VLoc$ and $ULoc$. The predicates $sameblock$, $reach$ and $mreach$ are low-level variants of reachability predicates used in shape analysis[17]. Note that $reach\,s\,l_1\,l_2$ means l_2 is reachable from l_1 through memory indirection.

By using these predicates, we can formally define $VLoc$ and $ULoc$ as below:

$$
\begin{aligned}
VLoc\,s &= \{l \mid \exists l' \in C_{pp}\,s \cup C_{ip}\,s.\,reach\,s\,l'\,l\} \\
ULoc\,s &= \{l \mid \exists l' \in C_{ip}\,s.\,sameblock\,s\,l\,l'\} \\
&\quad \cup \{l \mid \exists l' \in VLoc\,s.\,mreach\,s\,l'\,l\} \cup ELoc
\end{aligned}
$$

Roughly, $VLoc\,s$ is the set of locations $reach$-able from locations converted to another pointer type, and $ULoc\,s$ is the set of locations converted to integer, or $mreach$-able from locations of blocks whose layout may be changed. We also include $ELoc$ in $ULoc$ as it is the predetermined set of locations allowed to be created from system-specific integer values.

$VLoc$ and $ULoc$ are upper-approximations of the semantics in the following sense:

1. Every location which is accessed as a type different from that of allocation time is included in $VLoc$;
2. Every location which satisfies the language restriction of Definition 1 is included in $ULoc$.

The correctness of these properties is obvious from the definition of $VLoc$ and $ULoc$.

3. 2 Abstract Locations

As the set of concrete locations is infinite, we need to have abstract representation (called *abstract location*) for them. The choice determines the solution space, and thus has a great impact on the precision and efficiency of the analysis. This is where the field-sensitivity emerges.

Basically, we use the conventional base/offset representation[20]. For a location l, we denote its abstraction as $\bar{l}$. The base part of $\bar{l}$ abstracts the block of l, and the offset part of $\bar{l}$ abstracts the byte offset of l from the top of the block. This provides a uniform representation under arbitrary type conversions and internal pointers.

However, we do not want to allow arbitrary offsets inside a block in an abstract location like previous works[19], because it makes the analysis complicated and slower, and causes *positive weight cycle* problem[16]. Instead, we exploit the declared type (in RML, the declared type means the type at allocation) and only use offsets of fields in the declared type of the block. This seems dangerous as there may be accesses to a location whose offset does not correspond to any field in the declared type by the use of casting. However, we can handle it safely by keeping field-sensitivity only for blocks which are definitely not in $VLoc$.

$$\frac{}{typeof\ T_N\ t\ 0\ t} \qquad \frac{\begin{array}{c} typeof\ T_N\ t\ n\ (str\ [t_0,\ldots,t_m]) \\ offsetof\ T_N\ (str\ [t_0,\ldots,t_m])i=k \end{array}}{typeof\ T_N\ t\ (n+k)\ t_i}$$

$$\frac{typeof\ T_N\ t\ n\ (arr\ t_a\ n_a)}{typeof\ T_N\ t\ n\ t_a} \qquad \frac{typeof\ T_N\ t\ n\ (name\ x)}{typeof\ T_N\ t\ n\ (T_N\ x)}$$

Figure 2. Typing Rules

$$(A)\ \frac{I\ c=alloc\ r}{(c,0)\in R\ s\ r} \qquad (C)\ \frac{\begin{array}{c} I\ c=copy\ r_1\ r_2 \\ isptr\ (T_R\ r_1) \\ \bar{l}\in\bar{R}\ \bar{s}\ r_2 \end{array}}{\bar{l}\in\bar{R}\ \bar{s}\ r_1}$$

$$(I)\ \frac{\begin{array}{c} I\ c=index\ r_1\ r_2\ r_3 \\ \bar{l}\in\bar{R}\ \bar{s}\ r_2 \end{array}}{\bar{l}\in\bar{R}\ \bar{s}\ r_1} \qquad (L)\ \frac{\begin{array}{c} I\ c=load\ r_1\ r_2 \\ isptr\ (T_R\ r_1) \\ \bar{l_1}\in\bar{R}\ \bar{s}\ r_2 \\ \bar{l_2}\in\bar{M}\ \bar{s}\ \bar{l_1} \end{array}}{\bar{l}\in\bar{R}\ \bar{s}\ r_1}$$

$$(S)\ \frac{\begin{array}{c} I\ c=store\ r_1\ r_2 \\ isptr\ (T_R\ r_2) \\ \bar{l_1}\in\bar{R}\ \bar{s}\ r_1 \\ \bar{l_2}\in\bar{R}\ \bar{s}\ r_2 \end{array}}{\bar{l_2}\in\bar{M}\ \bar{s}\ \bar{l_1}} \qquad (F)\ \frac{\begin{array}{c} I\ c=field\ r_1\ n\ r_2 \\ T_R\ r_1=ptr\ t \\ offset\ T_N\ t\ n=k \\ (c',o)\in\bar{R}\ \bar{s}\ r_2 \\ (c',o+k)\in\overline{Loc} \end{array}}{(c',o+k)\in\bar{R}\ \bar{s}\ r_1}$$

Figure 3. Analysis Rules: Core

Abstract Offsets For a block of type t_o, we define the set of abstract offsets as below:

$$\overline{Offset}\ T_N\ t_o = \{n \in Nat | \exists t.\ typeof\ T_N\ t_o\ n\ t,\ primitive\ t\}$$

where T_N is a typename environment for a program, and $primitive\ t$ holds iff t is integer, pointer, or union type.

The predicate $typeof$ is inductively defined by the rules in Figure 2. It determines the type of each offset from outer to inner. This matches the computation order of offset calculation for successive field references, and makes the proof easier. $offsetof$ is similar to offsetof macro in C, except for taking a type name environment as an extra argument to resolve type names.

Note that there is no rule to go inside union. We regard union as atomic and treat field-insensitively. Also, note that offsets of all elements of an array are represented as the offset of the first element of the array. We do not treat an array as atomic, and if the type of elements is a structure, we distinguish each field inside the structure. This makes the analysis more precise, but as we will show in Section 4, if this is applied unrestrictedly, it may make the analysis unsound. We avoid this problem by treating locations in $VLoc$ field-insensitively as shown later.

Abstract Blocks Basically, we abstract blocks created during the execution by its allocated location $c \in CLoc$. For the block for $ELoc$, the set of a system-specific constant locations, we abstract the block as 0, which is excluded from $CLoc$

Abstract Locations For a program $P = (T_N, T_R, I, N, entry)$, the set of abstract locations $\overline{Loc}(P)$ can be defined as follows:

$$\{(c,n) | I\ c = alloc\ r, n \in \overline{Offset}\ T_N\ (T_R\ r)\} \cup \{(0,0)\}$$

We use $(0,0)$ to abstract all locations in $ELoc$. $\overline{Loc}(P)$ is the fixed set of abstract locations used for the analysis. For simplicity, we usually abbreviate $\overline{Loc}(P)$ as $\overline{Loc}$.

For the lack of space, we do not show the formal definition of the abstraction function between concrete and abstract locations. For *l* not contained *VLoc*, it can be defined naturally based on the above descriptions for abstract locations and blocks. For $l \in VLoc$, we abstract it field-insensitively to (*b*, 0) where b is the abstract block of *l*. We lift the abstraction function to the set; for a set of locations *X*, $\bar{X} = \cup \{\bar{l} | l \in X\}$.

3. 3 Algorithm

Now we give the algorithm for our pointer analysis. It is Andersen-style[1], that is, flow-insensitive (ignores all control flows) and subset-based (allows a pointer to have multiple targets).

The solution of the analysis is the abstract state $\bar{s}$ which summarizes all possible states in executions of the program. $\bar{s}$ consists of $\bar{R}, \bar{M}, \overline{C_{pp}}, \overline{C_{ip}}, \overline{ULoc}$, and $\overline{VLoc}$. We represent the component X of $\bar{s}$ by $X\,\bar{s}$. The meaning of each components is self-explanatory.

For a program $P = (T_N, T_R, I, N, entry)$, we formalize our analysis as a deductive system *DS*, which is a set of rules to deduce constraints of each components of $\bar{s}$ (Figures 3 and 4).

$$(PP)\ \frac{\begin{array}{c} I\ c{=}conv\ r_1\ r_2 \\ isptr\ (T_R\ r_1) \\ isptr\ (T_R\ r_2) \\ \bar{l}{\in}\bar{R}\ \bar{s}\ r_2 \end{array}}{\bar{l}{\in}\bar{R}\ \bar{s}\ r_1 \quad \bar{l}{\in}\overline{C_{pp}}\ \bar{s}} \qquad (IP)\ \frac{\begin{array}{c} I\ c{=}conv\ r_1\ r_2 \\ \neg isptr\ (T_R\ r_1) \\ isptr\ (T_R\ r_2) \\ \bar{l}{\in}\bar{R}\ \bar{s}\ r_2 \end{array}}{\bar{l}{\in}\overline{C_{\iota p}}\ \bar{s}}$$

$$(PI)\ \frac{\begin{array}{c} I\ c{=}conv\ r_1\ r_2 \\ isptr\ (T_R\ r_1) \\ \neg isptr\ (T_R\ r_2) \\ \bar{l}{\in}\overline{ULoc}\ \bar{s} \end{array}}{\bar{l}{\in}\bar{R}\ \bar{s}\ r_1} \qquad (V1)\ \frac{\bar{l}{\in}\overline{C_{pp}}\ \bar{s} \cup \overline{C_{\iota p}}\ \bar{s}}{\bar{l}{\in}\overline{VLoc}\ \bar{s}}$$

$$(V2)\ \frac{\begin{array}{c} I\ c{=}alloc\ r \\ T_R\ r{=}ptr\ t \\ typeof\ T_N\ t\ n\ (uni\ ts) \end{array}}{(c,n){\in}\overline{C_{pp}}} \qquad (V3)\ \frac{\begin{array}{c} \bar{l_1}{\in}\overline{VLoc}\ \bar{s} \\ \overline{reach}\ \bar{s}\ \bar{l_1}\ \bar{l_2} \end{array}}{\bar{l_2}{\in}\overline{VLoc}\ \bar{s}}$$

$$(U1)\ \frac{\bar{l}{\in}\ \overline{C_{\iota p}}\ \bar{s}}{\bar{l}{\in}\overline{ULoc}\ \bar{s}} \qquad (U2)\ \frac{}{(0,0){\in}\overline{ULoc}\ \bar{s}}$$

$$(U3)\ \frac{\begin{array}{c} \bar{l_1}{\in}\overline{ULoc}\ \bar{s} \\ \overline{sameblock}\ \bar{s}\ \bar{l_1}\ \bar{l_2} \end{array}}{\bar{l_2}{\in}\overline{ULoc}\ \bar{s}} \qquad (U4)\ \frac{\begin{array}{c} \bar{l_1}{\in}\overline{VLoc}\ \bar{s} \\ \overline{mreach}\ \bar{s}\ \bar{l_1}\ \bar{l_2} \end{array}}{\bar{l_2}{\in}\overline{ULoc}\ \bar{s}}$$

$$(M)\ \frac{\begin{array}{c} \bar{l_1}{\in}\overline{VLoc}\ \bar{s} \\ \bar{l_2}{\in}\overline{ULoc}\ \bar{s} \end{array}}{\bar{l_2}{\in}\bar{M}\ \bar{s}\ \bar{l_1}} \qquad (R)\ \frac{\begin{array}{c} (c,n){\in}\overline{VLoc}\ \bar{s} \\ (c,n){\in}\bar{R}\ \bar{s}\ r \end{array}}{(c,0){\in}\bar{R}\ \bar{s}\ r}$$

Figure 4. Analysis Rules: Extension

The solution is the least $\bar{s}$ closed under *DS*. As *DS* is monotone and we fix the set of abstract location for the target program, the termination is obvious.

Figure 3 shows the rules for analyzing programs which do not use casts, union types, and constant locations. It is straightforward and needs no explanation

Figure 4 shows the rules for handling all complications described earlier. Here we use the predicates $\overline{sameblock}$, $\overline{reach}$ and $\overline{mreach}$, which abstract $sameblock$, $reach$, and $mreach$ in an intuitive way.

Rules *PP*, *IP*, and *PI* detects three kinds of type conversions caused by *conv*. Rules *V1*, *V2*, *V3*, *U1*, *U2*, *U3*, *U4* are straightforward abstractions of $VLoc$ and $ULoc$. Rule *V2* treats all locations of union type as statically converted to another pointer type. Rule *U2* includes (0,0), that is, the abstract location for $ELoc$ into $\overline{ULoc}$. Rule *M* reflects that all locations in $VLoc$ may have arbitrary integer value by the change of layout of the block. Rule *R* compensate the imprecision of offset calculations on locations in $\overline{VLoc}$ by adjusting to 0 , which is the representative offset of all locations in $\overline{VLoc}$.

3. 4 Soundness

We use the following predicates to prove the soundness of analysis.

$$INV_R\ s\ \bar{s} = \forall r.\ isptr(T_R\ r) \Rightarrow \overline{R\ s\ r} \in \bar{R}\ \bar{s}\ \bar{r}.$$
$$INV_M\ s\ \bar{s} = \forall l \in L\ s.\ isptr(T_M\ s\ l) \Rightarrow$$
$$\overline{M\ s\ l\ (T_M\ s\ l)} \in \bar{M}\ \bar{s}\ \bar{l}.$$
$$INV_{C_{pp}}\ s\ \bar{s} = \overline{C_{pp}\ s} \subseteq \overline{C_{pp}}\ \bar{s}.$$
$$INV_{C_{\iota p}}\ s\ \bar{s} = \overline{C_{\iota p}\ s} \subseteq \overline{C_{\iota p}}\ \bar{s}.$$
$$INV_r\ s\ \bar{s} = \forall l_1, l_2 \in L\ s. reach\ s\ l_1\ l_2 \Rightarrow \overline{reach}\ \bar{s}\ \bar{l_1}\ \bar{l_2}.$$
$$INV_{mr}\ s\ \bar{s} = \forall l_1, l_2 \in L\ s. mreach\ s\ l_1\ l_2$$
$$\Rightarrow \overline{mreach}\ \bar{s}\ \bar{l_1}\ \bar{l_2}.$$
$$INV_V\ s\ \bar{s} = \overline{VLoc\ s} \subseteq \overline{VLoc}\ \bar{s}.$$
$$INV_U\ s\ \bar{s} = \overline{ULoc\ s} \subseteq \overline{ULoc}\ \bar{s}.$$

Each invariant INV_X, represents that the component $\bar{X}$ of $\bar{s}$ is an upper-approximation of the corresponding component X of all possible states with respect to the abstraction function. We denote the conjunction of all invariants as $INV\ s\ \bar{s}$. Thanks to our formal model of memory, we can prove the soundness in a standard way.

Lemma 1 (Initialization). Let $\bar{s}$ be the solution of the analysis for a program P. For every initial state $s \in StateInit(P)$, $INV\ s\ \bar{s}$ hold.

Lemma 2 (Preservation). Let $\bar{s}$ be the solution of the analysis for a program P. For any states $s_1, s_2 \in State(P)$, if $INV\ s_1\ \bar{s}$holds and $s_1 \rightarrow s_2$, then $INV\ s_2\ \bar{s}$ holds.

Proof. We can prove that INV_r, INV_{mr}, INV_V, and INV_U,are derived from other invariants. For INV_R, INV_M, INV_{cpp}, and INV_{cip}, we can prove by case analysis of instructions. ■

Theorem 3 (Soundness). Let $\bar{s}$ be the solution of the analysis for a program P. For every state $s \in State(P)$, $INV\ s\ \bar{s}$ holds.

Proof. By induction on the length of the execution of P with Lemma 1 and Lemma 2. ■

4. Related Work

Many field-sensitive analysis cannot handle casting nor union[1,5,7,16], and give unsound solution for non strictly-conforming programs. For example, Cheng et al. model the layout of object by base and access paths[5]. Roughly, an access path is a list of field references represented as a pair (offset, size). They ignore array subscriptions completely. This implicitly means that all elements of an array are abstracted to the first element of the array. This gives unsound result in the presence of casting or union. For example, consider the following union.

```
union U {
  struct S { int *f1,*f2,*f3,*f4; } s;
  struct T { int *g1,*g2; } a[2];
}
```

While `u.s.f3` and `u.a[1].g1` refers to the same location, their access paths are (8,4) and (0,4), which do not overlap.

The seminal work by Yong et al.[20] is most closely related to our goal. They give four algorithms for modeling the layout of object under the presence of pointer casting. Our work has three advancements from their work. First, all of their algorithms treat an array as atomic. This means that they cannot distinguish fields inside an array of structure; for example, `a[0].f1` and `a[0].f2` are considered the same while our algorithm can distinguish them as long as they are not in $\overline{VLoc}$. This also makes the layout of object complicated. As they commented in footnote, their $lookup$ function in offset-approach cannot handle an array of structure safely. Second, their formalization does not include union types. As our analysis shows, union type needs a special treatment in many cases and to incorporate it into the formal model is not trivial. Third, their algorithms cannot handle integer/pointer conversion soundly. As their algorithms and ours are both flow-insensitive, subset-based, and context-insensitive, this performance loss may well be caused by the way of handling the layout of object. The algorithm given in Steendsgaard is similar to Yong's common initial sequence approach, but less precise[18].

Wilson et al. use the triple of (base, offset, stride) to model the layout of object[19]. Roughly, a triple (b,o,s) represents the set of locations $\{b + o + ks | k \in Nat\}$. They completely ignore the declared type of object. This means that they cannot use the constraints of the language specification such as referring to a location out of array is undefined behaviour. If there is a field of an array in a structure, it imprecisely assumes that the array is expanded to the whole structure. Calculation of a triple is complex and their algorithm is slow. Also, Their algorithm cannot handle integer/pointer conversion soundly.

Chandra et al. investigates the usage pattern of casting[3]. They proposes the concept of physical subtyping, which is a subtyping relation with respect to the layout of object. We can improve the precision of our work by incorporating physical subtyping to ignore ``upcast'' which does not affect the layout of object.

Lattner proposes a data structure analysis which checks the data structures of lhs and rhs in each assignments and if they are incompatible, collapses them and gives up field-sensitivity[11]. This is similar to *collapse-on-cast* approach in Yong, but less precise as their analysis is unification-based, that is, multiple targets of a pointer should be merged. Such collapsing merges pointer fields and integer fields into the one and is unsound in the presence of integer/pointer conversion.

Necula et al. propose CCured, which is a program transformation system that adds memory safety guarantees to C programs[15]. Though CCured is not pointer analysis, their work has some similarity with ours in doing some classification about memory usage. They classify pointers, while we classify locations. Their approach is less precise than ours. If a pointer p points to a non-well-typed location, all other locations pointed by p are imprecisely considered to be a non-well-typed.

There are several works that develop the reachability predicates on low-level memory model such as an array of byte[2,4,8]. These predicates are designed for specifying the properties of linked data structures, and more expressive than ours. On the other hand, they cannot handle the integer/pointer conversion. Our analysis is less precise, but efficient and fully automatic.

We implemented our analysis based on the set-based constraint system. It can also be implemented efficiently by BDD-based bottom-up solver by preparing all kinds of offset calculations by predicates[13], that is possible as our abstract locations are finite and fixed.

5. Conclusion

We gave a field-sensitive pointer analysis for C with integer/pointer conversion. Based on our formal model of memory, we could reduce all the intricacies of arbitrary type conversions to the simple reachability problem. Our analysis was derived as a natural abstraction of the semantics, and easily shown to be sound. As our analysis can handle all low-level features of C, and has no assumption on the implementation-dependent features, it is directly applicable to many compilers. Also, analyses just for type-safe subset of C can be combined with our analysis and extended to handle full C. We hope that our work releases compiler developers from the struggle with subtle bugs caused by the combination of field-sensitivity and arbitrary type conversions.

References

1) Andersen, L. O.: Program Analysis and Specialization for the C Programming Languages, Ph.D. thesis, DIKU, University of Copenhagen, 1994.
2) Calcagno, C., Distefano, D., O'Hearn, P. W. and Yang, H.: Beyond Reachability: Shape Abstraction In the Presence of Pointer Arithmetics, SAS, 2006.
3) Chandra, S. and Reps, T. W.: Physical Type Checking for C, PASTE, 1999.
4) Chatterjee, S., Lahiri, S. K., Qadeer, S. and Rakamaric, Z.: A Reachability Predicate for Analyzing Low-Level Software, TACAS, 2007.
5) Cheng, B.-C. and Hwu, W.-M. W.: Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation, PLDI, 2000.
6) Das, M.: Static Analysis of Large Programs: Some Experiences, PEPM, 2000.
7) Emami, M., Ghiya, R. and Hendren, L.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers, PLDI, 1994.
8) Gulwani, S. and Tiwari, A.: An Abstract Domain for Analyzing Heap-Manipulating Low-Level Software, CAV, 2007.
9) Heintze, N. and Tardieu, O.: Ultra-fast Aliasing Analysis Using CLA: a Million Lines of C Code in a Second, PLDI, 2001.
10) Hind, M.: Pointer Analysis: Haven't We Solved This Problem Yet?, PASTE, 2001.
11) Lattner, C.: Macroscopic Data Structure Analysis and Optimization, Ph.D. thesis, Computer Science Department, University of Illinois at Urbana-Campaign, 2005.
12) Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, CGO, 2004.
13) Lhotak, O. and Hendren, L.: Jedd: a BDD-based Relational Extension of Java, PLDI, 2004.
14) Mine, A.: Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics, LCTES, 2006.
15) Necula, G. C., McPeak, S. and Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Code, POPL, 2002.
16) Pearce, D. J., Kelly, P. H. J. and Hankin, C.: Efficient Field-Sensitive Pointer Analysis for C, PASTE, 2004.
17) Sagiv, M., Reps, T. and Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic, TOPLAS, 2002.
18) Steensgaard, B.: Points-to Analysis by Type Inference of Programs with Structures and Unions, Computational Complexity, 1996.
19) Wilson, R. P. and Lam, M. S.: Efficient Context-Sensitive Pointer Analysis for C Programs, PLDI, 1995.
20) Yong, S. H., Horwitz, S. and Reps, T. W.: Pointer Analysis for Programs with Structures and Casting, PLDI, 1999