Data-Flow Analysis of Program Fragments*

Atanas Rountev[1], Barbara G. Ryder[1], and William Landi[2]

[1] Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
{rountev,ryder}@cs.rutgers.edu
[2] Siemens Corporate Research Inc, 755 College Road East, Princeton, NJ 08540, USA
landi@scr.siemens.com

Abstract. Traditional interprocedural data-flow analysis is performed on whole programs; however, such *whole-program analysis* is not feasible for large or incomplete programs. We propose *fragment data-flow analysis* as an alternative approach which computes data-flow information for a specific program fragment. The analysis is parameterized by the additional information available about the rest of the program. We describe two frameworks for interprocedural flow-sensitive fragment analysis, the relationship between fragment analysis and whole-program analysis, and the requirements ensuring fragment analysis safety and feasibility. We propose an application of fragment analysis as a second analysis phase after an inexpensive flow-insensitive whole-program analysis, in order to obtain better information for important program fragments. We also describe the design of two fragment analyses derived from an already existing whole-program flow- and context-sensitive pointer alias analysis for C programs and present empirical evaluation of their cost and precision. Our experiments show evidence of dramatically better precision obtainable at a practical cost.

1 Introduction

Many phases of the software development cycle require information about the properties of large and complex programs. *Data-flow analysis* extracts semantic information which can be used for code optimization, program slicing, semantic change analysis, program restructuring, and code testing. In many cases, *interprocedural* data-flow analysis is needed to obtain information about program properties that depend on the interaction between different procedures. *Flow-insensitive* analysis ignores the ordering of statements and computes one solution for the whole program; in contrast, *flow-sensitive* analysis follows the control flow order of statements and computes different solutions at distinct program points. *Context-sensitive* analysis considers (sometimes approximately) only paths along which calls and returns are properly matched, while *context-insensitive* analysis does not make this distinction.

Traditionally, interprocedural data-flow analysis is designed to analyze whole programs; however, in many cases such *whole-program analysis* is infeasible. For

* This research was supported, in part, by NSF grants CCR-9501761 and CCR-9804065, and Edison Design Group.

very large programs with hundreds of thousands or even millions lines of code, the time required to build a whole-program representation and the space needed to store it are prohibitive [1]. In many cases, the programs are incomplete — the source code for parts of the program (e.g., libraries) is not available. Empirical evidence suggests that precise interprocedural flow-sensitive analysis does not scale for large programs [18]. In some cases the analysis results are not needed for the whole program, but only for a relatively small part of it — for example, a maintenance task for a specific program fragment may only require data-flow information for program points inside the fragment, both before and after the maintenance change.

This paper proposes an alternative approach for program analysis. Instead of addressing the problem of computing data-flow information for the whole program, we address the problem of computing data-flow information for a specific program fragment. The problem is parameterized by the additional information available about the rest of the program. Such *fragment data-flow analysis* can avoid the problems of the traditional whole-program analysis. For example, information can be obtained about fragments of very large programs for which whole-program analysis is prohibitively expensive. Similarly, analysis can be performed on fragments of incomplete programs; for such programs, traditional whole-program analysis is not possible. Finally, fragment analysis computes information about only the "interesting portion" of the program, which can be significantly smaller than the program itself.

This paper is a first step in investigating the theory and practice of fragment data-flow analysis. It only considers flow-sensitive fragment analysis. The main contributions of this work can be summarized as follows:

– We describe two frameworks for interprocedural flow-sensitive fragment analysis, derived from existing frameworks for whole-program analysis. We discuss the relationship between fragment analysis and whole-program analysis, and the requirements ensuring fragment analysis safety and feasibility.
– We propose an application of fragment analysis as a second analysis phase after an inexpensive flow-insensitive whole-program analysis, in order to obtain better information for important program fragments. This approach can be used for programs that are too big to be analyzed by flow-sensitive whole-program analysis, yet allow flow-insensitive whole-program analysis — for example, C programs with around 100,000 lines of code [1, 18, 17].
– We describe the design of two fragment analyses derived from a whole-program flow- and context-sensitive pointer alias analysis [10] for C programs.
– We present empirical evaluation of the cost and precision of these two fragment pointer alias analyses. We show that the time and space costs of the analyses are practical. In about 75% of our experiments, the better of the two analyses results in a fourfold or higher precision improvement over the whole-program flow-insensitive solution.

The rest of the paper is organized as follows: Section 2 describes frameworks for flow-sensitive whole-program analysis. Section 3 discusses frameworks for fragment analysis and the issues involved in their design. Section 4 describes the whole-program pointer alias analysis from [10], and Section 5 describes the design of two fragment pointer alias analyses. Empirical results are presented in Section 6. Section 7 describes related work, and Section 8 presents our conclusions.

2 Whole-Program Data-Flow Analysis

This section presents two well-known frameworks for whole-program interprocedural flow-sensitive data-flow analysis. Without loss of generality, we will only consider analysis for forward data-flow problems [12]. Given a whole program to be analyzed, a *whole-program analysis* constructs a data-flow framework $<G, L, F, M, \eta>$, where:

- $G = (N, E, \rho)$ is a directed graph with node set N, edge set E and starting node $\rho \in N$ (for our purposes, G is an interprocedural control flow graph).
- $<L, \leq, \wedge>$ is a meet semi-lattice [12] with partial order $\leq$ and meet $\wedge$. For simplicity, we only consider L which is finite[1] and has a top element $\top$.
- $F \subseteq \{f \mid f : L \to L\}$ is a function space closed under composition and arbitrary meets. We assume that F is monotone [12].
- $M : N \to F$ is an assignment of transfer functions to the nodes in G (without loss of generality, we assume no edge transfer functions). The transfer function for node n will be denoted by f_n.
- $\eta \in L$ is the solution at the bottom of ρ.

The program is represented by an *interprocedural control flow graph* (ICFG) [10], which contains control flow graphs for all procedures in the program. Each procedure has a single *entry node* (node ρ is the entry node of the starting procedure) and a single *exit node*. Each call statement is represented by a pair of nodes, a *call node* and a *return node*. There is an edge from the call node to the entry node of the called procedure; there is also an edge from the exit node of the called procedure to the return node in the calling procedure.

A path from node n_1 to node n_k is a sequence of nodes $p = (n_1, \ldots, n_k)$ such that $(n_i, n_{i+1}) \in E$. Let $f_p = f_{n_1} \circ f_{n_2} \circ \ldots \circ f_{n_k}$. A *realizable path* is a path on which every procedure returns to the call site which invoked it [16, 10, 13]; only such paths represent potential sequences of execution steps. A *same-level realizable path* is a realizable path whose first and last nodes belong to the same procedure, and on which the number of call nodes is equal to the number of return nodes. Such paths represent sequences of execution steps during which the call stack may temporarily grow deeper, but never shallower that its original depth, before eventually returning to its original depth [13]. The set of all realizable paths from n to m will be denoted by $RP(n, m)$; the set of all same-level realizable paths from n to m will be denoted by $SLRP(n, m)$.

Definition 1. *For each $n \in N$, the* ***meet-over-all-realizable-paths*** *(MORP) solution at n is defined as $MORP(n) = \bigwedge_{p \in RP(\rho, n)} f_p(\eta)$.*

[1] The results can be easily generalized for finite-height semi-lattices.

Context-Insensitive Analysis. After constructing $< G, L, F, M, \eta >$, a whole-program analysis computes a solution $S : N \to L$; the data-flow solution at the bottom of node n will be denoted by S_n. The solution is *safe* iff $S_n \leq \mathit{MORP}(n)$ for each node n; a *safe analysis* computes a safe solution for each valid input program. Traditionally, a system of equations is constructed and then solved using fixed-point iteration. In the simplest case, a context-insensitive analysis constructs a system of equations of the form

$$S_\rho = \eta, \quad S_n = \bigwedge_{m \in \mathit{Pred}(n)} f_n(S_m)$$

where $\mathit{Pred}(n)$ is the set of predecessor nodes for n. The initial solution has $S_\rho = \eta$ and $S_n = \top$ for any $n \neq \rho$; the final solution is a fixed point of the system and is also a safe approximation of the MORP solution.

Context-Sensitive Analysis. The problem with the above approach is that information is propagated from the exit of a procedure to *all* of its callers. Context-sensitive analysis is used to handle this potential source of imprecision. One approach is to propagate elements of L together with tags which approximate the calling context of the procedure. At the exit of the procedure, these tags are consulted in order to back-propagate information only to call sites at which the corresponding calling context existed.

The "functional approach" to context sensitivity [16] uses a new lattice that is the function space of functions mapping L to L. Intuitively, if the solution at node n is a map $h_n : L \to L$, then for each $x \in L$ we can use $h_n(x)$ to approximate $f_p(x)$ for each path $p \in \mathit{SLRP}(e, n)$, where e is the entry node of the procedure containing n. In other words, $h_n(x)$ approximates the part of the solution at n that occurs under calling context x at e. If calling context x never occurs at e, then $h_n(x) = \top$. The solution of the original problem is obtained as $\bigwedge_{x \in L} h_n(x)$.

In general, this approach requires compact representation of functions and explicit functional compositions and meets, which are usually infeasible. When L is finite, a feasible version of the analysis can be designed [16]. Figure 1 presents a simplified description of this version. $H[n,x]$ contains the current value of $h_n(x)$; the worklist contains pairs (n,x) for which $H[n,x]$ has changed and has to be propagated to the successors of n. If n is a call node, at line 7 the value of $H[n,x]$ is propagated to the entry node of the called procedure. If n is an exit node, the value of $H[n,x]$ is propagated only to return nodes at whose corresponding call nodes x occurs (lines 8-10).

For distributive frameworks [12], this algorithm terminates with the MORP solution; for non-distributive monotone frameworks, it produces a safe approximation of the MORP solution [16]. When the lattice is the power set of some basic finite set D of data-flow facts (e.g., the set all potential aliases or the set of all variable definitions), the algorithm can be modified to propagate elements of D instead of elements of 2^D. For distributive frameworks, this approach produces a precise solution [13]. For non-distributive monotone frameworks, restricting the context to a singleton set necessarily introduces some approximation; the whole-program pointer alias analysis from [10] falls in this category.

```
input      $<G, L, F, M, \eta>$; $L$ is finite
output     $S$: array[$N$] of $L$
declare    $H$: array[$N$,$L$] of $L$; initial values $\top$
           $W$: list of $(n,x)$, $n \in N$, $x \in L$; initially empty
[1]   $H[\rho,\eta] := \eta$; $W := \{(\rho,\eta)\}$;
[2]   while $W \neq \emptyset$ do
[3]       remove $(n,x)$ from $W$; $y := H[n,x]$;
[4]       if $n$ is not a call node or an exit node then
[5]           foreach $m \in Succ(n)$ do propagate($m$,$x$,$f_m(y)$);
[6]       if $n$ is a call node then
[7]           $e$ := called_entry($n$); propagate($e$,$y$,$f_e(y)$);
[8]       if $n$ is an exit node then
[9]           foreach $r \in Succ(n)$ and $l \in L$ do
[10]              if $H$[call_node($r$),$l$] $= x$ then propagate($r$,$l$,$f_r(y)$);
[11]  foreach $n \in N$ do
[12]      $S[n] := \bigwedge_{l \in L} H[n,l]$;
[13]  procedure propagate($n$,$x$,$y$)
[14]      $H[n,x] := H[n,x] \wedge y$;  if $H[n,x]$ changed then add $(n,x)$ to $W$;
```

Fig. 1. Worklist implementation of context-sensitive whole-program analysis

3 Fragment Data-Flow Analysis

This section describes how interprocedural flow-sensitive whole-program analysis can be modified to obtain *fragment data-flow analysis*, which analyzes a program fragment instead of a whole program. The structure of context-insensitive and context-sensitive fragment analysis is discussed, as well as the issues involved in the design of fragment analysis and the requirements ensuring its safety.

3.1 Fragment Analysis Structure

We assume that the analysis input includes a *program fragment* F, which is an arbitrary set of procedures. We expect these procedures to be strongly interrelated; otherwise, the analysis may yield information that is too imprecise. The input also contains *whole-program information* I, which represents the knowledge available about the programs to which F belongs. The whole-program information depends on the particular software development environment and the process in which fragment analysis is used; the role of I is further discussed in Sect. 3.2. We will use $\mathcal{P}_I(F)$ to denote the set of all valid whole programs that contain F and for which I is true; depending on I, $\mathcal{P}_I(F)$ can be anything from a singleton set (e.g., when the source code of the whole program is available) to an infinite set.

Given F and I, a fragment analysis extracts several kinds of information, shown in Table 1. Graph $G' = (N', E')$ is the ICFG for the fragment and can be constructed similarly to the whole-program case, except for calls to procedures outside of F, which are not represented by any edges in G'. Set *BoundaryEntries* contains every entry node $e \in N'$ which has a predecessor $c \notin N'$ in some program from $\mathcal{P}_I(F)$. Similarly, *BoundaryCalls* contains every call node $c \in N'$ which has a successor $e \notin N'$ in some program from $\mathcal{P}_I(F)$. Set *BoundaryReturns* contains the return nodes corresponding to call nodes from *BoundaryCalls*.

Table 1. Information extracted by a fragment analysis

Information	Description
$<G', L', F', M', \eta'>$	Data-flow framework
BoundaryEntries	Entry nodes of procedures called from outside of F
BoundaryCalls	Call nodes to procedures outside of F
BoundaryReturns	Return nodes from procedures outside of F
$\beta_e : \mathit{BoundaryEntries} \rightarrow L'$	Summary information at boundary entry nodes
$\beta_c : \mathit{BoundaryCalls} \rightarrow F'$	Summary information at boundary call nodes

In the data-flow framework, L' is a finite meet semi-lattice with partial order $\leq$, meet $\wedge$ and a top element $\top'$. $F' \subseteq \{f \mid f : L' \rightarrow L'\}$ is a monotone function space closed under composition and arbitrary meets. M' is an assignment of transfer functions to the nodes in G'; the transfer function for node n will be denoted by f'_n. Value $\eta' \in L'$ is the solution at the bottom of the entry node of the program; it is needed only if F contains the starting procedure of the program.

In order to summarize the effects of the rest of the program on F, a fragment analysis constructs two maps. Map $\beta_e : \mathit{BoundaryEntries} \rightarrow L'$ assigns to each boundary entry node e a value which approximates the part of the solution at e that is due to realizable paths that reach e from outside of F. Map $\beta_c : \mathit{BoundaryCalls} \rightarrow F'$ assigns to each boundary call node a function which summarizes the effects of all same-level realizable paths from the entry to the exit of the called procedure. Both maps are discussed in Sect. 3.3.

Based on the extracted information from Table 1, a *context-insensitive* fragment analysis solves the system of equations shown in Fig. 2. A *context-sensitive* fragment analysis can be constructed as in Sect. 2; a worklist implementation is shown in Fig. 3. This algorithm is a modified version of the one presented in Fig. 1, with new or modified lines labeled with asterisks.

3.2 Fragment Analysis Design

Designers of a specific fragment analysis have to address several important problems. One problem is to decide what kind of whole-program information I to use. The decision depends on the software development environment, as well as the process in which the fragment analysis is used. In this paper, we are particularly interested in a process where an inexpensive flow-insensitive whole-program

$$
\begin{array}{ll}
S'_n = \bigwedge_{m \in Pred(n)} f'_n(S'_m) & \text{if } n \notin \mathit{BoundaryEntries} \cup \mathit{BoundaryReturns} \\
S'_n = \beta_e(n) \ \wedge \ \bigwedge_{m \in Pred(n)} f'_n(S'_m) & \text{if } n \in \mathit{BoundaryEntries} \\
S'_n = f'_n(\beta_c(m)(S'_m)) & \text{if } n \in \mathit{BoundaryReturns} \text{ and } m = \text{call\_node}(n) \\
S'_\rho = \eta' & \text{if } \rho \in N'
\end{array}
$$

Fig. 2. Context-insensitive fragment analysis

```
input       $<G', L', F', M', \eta'>$; $L'$ is finite
output      $S$: array[$N'$] of $L'$
declare     $H$: array[$N'$,$L'$] of $L'$; initial values $\top'$
            $W$: list of ($n$,$x$), $n \in N'$, $x \in L'$; initially empty
[1a*]   if $\rho \in N'$ then
[1b]        $H[\rho,\eta'] := \eta'$; add ($\rho$,$\eta'$) to $W$;
[1c*]   foreach $n \in \mathit{BoundaryEntries}$ do
[1d*]       $H[n,\beta_e(n)] := \beta_e(n)$; add ($n$,$\beta_e(n)$) to $W$;
[2]     while $W \neq \emptyset$ do
[3]         remove ($n$,$x$) from $W$; $y$:=$H[n,x]$;
[4]         if $n$ is not a call node or an exit node then
[5]             foreach $m \in \mathit{Succ}(n)$ do propagate($m$,$x$,$f'_m(y)$);
[6a*]       if $n$ is a call node and $n \notin \mathit{BoundaryCalls}$ then
[7a]            $e$ := called_entry($n$); propagate($e$,$y$,$f'_e(y)$);
[6b*]       if $n$ is a call node and $n \in \mathit{BoundaryCalls}$ then
[7b*]           $r$ := ret_node($n$); propagate($r$,$x$,$f'_r(\beta_c(n)(y))$);
[8]         if $n$ is an exit node then
[9]             foreach $r \in \mathit{Succ}(n)$ and $l' \in L'$ do
[10]                if $H$[call_node($r$),$l'$] = $x$ then propagate($r$,$l'$,$f'_r(y)$);
[11]    foreach $n \in N'$ do
[12]        $S[n] := \bigwedge_{l' \in L'} H[n,l']$;
```

Fig. 3. Worklist implementation of context-sensitive fragment analysis

analysis is performed first, and then a more precise flow-sensitive fragment analysis is used on fragments for which better information is needed. This approach can be used for programs that are too big to be analyzed by flow-sensitive whole-program analysis, yet allow flow-insensitive whole-program analysis — for example, C programs with around 100,000 lines of code [1, 18, 17]. In this scenario, the first stage computes a whole-program flow-insensitive solution and the program call graph. The second stage uses the call graph and the flow-insensitive solution as its whole-program information I. The two fragment analyses in Sect. 5 are designed in this manner.

Another problem is to construct the information described in Table 1. Sets *BoundaryCalls* and *BoundaryReturns* can be determined from F. Set *BoundaryEntries* by definition depends on I. The summary information at boundary nodes also depends on I. When the fragment analysis follows a flow-insensitive whole-program analysis, I contains the call graph of the program, from which the boundary entries can be easily determined. In this case, β_e and β_c can be extracted from the whole-program flow-insensitive solution, as shown in Sect. 5.

The semi-lattice L' depends mostly on F. However, it may be also dependent on I — for example, when the fragment analysis follows a flow-insensitive whole-program analysis, I contains a whole-program solution which may need to be represented (possibly approximately) using L'. The fragment analysis complexity is bounded by a function of the size of L'; therefore, it is crucial to ensure that the size of L' depends *only* on the size of F and not the size of I. This requirement guarantees that the fragment analysis will be feasible for relatively small fragments of very large programs. The fragment analyses from Sect. 5 illustrate this situation.

3.3 Fragment Analysis Safety

The fragment analyses outlined above are similar in structure to the whole-program analyses from Sect. 2. In fact, we are only interested in fragment analyses that are *derived* from whole-program analyses. Consider a safe whole-program analysis which we want to modify in order to obtain a safe fragment analysis. The most important problem is to define the relationship between the semi-lattice L' for a fragment F and the whole-program semi-lattices L_p for the programs $p \in \mathcal{P}_I(F)$. For each such L_p, the designers of the fragment analysis must define an *abstraction relation* $\alpha_p \subseteq L_p \times L'$. This relation encodes the notion of safety and is used to prove that the analysis is safe; it is never explicitly constructed or used during the analysis. If $(x, x') \in \alpha_p$, we will write "$\alpha_p(x, x')$".

Intuitively, the abstraction relation α_p defines the relationship between the "knowledge" represented by values from L_p and the "knowledge" represented by values from L'. If $\alpha_p(x, x')$, the knowledge associated with x' "safely abstracts" the knowledge associated with x; thus, α_p is similar in nature to the abstraction relations used for abstract interpretation [19, 6]. The choice of α_p depends both on the original whole-program analysis and the intended clients of the fragment analysis solution. Sect. 5 presents an example of one such choice.

Definition 2. *A solution produced by a fragment analysis for an input pair* (F, I) *is* **safe** *iff* $\alpha_p(MORP_p(n), S'_n)$ *for each* $p \in \mathcal{P}_I(F)$ *and each* $n \in N'$*, where* S'_n *is the fragment analysis solution at* n *and* $MORP_p(n)$ *is the MORP solution at* n *in* p*. A* **safe fragment analysis** *yields a safe solution for each valid input pair* (F, I)*.*

With this definition in mind, we present a set of sufficient requirements that ensures the safety of the fragment analysis. Intuitively, the first requirement ensures that a safe approximation in L', as defined by the partial order $\leq$, is also a safe abstraction according to α_p. The second requirement ensures that if $x' \in L'$ safely abstracts the effects of each realizable path ending at a given node n, it can be used to safely abstract the MORP solution at n. Formally, for any $p \in \mathcal{P}_I(F)$ and its α_p, any $x, y \in L_p$ and any $x', y' \in L'$, the following must be true:

Property 1: if $\alpha_p(x, x')$ and $y' \leq x'$, then $\alpha_p(x, y')$
Property 2: if $\alpha_p(x, x')$ and $\alpha_p(y, x')$, then $\alpha_p(x \wedge y, x')$

The next requirement ensures that the transfer functions in the fragment analysis safely abstract the transfer functions in the whole-program analysis. Let $f_{n,p}$ be the transfer function for $n \in N'$ in the whole-program analysis for any $p \in \mathcal{P}_I(F)$. For any $n \in N'$, $x \in L_p$ and $x' \in L'$, the following must be true:

Property 3: if $\alpha_p(x, x')$, then $\alpha_p(f_{n,p}(x), f'_n(x'))$

If F contains the starting procedure of the program, then $\rho \in N'$ and the fragment analysis solution at the bottom of ρ is η'. Let η_p be the whole-program solution at the bottom of ρ for any $p \in \mathcal{P}_I(F)$. Then the following must be true:

Property 4: $\alpha_p(\eta_p, \eta')$

The next requirement ensures that for each boundary entry node e, the summary value $\beta_e(e)$ safely abstracts the effects of each realizable path that reaches e from outside of F. For any $p \in \mathcal{P}_I(F)$, let $RP_p^{out}(\rho, e)$ be the set of all realizable paths $q = (\rho, \ldots, c, e)$ in p such that $c \notin N'$; recall that f_q is the composition of the transfer functions for the nodes in q. Then the following must be true:

Property 5: $\forall q \in RP_p^{out}(\rho, e) : \ \alpha_p(f_q(\eta_p), \beta_e(e))$

The last requirement ensures that the summary function $\beta_c(c)$ for each boundary call node c safely abstracts the effects of each same-level realizable path from the entry to the exit of the called procedure. Consider any $p \in \mathcal{P}_I(F)$ in which c has a successor entry node $e \notin N'$ and let t be the exit node corresponding to e. Let $SLRP_p(e, t)$ be the set of all same-level realizable paths in p from e to t. Intuitively, each realizable path ending at t has a suffix which is a same-level realizable path; thus, $\beta_c(c)$ should safely abstract each path from $SLRP_p(e, t)$. Formally, for any $x \in L_p$ and $x' \in L'$, the following must be true:

Property 6: $\forall q \in SLRP_p(e, t)$: if $\alpha_p(x, x')$, then $\alpha_p(f_q(x), \beta_c(c)(x'))$

If the above requirements are satisfied, the context-insensitive fragment analysis derived from a safe context-insensitive whole-program analysis is safe, according to Definition 2. The proof considers a fixed-point solution of the system in Fig. 2. It can be shown that for each $p \in \mathcal{P}_I(F)$, each $n \in N'$ and each realizable path $q = (\rho_p, \ldots, n)$, it is true that $\alpha_p(f_q(\eta_p), S'_n)$. Each such q is the concatenation of two realizable paths r' and r''. Path r' starts from ρ_p and lies entirely outside of F; if $\rho_p \in N'$, r' is empty. Path $r'' = (e, \ldots, n)$ has $e \in N'$ and $n \in N'$ and is the *fragment suffix* of q; it may enter and leave F arbitrarily. The proof is by induction on the length of the fragment suffix of q.

Similarly, the context-sensitive fragment analysis derived from a safe context-sensitive whole-program analysis is safe. It is enough to show that for each $p \in \mathcal{P}_I(F)$, each $n \in N'$ and each realizable path $q = (\rho_p, \ldots, n)$, there exists a value $l' \in L'$ such that $\alpha_p(f_q(\eta_p), H[n, l'])$. Again, this can be proven by induction on the length of the fragment suffix of q.

4 Whole-Program Pointer Alias Analysis

This section presents a simplified high-level description of the Landi-Ryder whole-program pointer alias analysis [10] for the C programming language. The analysis considers a set of names that can be described by the grammar in Fig. 4. Each of the names corresponds to one or more run-time memory locations. An *alias pair* (or simply an *alias*) is a pair of names that potentially represent the same memory location.

```
<Name> ::= <SimpleName> | <Deref> | <ArrayElem>
         | <FieldAccess> | <K-Limited>
<SimpleName>   ::=   identifier       /* variable name */
               |     heap_n           /* heap location */
<Deref>        ::=   *(<Name>+?)      /* dereference */
<ArrayElem>    ::=   <Name>[?]        /* array element */
<Fld>          ::=   identifier       /* field name */
<FieldAccess>  ::=   <Name>.<Fld>     /* field of a structure */
<K-Limited>    ::=   <Name>#          /* k-limited name */
```

Fig. 4. Grammar for names

If a program point n is a call to `malloc` or a similar function, name $heap_n$ represents the set of all heap memory locations created at this point during execution. If there are recursive data structures (e.g., linked lists), the number of names is potentially infinite. The analysis limits the number of dereferences in a name to a given constant k; any name with more that k dereferences is represented by a *k-limited name*. For example, for $k = 1$, name $(*((*p).f)).f$ is represented by the k-limited name $((*p).f)\#$.

A *simple name* is a name generated by the `SimpleName` nonterminal in the grammar. The *root name* of a name n is the simple name used when n is generated by the grammar; for example, the root name of $(*p).f$ is p. Name n is a *fixed location* if it does not contain any dereferences. Name $(*p).f$ is not a fixed location, while $s[?].f$ is a fixed location.

The lattice of the analysis is the power set of the set of all pairs of names; the meet operator is set union and the partial order is the "is-superset-of" relation. The analysis is flow- and context-sensitive, and conceptually follows the Sharir-Pnueli algorithm described in Sect. 2. However, since the lattice is a power set, the algorithm can be modified to propagate single aliases instead of sets of aliases — the worklist contains triples (n,RA,PA), where n is a node, RA is a reaching alias that is part of the solution at the entry of the procedure to which n belongs, and PA is a possible alias at the bottom of n. Restricting the reaching alias set to a single alias introduces some approximation; therefore, the actual Landi-Ryder algorithm can be viewed as an approximation algorithm solving the more precise Sharir-Pnueli formulation of the problem.

The fragment analyses in Sect. 5 are derived from a modified version of the Landi-Ryder analysis. This version considers only aliases containing a fixed location; aliases with two non-fixed locations (e.g., an alias between $(*p).f$ and $*q$) are ignored during propagation. It can be shown that this is safe, as long as the program does not contain assignments in the form `*p=&x`; if such assignments exist, they can be removed by introducing intermediate temporary variables — for example, `t=&x; *p=t`. An assignment whose left-hand side is a non-fixed location (i.e., *through-deref assignment*) potentially modifies many fixed locations. A safe solution for the modified analysis is sufficient to estimate all fixed locations possibly modified by through-deref assignments. We refer to this process as *resolution* of through-deref assignments.

5 Fragment Pointer Alias Analyses

This section describes two fragment pointer alias analyses — basic analysis A_1 and extended analysis A_2 — derived from the modified whole-program analysis in Sect. 4. They are used for resolution of through-deref assignments. Their design is based on a process in which flow-insensitive whole-program pointer alias analysis is performed first, and then more precise flow-sensitive fragment pointer alias analysis is used on fragments for which better information is needed. As described in Sect. 3.2, in this case the whole-program information I contains a whole-program flow-insensitive solution and the program call graph.

Analysis Input. The flow-insensitive solution S_{FI} is obtained using a whole-program flow- and context-insensitive analysis similar to the one in [21]. Intuitively, the names in the program are partitioned into equivalence classes; if two names can be aliases, they belong to the same class in the final solution. Every name starts in its own equivalence class and then classes are joined as possible aliases are discovered. For example, statement `p=&x` causes the equivalence classes of $*p$ and x to merge. Overall, the analysis is similar to other flow- and context-insensitive analyses with almost-linear cost [17, 15]. Then S_{FI} is used to resolve calls through function pointers and to produce a safe approximation of the program call graph. The sets of boundary entry, call and return nodes can be easily determined from this graph.

The basic analysis A_1 takes as input S_{FI} and the program call graph, as well as the source code for the analyzed fragment F. The extended analysis A_2 requires as additional input the source code for all procedures directly or indirectly called by procedures in F. These additional procedures together with the procedures from F form the *extended fragment* F_{ext}. Including all transitively called procedures allows A_2 to estimate better the effects of calls to procedures outside of F; the tradeoffs of this approach are further discussed in Sect. 8.

Analysis Lattices. The whole-program lattice is based on the set of names generated by the grammar in Fig. 4. Each such name can be classified as either relevant or irrelevant. A *relevant name* has a root name with a syntactic occurrence in the fragment; all other names are irrelevant. Since the fragment analysis solution is used to resolve through-deref assignments, only aliases that contain a relevant non-fixed location are useful. If the new lattice L' contains all such aliases, its size still potentially depends on the number of fixed locations in the whole program. This problem can be solved by using special *placeholder variables* to represent sets of related irrelevant fixed locations; this approach is similar to the use of representative data-flow values in other analyses [11, 10, 7, 20, 4].

Consider an equivalence class $E_i \in S_{FI}$ that contains at least one relevant name. If E_i contains irrelevant fixed locations, a placeholder variable ph_i is used to represent them. The aliases from L' fall in two categories: (1) a pair of relevant names, exactly one of which is a fixed location, and (2) a relevant non-fixed location and a placeholder variable. Clearly, the number of relevant names and the number of placeholder variables depend only on the size of the fragment; thus, the size of L' is independent of the size of the whole program.

In the final solution, an alias with two relevant names represents itself, while an alias with a placeholder ph_i represents a set of aliases, one for each irrelevant fixed location represented by ph_i. This can be formalized by defining an abstraction relation $\alpha_p \subseteq L_p \times L'$ for each $p \in \mathcal{P}_I(F)$. For each pair of sets $S \in L_p$ and $S' \in L'$, we have $\alpha_p(S, S')$ iff each alias from S is safely abstracted by S'. Let x be the non-fixed location in alias $(x,y) \in S$; then α_p is defined as follows:

- If x is an irrelevant name, then (x,y) is safely abstracted by S'
- If x and y are relevant and $(x,y) \in S'$, then (x,y) is safely abstracted by S'
- If x is relevant and y is irrelevant, E_i is y's equivalence class, and $(x,ph_i) \in S'$, then (x,y) is safely abstracted by S'

Intuitively, the above definition says that aliases involving an irrelevant non-fixed location can be safely ignored. It also says that irrelevant fixed locations from the same equivalence class have equivalent behavior and need not be considered individually. The safety implications of this definition are discussed later. Note that if a fragment alias solution is safe according to the above definition, it can be used to safely resolve through-deref assignments.

Summaries at Boundary Nodes. Based on S_{FI}, A_1 and A_2 construct a set of aliases $\sigma \in L'$. Each equivalence class E_i is examined and for each pair of names (x,y) of a non-fixed location x and a fixed location y from the class, the following is done: first, if x is an irrelevant name, the pair is ignored. Otherwise, if y is a relevant name, (x,y) is added to σ. Otherwise, if y is an irrelevant name, (x,ph_i) is added to σ. Clearly, (x,y) is safely abstracted by σ according to the definition of α_p given above.

The basic analysis A_1 defines $\beta_e(e) = \sigma$ for each boundary entry node e and and $\beta_c(c)(x) = \sigma$ for each $x \in L'$ and each boundary call node c. This essentially means that S_{FI} is used to approximate the solutions at boundary entry nodes and boundary return nodes. For the extended analysis A_2, there are no boundary call nodes. For each boundary entry node e that belongs to the original fragment F, the analysis defines $\beta_e(e) = \sigma$. If e is part of the extended fragment F_{ext}, but is not part of the original fragment F, the analysis defines $\beta_e(e) = \emptyset$.

Analysis Safety. To show the safety of A_1, it is enough to show that the requirements from Sect. 3.3 are satisfied. Properties 1 and 2 are trivially satisfied. Since both η_p and η' are the empty set, Property 4 is also true.

Showing that Property 3 is true requires careful examination of the transfer functions from [10]. The formal proof is based on two key observations. The first one is that aliases involving an irrelevant non-fixed location are only propagated through the nodes in the fragment, without actually creating any new aliases; therefore, they can be safely ignored. The second observation is that aliases with the same non-fixed location and with different irrelevant fixed locations have equivalent behavior. For example, alias $(*p,x)$ at the top of statement `q=p;` results in $(*p,x)$ and $(*q,x)$ at the bottom of the statement. Similarly, $(*p,y)$ results in $(*p,y)$ and $(*q,y)$. If both x and y are represented by a placeholder ph_i, alias $(*p,ph_i)$ results in $(*p,ph_i)$ and $(*q,ph_i)$, which satisfies Property 3.

The set σ described above is extracted from the whole-program flow-insensitive alias solution, which is safe. Therefore, σ safely abstracts any alias that could be true at a node in the program; thus, Properties 5 and 6 are true. Since all requirements are satisfied, A_1 is safe. Similarly to the whole-program case, the actual implementation propagates single aliases instead of sets of aliases. As a result, the reaching alias set is restricted to a single alias and therefore some approximation is introduced. Of course, the resulting solution is still safe.

For the extended analysis A_2, it is *not true* that the solution is safe at each node in the extended fragment. For example, consider the entry node of a procedure that was not in the original fragment F. If this procedure is called from outside of the extended fragment F_{ext}, aliases could be propagated along this call edge during the whole-program analysis, but would be missing in the fragment analysis. However, it can be proven that for each node in the original fragment F, the solution is safe. The proof is very similar to the one outlined in Sect. 3.3, and still requires that Properties 1 through 4 are true. For each realizable path q starting at ρ and ending at a node in F, its fragment suffix is the subpath starting at the first node in q that belongs to F. The proof is by induction on the length of the fragment suffix; the base case of the induction depends on the fact that $\beta_e(e) = \sigma$ for boundary entry nodes in F, and therefore the solution at such nodes is safe.

6 Empirical Results

Our implementation uses the PROLANGS Analysis Framework[2] (version 1.0), which incorporates the Edison Design Group front end for C/C++. The results were gathered on a *Sun Sparc-20* with 352 MB of memory. The implementation analyzes a reduced version of C that excludes signals, `setjmp`, and `longjmp`, but allows function pointers, type casting and union types.

Table 2 describes the C programs used in our experiments. It shows the program size in lines of code and number of ICFG nodes, the number of procedures, the total number of assignments, and the number of through-deref assignments.

[2] **http://www.prolangs.rutgers.edu**

Table 2. Analyzed Programs

Program	LOC	ICFG Nodes	Procs	Assignments		Program	LOC	ICFG Nodes	Procs	Assignments	
				All	Deref					All	Deref
`zip`	8177	6172	122	3443	582	`espresso`	14910	15339	372	7822	1322
`sc`	8530	6678	160	3440	159	`tsl`	16053	15469	471	7249	507
`larn`	10014	12063	298	6021	389	`moria`	25292	20213	458	10557	1358

For each of the data programs, we extracted by hand subsets of related procedures that formed a cohesive fragment. Significant effort was put in determining realistic fragments — the program source code was thoroughly examined, the program call graph was used to obtain better understanding of the calling relationships in the program, and the documentation (both external and inside the source code) was carefully analyzed. For each program, two fragments were extracted. For example, for `zip` one of the fragments consisted of all procedures implementing the implosion algorithm. For `espresso`, one of the fragments consisted of the procedures for reducing the cubes of a boolean function. For `sc`, one of the fragments consisted of the procedures used to evaluate expressions in a spreadsheet. Some characteristics of the fragments are given in Table 3.

For each fragment, three experiments were performed; the results are shown in Table 4. In the first experiment, the solution from the whole-program flow-insensitive (FI) analysis was used at each through-deref assignment to determine the number of simple names possibly modified by the assignment. For a fixed location that was not a simple name, a modification of its root simple name was counted (e.g., a modification of $s.f$ was counted as a modification of s). Then

Table 3. Analyzed Fragments

Fragment	ICFG Nodes	% Total	Procs	Boundary		Assignments	
				Entries	Calls	All	Deref
`zip.1`	1351	21.9	28	5	17	776	59
`zip.2`	429	7.0	9	5	19	255	50
`sc.1`	1238	18.5	30	3	30	609	8
`sc.2`	793	11.9	13	3	10	459	23
`larn.1`	345	2.9	4	4	35	188	46
`larn.2`	420	3.5	11	9	44	216	4
`espresso.1`	440	2.9	6	2	40	234	38
`espresso.2`	963	6.3	19	5	113	461	53
`tsl.1`	355	2.3	13	4	33	175	15
`tsl.2`	1004	6.5	29	7	134	459	17
`moria.1`	2678	13.2	43	14	348	1634	392
`moria.2`	1221	6.0	27	7	149	644	49

Table 4. Precision Comparison

Fragment	WholePrgm FI	Basic FFS	Percent	Extended FFS	Percent
`zip.1`	15.78	12.98	82.3	12.98	82.3
`zip.2`	3.88	1.02	26.3	1.02	26.3
`sc.1`	9.50	9.00	94.7	1.00	10.5
`sc.2`	13.14	3.29	25.0	3.29	25.0
`larn.1`	17.00	14.57	85.7	1.00	5.9
`larn.2`	9.00	9.00	100.0	1.00	11.1
`espresso.1`	107.10	107.10	100.0	34.00	31.7
`espresso.2`	57.55	57.55	100.0	35.11	61.0
`tsl.1`	41.87	17.07	40.8	1.00	2.4
`tsl.2`	41.88	24.82	59.3	1.00	2.4
`moria.1`	132.80	1.46	1.1	1.46	1.1
`moria.2`	31.04	3.55	11.4	3.55	11.4

the average across all through-deref assignments in the fragment was taken. In the second experiment, the FI analysis was followed by the basic fragment flow-sensitive (FFS) analysis. Again, for each through-deref assignment the number of simple names possibly modified was determined; placeholder variables were expanded to determine the actual simple names modified. Then the average across all through-deref assignments was taken. Each of these averages is shown as an absolute value and as a percent of the FI average. The third experiment used the extended fragment analysis instead of the basic fragment analysis.

Overall, the results show that the precision of the extended analysis is very good. In particular, for seven fragments it produces averages very close to 1, which is the lower bound. The averages are bigger than 4 for only three fragments; all three take as input a pointer to an external data structure, and a large number of the through-deref assignments in the fragment are through this pointer. The pointer itself is not modified, and each modification through it resolves to the same number of simple names as in the FI solution.

The performance of the basic analysis is less satisfactory. For five fragments, it achieves the same precision as the extended analysis. For the remaining fragments, in five cases the solution is close to the FI solution. The main reason is that precision gains from flow-sensitivity are lost at calls to procedures outside of the fragment.

Table 5 shows the running times of the analyses in minutes and seconds. The last two columns give the used space in kilobytes. The results show that the cost of the fragment analyses is acceptable, in terms of both space and time.

7 Related Work

In Harrold and Rothermel's separate pointer alias analysis [8], a software module is analyzed separately and later linked with other modules. The analysis is based

Table 5. Analysis Time and Space

Fragment	WholePrgm FI	Basic FFS	Extended FFS	Basic Space	Extended Space
`zip.1`	0:02	1:01	1:35	2544	3784
`zip.2`	0:02	0:18	0:18	2064	2064
`sc.1`	0:02	0:18	0:18	2504	2504
`sc.2`	0:02	0:25	0:33	2664	2824
`larn.1`	0:03	0:22	0:27	3760	6560
`larn.2`	0:03	0:20	0:28	3680	6556
`espresso.1`	0:07	0:52	1:58	5504	17400
`espresso.2`	0:07	0:56	3:07	5504	26904
`tsl.1`	0:08	0:33	0:43	5776	8672
`tsl.2`	0:08	1:15	1:25	12480	18648
`moria.1`	0:09	1:22	1:29	9504	10464
`moria.2`	0:09	1:39	1:37	7608	17440

on the whole-program analysis from [10]; it simulates the aliasing effects that are possible under all calling contexts for the module. Placeholder variables are used to represent sets of variables that are not explicitly referenced in the module, similarly to the placeholder variables in our fragment pointer alias analyses. Aliases are assigned extra tags describing the module calling context.

There are several differences between our work and [8]. First, we have designed a general framework for fragment analysis and emphasized the importance of the theoretical requirements that ensure analysis safety and feasibility. Second, the intended application of our fragment pointer alias analyses is to improve the information about a part of the program after an inexpensive whole-program analysis; the application in [8] is separate analysis of single-entry modules. Finally, [8] does not present empirical evaluation of the performance of the analysis; we believe that their approach may have scalability problems.

Reference [11] presents an analysis that decomposes the program into regions in which several local problems are solved. Representative values are used for actual data-flow information that is external to the region; our placeholder variables are similar to these representative values. Other similar mechanisms are the non-visible names from [10, 7], extended parameters from [20], and unknown initial values from [4]. We use an abstraction relation to capture the correspondence between the representative and the actual data-flow information; this is similar to the use of abstraction relations in the field of abstract interpretation [19, 6].

The work in [3] also addresses the analysis of program fragments and uses the notion of representative data-flow information for external data-flow values. However, in [3] the specific fragments are libraries with no boundary calls, the analysis computes def-use associations in object-oriented languages with exceptions, and there is no assumption of available whole-program information.

Cardelli [2] considers separate type checking and compilation of program fragments. He proposes a theoretical framework in which program fragments are separately compiled in the context of some information about missing fragments, and later can be safely linked together.

Model checking is a technique for verifying properties of finite-state systems; the desired properties are specified using temporal-logic formulae. Modular model checking verifies properties of system modules, under some assumptions about the environment with which the module interacts [9]. These assumptions play an analogous role to that of the whole-program information in fragment analysis. Further discussion of the relationship between data-flow analysis and model checking is given in [14].

There is some similarity in the problem addressed by our work and that in [5], which presents an analysis of modular logic programs for which a compositional semantics is defined. Each module can be analyzed using an abstract interpretation of the semantics. The analysis results for separate modules can be composed to yield results for the whole program, or alternatively, the results for one module can be used during the analysis of another module.

8 Conclusions

This paper is a first step in investigating the theory and practice of fragment data-flow analysis. It proposes fragment analysis as an alternative to traditional whole-program analysis. The theoretical issues involved in the design of safe and feasible flow-sensitive fragment analysis are discussed. One possible application of fragment analysis is to be used after a whole-program flow-insensitive analysis in order to improve the precision for an interesting portion of the program. This paper presents one such example, in which better information about modifications through pointer dereference is obtained by performing flow- and context-sensitive fragment pointer alias analysis. The design of two such analyses is described, and empirical results evaluating their cost and precision are presented.

The empirical results show that the extended analysis presented in Sect. 5 can achieve significant precision benefits at a practical cost. The performance of the basic analysis is less satisfactory, even though in about half of the cases it achieves the precision of the extended analysis. Clearly, in some cases the whole-program flow-insensitive solution is not a precise enough estimate of the effects of calls to external procedures. The extended analysis solves this problem for the fragments used in our experiments. We expect this approach to work well in cases when the extended fragment is not much bigger than the original fragment. One typical example would be a fragment with calls only to relatively small external procedures which provide simple services; in our experience, this is a common situation. However, in some cases the extended fragment may contain a prohibitively large part of the program; furthermore, the source code for some procedures may not be available. We are currently investigating different solutions to these problems.

Acknowledgments. We would like to thank Thomas Marlowe, Matthew Arnold, and the anonymous reviewers for their valuable comments.

References

1. D. Atkinson and W. Griswold. Effective whole-program analysis in the presence of pointers. In *Proc. Symp. on the Foundations of Software Engineering*, pages 46–55, 1998.
2. L. Cardelli. Program fragments, linking and modularization. In *Proc. Symp. on Principles of Prog. Lang.*, pages 266–277, 1997.
3. R. Chatterjee and B. G. Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-382, Rutgers University, 1999.
4. R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Proc. Symp. on Principles of Prog. Lang.*, pages 133–146, 1999.
5. M. Codish, S. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. Symp. on Principles of Prog. Lang.*, pages 451–464, 1993.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Proc. Symp. on Principles of Prog. Lang.*, pages 238–252, 1977.
7. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. Conf. on Prog. Lang. Design and Implementation*, pages 242–257, 1994.
8. M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7):442–460, July 1996.
9. O. Kupferman and M. Y. Vardi. Modular model checking. In *Proc. Symp. on Compositionality*, LNCS 1536, pages 381–401, 1997.
10. W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proc. Conf. on Prog. Lang. Design and Implementation*, pages 235–248, 1992.
11. T. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proc. Symp. on Principles of Prog. Lang.*, pages 184–196, 1990.
12. T. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.
13. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. Symp. on Principles of Prog. Lang.*, pages 49–61, 1995.
14. D. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc. Symp. on Principles of Prog. Lang.*, pages 38–48, 1998.
15. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. Symp. on Principles of Prog. Lang.*, pages 1–14, 1997.
16. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.
17. B. Steensgaard. Points-to analysis in almost linear time. In *Proc. Symp. on Principles of Prog. Lang.*, pages 32–41, 1996.
18. P. Stocks, B.G. Ryder, W. Landi, and S. Zhang. Comparing flow- and context-sensitivity on the modification side-effects problem. In *Proc. International Symposium on Software Testing and Analysis*, pages 21–31, 1998.
19. R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.
20. R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. Conf. on Prog. Lang. Design and Implementation*, pages 1–12, 1995.
21. S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proc. Symp. on the Foundations of Software Engineering*, pages 81–92, 1996.