Efficient Field-Sensitive Pointer Analysis for C

David J. Pearce
Department of Computing,
Imperial College, London,
SW7 2BZ, UK
djp1@doc.ic.ac.uk

Paul H. J. Kelly
Department of Computing,
Imperial College, London,
SW7 2BZ, UK
phjk@doc.ic.ac.uk

Chris Hankin
Department of Computing,
Imperial College, London,
SW7 2BZ, UK
clh@doc.ic.ac.uk

ABSTRACT

The subject of this paper is flow- and context-insensitive pointer analysis. We present a novel approach for precisely modelling `struct` variables and indirect function calls. Our method emphasises efficiency and simplicity and extends the language of set-constraints. We experimentally evaluate the precision cost trade-off using a benchmark suite of 7 common C programs between 5,000 to 150,000 lines of code. Our results indicate the field-sensitive analysis is more expensive to compute, but yields significantly better precision.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program Analysis

General Terms

Algorithms, Theory, Languages, Verification

Keywords

Set-Constraints, Pointer Analysis

1. INTRODUCTION

Pointer analysis is the problem of statically determining the runtime targets of pointer variables in a program. We say that a solution is *sound* if the inferred target set for each variable contains all actual runtime targets for that variable. A solution is *imprecise* if, for any variable, the inferred target set is larger than necessary. Thus, the most imprecise but sound solution has each variable pointing to every other. Obtaining a perfect (i.e. flow- and context-sensitive) solution, however, is undecidable in general [14] and, in practice, obtaining even relatively imprecise information (i.e. flow- and context-insensitive) is expensive [13]. The main contributions of this paper are:

1. A small extension to the language of set-constraints, which elegantly formalises a field-sensitive pointer analysis for the C language. As a byproduct, function pointers are supported for free with this mechanism.
2. The largest experimental investigation to date into the trade-offs in time and precision of field-insensitive and -sensitive analyses for C.

In some sense, our formulation can be regarded as an instance of the general framework for field-sensitive pointer analysis by Yong *et al.* [28]. In particular, we argue it is equivalent to the most precise, yet portable analysis their system can describe. However, this work goes beyond their initial treatment by considering efficient implementation and some important algorithmic issues which were not addressed.

2. CONSTRAINT-BASED ANALYSIS

Set-constraints [1] were first used by Andersen for performing pointer analysis [2]. Since then, this approach has become popular and, recently, was shown capable of analysing million line programs [10, 15]. We use the following set-constraint language to formulate our pointer analysis:

$$p \supseteq q \mid p \supseteq \{q\} \mid p \supseteq *q \mid \; *p \supseteq q \mid \; *p \supseteq \{q\}$$

Where p and q are constraint variables and $*$ is the dereference operator. We can think of each variable as containing the set it points to. Thus, $p \supseteq \{x\}$ states that *p points to x*. Note, constraints involving "$*$" are called *complex*. To perform the analysis we first translate the source program into this language, by mapping each variable to a unique constraint variable and converting assignments to constraints. Consider the following, where the solution (shown below the line) is obtained using the rules of Figure 1:

```
int *f(int *p){return p;} (1) f_* ⊇ f_p
int g() { int x,y,*p,*q,**r,**s;
 s=&p;                    (2)  g_s ⊇ {g_p}
 if(...) p=&x;            (3)  g_p ⊇ {g_x}
 else p=&y;               (4)  g_p ⊇ {g_y}
 r=s;                     (5)  g_r ⊇ g_s
 q=f(*r);                 (6)  f_p ⊇ *g_r
}                         (7)  g_q ⊇ f_*
```

(8)	$g_r \supseteq \{g_p\}$	$(trans, 2+5)$
(9)	$f_p \supseteq g_p$	$(deref_2, 6+8)$
(10)	$f_p \supseteq \{g_x\}$	$(trans, 3+9)$
(11)	$f_p \supseteq \{g_y\}$	$(trans, 4+9)$
(12)	$f_* \supseteq \{g_x\}$	$(trans, 1+10)$
(13)	$f_* \supseteq \{g_y\}$	$(trans, 1+11)$
(14)	$g_q \supseteq \{g_x\}$	$(trans, 7+12)$
(15)	$g_q \supseteq \{g_y\}$	$(trans, 7+13)$

Variable names are augmented with scope information to ensure uniqueness. Also, f_* represents the return value of f.

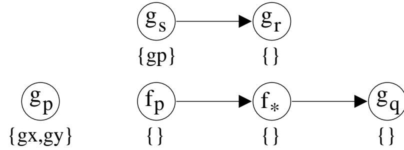
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASTE'04, June 7-8, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-910-1/04/0006 ...$5.00.

$$\frac{\tau_1 \supseteq \{\tau_2\} \quad \tau_3 \supseteq \tau_1}{\tau_3 \supseteq \{\tau_2\}} \; [trans] \qquad \frac{\tau_1 \supseteq *\tau_2 \quad \tau_2 \supseteq \{\tau_3\}}{\tau_1 \supseteq \tau_3} \; [deref_1] \qquad \frac{*\tau_1 \supseteq \tau_2 \quad \tau_1 \supseteq \{\tau_3\}}{\tau_3 \supseteq \tau_2} \; [deref_2] \qquad \frac{*\tau_1 \supseteq \{\tau_2\} \quad \tau_1 \supseteq \{\tau_3\}}{\tau_3 \supseteq \{\tau_2\}} \; [deref_3]$$

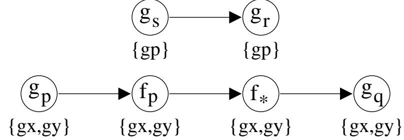
Figure 1: An inference system for pointer analysis

From the derivation, we can build the target set of a variable v from constraints of the form $v \supseteq \{x\}$. Thus, constraints 14+15 give $q = \{g_x, g_y\}$, meaning q may point to g_x or g_y anywhere in the program. To ensure the solution is sound, we must derive all facts. One feature of the system is that control-flow and calling context are ignored. This is called *flow-* and *context-insensitivity* and causes imprecision. But, without these simplifications, our analysis would not scale to large programs.

To perform the analysis efficiently we use a directed graph, where each variable is represented by a unique node and each constraint $p \supseteq q$ by an edge $p \leftarrow q$. For each node n we also have a solution set, $Sol(n)$, initialised from all constraints of the form $n \supseteq \{x\}$. So, for the previous example, we have:



Here, each solution is placed below its node. The graph is then solved by propagating the solution of each node into all reachable successors. During this, complex constraints involving n are evaluated as $Sol(n)$ changes. For a constraint $*n \supseteq q$, this is done by adding an edge $v \leftarrow q, \forall v \in Sol(n)$. The final graph looks like:



Notice the new edge, caused by constraint 6 ($f_p \supseteq *g_r$). Various techniques for speeding up this computation have been proposed in the literature and the reader is referred to [19, 10, 15, 7, 24, 20]. One important consideration is that, for efficiency reasons, it is desirable to implement $Sol(n)$ as an integer set. This permits the use of data structures supporting efficient set union, such as bit vectors or sorted arrays, and is achieved by indexing each variable.

There are two limitations to the system described so far: *it cannot handle aggregates or function pointers*. For the former, three approaches are common: *field-insensitive*, where field information is discarded by modelling an aggregate with a single constraint variable; *field-based*, where all instances of a particular field are modelled with one variable; *field-sensitive*, where each instance of a field is modelled with a separate variable. The following clarifies this:

`typedef struct { int *f1; int *f2; } aggr;`

	(insensitive)	(based)	(sensitive)
`aggr a,b;`			
`int *c,d,e,f;`			
`a.f1 = &d;`	$a \supseteq \{d\}$	$f1 \supseteq \{d\}$	$a_{f1} \supseteq \{d\}$
`a.f2 = &f;`	$a \supseteq \{f\}$	$f2 \supseteq \{f\}$	$a_{f2} \supseteq \{f\}$
`b.f1 = &e;`	$b \supseteq \{e\}$	$f1 \supseteq \{e\}$	$b_{f1} \supseteq \{e\}$
`c = a.f1;`	$c \supseteq a$	$c \supseteq f1$	$c \supseteq a_{f1}$

The field-insensitive and field-based solutions are imprecise in different ways, with their relative precision depending upon the program in question. The focus of this paper is in extending our system to be field-sensitive. In fact, there exist several field-sensitive pointer analyses for Java, which are formulated using set constraints [21, 15, 17, 26]. However, Java presents a simpler problem and we must go beyond these to achieve our goal.

In the literature, function pointers are either dealt with in ad hoc ways (e.g. [10, 15]) or through a special *lam* constructor (e.g. [9, 8]). The latter gives something like:

`int f(int *p) { return p; }`	$f_* \supseteq f_p$
`int (*p)(int*) = &f`	$p \supseteq \{\ lam_f(f_p)\ \}$
`int *q = ...`	$q \supseteq \{\ \ldots\ \}$
`p(q)`	$*p(q)$

with a corresponding rule for function application:

$$\frac{*p(\tau_1, \ldots, \tau_n) \qquad p \supseteq \{\ lam_v(v_1, \ldots, v_n)\ \}}{\forall 1 \leq i \leq n.\ v_i \supseteq \tau_i}$$

The main issue here is the implementation of *lam*. Certainly, we don't wish to sacrifice the ability to implement solutions as integer sets. One approach is to place the *lam* constructs into a table, so they are identified by index. Thus, if care is taken to avoid clashes with the variable identifiers, the two element types can co-exist in the same solution set. However, this is inelegant as we must litter our algorithm with special type checks. For example, when dealing with $*p \supseteq q$, we must check for *lam* values in $Sol(p)$. In the next Section, we present a simple alternative, which forms part of our field-sensitive formulation — meaning field-sensitive analyses can model function pointers for free.

3. EXTENDING THE BASIC MODEL

The main observation behind our method is that, since variables are identified by integers, we can reference one as an offset from another. Thus, we introduce the following forms:

$$p \supseteq *(q+k) \mid *(p+k) \supseteq q \mid *(p+k) \supseteq \{q\}$$

Here k is an arbitrary constant and $*(p+k)$ means "load p into a temporary set, add k to each element and dereference as before". When $k=0$, these forms are equivalent to those of the original language. The corresponding inference rules are given in Figure 2, where *idx* maps variables to their index. Now, suppose in our source program there is some function f, accepting x parameters. If the address of f has been taken, we create a block of x consecutively indexed variables, where the first represents the first parameter of f and so on. Thus, we model the address of f using the first index in the block, allowing us to reference the other parameters as an offset. We can also model return values using this mechanism by allocating another variable after the last parameter. Thus, we can determine the offset of the return value from the type of the function pointer being dereferenced. The following aims to clarify this:

$$\frac{\tau_1 \supseteq *(\tau_2{+}k) \quad \tau_2 \supseteq \{\tau_3\} \quad idx(\tau_4) = idx(\tau_3){+}k}{\tau_1 \supseteq \tau_4}\ [deref_4]$$

$$\frac{*(\tau_1{+}k) \supseteq \tau_2 \quad \tau_1 \supseteq \{\tau_3\} \quad idx(\tau_4) = idx(\tau_3){+}k}{\tau_4 \supseteq \tau_2}\ [deref_5]$$

$$\frac{*(\tau_1{+}k) \supseteq \{\tau_2\} \quad \tau_1 \supseteq \{\tau_3\} \quad idx(\tau_4) = idx(\tau_3){+}k}{\tau_4 \supseteq \{\tau_2\}}\ [deref_6]$$

$$\frac{\tau_1 \supseteq \tau_2{+}k \quad \tau_2 \supseteq \{\tau_3\} \quad idx(\tau_4) = idx(\tau_3){+}k}{\tau_1 \supseteq \{\tau_4\}}\ [add]$$

Figure 2: Extended inference rules. Note, *add* is only needed for the second half of Section 3

```
void f(int**p,int*q) (1,2) idx(f_p) = 0, idx(f_q) = 1
{ *p = q;              (3) *f_p ⊇ f_q
}
void g(...) { void (*p)(int**,int*);
 int *a,*b,c;        (4,5) idx(g_p)=2, idx(g_a)=3
                     (6,7) idx(g_b)=4, idx(g_c)=5
 p = &f;               (8) g_p ⊇ {f_p}
 b = &c;               (9) g_b ⊇ {g_c}
 p(&a,b);             (10) *(g_p+0) ⊇ {g_a}
}                     (11) *(g_p+1) ⊇ g_b
---------------------------------------------------
                      (12) f_p ⊇ {g_a}    (deref_6 8,10,1)
                      (13) f_q ⊇ g_b      (deref_5 1,2,8,11)
                      (14) f_q ⊇ {g_c}    (trans 9,13)
                      (15) g_a ⊇ f_q      (deref_2 3,12)
                      (16) g_a ⊇ {g_c}    (trans 14,15)
```

Thus, we see `&f` is translated as f_p — its first parameter. One difficulty is the use of invalid casts:

```
void f(int *p) { ... }                 idx(f_p)=0
int g(int *a,int *b) {                 idx(g_a)=1, idx(g_b)=2
 void (*p)(int*,int*);                 idx(g_p)=3
 p = (void(*)(int*,int*)) &f;          g_p ⊇ {f_p}
 *p(a,b);                              *(g_p+0) ⊇ g_a
}                                      *(g_p+1) ⊇ g_b
```

Here, $*(g_p{+}1) \supseteq b$ derives $g_a \supseteq b$ as $idx(g_a) = idx(f_p) + 1$. This is unfortunate, although it is unclear how to model the above anyway. To prevent such unwanted propagation we can extend our mechanism with *end*() information for each variable. This determines where the enclosing block of consecutively allocated variables ends and we only permit offsets which remain within this. For example, in the above, $end(f_p) = 0$ and $end(g_a) = end(g_b) = 2$ and we can identify the problem as $idx(*(g_p{+}1)) > end(*g_p)$.

We now consider how this system can be made field-sensitive, which we have already indicated is easier for Java than C. So, what is the difference? The answer is that, in C, we can take the address of a field. Indeed, it turns out the language of the previous Section is enough for field-sensitive analysis of Java. This is achieved by using blocks of constraint variables, as we did for functions, to represent aggregates. For example:

```
typedef struct { int *f1; int *f2; } aggr;
aggr a,*b;         idx(a_f1)=0, idx(a_f2)=1, idx(b)=2
int *p,**q,c;      idx(p)=3, idx(q)=4, idx(c)=5
b=&a               b ⊇ {a_f1}
b->f2=&c;          *(b+1) ⊇ {c}
p=b->f2;           p ⊇ *(b+1)
```

But, how can we translate "`q=&(b->f2);`"? The problem is that we want to load the *index* of a_{f2} into $Sol(q)$, but there is no mechanism for this. So, we extend the language to permit the translation: $q \supseteq b{+}1$, meaning *load b into a temporary, add 1 to each element and merge into q*. Note the inference rule in Figure 2. This form can be represented by turning the constraint graph into a weighted multigraph, where weight determines increment. However, this introduces the *Positive Weight Cycle (PWC)* problem:

```
aggr a,*p; void *q;
q=&a;                      q ⊇ {a}
p=q;                       p ⊇ q
q=&p->f2;                  q ⊇ p+1
/* now use q as int* */
```

This is legal and well-defined C code. The cycle arises from flow-insensitivity and, we argue, any such analysis must deal with this. Note, cycles can also arise from an imprecise model of the heap (see below). In general, the problem is that cycles describe infinite derivations. To overcome this, we use *end()* information, as with function pointers, so that a variable is only incremented within its enclosing block. Another problem with weighted edges is that *cycle elimination* is now unsafe. Eliminating cycles is a common optimisation for speeding up the analysis, which exploits the fact that nodes in a cycle have the same solution [19, 10, 7]. Thus, each cycle is replaced by a single representative — reducing the size and complexity of the graph. Returning to our problem, we observe that cycles can be collapsed if there is a zero weighted path between all nodes and intra-cycle weighted edges are preserved as self loops.

The heap is a further source of complication. One approach, used by most pointer analyses, is to model all objects returned by a particular call to `malloc` with one variable. This has some implications, highlighted in the following:

```
typedef struct {double d1; int *f2;} aggr1;
typedef struct {int   *f1; int *f3;} aggr2;
void *f(int s) {return malloc(s);}  f_* ⊇ {HEAP0}
void *g(int s) {return malloc(s);}  g_* ⊇ {HEAP1}
aggr1 *p = f(sizeof(aggr1));        p ⊇ f_*
aggr2 *q = f(sizeof(aggr2));        q ⊇ f_*
int *x = f(100);                    x ⊇ f_*
int *y = g(100);                    y ⊇ g_*
```

The issue is that we cannot, in general, determine which heap variables will be used as aggregates. Indeed, the same variable can be used as both aggregate and scalar (e.g. *HEAP0* above). Thus, we either model heap variables field-insensitively or assume they can be treated as aggregates. Our choice is the latter, raising a further problem: *how many fields should each heap variable have?* A simple solution is to give them the same number as the largest `struct` in the program. Effectively, then, each heap variable is modelling the C `union` of all `struct`s. So, in the above, *HEAP0* and *HEAP1* both model `aggr1` and `aggr2` and are implemented with two constraint variables: the first representing fields `f1` and `d1`; the second `f2` and `f3`. The observant reader will have noticed something strange here: *the first constraint variable models fields of different sizes.* This seems a problem as, for example, writing to `d1` should invalidate `f1` and `f3`. In practice,

	Ver	LOC	Triv	Simp	Comp	Vars	# Heap	# PWC
make	3.79	16164	1427	4417	1557	4773 / 6920	69 / 1794	0
gawk	3.1.0	19598	2263	7797	2265	7288 / 10125	96 / 2496	0
bash	2.05	55324	3594	12076	2659	10831 / 13109	36 / 936	0
emacs	20.7	93151	11715	10437	5135	17961 / 38170	172 / 12900	0
sendmail	8.11.4	49053	5444	9595	2286	10218 / 12869	13 / 949	1
named	9.2.0	75599	17848	28972	24088	34649 / 47101	24 / 1704	1
gs	6.51	159853	21653	44030	36431	63568 / 100209	17 / 1887	2

Table 1: LOC measures non-comment, non-blank lines. Initial constraints are Trivial ($p \supseteq \{q\}$), Simple ($p \supseteq q$) and Complex (involving '*'). Total number of constraint variables is given in $\#Vars$, with $\#Heap$ showing number modelling the heap. For each, the two numbers are for the field-insensitive and -sensitive analyses respectively. Finally, "# PWC" counts positive weight cycles in the final graph (for the sensitive analysis).

however, this cannot be exploited without using undefined C, such as:

```
aggr1 *p = malloc(sizeof(aggr1));   idx(HEAP0_0)=0
                                    idx(HEAP0_1)=1
int a,*r;                           p ⊇ {HEAP0_0}
aggr2 *q = (aggr2 *) p;             q ⊇ p
q->f3 = &a;                         *(q+1) ⊇ {a}
p->f1 = 1.0; /* clobbers q->f3 */   *(p+0) ⊇ {?}
r = q->f3;                          r ⊇ *(q+1)
```

Here, our analysis unsoundly concludes that r only points to a. Note the special value "?", used to indicate that a pointer may target anything. In general, we are not concerned with this issue as our objective is to model portable C programs only. Finally, nested `struct`s are easily dealt with by "inlining" them into their enclosing `struct`, so that each nested field is modelled by a distinct constraint variable.

4. EXPERIMENTAL STUDY

We now present empirical data on a range of benchmarks comparing two example field-sensitive and -insensitive solvers. Figure 3 provides pseudo-code for the field-sensitive solver. The insensitive algorithm is similar, but operates on the simpler language of Figure 1. Notice that cycles are identified with Tarjan's algorithm [25] and not the partial online detector from [7]. In our experience, we have found this configuration to be highly efficient, not least because Tarjan's algorithm can topologically sort the graph for free. Our implementation also used bit vectors for the solution sets, applied the variable substitution methods of [20] and the (hash-based) duplicate set compaction scheme from [10].

To generate constraints, the SUIF 2.0 compiler was employed and a few points must be made about this: the approach of Section 3 was used for modelling the heap; string constants were all treated as one object; lastly, external function calls were modelled with hand crafted summary functions. Note, we were able to compile all the benchmarks with only superficial modifications, such as adding extra "`#include`" directives for missing standard library headers.

Finally, the experimental machine was a 900Mhz Athlon with 1Gb of main memory, running Redhat 8.0 (Psyche). The executables were compiled using gcc 3.2, with "`-O3`".

Table 1 provides information on our benchmarks, which are all open source and available online. In particular, we note the number of variables differs between the insensitive and sensitive analyses. This was expected as, in the latter, each aggregate is now modelled using several variables. In fact, there will be more constraints for similar reasons, although these are omitted for brevity as they are essentially

```
foreach y ∈ V do changed(y) = true;

while ∃y.changed(y) do
 collapse zero weight cycles with Tarjan's algorithm
 foreach n ∈ V in (weak) topological order do
  if changed(n) then
   changed(n) = false;
   // process complex constraints involving *n
   foreach c ∈ C(n) do case c of
   *(n + k) ⊇ w:
     foreach v ∈ Sol[n] do
      x = v + k;
      if x ≤ end(v) ∧ w → x ∉ E do
       E ∪= w → x;
       if Sol[w] ⊈ Sol[x] do
        Sol[x] ∪= Sol[w]; changed(x) = true;
   w ⊇ *(n + k):
     foreach v ∈ Sol[n] do
      x = v + k;
      if x ≤ end(v) ∧ x → w ∉ E do
       E ∪= x → w;
       if Sol[x] ⊈ Sol[w] do
        Sol[w] ∪= Sol[x]; changed(w) = true;
   *(n + k) ⊇ {w}:
     foreach v ∈ Sol[n] do
      x = v + k;
      if x ≤ end(v) ∧ w ∉ Sol[x] do
       Sol[x] ∪= {w}; changed(x) = true;
   // process outgoing edges from n
   foreach n -k-> w ∈ E do
    foreach v ∈ Sol[n] do
     x = v + k;
     if x ≤ end(v) do Sol[w] ∪= {x};
    if Sol[w] changed then changed(w) = true;
```

Figure 3: The Field-sensitive Pointer Analysis Algorithm. All scalar variables (e.g. n and x) have integer type and Sol is an array of integer sets, where $Sol[n]$ is initialised by constraints of the form $n \supseteq \{q\}$. $C(n)$ contains all constraints involving "$*n$". The algorithm consists of an outer loop which iterates until no change is observed. On each iteration, zero weighted cycles are collapsed and then all remaining are visited in topological order. To visit a node n, any complex constraints involving it are evaluated and then $Sol[n]$ is propagated along all outgoing edges. Note the use of end information to prevent infinite loops arising from positive weight cycles.

		Time / s	Avg Working Set Size	Avg Deref Size (N)	Dereference Sites (% of normalised total)						
					0	1	2	3-10	11-100	101-1000	1000+
make	fdi	0.07	10.3	336.7	5.7	7.3	0.77	5.0	16.0	66.0	0.0
	fds	0.17	7.0	17.2	5.9	20.0	6.3	4.2	64.0	0.0	0.0
gawk	fdi	0.12	28.7	633.9	6.2	7.8	6.3	4.3	27.0	0.93	48.0
	fds	0.3	15.3	22.4	7.1	24.0	21.0	7.0	41.0	0.0	0.0
bash	fdi	0.51	84.5	543.0	5.4	7.6	2.7	3.4	20.0	61.0	0.0
	fds	0.53	52.5	86.7	5.4	24.0	6.1	2.9	5.8	56.0	0.0
emacs	fdi	0.4	12.2	79.3	24.0	17.0	27.0	2.7	7.9	21.0	0.53
	fds	0.69	2.6	5.4	25.0	25.0	32.0	10.0	7.6	0.37	0.03
sendmail	fdi	0.49	58.7	558.4	3.5	17.0	1.8	4.8	8.0	65.0	0.0
	fds	2.05	106.5	214.2	4.6	23.0	3.3	4.2	4.4	60.0	0.0
named	fdi	30.0	570.5	2865.5	3.4	3.6	3.9	28.0	1.4	1.2	58.0
	fds	129.1	2042.9	2167.7	5.1	8.4	1.3	28.0	2.2	2.7	52.0
gs	fdi	277.4	1148.7	7703.1	32.0	2.7	1.2	3.1	3.8	1.7	56.0
	fds	2510.4	5977.0	7365.2	33.0	7.2	2.9	2.1	0.25	0.35	54.0

Table 2: Experimental data on the field-sensitive (fds) and field-insensitive (fdi) formulations of our analysis.

the same. This table also looks at the number of positive weight cycles in the final graph. It is important to realise the count may be higher during solving, as some cycles may end up being combined. Nevertheless, we believe this figure indicates that positive weight cycles are rare.

Table 2 looks at the effect on time and precision of using our field-sensitive analysis versus its insensitive counterpart. The data clearly shows that field-sensitivity is more expensive to compute. The average working set size gives the average size of the final solution sets computed by the algorithm. This figure gives insight into the cost of a set union operation during solving and, hence, we expected a correlation with execution time. Unfortunately, only three benchmarks appear to support this and one explanation might be that the greater accuracy of the sensitive analysis means there are fewer cycles to collapse. Furthermore, we note the number of positive weight cycles appears to impact upon the average working set size. "Avg Deref" reports the average set size at dereference sites. However, to facilitate a comparison (in terms of precision) between the two analyses we must normalise this value. To understand why, consider a pointer p which targets the first three fields of some `struct` a. For the insensitive analysis, we have the solution $p \supseteq \{a\}$, whilst the sensitive analysis gives $p \supseteq \{a_{f1}, a_{f2}, a_{f3}\}$. Thus, the latter seems less accurate since it is larger. However, this is misleading as the insensitive analysis actually concludes that p may point to *any field* of a. Therefore, we normalise the insensitive solution by counting each aggregate by the number of variables representing it in the sensitive formulation. We also break up the average deref figure to show its distribution. Note that zero sized sets arise from unreachable code, typically occurring when a function is linked with the program, but not actually called. The results are encouraging and show the field-sensitive analysis to give more precise results across the board. However, it appears the payoff decreases with program size. In particular, a large proportion of sets for the two largest benchmarks have a thousand elements or more. We believe the main reason for this trend can be attributed to the number of variables modelling the heap — which does not increase with program size. Thus, these variables will likely be modelling an increasingly large number of actual heap objects. In fact, the data for `emacs` appears to support this, since this has an unusually high number of heap variables and appears to have a much better distribution of sets than the others.

5. RELATED WORK

Flow- and context-insensitive pointer analysis has been studied extensively in the literature (see e.g. [19, 15, 7, 10, 20, 2, 23, 5]). These works can, for the most part, be placed into two camps: extensions of either Andersen's [2] or Steensgaard's [23] algorithm. The former use *inclusion constraints* (i.e. set-constraints) and are more precise but slower, while the latter adopt *unification systems* and sacrifice precision in favour of speed. Thus, new developments tend to be focused either on speeding up Andersen's algorithm (e.g. [10, 7, 20, 19]) or on improving the precision of Steensgaard's (e.g. [5, 6, 16]). Furthermore, there have been numerous studies on the relative precision of these two approaches (see e.g. [17, 9, 12, 22, 5, 6], with the results confirming that set-constraints offer useful improvements in precision. We refer the reader to [11] for a more thorough survey of pointer analysis.

For field-sensitive pointer analysis of C, there are several previous works (e.g. [27, 28, 3, 16]), although only two are for the flow- and context-insensitive setting. The first, due to Yong *et al.* [28], is a framework covering a spectrum of analyses from complete field-insensitivity through various levels of field-sensitivity. The main difference from our work is the approach taken to modelling field-addresses where, instead of using integer offsets, the actual field names themselves are used. To understand this, consider:

```
typedef struct { int *f1; int *f2; } aggr1;

aggr1 a,*b; int *p,c;
a.f1 = &c;
b = &a;
p = b->f2;
```

$a.f1 \supseteq \{c\}$, $b \supseteq \{a\}$, $p \supseteq (*b)||f2$

Here, the $||$ operator is just string concatenation, where $a||b \Rightarrow a.b$ and $(*a)||b \Rightarrow c.b$, if $a \supseteq \{c\}$. Thus, $p \supseteq (*b)||x$ replaces $p \supseteq *(b+k)$ from our system. While this difference appears trivial, it hides some complications. For example:

```
typedef struct { int *f1; int *f2; } aggr1;
typedef struct { int *f3; int *f4; } aggr2;
aggr1 a; aggr2 b; void *c; int d;
b.f3 = &d                        b.f3 ⊇ {d}
c = &b;                          c ⊇ {b}
a = (struct aggr1) *c;           a.f1 ⊇ (*c)||f1
                                 a.f2 ⊇ (*c)||f2
```

The above is well-defined C code, but the last statement presents an issue for the name string approach. This is because the type of "a" determines which fields are involved in the assignment. The problem is that the constraint variables $b.f1$ and $b.f2$ (arising from $(*c)||f1$ and $(*c)||f2$) do not exist as "b" has a different, but *compatible* type to "a". To overcome this, Yong *et al.* introduce three functions, *normalise, lookup* and *resolve*, whose purpose is to bridge the gap between different names representing the same location (such as $b.f1$ and $b.f3$ in the above). The key point is that, by using offsets instead of name strings, our system avoids these issues entirely and, thus, provides a simpler and more elegant formalisation.

An important feature of the Yong *et al.* framework is the ability to describe both portable and non-portable analyses. The latter can be used to support commonly found, but undefined C coding practices which rely on implementation-specific information, such as type size and alignment. In contrast, our system as described cannot safely handle such practices. However, with some small modification, it could be made to do so, whilst retaining its relative simplicity.

Yong *et al.* also examine the precision obtainable with field-sensitivity and, although smaller benchmarks were used, their findings match ours. Finally, they do not discuss the PWC problem, perhaps because it is only relevant to particular instances of their framework. Nevertheless, to obtain an equivalent analysis to ours, this issue must be addressed. Indeed, Chandra and Reps do so in their analyses, which they describe as an instance of the Yong *et al.* framework [3, 4]. Their solution is to adopt a worse-case assumption about pointers in positive weight cycles (i.e. they point to every field of each target). Unfortunately, they do not provide any experimental data which could be used as the basis of a comparison with our system.

6. CONCLUSION

We have presented a novel approach to modelling indirect function calls and aggregates for pointer analysis of C. Furthermore, we we evaluated its effect on time and precision using an example implementation. Our results indicate that field-sensitivity, while offering greater precision, is expensive to compute. In the future, we are interested in extending our formulation to be flow-sensitive and investigating the pros and cons of doing this. We have also been investigating the potential of several new graph algorithms for speeding up pointer analysis [19, 18]. Finally, the reader is referred to [18] for a more thorough examination of this material.

7. REFERENCES

[1] A. Aiken. Introduction to set constraint-based program analysis. *Sci. Comp. Prog.*, 35(2–3):79–111, 1999.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[3] S. Chandra and T. Reps. Physical type checking for C. In *Proc. ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 66–75, 1999.

[4] S. Chandra and T. Reps. Physical type checking for C. Technical Report BL0113590-990302-04, Bell Laboratories, Lucent Technologies, 1999.

[5] M. Das. Unification-based pointer analysis with directional assignments. In *Proc. ACM Conf. Programming Language Design and Implementation*, pages 35–46, 2000.

[6] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proc. Static Analysis Symposium*, volume 2126 of *LNCS*, pages 260–278. Springer, 2001.

[7] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. ACM Conf. Programming Language Design and Implementation*, pages 85–96, 1998.

[8] J. S. Foster, M. Fähndrich, and A. Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report CSD-97-964, University of California, Berkeley, 1997.

[9] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proc. Static Analysis Symposium*, volume 1824 of *LNCS*, pages 175–198. Springer-Verlag, 2000.

[10] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proc. ACM Conf. Programming Language Design and Implementation*, pages 254–263, 2001.

[11] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[12] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proc. ACM Symp. Software Testing and Analysis*, pages 113–123, 2000.

[13] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Transactions on Programming Languages And Systems*, 19(1):1–6, Jan. 1997.

[14] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[15] O. Lhoták and L. J. Hendren. Scaling Java points-to analysis using SPARK. In *Proc. Conf. Compiler Construction*, volume 2622 of *LNCS*, pages 153–169. Springer, 2003.

[16] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proc. Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 199–215. 1999.

[17] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Proc. ACM Workshop Program Analyses for Software Tools and Engineering*, pages 73–79, 2001.

[18] D. J. Pearce. *Some directed graph algorithms and their application to pointer analysis (work in progress).* PhD thesis, Imperial College, London, 2004.

[19] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proc. IEEE Workshop on Source Code Analysis and Manipulation*, pages 3–12, 2003.

[20] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proc. ACM Conf. Programming Language Design and Implementation*, pages 47–56, 2000.

[21] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proc. ACM Conf. Object Oriented Programming Systems, Languages and Applications*, pages 43–55, 2001.

[22] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. ACM symposium on Principles of Programming Languages*, pages 1–14, 1997.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. ACM Symp. Principles of Programming Languages*, pages 32–41, 1996.

[24] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *ACM Symp. Principles of Programming Languages*, pages 81–95, 2000.

[25] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[26] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proc. Static Analysis Symp.*, volume 2477 of *LNCS*, pages 180–195, 2002.

[27] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. ACM Conf. Programming Language Design and Implementation*, pages 1–12, 1995.

[28] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proc. ACM Conf. Programming Language Design and Implementation*, pages 91–103, 1999.