On The Complexity of Function Pointer May-Alias Analysis*

Robert Muth Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{muth, debray}@cs.arizona.edu

October 25, 1996

Abstract

This paper considers the complexity of interprocedural function pointer may-alias analysis, i.e., determining the set of functions that a function pointer (in a language such as C) can point to at a point in a program. This information is necessary, for example, in order to construct the control flow graphs of programs that use function pointers, which in turn is fundamental for most dataflow analyses and optimizations. We show that the general problem is complete for deterministic exponential time. We then consider two natural simplifications to the basic (precise) analysis and examine their complexity. The approach described can be used to readily obtain similar complexity results for related analyses such as reachability and recursiveness.

1 Introduction

Recent years have seen a great deal of interest in interprocedural compile-time analyses and optimizations (see, for example, [CBC93, LR92, LRZ93, Mye81, SP81, WL95]). Fundamental to any such effort is the determination of interprocedural control flow. In the presence of function pointers (or procedure-valued variables) this requires the determination of the set of functions that a function pointer may point to at any program point, i.e., the set of its possible aliases. In this paper, we examine the theoretical complexity of this problem, which we refer to as the *interprocedural function pointer may-alias analysis* (*FP-MayAlias*).

The problem of determining interprocedural control flow in the presence of procedure-valued arguments was first investigated in the context of call (multi)graph construction for Fortran programs, which do not allow functions to return procedure values [Ryd79, CCHK90]. More recently, Shivers uses abstract

* This work was supported in part by the National Science Foundation under grant number CCR-9502826.

interpretation to examine the problem in the context of higher-order languages such as Scheme [Shi88, Shi91]: His major concerns are with semantic aspects of the problem. Lakhotia studies the general problem for a language where procedures may be assigned to variables, invoked through variables, and returned as results [Lak93]. Lakhotia gives a polynomial time algorithm whose efficiency gains come at the cost of considerable imprecision in the analysis.

Zhang and Ryder [ZR94] examine the complexity of interprocedural function pointer may-alias analysis for the programming language C. They are the first to define, in a precise way, what it means for such an analysis to be *precise*,[1] and consider the complexity of the problem with respect to the presence or absence of various program constructs, such as global function pointers, assignment to function pointers, invocation through function pointers, etc. They show that while polynomial-time algorithms exist for precise solutions to the problem in the presence of some restricted combinations of such program constructs, the problem is, in most cases, NP-hard.

This paper examines in detail the computational complexity of a number of variations on interprocedural function pointer may-alias analysis. We first show that the computation of precise solutions requires the use of the relational attributes method [JM81], which, in turn, implies NP-hardness even in the absence of function calls, and show that the problem is complete for deterministic exponential time. We then examine two natural ways to simplify the analysis at the cost of precision. The first is to use an independent attributes analysis [JM81], i.e., ignore dependences between the aliases of different variables. Somewhat surprisingly, the problem remains EXPTIME-complete in this case: the simplification produces no improvement in the theoretical worst-case complexity. The second simplification is to abandon context information for function calls and resort to what Shivers refers to as 0-CFA (zeroth-order control flow analysis). It turns out that this simplification admits polynomial-time algorithms, though potentially at the cost of considerable sacrifice in precision.

The remainder of this paper is organized as follows. Section 2 discusses background information and defines the *FP-MayAlias* problem. Section 3 discusses the problem of obtaining precise solutions to this problem. Section 4 considers the complexity of function pointer alias analysis using the independent attribute method, and Section 5 considers a further sacrifice in precision involving context-insensitive analysis. Section 6 briefly considers the related problem of function pointer must-alias analysis. Finally, Section 7 concludes.

2 Preliminaries

For code analysis and optimization purposes, compilers typically construct a control flow graph for each function in a program [ASU86]. This is a directed graph where each node represents a segment of executable code that has a single

[1] The determination of whether some (nontrivial) property will actually hold at a particular program point at runtime is, of course, undecidable. A standard assumption in the dataflow analysis literature is that both branches of a conditional can be executed: this usually suffices to sidestep the problem of undecidability, and "precision" of program analyses is typically defined with respect to this assumption.

entry point and a single exit point, and where there is an edge from a node A to a node B if and only if it is possible for execution to leave node A and immediately enter node B. If there is an edge from a node A to a node B, then A is said to be a *predecessor* of B and B is a *successor* of A; the set of all predecessors of a node A is denoted by $pred(A)$, while the set of all successors of A are denoted by $succ(A)$. For a node with a single predecessor, we abuse notation and use $pred(A)$ to refer to the predecessor itself rather than the singleton set containing the predecessor, and similarly with successors.

Control flow graphs in the traditional sense describe the flow of control within a function, but do not account for control flow across function boundaries. An interprocedural control flow graph (ICFG) for a program consists of the control flow graphs of all the functions in the program, together with edges representing calls and returns that link the flow graphs of different functions. A function call is represented using a pair of nodes, a *call node* and a *return node*: the successors of a call node consist of the corresponding return node together with the entry node of each function that can be called from that node (in the case of indirect calls through function pointers, there is an edge to the entry node of each function in the program), while the predecessors of a return node consist of the corresponding call node together with the exit node of each function that could have been called from that call node. The function that is called from a call node n is denoted by $callee(n)$. To prove that a property holds at a program point, an analysis must consider *statically executable* paths from the entry point of the program upto that point: roughly speaking, these are paths that can actually be taken during execution, modulo the assumption (standard in dataflow analysis) that both branches of a conditional can be taken [ZR94]. More formally, these paths can be defined as follows:

Definition 2.1 [Statically executable path] A path starting at the root[2] of the interprocedural control flow graph is *statically executable* if it satisfies the following two conditions:

1. The path has a *proper* subpath containing all the return nodes. A path is *proper* if
   - It contains no return and no call nodes.
   - It is the concatenation of two proper paths.
   - Its first call node c and its last return node r stem from the same call site; the successor c' of c and the predecessor r' of r belong to the same function; and the path from c' to r' is also proper.
2. For each indirect call edge through a function pointer x to a function f, x must point to f, i.e., the last assignment to x along the path must have been to f.

∎

[2] In the sequel we will call this root $entry(\texttt{main})$

Definition 2.2 [Function Pointer May Aliasing Problem] Given a node n in the ICFG and a variable v the function pointer may aliasing problem is to find all procedures p so that there is a statically executable path from the entry point of the program to the node n at the end of which v points to p. ∎

We write $[n, \langle v, p\rangle]$ to indicate that v may be aliased to p, i.e., may point to p, at a program point n. An interprocedural function pointer may-alias analysis is said to be *precise* if, for each program point n of each program P, the set of aliases it computes is exactly the set $[n, \langle v, p\rangle]$. While an analysis may not be precise in general, it is required to be *safe*, i.e., compute at least those aliases that hold at each program point. We will show that the problem of precise function pointer may-alias analysis is complete for the complexity class EXPTIME, i.e., deterministic exponential time, which is defined as EXPTIME $= \bigcup_{c \geq 0}$ DTIME$[2^{n^c}]$.

We use the following notation in the remainder of the paper. The powerset of a set S is denoted by $\mathcal{P}(S)$, the n-fold Cartesian product of S with itself is denoted by S^n, the set of monotone functions from S^n to S—assuming that S is ordered—is denoted by $[S^n \to S]$. If S forms a (complete) lattice under a partial order $\sqsubseteq$, with meet and join operations $\sqcap$ and $\sqcup$, then S^n and $[S^n \to S]$ also form (complete) lattices with $\sqsubseteq$, $\sqcap$ and $\sqcup$ extended componentwise and pointwise in the obvious way. Given a recursive equation $f(\bar{x}) = \mathcal{E}(f, \bar{x})$ over a complete lattice $(S, \sqsubseteq)$ with meet and join operations $\sqcap$ and $\sqcup$ respectively, let $\tau(f) : S \to S$ be the functional corresponding to the right hand side of this equation: if τ is monotone and continuous (note that a monotone function over a finite(-height) lattice is necessarily continuous), then from the Knaster-Tarski fixpoint theorem [Tar55], it has a least fixpoint given by $\bigsqcup_{i \geq 0}\{\tau^i(\bot) \mid i \geq 0\}$, where $\bot$ is the least element of S and τ^i denotes the i-times iterated unfolding of τ. We use $lfp(f)$ and sometimes f^* to denote this least fixpoint. Finally, $f[a \mapsto b]$ denotes the function that coincides with f except at a, where it evaluates to b: $f[a \mapsto b] \triangleq \lambda x.\textbf{if } x = a \textbf{ then } b \textbf{ else } f(x)$.

Since we focus purely on the problem of function pointer aliasing, to simplify the discussion we explicitly disregard issues that do not bear directly on this. In particular, we assume that there are no arrays or records, nor any reference parameters or pointer-induced aliasing (except for aliases due to function pointers).

For notational simplicity in the discussion that follows, we assume that programs obey the following syntactic restrictions. We assume that all functions have the same set of local variable names, denoted by Var, and the same set of formal parameters $\mathsf{Fml} = \{fml_1, \ldots, fml_k\} \subseteq \mathsf{Var}$. These formals are assumed to be read-only, i.e., they cannot be changed within a function. This makes it easier to match up environments at the entry to, and exit from, a function, and can be easily met by copying formals to other local variables where necessary. Additionally, each function is assumed to have a special variable $ret \in \mathsf{Var}$: the value returned by the function is loaded into this variable before control returns to its caller. To model parameter passing, we assume that each function has a special set of variables $\mathsf{Arg} = \{arg_1, \ldots, arg_k\} \subseteq \mathsf{Var}$, and that the value of the

i^{th} argument is assigned to arg_i before a function call ($1 \leq i \leq k$). Additionally, each function is assumed to have a special variable $res \in \mathsf{Var}$: whenever a function calls another function, the result of the function call is assumed to be assigned to this variable when control returns to the caller. Finally, it is assumed that the flow graph for each function f has distinguished entry and exit nodes, $entry(f)$ and $exit(f)$ respectively, where execution enters f and leaves f.

We sidestep the issue of indirect calls through an undefined function pointer variable by assuming that there is a special function `nil` $\in$ Fun, where Fun denotes the set of function names in a program, that always returns a pointer to itself. Initially, all variables are assumed to be initialized to point to `nil`. The entry point of a program is a distinguished function `main` $\in$ Fun. We assume that there are no global variables. This restriction is primarily to simplify our dataflow equations: it is straightforward to extend the equations to take globals into account, but this does not shed any additional insight into complexity issues relating to this analysis or affect our results in any way.

3 Precise Function Pointer Alias Analysis

3.1 Relational Attributes vs. Independent Attributes

Program analysis involves keeping track of (descriptions of) the values different variables can take on at different program points. In general, the values of different variables may depend on each other. When tracking the values that variables can take on, we may choose to keep track of such dependencies (leading to analysis information of the form "$[n, \langle x, a\rangle]$ and $[n, \langle y, b\rangle]$; or $[n, \langle x, c\rangle]$ and $[n, \langle y, d\rangle]$ "), or we may choose to ignore such dependencies (leading to information of the form "$[n, \langle x, a\rangle]$ or $[n, \langle x, c\rangle]$; and $[n, \langle y, b\rangle]$ or $[n, \langle y, d\rangle]$ "). Jones and Muchnick refer to the former kind of analysis as the *relational attributes* method, and the latter kind as the *independent attributes* method [JM81]. In practice, program analyses typically use the independent attributes method because it tends to be simpler and more efficient to implement.

In the context of function pointer may-alias analysis, a precise analysis algorithm cannot use the independent attributes method in general. This is illustrated by the following example:

Example 3.1 Let `PF` denote the type of a pointer to a function that takes an argument of type `PF` and returns a result of type `PF`.[3] Consider the following program:

```
PF id(PF x) { return x; }
PF nil(PF x) { return &nil; }
main()
{
  PF z;
```

[3]This recursive type cannot be properly expressed in C, though it is possible to use void pointers and casting to achieve the same results. To simplify the presentation, however, we will use `PF` to refer to such pointers.

```
    if (...) { x = &id; y = &nil; }
    else     { x = &nil; y = &id; }
    z = (*x)(y);
    ...
}
```

It is not difficult to determine that, regardless of which branch of the conditional is taken, the value assigned to `z` must be a pointer to `nil`. However, an independent attribute analysis would determine the set of possible aliases for both `x` and `y`, at the point immediately after the conditional, to be {`id`, `nil`}. Then, when considering the indirect call `(*x)(y)` we would be forced to consider the possibility that both `x` and `y` are pointers to `id`, implying that a possible value that could be assigned to `z` is a pointer to `id`. This is imprecise, and the imprecision is due solely to the fact that the connection between the aliases of different variables is lost during an independent attributes analysis. ■

3.2 A Framework for Function Pointer May-Alias Analysis

As Example 3.1 illustrates, a precise analysis requires what Jones and Muchnick have referred to as a *relational attributes* analysis, i.e., where connections between the possible aliases of different variables are maintained [JM81]. We will keep track of such connections using *environments*, which map local variables to the functions they are aliased to (point to). Environments are finite maps; an environment of the form $[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ represents the function

$$\lambda x.\mathbf{if}\ x = a_1\ \mathbf{then}\ b_1;\ \cdots;\ \mathbf{else\ if}\ x = a_n\ \mathbf{then}\ b_n;\ \mathbf{else}\ \mathtt{nil}$$

The set of environments is $\mathsf{Env} = \mathsf{Var} \to \mathsf{Fun}$. The function $\mathit{Lookup} : (\mathsf{Var} \cup \mathsf{Fun}) \times \mathsf{Env} \to \mathsf{Fun}$ evaluates the expressions in call and assignment nodes. An expression can either be a variable or a constant:

$$\mathit{Lookup}(expr, env) = \begin{cases} expr & \mathtt{if}\ expr \in \mathsf{Fun} \\ env(expr) & \mathtt{if}\ expr \in \mathsf{Var} \end{cases}$$

The dataflow analysis associates, with each node n in the ICFG, an element $\mathsf{AEnv}(n) \in \mathcal{P}(\mathsf{Env})$. Since all variables are undefined, and hence assumed to be initialized to `nil` at the entry to the program (see Section 2), for the root node r $(= entry(\mathtt{main}))$ of the ICFG we set $\mathsf{AEnv}(r) = \{\lambda x.\mathtt{nil}\}$. The environments for the other nodes are defined via dataflow equations as follows:

1. n is the entry node for a function f. Let $\mathit{CallEnv}(n, f)$ be a the subset of environments currently associated with call node n that could have caused execution to enter function f:

   $$\mathit{CallEnv}(n, f) = \{e \in \mathsf{AEnv}(n) \mid f = \mathit{Lookup}(callee(n), e)\}.$$

   Then, $\mathsf{AEnv}(n)$ is given by

   $$\mathsf{AEnv}(n) = \textstyle\bigcup_{p \in pred(n)} \{[fml_1 \mapsto e(arg_1), \ldots, fml_k \mapsto e(arg_k)] \mid e \in \mathit{CallEnv}(p, f)\}.$$

2. n is an assignment node '`x := u`'. The only effect of this is to update the binding of x in the environment to the value(s) denoted by u:

$$\mathsf{AEnv}(n) = \bigcup_{p \in pred(n)} \{e[x \mapsto \mathit{Lookup}(u, e)] \mid e \in \mathsf{AEnv}(p)\}.$$

3. n is a return node for a function call. Let n' be the call node corresponding to n. The possible return values can be obtained from the environments at the exit nodes of the functions called by n'. However, we have to make sure that we consider only realizable paths: this can be done by considering only those environments at the exit nodes whose formals match the arguments at the call node n'. Therefore, we define:

$$\mathit{ReturnEnv}(n, e) = \{e' \in \mathsf{AEnv}(exit(f)) \mid f = \mathit{Lookup}(\mathit{callee}(n), e) \wedge \bigwedge_{1 \le i \le k} e(arg_i) = e'(\mathit{fml}_i)\}.$$

$\mathsf{AEnv}(n)$ is then simply the set of environments at the call node n' appropriately updated with the values that could be returned by the called function:

$$\mathsf{AEnv}(n) = \{e[res \mapsto e'(ret)] \mid e \in \mathsf{AEnv}(n') \;\wedge\; e' \in \mathit{ReturnEnv}(n', e)\}$$

4. n is a conditional node, an exit node, or a call node. In each case, $\mathsf{AEnv}(n)$ is obtained by copying the environments of the only predecessor node:

$$\mathsf{AEnv}(n) = \mathsf{AEnv}(pred(n)).$$

5. n is a junction node. In this case, $\mathsf{AEnv}(n)$ is obtained as the union of its predecessors' envoronments:

$$\mathsf{AEnv}(n) = \bigcup_{p \in pred(n)} \mathsf{AEnv}(p).$$

The equation for the entry nodes make sure that not all possible function arguments are considered but only those that can actually happen during execution. This essentially resembles the minimal function graphs approach of [JM86].

We use AEnv^* to denote the least fixpoint of the system of equations given above for AEnv . Since the sets Var and Fun are finite, so is the set $\mathsf{Env} = \mathsf{Var} \rightarrow \mathsf{Fun}$. This implies that $(\mathcal{P}(\mathsf{Env})\ , \subseteq)$ is a finite lattice, and therefore that $\mathsf{AEnv}^* \in \mathcal{P}(\mathsf{Env})$ can be computed using the iterative algorithm shown below.

Algorithm 3.1

for *all nodes* n **do**
  if $n = r$ **then** $\mathsf{AEnv}(n) = \{\lambda x.\mathtt{nil}\}$ **else** $\mathsf{AEnv}(n) = \emptyset$
repeat
  for *all nodes* n *except* r **do in parallel**
    recompute $\mathsf{AEnv}(n)$ *from the* AEnv *value(s) of the predecessor(s) of* n
until *there is no change to* $\mathsf{AEnv}(n)$ *for any node* n

The fixpoint captures the aliasing behavior of the program precisely (upto the standard assumptions of dataflow analysis):

Lemma 3.1 *The precise set of aliases at any program point n in a program is given by* $\mathsf{AEnv}^*(n)$. *In other words, for any point n in a program,* $[n, \langle v, p\rangle]$ *if and only if* $\exists e \in \mathsf{AEnv}^*(n) : e(v) = p$.

Proof The proof that $[n, \langle v, p\rangle]$ implies $\exists e \in \mathsf{AEnv}^*(n) : e(v) = p$ is by induction on the length of statically executable paths leading upto n. The other direction is by fixpoint induction on the equations defining AEnv. ∎

Theorem 3.1 *FP-MayAlias* $\in$ EXPTIME.

Proof We show that AEnv^* can be computed in time $O(n^3 \cdot (f^v)^2) = O(n^3 \cdot 2^{2n \log n})$, where n is the number of nodes in the ICFG,$f = |\mathsf{Fun}|$, and $v = |\mathsf{Var}|$. We assume that each set $\mathsf{AEnv}(n)$ is represented by a bitvector of length f^v. Since each bit once set to 1 will never change back to 0 there can be at most $n \cdot f^v$ iterations. In each iteration we have to consider n nodes. The most costly operation is the computation for a junction node which is bounded above by $O(n \cdot f^v)$. Furthermore, we have $f = O(n)$, and $v = O(n)$. Since the complexity class EXPTIME is defined as EXPTIME $= \cup_{c \geq 0}$DTIME$[2^{n^c}]$, the theorem follows.

∎

3.3 *FP-MayAlias* is EXPTIME-Hard

Theorem 3.1 indicates that in the worst case, time that is exponential in the size of the input program is sufficient for the *FP-MayAlias* problem. In this section, we show that this analysis problem is hard for deterministic exponential time, i.e., it may require, in the worst case, time that is (at least) exponential in the input size. Our proof is by reduction from a problem of evaluating recursive monotone Boolean functions over the lattice $\mathcal{B} = \{\mathbf{0}, \mathbf{1}\}$, the boolean lattice with $\mathbf{0} \sqsubseteq \mathbf{1}$, and meet and join operations $\sqcap$ and $\sqcup$.

Definition 3.1 [Recursive monotone boolean function (RMBF)]
A recursive monotone boolean function (RMBF) is an equation

$$F(x_1, \ldots, x_k) = expr$$

where *expr* is recursively defined by the following BNF productions:

$$expr ::= \mathbf{0} \mid \mathbf{1} \mid x_i (1 \leq i \leq k) \mid expr \wedge expr \mid expr \vee expr \mid F(expr, \ldots, expr)$$

expr induces a monotone and continous functional τ_1 on $[\mathcal{B}^k \to \mathcal{B}]$.[4] The function denoted by the equation is then its least fixpoint $\mathit{lfp}(F)$ in $[\mathcal{B}^k \to \mathcal{B}]$. ∎

[4] Here $\mathbf{0}$,$\mathbf{1}$, resp. x_i are abbreviations for $\lambda\vec{x}.\mathbf{0}$, $\lambda\vec{x}.\mathbf{1}$, resp. $\lambda\vec{x}.x_i$

Definition 3.2 [RMBF Problem]
Given a pair $(eq, \vec{z})$ where eq is a RMBF and $\vec{z} \in \mathcal{B}^k$ the RMBF problem is to evaluate $(lfp(F))(\vec{z})$. ∎

Theorem 3.2 (Hudak and Young [HY86])
The RMBF *problem is* EXPTIME*-complete in the length of the pair* $(eq, \vec{z})$.

Given an instance $\varphi = (eq, \vec{z})$ of the RMBF problem, our strategy will be to generate a program P_φ such that the results of function pointer alias analysis on P_φ can be used to solve φ. (The generation of the corresponding ICFG is straightforward). Given any numbering of the syntax tree of eq that assigns distinct numbers to distinct nodes, let the subtree of the syntax tree rooted at the node numbered ℓ be denoted by E_ℓ and let ℓ_r be the number of the root node. Then, the program P_φ is defined as follows:

1. It contains the definitions

   ```
   typedef PF ...;
   PF nil(PF arg) {return &nil;}
   PF id(PF arg) {return arg;}
   ```

   Here, `PF` is a pointer to a function that returns a result of type `PF` and takes an argument of type `PF` (see Example 3.1).

2. Corresponding to each subexpression E_ℓ of the body of the recursive equation eq, there is a C function $\mathtt{f}_\ell$, defined as follows:

   (a) If $E_\ell \equiv \mathbf{1}$ then $\mathtt{f}_\ell$ is: `PF` $\mathtt{f}_\ell$`(PF x1,...,PF xk) {return &id;}`

   (b) If $E_\ell \equiv \mathbf{0}$ then $\mathtt{f}_\ell$ is: `PF` $\mathtt{f}_\ell$`(PF x1,...,PF xk) {return &nil;}`

   (c) If $E_\ell \equiv x_i$ then $\mathtt{f}_\ell$ is: `PF` $\mathtt{f}_\ell$`(PF x1,...,PF xk) {return x`i`;}`

   (d) If $E_\ell \equiv E_{\ell_1} \wedge E_{\ell_2}$ then $\mathtt{f}_\ell$ is:

   `PF` $\mathtt{f}_\ell$`(PF x1,...,PF xk)`
   `{return` $\mathtt{f}_{\ell_1}$`(x1,...,xk)(`$\mathtt{f}_{\ell_2}$`(x1,...,xk));}`

   (e) If $E_\ell \equiv E_{\ell_1} \vee E_{\ell_2}$ then $\mathtt{f}_\ell$ is:

   `PF` $\mathtt{f}_\ell$`(PF x1,...,PF xk)`
   `{return (...)?` $\mathtt{f}_{\ell_1}$`(x1,...,xk):` $\mathtt{f}_{\ell_2}$`(x1,...,xk);}`

   (f) If $E_\ell \equiv F(E_{\ell_1}, ..., E_{\ell_k})$ then $\mathtt{f}_\ell$ is:

   `PF` $\mathtt{f}_\ell$`(PF x1,...,PF xk)`
   `{return` $\mathtt{f}_{\ell_r}$ `(`$\mathtt{f}_{\ell_1}$`(x1,...,xk),...,`$\mathtt{f}_{\ell_k}$`(x1,...,xk));}`

3. Let $\vec{z'}$ be obtained from $\vec{z}$ by mapping $\mathbf{1}$ to `id` and $\mathbf{0}$ to `nil` componentwise. Then, the entry point for P_φ is defined by the C function

   `void main() { PF result =` $\mathtt{f}_{\ell_r}$`(&`z'_1`,...,&`z'_n`); }`

Example 3.2 Consider the RMBF instance

$$\varphi = (F(x_1, x_2, x_3) = F(F(x_1, x_3, x_2), x_3 \vee x_2, \mathbf{1}) \wedge x_1, (\mathbf{1}, \mathbf{0}, \mathbf{1}))$$

Its syntax tree, with the (preorder) number of each node shown next to the node, together with the program P_φ corresponding to φ, is shown in Figure 1. ■

The following result is straightforward:

Lemma 3.2 *Given any instance φ of* RMBF *, the* ICFG *for program P_φ can be generated in time polynomial in $|\varphi|$.*

Since aliases in the programs so generated are generated through function calls only, variables can point only to `nil` and/or `id`, and the aliases of a particular incarnation of a variable never changes, we can use a somewhat simpler approach for the analysis than that outlined in Section 3.2. The following theorem establishes the relationship between the alias analysis and the solution of the RMBF problem. The rest of the section is devoted to its proof.

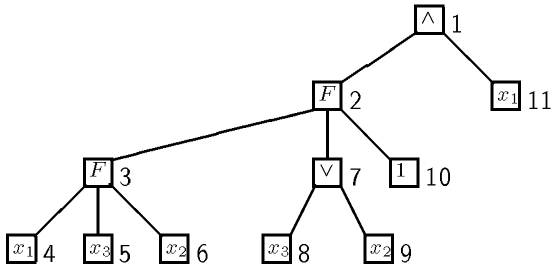
Theorem 3.3 (Main Theorem)
Let $\varphi = (eq, \vec{z})$ be an RMBF *problem and P_φ the corresponding program generated by our reduction. Then,*

$$(\mathit{lfp}(F))(\vec{z}) = \mathbf{1} \quad \textit{if and only if} \quad [\mathit{exit}(\texttt{main}), \langle \texttt{result}, \texttt{id} \rangle] \quad \textit{holds in } P_\varphi$$

In order to prove Theorem 3.3, it suffices to focus on the possible return values of functions. This motivates the definition of the mapping $\mathsf{AFunc} : \mathsf{Fun} \to \mathcal{P}(\mathsf{Fun})^{\ k} \to \mathcal{P}(\mathsf{Fun})$ that models the aliasing behavior of an entire function. $\mathsf{AFunc}(f)$ maps argument aliases of f into return aliases of f. $\mathsf{AFunc}(f)$ is defined by a system of recursive equations, one equation for each function $\mathtt{f}_\ell$ corresponding to the subexpression E_ℓ, as given in Table 1, with the binary operation $\star$ is defined as follows:

$$\begin{aligned} a \star b \triangleq\ & \texttt{if } (a = \emptyset \vee b = \emptyset) \texttt{ then } \emptyset \\ & \texttt{elseif } (a = \{\texttt{nil}\} \vee b = \{\texttt{nil}\}) \texttt{ then } \{\texttt{nil}\} \\ & \texttt{else } a \cup b. \end{aligned}$$

Let $\mathcal{L}$ be the lattice $(\mathcal{P}(\{\texttt{nil}, \texttt{id}\})\ , \subseteq)$. All the functions occuring in the system of equations defining AFunc are in $[\mathcal{L}^k \to \mathcal{L}]$, i.e., are monotone functions over a finite complete lattice. These equations therefore have a least fixpoint, which we denote by AFunc^*. Furthermore, we can reduce this system of equations (by successive substitution) to a single recursive equation in $\mathsf{AFunc}(\mathtt{f}_{\ell_r})$ The syntax tree of this equation is isomorphic to that for eq: only the labels are different, but they correspond as follows: a node labelled $\mathbf{0}$ in the tree for eq corresponds to a node $\lambda\vec{x}.\{\texttt{nil}\}$ in the tree for the equation for $\mathsf{AFunc}(\mathtt{f}_{\ell_r})$; $\mathbf{1}$ corresponds to $\lambda\vec{x}.\{\texttt{id}\}$; x_i corresponds to $\lambda\vec{x}.x_i$; $\wedge$ corresponds to $\star$; $\vee$ corresponds to $\cup$; and a node labelled $F(\cdots)$ corresponds to one labelled $\mathsf{AFunc}(\mathtt{f}_{\ell_r})$. The functional τ_2




```
typedef PF ...;
PF nil(PF arg) {return &nil;}
PF id(PF arg)  {return arg;}
void main() { PF result = f1(&id,&nil,&id) }

PF f1(PF x1, PF x2, PF x3)
{   return (f2(x1,x2,x3))( f11(x1,x2,x3) ); }
PF f2(PF x1, PF x2, PF x3)
{   return f1( f3(x1,x2,x3),f7(x1,x2,x3),f10(x1,x2,x3) ); }
PF f3(PF x1, PF x2, PF x3)
{    return f1( f4(x1,x2,x3),f5(x1,x2,x3),f6(x1,x2,x3) ); }
PF f4(PF x1, PF x2, PF x3)
{    return x1;  }
PF f5(PF x1, PF x2, PF x3)
{    return x3;  }
PF f6(PF x1, PF x2, PF x3)
{    return x2;  }
PF f7(PF x1, PF x2, PF x3)
{   return (...) ? f8(x1,x2,x3) : f9(x1,x2,x3); }
PF f8(PF x1, PF x2, PF x3)
{   return x3; }
PF f9(PF x1, PF x2, PF x3)
{    return x2;  }
PF f10(PF x1, PF x2, PF x3)
{    return &id;  }
PF f11(PF x1, PF x2, PF x3)
{    return x1;  }
```

Figure 1: The syntax tree and generated program for Example 3.2

E_ℓ	Equation corresponding to $\mathtt{f}_\ell$
$\mathbf{1}$	$\mathsf{AFunc}(\mathtt{f}_\ell) = \lambda\vec{x}.\{\mathtt{id}\}$
$\mathbf{0}$	$\mathsf{AFunc}(\mathtt{f}_\ell) = \lambda\vec{x}.\{\mathtt{nil}\}$
x_i	$\mathsf{AFunc}(\mathtt{f}_\ell) = \lambda\vec{x}.x_i$
$E_{\ell_1} \wedge E_{\ell_2}$	$\mathsf{AFunc}(\mathtt{f}_\ell) = \mathsf{AFunc}(\mathtt{f}_{\ell_1}) \star \mathsf{AFunc}(\mathtt{f}_{\ell_2})$
$E_{\ell_1} \vee E_{\ell_2}$	$\mathsf{AFunc}(\mathtt{f}_\ell) = \mathsf{AFunc}(\mathtt{f}_{\ell_1}) \cup \mathsf{AFunc}(\mathtt{f}_{\ell_2})$
$F(E_{\ell_1}, \ldots, E_{\ell_k})$	$\mathsf{AFunc}(\mathtt{f}_\ell) = \mathsf{AFunc}(\mathtt{f}_{\ell_r})(\mathsf{AFunc}(\mathtt{f}_{\ell_1}), \ldots, \mathsf{AFunc}(\mathtt{f}_{\ell_k}))$

Table 1: Equations for AFunc

represented by the right hand side of the resulting equation allows us to express AFunc^* as $\sqcup\,\{\tau_2^{<i>}(\bot_2) \mid i \geq 0\}$ where $\bot_2 = \lambda\vec{x}.\emptyset$. Since $[\mathcal{L}^k \to \mathcal{L}]$ is a finite lattice, it follows that $\mathsf{AFunc}^* = \tau_2{}^{<k>}(\bot_2)$ for some finite k.

$\mathsf{AFunc}^*(f)$ is closely related to the set $\mathsf{AEnv}^*(exit(f))$: the set of function pointers that can be returned by a function f, as determined by $\mathsf{AFunc}^*(f)$, is precisely the set of return aliases for the exit node of f as determined by AEnv^*:

Lemma 3.3 (Relationship between $\mathsf{AFunc}^*(f)$ and $\mathsf{AEnv}^*(exit(f))$)
For any $f, v_1, \ldots, v_k \in \mathsf{Fun}$ with $[fml_1 \mapsto v_1, \ldots, fml_k \mapsto v_k] \in \mathsf{AEnv}^*(init(f))$,

$$\mathsf{AFunc}^*(f)(\{v_1\}, \ldots, \{v_k\}) = \{e(ret) \mid e \in \mathsf{AEnv}^*(exit(f)) \wedge \bigwedge_{i=1}^{k} e(fml_i) = v_i\}.$$

Proof Omitted. ∎

In contrast with the minimal function graph approach for AEnv^* where we were only interested in arguments of each function that could actually occur during program execution, AFunc^* considers all possible arguments. However, the preceding lemma shows that AFunc^* agrees with AEnv^* for those arguments that can occur.

Next we show that given a RMBF instance φ defining a function F, the set of aliases AFunc^* computed for the corresponding program P_φ is essentially equivalent to $lfp(F)$, if we associate aliases to `nil` with $\mathbf{0}$ and aliases to `id` with $\mathbf{1}$: define the function $h : \mathcal{L} \to \mathcal{B}$ as follows:

$$h(x) = \begin{cases} \mathbf{1} & \text{if } \mathtt{id} \in x \\ \mathbf{0} & \text{otherwise} \end{cases}$$

Let $\vec{h} : \mathcal{L}^n \to \mathcal{B}^n$ be the componentwise extension of h. The connection between AFunc^* and $lfp(F)$ can now be made precise via the notion of one function being *faithful* to another. Intuitively, $g : \mathcal{L}^n \to \mathcal{L}$ is faithful to $f : \mathcal{B}^n \to \mathcal{B}$ if $g(\vec{x})$ can return a pointer to the function `id` if and only if $f(\vec{x})$ evaluates to the truth-value $\mathbf{1}$:

Definition 3.3 A function $g \in [\mathcal{L}^n \to \mathcal{L}]$ is *faithful* to a function $f \in [\mathcal{B}^n \to \mathcal{B}]$, written $g \rhd f$, if and only if $f \circ \vec{h} = h \circ g$. ∎

Theorem 3.4 $\mathsf{AFunc}^* \rhd lfp(F)$.

Proof: The proof is by a double induction: the outer level is an arithmetic induction on i, the number of iterations of the functionals corresponding to AFunc and F, while the inner level is a structural induction on the formula E_ℓ and the corresponding expression $\mathsf{AFunc}(\mathbf{f}_\ell)$ obtained after i unfoldings of these functionals. The base case for either induction follows directly from the definitions of $\mathcal{L}$, $\mathcal{B}$, and AFunc; the inductive case uses the straightforward auxiliary results that $\star \rhd \sqcap$ and $\cup \rhd \sqcup$, and that faithfulness is preserved under function composition. ∎

The Main Theorem is an easy corollary of this result:

Corollary 3.1 *FP-MayAlias is* EXPTIME-*complete.*

It is interesting, at this point, to revisit the NP-hardness result for function pointer may-alias analysis due to Zhang and Ryder [ZR94]. As shown in Section 3.1, a relational attributes analysis is necessary for precise function pointer may alias analysis. It turns out that once we have a relational attributes analysis, the problem becomes NP-hard even for the intra-procedural case: in other words, aliasing effects are enough to give rise to NP-hardness, even if we dispense with the additional complications due to interprocedural analysis. This can been seen by a reduction from 3-SAT which we illustrate by an example. Given the 3-SAT problem $(x \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y} \vee z) \wedge (x \vee \bar{y} \vee \bar{z})$ we generate the following program:

```
main()
{
    if (...) {x=&id;nx=&nil}   else   {x=&nil;nx=&id;}
    if (...) {y=&id;ny=&nil}   else   {y=&nil;ny=&id;}
    if (...) {z=&id;nz=&nil}   else   {z=&nil;nz=&id;}

    if (...) c1=x    else if (...) c1=y    else c1=nz;
    if (...) c2=nx   else if (...) c2=ny   else c2=z;
    if (...) c3=x    else if (...) c3=ny   else c3=nz;
}
```

Here `nx,ny,nz` represent the negation of the variables `x,y,z` and `c1,c2,c3` represent the 3 clauses. Each computation path in the first group of `if`-statements corresponds to a truth assignment for the variables of the clause. Each `if`-statement in the second group of statements then corresponds to the evaluation of the truth value of the corresponding clause: the i^{th} clause evaluates to true if and only if there is a computation path through the i^{th} `if`-statement that causes ci to be aliased to `id`. It follows that the original 3-SAT problem is satisfiable if and only if `c1,c2,c3` may be simultaneously aliased to `id` at $exit($`main`$)$.

4 Approximation I: Independent Attributes Analysis

As mentioned in Section 3.1, for pragmatic reasons most program analyses do not use the relational attribute method considered in the previous section: instead, they ignore dependences between the values taken on by different variables in order to improve efficiency. In this section, we consider the complexity of function pointer may-alias analysis based on this simplification. The dataflow framework in this case can be derived from that of Section 3.2 by systematically modifying equations to ignore dependences between variables: we underline identifiers that are changed in this manner to indicate that this has been done. Environments now associate, at each program point, each variable with the set of its aliases: $\underline{\mathsf{Env}} = \mathcal{P}(\mathsf{Fun})^{\mathsf{Var}}$. Small changes are necessary for the function *CallEnv* which becomes

$$\underline{CallEnv}(n, f) = \{e \in \underline{\mathsf{AEnv}}(n) \mid f \in \underline{Lookup}(callee(n), e)\}$$

and the function *ReturnEnv* which becomes

$$\underline{\mathit{ReturnEnv}}(n, e) = \{e' \in \underline{\mathsf{AEnv}}(exit(f)) \mid f \in \underline{\mathit{Lookup}}(callee(n), e) \wedge \bigwedge_{1 \leq i \leq k} e(arg_i) = e'(fml_i)\}.$$

The major change is with the equation for junction nodes where we now "merge" environments with matching formals. As a result there will be at most one environment for every combination of formals at any node.

Next we define two auxiliary function which describe the merging process. $\underline{\mathit{Merge}}$ merges all environments, $\underline{\mathit{Merge}}_V$ only those that agree for the variables in $V \subset \mathsf{Var}$:

$$\underline{\mathit{Merge}}(E) = \{e \in \underline{\mathsf{Env}} \mid \forall\, v \in \mathsf{Var} : \exists\, e' \in E : e'(v) = e(v)\}$$
$$\underline{\mathit{Merge}}_V(E) = \bigcup_{e \in E} \underline{\mathit{Merge}}(\{e' \in E \mid \forall\, v \in V : e'(v) = e(v)\}).$$

The new equation for junction nodes is now:

$$\underline{\mathsf{AEnv}}(n) = \underline{\mathit{Merge}}_{\mathsf{Fml}}(\textstyle\bigcup_{p \in pred(n)} \underline{\mathsf{AEnv}}(p)).$$

Exponential time is still sufficient to solve the relaxed problem. Exponential time is also necessary which can be proven reusing the reduction from section 3.3 and the following lemma, which expresses an intuition very similar to that of Lemma 3.3:

Lemma 4.1 (Relationship between $\mathsf{AFunc}^*(f)$ and $\underline{\mathsf{AEnv}}^*\ (exit(f))$) *For any function f and alias sets $v_1, \ldots, v_k$ with $[fml_1 \mapsto v_1, \ldots, fml_k \mapsto v_k] \in \underline{\mathsf{AEnv}}^*\ (init(f))$,*

$$\mathsf{AFunc}^*(f)(v_1, \ldots, v_k) = \{e(ret) \mid e \in \underline{\mathsf{AEnv}}^*\ (exit(f)) \wedge \bigwedge_{i=1}^{k} e(fml_i) = v_i\}.$$

Proof Omitted. ∎

Corollary 4.1 *Function pointer may-alias analysis remains complete for deterministic exponential time even when the independent attributes method is used.*

This result comes as something of a surprise, since it is usually the case that concessions in the precision of analysis are accompanied by improvements in the complexity of the analysis algorithms. In practice, program analyses usually abandon the relational method in favor of the independent attributes method because the latter tend to be simpler and more efficient. This result indicates, however, that in this case the sacrifice in precision (illustrated in Example 3.1) does nothing to improve the worst case complexity of this analysis problem.

5 Approximation II: Context-Insensitive Analysis

The analysis discussed in the previous section "merges" environments at a node if their formals match, i.e. if they are the result of the same function invocation, but distinguishes between different invocations of the same function.

The completeness result of the last section suggests that there can be an exponential number of different invocations and hence an exponential number of environments at a node, and keeping track of these different invocations can be expensive. Our next approximation will be to merge environments even if they come from different invocations. The effect of this is that the analysis no longer distinguishes between different invocations of a function with different sets of aliases for the formals. As a result, when propagating the results of a function call back to the caller at one point, we also propagate aliases arising from invocations from other program points. In effect, the analysis of a function invocation does not maintain any information about the context from which it arose: for this reason, this has also been referred to as "zeroth-order control flow analysis" (0-CFA) [Hei94, Shi88, Shi91].

We can capture the effects of this approximation by changing the equations for return and entry nodes. The equation for entry nodes becomes:

$$\underline{\mathsf{AEnv}}(n) = \underline{\mathit{Merge}}(\textstyle\bigcup_{p \in pred(n)} \{\underline{\mathit{InitEnv}}(e) \mid e \in \underline{\mathit{CallEnv}}(p, f)\}).$$

For return nodes, we get:

$$\begin{aligned}\underline{\mathsf{AEnv}}(n) = \underline{\mathit{Merge}}_{\mathsf{Var}\setminus\{res\}}(\{e[res \mapsto e'(ret)] \mid {} & e \in \underline{\mathsf{AEnv}}(n') \\ & \wedge\, e' \in \underline{\mathit{ReturnEnv}}(n', e)\}).\end{aligned}$$

It is not hard to see that in the resulting framework there will be at most one environment at any node. Hence the problem has been simplified considerably. In fact, it is equivalent to a problem discussed by Lakhotia [Lak93] who also shows how to solve it polynomial time.[5]

6 Interprocedural Function Pointer Must Alias Analysis

The discussion thus far has focused on interprocedural function pointer may-alias analysis, which is concerned with determining whether there exists a computation path through the program along which certain aliases can occur. One can also consider an analysis that is concerned with determining whether certain aliases must occur along every computation path from the entry point of the program to some particular program point. Such an analysis is called a "must alias" analysis:

Definition 6.1 [Function Pointer Must Aliasing Problem] Given a node n in the ICFG and a variable v the function pointer must aliasing problem is to determine if there is a single procedure p so that at the end of *all* statically executable path from $entry(\texttt{main})$ to n v points to p.

We write $[n, \langle v, p\rangle]_{\ must}$ indicating that v must point to p at n. ▮

Lemma 6.1 $[n, \langle v, p\rangle]_{\ must} \Leftrightarrow \{q \mid [n, \langle v, p\rangle]\ \} = \{p\}$

Given the results of the previous sections, the following result is not a great surprise:

[5] Lakhotia assumes a slightly more elaborate parameter passing mechanism.

Theorem 6.1 *The function pointer must aliasing problem is* EXPTIME-*complete.*

Proof Given an RMBF problem $(F(\vec{x}) = expr, \vec{z})$ we consider the logically equivalent problem $(F(\vec{x}) = false \vee expr, \vec{z})$. Then we have $(lfp(F))(\vec{z}) = 0 \Leftrightarrow$ $[exit(\texttt{main}), \langle \texttt{result}, \texttt{nil} \rangle]_{must}$ holds. ∎

7 Conclusion

The construction of a interprocedural control flow graph is the first step in any interprocedural dataflow analysis. In programs involving function pointers (or function-valued variables), this requires the determination of the possible values such pointers can take on. In this paper, we consider complexity issues for a variety of approaches to this problem. We show that a relational attribute analysis is necessary if precise results are to be obtained; extend earlier results by Zhang and Ryder [ZR94] to show that the problem is complete for deterministic exponential time; and show that for precise analyses, Zhang and Ryder's NP-hardness result holds even for intra-procedural analyses: that is, aliasing effects alone give rise to NP-hardness even when inter-procedural effects are absent. We then show that sacrificing precision by resorting to an independent attribute analysis does not change the complexity result: the problem remains EXPTIME-complete. However, if context-sensitivity is abandoned as well, it is possible to get polynomial-time algorithms.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. principles, techniques, and tools.* Addison-Wesley, 1986.

[CCHK90] D. Callahan, A. Carle, M. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Trans. on Softw. Eng.*, 16(4):483, April 1990.

[CBC93] J.-D. Choi, M. Burke, and P. Carini, "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects", *Proc. 20th. ACM Symposium on Principles of Programming Languages*, Jan. 1993, pp. 232–245.

[Hei94] Nevin Heintze. Control flow analysis and type systems. Technical Report CMU-CS-94-227, School of Computer Science, Carnegie Mellon University, December 1994.

[HY86] Paul Hudak and Jonathan Young. Higher-order strictness analysis in untyped lambda calculus. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 97–109, St. Petersburg Beach, Florida, January 1986.

[JM81] Neil D. Jones and Steven S. Muchnick. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 12, pages 380–393. Prentice-Hall, 1981.

[JM86] Neil D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs: abridged version. In *Proc. 13th ACM*

Symp. on Principles of Programming Languages, pages 296–306, St. Petersburg, FL, January 1986.

[Lak93] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Proc. 20th ACM Symp. on Principles of Programming Languages*, pages 273–284, Charleston, South Carolina, January 1993.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural side effect analysis with pointer aliasing. *SIGPLAN Notices*, 28(6):56–67, June 1993. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.

[Mye81] Eugene W. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, Williamsburg, Virginia, January 1981.

[Ryd79] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transaction of Software Engineering*, SE-5(3):216–226, 1979.

[Shi88] Olin G. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, Atlanta, Georgia, June 1988.

[Shi91] Olin G. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnige-Mellon Univeristy, May 1991. Also available as CMU-CS-91-145.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[Tar55] Alfred Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.

[WL95] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 1–12.

[ZR94] Sean Zhang and Barbara G. Ryder. Complexity of single level function pointer aliasing analysis. Technical Report LCSR-TR-233, Laboratory of Computer Science Research, Rutgers University, October 1994.