On the Complexity of Partially-Flow-Sensitive Alias Analysis

N. Rinetzky*, G. Ramalingam†, M. Sagiv, and E. Yahav

We introduce the notion of a *partially*-flow-sensitive analysis based on the number of read and write operations that are guaranteed to be analyzed in a sequential manner. We study the complexity of partially-flow-sensitive alias analysis and show that precise alias analysis with a very limited flow-sensitivity is as hard as precise flow-sensitive alias analysis, both when dynamic memory allocation is allowed, as well as in the absence of dynamic memory allocation.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors-compilers; optimization; F.4.1 [**Mathematical Logic**]: Computability Theory; F.4.3 [**Formal Languages**]: Decision Problems

General Terms: Languages, Theory

Additional Key Words and Phrases: Alias analysis, Flow-sensitive, Flow-insensitive

1. INTRODUCTION

Most static analyses for modern programming languages depend significantly on various forms of pointer analysis, such as alias analysis, to deal with indirect data references and modifications. However, precise flow-sensitive alias analysis is known to be undecidable for single-procedure programs with loops, recursive data structures, and dynamically allocated storage even under the assumption that all paths in the program are feasible [Landi 1992b; Ramalingam 1994]. The problem remains undecidable even if the program manipulates only singly-linked lists [Chakaravarthy 2003]. This result is shown for flow-sensitive analysis: *i.e.*, the analysis is required to respect the order in which statements execute in a path.

Precise flow-insensitive alias analysis has been shown to be NP-hard for programs without dynamic allocation, but in which pointers can reference other variables [Horwitz 1997]. However, this proof assumes that there is no bound on the number of memory accesses occurring in a single statement.

In this paper, we present some new complexity results for alias analysis by considering analyses that have a very restricted, and precisely-defined, form of flow-sensitivity. Traditionally, the term *flow-insensitive analysis* has been used to refer to analyses which ignore

Author's address: N. Rinetzky, Tel Aviv University, maon@post.tau.ac.il; G. Ramalingam, Microsoft Research India, grama@microsoft.com; M. Sagiv, Tel Aviv University, msagiv@post.tau.ac.il; E. Yahav, IBM T.J. Watson Research Center, eyahav@us.ibm.com.

* Supported in part by the IBM Ph.D. Fellowship Program, and in part by a grant from the Israeli Academy of Science.

† Work done partly when the author was at IBM Research.

constraints on the order in which statements in a program can execute. However, such analyses typically do take into account the order in which computations within a single statement occur. E.g., the multiple pointer dereferences occurring in a single statement such as "`x = ***p`", must be treated atomically by a precise flow-insensitive analysis. Note that if this statement is broken into a sequence of statements, the analysis might produce a different result. (In other words, the choice of the set of the atomic statements affects the precision of the analysis.) Thus, an analysis which treats "`x = ***p`" as an atomic unit, may be viewed as being *partially flow-sensitive*.

In this paper, we first formalize the notion of partially-flow-sensitive analysis as follows. We consider programs written in a language with a set of primitive statements. (Each statement can dereference at most one pointer.) A *block-partitioned* program is one that has been partitioned into units of computation called *blocks*. Informally, an analysis is said to be *block-flow-sensitive* if it analyzes code within any given block in a flow-sensitive fashion, but the analysis may ignore the execution order between blocks. Intuitively, the ability to analyze certain adjacent, related, statements as a unit (*i.e.*, flow-sensitively) can obviously help improve the precision of flow-insensitive analyses.

We will particularly consider analyses that are guaranteed to be block-flow-sensitive for programs where the total number of read and write operations in a block are less than some given constants. This allows us to measure the degree of flow-sensitivity of an analysis by considering the maximal number of read and write operations in a block. We show that the problem of a precise flow-sensitive alias analysis can be reduced to the problem of a precise partially-flow-sensitive alias analysis with a very limited degree of flow sensitivity. This, combined with [Landi 1992a; 1992b; Ramalingam 1994; Muth and Debray 2000; Chakaravarthy 2003], leads to our main results: lower bounds on the complexity of partially-flow-sensitive alias analysis.

From a pragmatic perspective, the key results of this paper are a sequence of reductions that show that certain aspects of flow-sensitive alias analysis are not critical and can be eliminated via these reductions, allowing analysis designers to focus on simplified subproblems.

1.1 Outline

Section 2 formalizes the notion of partially-flow-sensitive alias analysis. Section 3 states our main results. Section 4 shows that precise partially-flow-sensitive may- and must- alias analysis is undecidable for programs that use dynamic allocation. Section 5 shows that precise partially-flow-sensitive may-alias analysis is PSPACE-complete for single-procedure programs that do not use dynamic allocation. Section 6 concludes.

2. PARTIALLY-FLOW-SENSITIVE ALIAS ANALYSIS

In this section, we formalize the notion of partially-flow-sensitive alias analysis. We define the syntax and the semantics of a language of primitive statements. In this language, each statement can dereference at most one pointer. We then define the notion of a block-partitioned program, which allows us to formalize the concept of a precise partially-flow-sensitive (alias) analysis.

Statement	Intended meaning
`noop`	A no-operation statement
$x = \texttt{NULL}$	Nullify variable x
$x = y$	Copy the value of variable y to variable x
$x = y \rightarrow f$	Copy the value of the f-field of the object pointed-to by variable y to variable x
$x \rightarrow f = y$	Copy the value of variable y to the f-field of the object pointed-to by variable x
$x = \texttt{alloc}\ T$	Allocate a fresh object of type T and assign its address to variable x
$x = \&rec$	Assign the address of record variable rec to pointer variable x

Table I. The set of primitive statements. x and y are arbitrary pointer variables and rec is an arbitrary record variable; f is an arbitrary field-identifier and T is an arbitrary type-identifier.

2.1 Language of Primitive Statements: Syntax

A program consists of a set of type definitions, a set of variable declarations and a single (non-recursive) procedure.[1]

The only types allowed are pointers and records, which consist of a set of pointer fields. As we address only alias analysis, we do not consider other primitive types. The variable declarations declare a finite set of variables, each of a given record type or pointer type. Records are allowed to have recursive fields.

We assume the syntactic domains $x \in VarId$, $f \in FieldId$, and $t \in TypeId$, of variable identifiers, field identifiers, and type identifiers, respectively. We assume that $flds(t) \subset FieldId$ denotes the (finite) set of fields comprising a record type $t \in TypeId$. For simplicity, we assume that field identifiers are globally unique, *i.e.*, for any type identifiers $t_1, t_2 \in TypeId$, if $t_1 \neq t_2$ then $flds(t_1) \cap flds(t_1) = \emptyset$.

We consider programs written in a language with the set of primitive statements shown in Table I. Note that each statement can dereference at most one pointer. Without loss of generality, we assume that all branches are non-deterministic and that only the addresses of record variables may be taken.

We utilize a control-flow graph *(CFG)* to represent a program P. The control-flow graph consists of a set of vertices (N_P), a set of edges (E_P), a designated entry vertex (n_P), and a map (M_P) that associates every edge with a primitive statement.

2.2 Language of Primitive Statements: Semantics

Programs in our language are executed using a standard two-level store semantics for pointer languages (see, *e.g.*, [Milne and Strachey 1977; Reynolds 2002]). We assume that the operational semantics has the following (rather standard) properties:

- The identifier `NULL` in our language denotes a special value *null* different from the address of any heap-allocated object or variable.
- When a program's execution starts, the contents of every memory cell is $null$.
- All fields of a newly allocated object are initialized to $null$.
- A program halts if it dereferences a $null$-valued pointer.

2.2.1 *A Formal Definition of a Store-based Semantics.* We formalize the notion of a precise alias analysis using the following (standard) definition of a two-level store seman-

[1]Thus, we do not consider interprocedural analysis in this paper; as the primary results of this paper are lower bound results, this is not particularly significant. In particular, our lower bounds also apply to procedural languages.

Domain	Description
$l \in Loc$	Locations
$v \in Val = Loc \cup \{null\}$	Values
$lv \in \mathcal{LV} = VarId \hookrightarrow Loc$	Environments
$rv \in \mathcal{RV} = Loc \hookrightarrow Val$	Stores
$\sigma \in \Sigma = 2^{Loc} \times \mathcal{LV} \times \mathcal{RV}$	Memory states

Fig. 1. Semantic domains.

tics. We note, however, the our results apply to any definition which satisfies the aforementioned assumptions.

2.2.1.1 *Memory States.* Figure 1 defines the semantic domains and the meta-variables ranging over them. We assume Loc to be an unbounded set of locations. Due to our simplifying assumptions, a value $v \in Val$ is either a memory location or *null* $\notin Loc$. A memory state is a 3-tuple $\sigma = \langle A, lv, rv \rangle$. A is the set of *used* (alternatively, *active* or *allocated*) locations. These locations store the contents (r-values [Strachey 1966]) of pointer variables and of fields. lv is the environment. It maps every pointer variable to the (immutable and unique) location in which its contents are stored, *i.e.*, lv maps a variable to its l-value [Strachey 1966]. In C [Kernighan and Ritchie 1988] terminology, $lv(\mathtt{x})$ denotes $\&\mathtt{x}$, the address of variable x, in σ. rv is the *store*; it maps a location to its contents. For example, $rv(lv(\mathtt{x}))$ denotes the value of variable x in σ.

The value of every field of every record variable and of every dynamically allocated object is kept in its own (unique) location in the store. In addition, we assume that every record variable and every dynamically allocated object is *identified* by a unique location in the store. The latter can be, for example, the address of one of the object's fields, as in C, or the location of the object's header, as in $Java$.

A common memory layout for objects is placing every field in a fixed offset from the location which identifies the object. This way is taken, for example, in [Reynolds 2002], where locations are integers and every object is identified by the location of its first field. The contents of the ith field of an object identified by location l are stored in location $l + i$.

To abstract away from issues such as specific memory layouts, we assume the existence of a *layout function* $lv_f : Loc \hookrightarrow Loc$ for every field identifier $f \in FieldId$. Given a location l identifying a dynamically allocated object (resp. a record variable), $lv_f(l)$ denotes the location in the store of σ in which the value of the f-field of l is kept. It is assumed that for every location $l \in Loc$ and for every pair of field identifiers $f_1, f_2 \in flds(t)$, if $f_1 \neq f_2$ then $lv_{f_1}(l) \neq lv_{f_2}(l)$.

EXAMPLE 2.1. Assume that P is a program which defines type T as `type T {T* a, T* b}`, *i.e.*, T is a record which has two (recursive) pointer fields, a and b. Assume that P declares `rec` as a record variable of type T and x as a pointer variable of type pointer to T. Let $\sigma = \langle A, lv, rv \rangle$ be a memory state of P.

$lv(\mathtt{rec})$ denotes the unique memory location identifying `rec` in memory state σ. In C terminology, $lv(\mathtt{rec})$ denotes &`rec`, the address of `rec`. $lv_a(lv(\mathtt{rec}))$ denotes the location which stores the value of the a-field of record `rec`. Similarly, $lv_b(lv(\mathtt{rec}))$ denotes the location which stores the value of the `rec`'s b-field. In C terminology, $lv_a(lv(\mathtt{rec}))$ denotes &(`rec.a`) and $lv_b(lv(\mathtt{rec}))$ denotes &(`rec.b`).

If x *points to* record `rec` in σ then $rv(lv(\mathtt{x})) = lv(\mathtt{rec})$. If x *points to* a dynamically allocated object identified by location l then $rv(lv(\mathtt{x})) = l$. $lv_a(l)$ denotes the location

Axiom	Side-condition	# Reads	# Writes
$\langle \mathtt{noop}, \sigma\rangle \rightsquigarrow \sigma$		0	0
$\langle x = \mathtt{NULL}, \sigma\rangle \rightsquigarrow \langle A, lv, rv[lv(x) \mapsto \mathit{null}]\rangle$		0	1
$\langle x = y, \sigma\rangle \rightsquigarrow \langle A, lv, rv[lv(x) \mapsto \rho(y)]\rangle$		1	1
$\langle x = y \rightarrow f, \sigma\rangle \rightsquigarrow \langle A, lv, rv[lv(x) \mapsto rv(lv_f(\rho(y)))]\rangle$	$\rho(y) \neq \mathit{null}$	2	1
$\langle x \rightarrow f = y, \sigma\rangle \rightsquigarrow \langle A, lv, rv[lv_f(\rho(x)) \mapsto \rho(y)]\rangle$	$\rho(x) \neq \mathit{null}$	2	1
$\langle x = \mathtt{alloc}\ T, \sigma\rangle \rightsquigarrow \langle A \cup F_T \cup \{l\}, lv, rv[lv(x) \mapsto l]\rangle$	$l \in Loc \setminus A,\ F_T \cap A = \emptyset$	0	1
$\langle x = \&rec, \sigma\rangle \rightsquigarrow \langle A, lv, rv[lv(x) \mapsto lv(rec)]\rangle$		0	1

Fig. 2. Meaning of statements. $\sigma = \langle A, lv, rv\rangle$. x and y are arbitrary pointer variables and rec is an arbitrary record variable; f is an arbitrary field-identifier and T is an arbitrary type-identifier. $\rho(y)$ is a shorthand for the value of y in σ, *i.e.*, $\rho(y) = rv(lv(y))$. Similarly, $\rho(x) = rv(lv(x))$. $F_T = \{lv_f(l) \mid f \in \mathit{flds}(T)\}$.

which stores the value of the a-field of l. Similarly, $lv_b(l)$ denotes the location which stores the value of the b-field of l. In C terminology, $lv_a(rv(lv(\mathtt{x})))$ denotes &(x→a) and $lv_b(rv(lv(\mathtt{x})))$ denotes &(x→b).

A memory state $\sigma = \langle A, lv, rv\rangle$ is an *admissible initial memory state* for a program P, if the following conditions hold:

(i) Every variable is mapped to a location, *i.e.*, for every variable x defined in P, $lv(x) \in Loc$.

(ii) The locations used to contain the values of different variables are disjoint: Let $base(x)$ be $\{lv(x)\}$, if x is a pointer variable, and $\{lv(x)\} \cup \{lv_f(l) \mid f \in \mathit{flds}(T)\}$, if x is a record variable of type T. If x and y are different variables defined in P then $base(x) \cap base(y) = \emptyset$.

(iii) Every memory location in the store is initialized to *null*, *i.e.*, $rv = \lambda l \in Loc.\mathit{null}$.

(iv) A, the set of used locations, contains all the locations used to store the values of the variables declared in P, and only these locations, *i.e.*, let V_P be the variables declared in P, then $A = \bigcup_{x \in V_P} base(x)$.

We assume that a program P always starts executing in an admissible initial memory state.

2.2.1.2 *Operational Semantics.* Figure 2 defines the meaning of statements in a standard two-level store semantics for pointer programs. The semantics is specified for every primitive statement $st \in \mathit{stms}$ of the form defined in Table I. The *meaning* of every statement st is given as a binary relation over a set of memory states $[\![st]\!] \subseteq \Sigma \times \Sigma$. A pair of memory states $\langle \sigma, \sigma'\rangle \in [\![st]\!]$ iff the execution of st in memory state σ may lead to memory state σ'. Figure 2 describes the semantics of a statement st in the form of axioms. The intention is that $\langle \sigma, \sigma'\rangle \in [\![st]\!]$ iff $\langle st, \sigma\rangle \rightsquigarrow \sigma'$. The *#Read* column shows the number of read memory access to the store done by each statement. The *#Write* column shows the number of write memory access to the store done by each statement.

The statement noop is a no-operation, *i.e.*, it does nothing. The statement x=NULL nullifies variable x. The statement x=y copies the value of variable y to variable x.

The statement x=y→f (field-dereference) reads the value of field f of the object pointed-to by y and writes that value to x. The statement x→f=y (destructive-update) writes the value of y to the f-field of the object pointed-to by x. In both statements, a side-condition ensures that the program does not dereference a null-valued pointer: The execution of the program halts if the dereferenced variable has a *null* value.

The statement `x=alloc T` (dynamic-allocation) allocates an object of type `T` and assigns its identifying location to variable `x`. The identifying location is guaranteed to be *fresh*, *i.e.*, it is not used in the current memory state. In addition, the statement reserves a set of fresh locations, F_T, to contain the values of the fields of the new object.

We require that for every type `T` there be an unbounded number of locations $l \in \mathit{Loc}$ such that $lv_f(l)$ is defined for every $f \in \mathit{flds}(\mathtt{T})$. This requirement ensures that it is possible to allocate an unbounded number of objects of type `T`. A similar requirement is placed in [Reynolds 2002].

For *simplicity*, we require that every location is allocated once during the execution of the program. This requirement is enforced by the side-condition of the `alloc` statement, and the maintenance of the set A of *all* allocated objects, including ones that are unreachable. Because every execution starts from an admissible initial memory state, this simplifying assumption also ensures that the fields of allocated objects are initialized to *null*.

The statement `x=&rec` assigns `&rec`'s identifying location to `x`.

Note that the lv component of a state is immutable. This immutability, combined with the assumption that a program always starts executing in an admissible initial memory state, ensures that reading or writing the value of a variable never leads to a null-dereference.

2.2.2 *Flow-Sensitive Executions.* We now formalize the (standard) notion of (flow-sensitive) executions.

DEFINITION 2.2. *A **sequence** π over a set S is a total function $\pi \in \{i \in \mathbb{N} \mid 1 \leq i \leq n\} \rightarrow S$ for some $n \in \mathbb{N}$. The **length of a sequence** π, denoted by $|\pi|$, is $|dom(\pi)|$.*

DEFINITION 2.3. *A **path** π of a program P is a sequence over E_P, the edges of the control-flow graph of P. A path π of a program P is **realizable** if (i) $\pi(1)$ originates from P's entry vertex, i.e., $\pi(1) = \langle n_P, n\rangle$ for some $n \in N_P$, and (ii) π forms a chain of edges, i.e., for every $1 \leq j < |\pi|$, if $\pi(j) = \langle n_j, n'_j\rangle$ and $\pi(j+1) = \langle n_{j+1}, n'_{j+1}\rangle$ then $n'_j = n_{j+1}$.*

DEFINITION 2.4. *A **trace** of a program P is a sequence τ over the set of memory states of program P. A trace τ is **induced by a path** π of program P if (i) $|\tau| = |\pi| + 1$ and (ii) $\langle \tau(j), \tau(j+1)\rangle \in [\![M(\pi(j))]\!]$ for every $1 \leq j \leq |\pi|$.*

Given a trace τ of a program P, we refer to $\tau(1)$ as τ's *initial memory state* and to $\tau(|\tau|)$ as τ's *terminal memory state*. If a trace τ' is induced by a path π, we say that τ *starts executing* in program point n', where n' is the source of the edge $\pi(1)$ and *ends executing* in program point n'', where n'' is the target of the edge $\pi(|\pi|)$. We refer to the terminal memory state of a trace τ as the *memory state resulting after the execution of* π.

DEFINITION 2.5. *A trace τ of a program P is a **flow-sensitive execution** of program P if (i) $\tau(1)$ is an admissible initial memory state and (ii) τ is induced by a realizable path.*

2.3 Flow Sensitive Alias Analysis

An analysis is said to be a *sound* (resp. *precise*) *flow-sensitive analysis* for a program P, if (resp. iff) the information it determines at program point (*CFG* node) n, is true at every program state that can result after any flow-sensitive execution of P ending in n.

DEFINITION 2.6. *A **flow-sensitive may-alias analysis** determines for a program P and a program point $n \in N_P$, a set S of pairs of program variables such that if $(x, y) \notin S$ then*

x and y never point to the same memory location after any flow-sensitive execution of P ending in program point n.

Note that the above definition implies that given two variables, say `x` and `y`, a *precise* flow-sensitive may-alias analysis also determines whether P has a flow-sensitive execution after which `x` and `y` point to the same memory location.

DEFINITION 2.7. *A **flow-sensitive must-alias analysis** determines for a program P and a program point $n \in N_P$, a set S of pairs of program variables such that if $(x, y) \in S$ then after any flow-sensitive execution in P ending in n, either both x and y have a $null$ value, or both point to the same memory location.*

2.4 Block Partitioned Programs

A *block-partitioned* program is one that has been partitioned into units of computation called *blocks*. A block consists of a sequence of primitive statements. Informally, an analysis is said to be block-flow-sensitive if it analyzes code within any given block in a flow-sensitive fashion, but the analysis may ignore the execution order between blocks.

We utilize a control-flow graph to represent a block-partitioned program, just as in Section 2.1. The only difference is that instead of associating edges with primitive statements, we associate them with blocks.

2.4.1 *Block-Flow-Sensitive Executions.* Intuitively, a *block-flow-sensitive execution* of a block-partitioned program P arbitrarily executes P's code blocks, while respecting the order of statements in every block. We now formalize the notion of block-flow-sensitive executions.

The semantics defined in Section 2.2.1 induces a (standard) *meaning* for every sequence of statements as the composition of the meanings of the statements comprising the sequence. We denote the composed meaning of a sequence of statements $block$ by $[\![block]\!]$, *i.e.*, $[\![block]\!] = [\![block(1)]\!] \circ \ldots \circ [\![block(|block|)]\!]$.

DEFINITION 2.8. *A trace τ of block-partitioned program P is a **block trace induced by a path** π of program P if (i) $|\tau| = |\pi| + 1$ and (ii) for every $1 \leq j \leq |\pi|$, $\langle \tau(j), \tau(j+1) \rangle \in [\![M(\pi(j))]\!]$.*

DEFINITION 2.9. *A trace τ of a program P is a **block-flow-sensitive execution** if (i) $\tau(1)$ is an admissible initial memory state and (ii) there exists a path π of program P such that τ is a block trace induced by π.*

Note: We define the notion of a *flow-sensitive* execution of a *block-partitioned program* by adapting Definition 2.5 to consider *block traces* instead of *traces*. Note that the modified definition ensures that the information determined by both flow-sensitive may-alias analysis and must-alias analysis of *block-partitioned programs* is oblivious to the intermediate memory states occurring during the execution of a block.

EXAMPLE 2.10. Figure 3.I shows the control flow graph of a block-partitioned program $P_{3.I}$. Program $P_{3.I}$, defines type `N` as `type N {N* n}`, *i.e.*, record `N` has a single recursive field, `n`. $P_{3.I}$ defines six variables of type pointer to `N`: `x`, `y`, `p`, `q`, `t`, and `z`. The entry vertex of $P_{3.I}$ is n_1. Every edge in $P_{3.I}$ is identified by a label of the form e_i, written above the edge. The code block associated with an edge is written below that edge. We sometimes refer to the code block associated with edge e_i as code block i.

I	CFG		$n_1 \xrightarrow[x=\texttt{alloc}\,N]{e_1} n_2 \xrightarrow[y=\texttt{alloc}\,N]{e_2} n_3 \xrightarrow[\substack{p=x;\\ q=y}]{e_3} n_4 \xrightarrow[\substack{p=y;\\ q=x}]{e_4} n_5 \xrightarrow[\substack{p\rightarrow n=q;\\ q\rightarrow n=p}]{e_5} n_6 \xrightarrow[\substack{z=p;\\ t=p\rightarrow n;\\ t=t\rightarrow n}]{e_6} n_7$
II	Realizable paths	π_1	$\cdot \xrightarrow[x=\texttt{alloc}\,N]{e_1} \cdot \xrightarrow[y=\texttt{alloc}\,N]{e_2} \cdot \xrightarrow[\substack{p=x;\\ q=y}]{e_3} \cdot \xrightarrow[\substack{p=y;\\ q=x}]{e_4} \cdot \xrightarrow[\substack{p\rightarrow n=q;\\ q\rightarrow n=p}]{e_5} \cdot \xrightarrow[\substack{z=p;\\ t=p\rightarrow n;\\ t=t\rightarrow n}]{e_6} \cdot$
		...	...
	Non-realizable paths	π_2	$\cdot \xrightarrow[y=\texttt{alloc}\,N]{e_2} \cdot \xrightarrow[\substack{p=y;\\ q=x}]{e_4} \cdot \xrightarrow[\substack{p=x;\\ q=y}]{e_3} \cdot \xrightarrow[\substack{p=y;\\ q=x}]{e_4} \cdot \xrightarrow[\substack{p=x;\\ q=y}]{e_3} \cdot$
		π_3	$\cdot \xrightarrow[x=\texttt{alloc}\,N]{e_1} \cdot \xrightarrow[\substack{p=x;\\ q=y}]{e_3} \cdot \xrightarrow[\substack{z=p;\\ t=p\rightarrow n;\\ t=t\rightarrow n}]{e_6} \cdot$
		...	...
III	Flow-sensitive	may	$\{q, x\}, \{p, y\}, \{z, p\}, \{z, y\}, \{t, p\}, \{t, y\}, \{t, z\}$
		must	$\{q, x\}, \{p, y\}, \{z, p\}, \{z, y\}, \{t, p\}, \{t, y\}, \{t, z\}$
	Block-flow-sensitive	may	$\{q, x\}, \{p, y\}, \{z, p\}, \{z, y\}, \{t, p\}, \{t, y\}, \{t, z\}, \{p, x\}, \{q, y\}, \{z, x\}, \{t, x\}$
		must	$\{t, z\}$

Fig. 3. I. Program $P_{3.I}$. II. Realizable paths vs. non-realizable paths. III. Flow-sensitive vs. block-sensitive alias analysis.

Figure 3.II shows three paths of $P_{3.I}$: π_1, π_2, and π_3. Path π_1 is a realizable path, while paths π_2 and π_3 are not realizable paths.

Path π_1 is the only realizable path ending in n_7. Furthermore, every realizable path of $P_{3.I}$ is a prefix of π_1. In every memory state resulting after a flow-sensitive execution induced by π_1, two objects are allocated. The pointer variables y, p, z, and t point to one of the objects. The pointer variables x and q point to the other object. The n-field of each object points to the other object.

Path π_2 is not realizable: The first edge in path π_2 is e_2. This edge does not have the entry vertex as its source. (A realizable path of $P_{3.I}$ must start with edge e_1.) Furthermore, code block 3 and code block 4 appear twice in π_2. This is not possible in a realizable path of $P_{3.I}$. At the end of the block-sensitive execution induced by π_2, one object is allocated. It is pointed-to by y and q. All other variables have a *null* value.

Path π_3 is comprised of edge e_1, e_3, and e_6. Note that an attempt to execute the program according to π_3 leads to a null-dereference: Code block 6 attempts to traverse the n-field twice, starting from the object pointed-to by p. However, at the program point in which code block 6 is executed, the n-field of the object pointed-to by p has a *null* value. As a result, the execution gets stuck: the second dereference cannot be executed.

2.5 Block Flow Sensitive Alias Analysis

An analysis is said to be a sound (resp. precise) *block-flow-sensitive analysis* for a block-partitioned program P, if (resp. iff) the information it determines is true at every program state that can result after block-flow-sensitive execution of P.

DEFINITION 2.11. *A **block-flow-sensitive may-alias analysis** determines for a block-partitioned-program P, a set S of pairs of program variables such that if $(x, y) \notin S$ then x and y never point to the same memory location after any block-flow-sensitive execution of P.*

Note that the above definition implies that given two variables, say x and y, a *precise* block-sensitive may-alias analysis also determines whether P has a block-flow-sensitive execution after which x and y point to the same memory location.

DEFINITION 2.12. *A **block-flow-sensitive must-alias analysis** determines for block-partitioned-program P, a set S of pairs of program variables such that if $(x, y) \in S$ then after any block-flow-sensitive execution in P, either both x and y have a $null$ value, or both point to the same memory location.*

Note: Note that the information determined by both block-flow-sensitive may-alias analysis and must-alias analysis is oblivious to the intermediate memory states occurring during the execution of a block. Also note that the exact program point in which the aliasing question is asked is immaterial in Definitions 2.11 and 2.12.

EXAMPLE 2.13. Figure 3.III-Flow-sensitive shows the precise solutions to the flow-sensitive may- and must - alias analyses of program $P_{3.I}$ at program point n_7. To avoid clutter, we exploit the symmetry of the aliasing relation and use $\{x, y\}$ as shorthand for (x, y) and (y, x). We also omit all pairs of the form (x, x).

The precise flow-sensitive solution of a may-alias analysis of $P_{3.I}$ at program point n_7 is shown in the row labeled *flow-sensitive may*: We list every pair of pointer variables which *may be* aliased, *i.e.*, point to the same location, after a flow-sensitive execution of program $P_{3.I}$ ending in program point n_7 (*i.e.*, an execution induced by path π_1). Note that the pair $\{p, q\}$ is not in the solution although after the execution of $y = p$ inside code block 4 p and q point to the same location: The analysis may ignore intermediate memory states occurring during the execution of a block.

The precise flow-sensitive solution of a must-alias analysis of $P_{3.I}$ at program point n_7 is shown in the row labeled *flow-sensitive must*. It contains every pair of pointer variables which must point to the same location or have a *null* value after every flow-sensitive execution of program $P_{3.I}$ ending in program point n_7. The solution coincides with the precise solution to the may-alias analysis because there is only one realizable path of $P_{3.I}$ ending in n_7.

Figure 3.III-Block-flow-sensitive shows the precise solutions to the block-flow-sensitive may- and must - alias analyses of program $P_{3.I}$.

The precise block-flow-sensitive solution of a may-alias analysis of $P_{3.I}$, is shown in the row labeled *block-flow-sensitive may*: We list every pair of pointer variables which *may be* aliased, *i.e.*, point to the same location, after a *block-flow-sensitive* execution of program $P_{3.I}$. Note that, by definition, every flow-sensitive execution is also a block-sensitive execution. Thus, every pair listed in the solution of the flow-sensitive may-alias analysis is listed here too. In addition, the precise block-sensitive solution contains four more pairs: The pairs $\{p, x\}, \{q, y\}$ are added by considering, *e.g.*, executions induced by the path e_1, e_2, e_3. The pairs $\{z, x\}, \{t, x\}$ are added by considering, *e.g.*, executions induced by the path e_1, e_2, e_3, e_5, e_6. Again, and for the same reasons mentioned above, the pair $\{p, q\}$ is not in the solution.

The precise block-flow-sensitive solution of a must-alias analysis of $P_{3.I}$ is shown in the row labeled *block-sensitive must*. Only t and z are determined to be must-alias: The values of `z` and `t` are always set by block 6: Variable `z` is assigned the value of `p` and variable `t` is assigned, in C notations , the value of `p→n→n`. The only block which sets the values of the `n`-fields is block 5. In *any* block-sensitive execution, if block 5 does not cause a null-dereference, then it sets the `n`-field of the object pointed-to by `p` to point to the object pointed-to by `q`, and vice versa. Note that because both destructive updates are in the same block, they are always executed as a unit. Thus, either the traversal of the `n`-fields done in block 6 is successful, and returns to the point of origin, *i.e.*, to the object pointed-to by `p`, or it leads to a null-dereference which halts the execution. In the latter case, the analysis also does not continue along the path.

The last point, *i.e.*, the analysis not "continuing" along a path after a null-dereference occurs, will play a key role in our arguments.

2.6 Partially Flow Sensitive Alias Analysis

We measure the degree of flow-sensitivity in block-flow-sensitive analysis of a program P by measuring the "size" of its blocks.

We can measure the "size" of a block in a number of ways. The simplest measure is to count the number of statements in a block. Thus, we may say that a block is a k-block if it has k statements. This measurement has the advantage of being both intuitive and simple. However, it blurs certain subtle distinctions between different k-blocks. For example, consider the following statements: `x=y` and `x=y→f`. Each statement is also a 1-block. However, the first block performs only one read operation while the second block performs two consecutive read operations. Intuitively, a precise analysis of the second block requires a higher degree of flow sensitivity.

Thus, we define a finer measurement by separately counting the number of memory locations read and the number written in a block. The exact bookkeeping method used for this purpose is not critical, (it changes our results only by a constant factor). In this paper, we use the following definition to get a reasonably intuitive measure.

We first introduce the notion of a local variable (or temporary). A local variable is one that is always initialized in a block before it is used. As a result, local variables cannot be used to communicate values between blocks (or between different executions of the same block). Thus, we may think of local variables as being "local" to each block. We will typically use variable names of the form r_i for local variables. For any block B, let $rd(B)$ be the number of non-local read occurrences in block B. Similarly, let $wr(B)$ be the number of non-local write occurrences in block B. A block B is said to be an (r, w)-block if $rd(B) = r$ and $wr(B) = w$. We say that a block-partitioned program P is an (r, w)-*block-partitioned program*, if every block B of P is such that $rd(B) \leq r$ and $wr(B) \leq w$.

Note that any primitive statement can be encoded by a $(2, 1)$-block. Furthermore, a "high level" statement of the form $x = y \rightarrow f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_k$ can be compiled into a $(k + 1, 1)$-block of primitive statements. This should illustrate the motivation behind the above definitions.

An analysis is said to be an (r, w)-*partially-flow-sensitive* analysis, if it is a block-flow-sensitive analysis for all (r, w)-block-partitioned programs. As a special case, an analysis is said to be (∞, w)-partially-flow-sensitive if it is block-flow-sensitive for all block-partitioned programs with blocks B such that $wr(B) \leq w$.

DEFINITION 2.14. *An* $(\mathbf{r}, \mathbf{w})$***-partially-flow-sensitive may-alias analysis*** *is a block-flow-sensitive may-alias analysis for all* (r, w)*-block-partitioned-programs.*

DEFINITION 2.15. *An* $(\mathbf{r}, \mathbf{w})$***-partially-flow-sensitive must-alias analysis*** *is a block-flow-sensitive must-alias analysis for all* (r, w)*-block-partitioned-programs.*

Note: As in Definitions 2.11 and 2.12, the exact program point in which the aliasing question is asked is immaterial in Definitions 2.14 and 2.15.

We can now summarize the Horwitz's result [Horwitz 1997] as: precise $(\infty, 1)$-partially-flow-sensitive alias analysis is NP-hard. On the other hand, Andersen's analysis [Andersen 1994] is a sound $(2, 1)$-partially-flow-sensitive may-alias analysis.

In this paper, we limit the allowed alias questions to be equality of variables, *i.e.*, we consider only questions of the form *are* x *and* y *may- (resp. must-) alias?*

3. MAIN RESULTS

In this paper, we show that precise flow-sensitive alias analysis can be reduced to precise (3,3)-partially-flow-sensitive alias analysis (with dynamic memory allocation) and to (5,2)-partially-flow-sensitive analysis (without dynamic memory allocation). This allows us to show that

- Precise (3,3)-partially-flow-sensitive may-alias and must-alias analysis is undecidable for programs that use dynamically allocated memory.
- Precise (5,2)-partially-flow-sensitive may-alias is PSPACE-complete for programs that do not use dynamically allocated memory.

We remind the reader that in this paper, we only consider pointer programs comprised of a single (non-recursive) procedure.

4. REDUCING FLOW-SENSITIVE ALIASING TO PARTIALLY-FLOW-SENSITIVE ALIASING

In this section, we show that alias analysis with a very limited flow-sensitivity is as hard as flow-sensitive alias analysis. Specifically, we show that any program P can be transformed into a $(3, 3)$-block-partitioned program Q such that the block-flow-sensitive solution for Q yields the flow-sensitive solution for P. Because precise flow-sensitive alias analysis is undecidable for heap manipulating programs [Landi 1992b; Ramalingam 1994; Chakaravarthy 2003], this shows that precise $(3, 3)$-partially-flow-sensitive alias analysis is also undecidable.

We present the reduction in two stages: Section 4.1 describes a reduction that uses an unbounded number of fields and Section 4.2 shows how the number of fields used can be bounded.

4.1 A Reduction with an Unbounded Number of Fields

As explained earlier, a program P is represented by a set of type definitions, a set of variable declarations, and a labeled CFG, where every edge of the CFG is annotated with a statement. We assume, without loss of generality, that all variables are pointers. (Thus,

Component		Program P	Program Q	Remark

Type definitions		type T {T* f; ... }
		 ...
			type T {T* fld; ... }
			 ...
			 type PState {
			 T* p;
			 ...
			 }
				Program Q contains all type definitions of program P, plus a new type called PState. PState has a field `p` of type pointer to T for every pointer variable `p` of type pointer to T in program P

Variables		T* p ...	PState* at_n_i ...	Program Q contains a variable at_n_i of type pointer to PState for every node n_i in P's CFG
CFG	Nodes	$N_P = \{n_1, \dots, n_m\}$	$N_Q = N_P \cup \{n_0\}$	Q's CFG is comprised of P's CFG, plus a new entry node, n_0, connected to P's entry node, n_1
	Entry	n_1	n_0	
	Edges	$E_P = \{e_1, \dots, e_k\}$	$E_Q = E_P \cup \{e_0\}$ where $e_0 = \langle n_0, n_1 \rangle$	
	Map	M_P maps P's edges to primitive statements	M_Q maps Q's edges to blocks of primitive statements	$M_Q(e_0)$ is `at_0 =alloc PState`, and $block(e)$ for $e \in E_P$ ($block$ is defined in the caption)
Alias question		Are `x` and `y` may- (resp. must-) alias at node n?	Are at_n→x and at_n→y may- (resp. must-) alias?	The program point is immaterial for the aliasing question in Q

Fig. 4. A transformation of an arbitrary program P into a (3,3)-block program Q. For an edge $e = \langle n_i, n_j \rangle \in E_P$, $block(e)$ is at_n_j = at_n_i; at_n_i = null; TRANS(M$_P$(e), at_n_j). The function TRANS is defined in Table II.

any record will have to be heap allocated. In particular, we rule out the use of statements which take the address of variables, *e.g.*, `x=&rec`.)

Figure 4 illustrates how we transform a given program P into a $(3, 3)$-block-partitioned program Q.

The first step in the transformation augments the type definitions in program P with the definition of a new type `PState` which contains a field `p` for every variable `p` in P. The idea is to use a single (heap-allocated) record of type `PState` to capture the values of all variables in P. The state of program P is captured in Q by a `PState` record plus the part of the heap reachable from that record.

As a result, we would like to replace the set of all variables declared in P with a single pointer variable of type pointer to `PState`. For reasons that will become clear soon, we actually use a pointer variable `at_n_i` of type pointer to `PState` for every vertex n_i in the CFG. These extra variables are used to ensure flow-sensitivity by converting control-flow information (the "program counter") into data, as outlined below.

We add the statement "`at_n_0 = alloc PState`" to program Q to create the single record that is used to store the value of all variables in P, where n_0 is the entry vertex of the CFG. We then transform every statement in program P associated with an edge from `n_i` to `n_j` into a block in program Q as follows: we first add a *transition guard* that copies `at_n_i` to `at_n_j` and then sets `at_n_i` to null; we then transform the original statement by replacing references to any variable `x` by a reference to `at_n_j`→`x`. The statement produced by this substitution may not be a primitive statement, but can be compiled into a sequence of primitive statements as shown in Table II. Thus, the execution of the block associated with edge `n_i` to `n_j` in program Q "passes the baton" to node `n_j`. Further, this block can execute successfully (without a null dereference) only after the execution of some other edge (with target `n_i`) passes the baton on to node `n_i`.

Statement	**Transformation**	**Encoding code block**
`noop`	`noop`	`noop`
$x = \texttt{NULL}$	$at\_n \rightarrow x = \texttt{NULL}$	$r_1 = at\_n;\ \ r_1 \rightarrow x = \texttt{NULL}$
$x = y$	$at\_n \rightarrow x = at\_n \rightarrow y$	$r_1 = at\_n;\ \ r_2 = r_1 \rightarrow y;\ \ r_1 \rightarrow x = r_2$
$x = y \rightarrow f$	$at\_n \rightarrow x = at\_n \rightarrow y \rightarrow f$	$r_1 = at\_n;\ \ r_2 = r_1 \rightarrow y;\ \ r_3 = r_2 \rightarrow f;\ \ r_1 \rightarrow x = r_3$
$x \rightarrow f = y$	$at\_n \rightarrow x \rightarrow f = at\_n \rightarrow y$	$r_1 = at\_n;\ \ r_2 = r_1 \rightarrow x;\ \ r_3 = r_1 \rightarrow y;\ \ r_2 \rightarrow f = r_3$
$x = \texttt{alloc}\ T$	$at\_n \rightarrow x = \texttt{alloc}\ T$	$r_1 = at\_n;\ \ r_2 = \texttt{alloc}\ T;\ \ r_1 \rightarrow x = r_2$

Table II. $TRANS(st, at\_n)$: Transformation of a primitive statement st which annotates a CFG edge entering node n (which is represented by the variable at_n).

Consider any path α in program P's CFG from its entry vertex to some vertex n, and let σ be the program state in P after execution along path α. Let α' be the corresponding path in program Q's CFG consisting of the edge e_0 from `n_0` to `n_1` followed by α, and let σ' be the program state in Q after execution along path α'. It should be clear that σ' is an equivalent representation of state σ. Specifically, it should be clear that pointer variables `x` and `y` will have the same value in σ iff `at_n`→`x` and `at_n`→`y` have the same value in state σ'.

The key aspect of the transformation, however, relates to a sequence ξ of edges in Q's CFG that does not constitute a path. Execution along any sequence ξ of edges in Q's CFG will either result in a null-dereference or will produce a state that is equivalent to the state produced by execution along a path from Q's entry vertex to n, where n is the target vertex of the last edge in sequence ξ.

Specifically, consider how the pointer to the record allocated in the entry edge is copied to other `at_n` variables. The transition guards that do this ensure that at most one `at_n` variable points to this object. Further, the sequence of `at_n` variables that point to this object must form a valid path in the CFG, starting from the entry vertex. However, note that in a block-sensitive analysis it is possible for the entry edge to be executed at any point, creating newer `PState` records. (Recall that the code block associated with the entry edge does *not* begin with a transition guard, i.e., it is comprised *only* of the statement "`at_n_0 = alloc PState`".) In the general case, it is possible for multiple `PState` records to exist, and for multiple `at_n` variables to be non-null (*i.e.*, to point to these records). However, no two `at_n` variables can point to the same record at the same time. This gives us the desired result, as shown below.

Let $\xi = [e_1, \cdots, e_q]$ be an arbitrary sequence of Q's edges and let n be the target of edge e_q. Consider the execution of the code block associated with any edge e_i that is not the entry edge. Let $e_i = (v, w)$. If the execution of the code block does not cause a null-dereference, then `at_v` must be non-null before the execution of the statements. However, `at_v` can be assigned a non-null value only by the execution of the block associated with some edge whose target is v. Let j be the largest integer less than i such that the target of e_j is v. (It follows from the previous argument that such a j exists.) We define j to be the *logical predecessor index* of i (in the sequence ξ).

We can identify a sequence of indices $[z_1, \cdots, z_{q'}]$ such that $z_{q'} = q$, and for $1 < i \leq q'$, z_{i-1} is the logical predecessor index of z_i in ξ, and e_{z_1} is the entry edge. The corresponding sequence γ of edges $[e_{z_1}, \cdots, e_{z_{q'}}]$ forms a realizable path from the entry vertex to vertex n in Q such that the state after the execution of ξ at n is equivalent to the state after execution of γ at n (where the notion of equivalence is as explained previously).

A key property that guarantees this result is the following. Let j be the logical predecessor index of i in ξ. Consider any p such that $j < p < i$. The execution of edge e_p does not affect the visible state seen during the execution of e_i. Specifically, let the source vertex of edge e_p be u. Then, the record that pointer `at_u` points to before the execution of e_p is different from the record that `at_v` points to before the execution of e_i. Further, the heap reachable from `at_u` is completely disjoint from the heap reachable from `at_v`. The disjointness is ensured by the following properties: (i) `at_u` and `at_v` point to different state records, and (ii) the transformation of the statements ensures that references to allocated objects are obtained by traversing through the same state record. Thus, once an object has been allocated and its location assigned to a field of one state record, it cannot be pointed-to by a field of any other state record.

Thus, execution along an arbitrary sequence ξ ends up simulating parallel executions along one or more realizable paths in the CFG, without any interference between these simulations. The key reason for the correctness of the transformation, whose proof follows immediately from the preceding discussion, is that executions halt once a null-pointer dereference occurs (see Section 2.2).

THEOREM 4.1. *The block-flow-sensitive aliasing solution for Q coincides with the flow-sensitive aliasing solution for P.*

Let us now measure the flow-sensitivity of the program Q. Note that the transition guard `at_n_j=at_n_i; at_n_i = NULL;` can be encoded by the following operations consisting of 1 read operation and 2 write operations:

$$r_1 = \texttt{at\_n\_i};\ \texttt{at\_n\_j} = r_1;\ \texttt{at\_n\_i} = \texttt{NULL}$$

Every simple statement can be encoded by at most 4 operations (see Table II) containing at most 3 reads and 1 write. Note that the first operation in the code block pertaining to any statement, with the exception of `noop`, is always to read the value of the variable pertaining to the "current" CFG node. However, because every transformed statement is preceded by a transition guard which stores a value into that variable, we can save this read operation.

The aliasing question `at_n→x == at_n→y` can also be encoded using a total of 4 operations with 3 reads:

$$r_1 = \mathit{at\_n};\ r_2 = r_1 \!\rightarrow\! x;\ r_3 = r_1 \!\rightarrow\! y;\ \mathit{equal} = \mathit{compare}(r_2, r_3)$$

The following corollary follows immediately from the above results:

COROLLARY 4.2. *Precise $(3,3)$-partially-flow-sensitive may-alias and must-alias analyses are undecidable in the presence of dynamic memory allocation.*

Note: It is possible to ask an alternative aliasing question, one which only requires determining information regarding aliasing of variables, using a slightly more complicated transformation: We add to program Q two *pointer variables,* say `at_n_x` and `at_n_y`, of the same types as `x` and `y`, respectively. Variables `at_n_x` and `at_n_y` can capture the values of `at_n→x` and `at_n→y`, respectively, whenever a state record is propagated to `at_n`. The values of `at_n→x` and `at_n→y` are fetched using the following code block:

$$r_1 = \texttt{at\_n};\ r_2 = r_1 \!\rightarrow\! \texttt{x};\ r_3 = r_1 \!\rightarrow\! \texttt{y};\ \texttt{at\_n\_x} = r_2;\ \texttt{at\_n\_y} = r_3.$$

This code block is *not* guarded. Thus, it can be encoded using 3 reads and 2 writes. The modified aliasing question is *are `at_n_x` and `at_n_y` may- (resp. must) alias?* Note that

Component	Program P	Program B	Remark

Type definitions	type T {T* f; ... }	type T {T* fld; ... }
		
		type VarList {
		VarList* n;
		T* px;
		}
			The one user-defined type.
			
			The variable list:
			· next variable
			· variable value

Variables	T* p_1, ... p_v
	...
	...
	...
		VarList* hd;
		VarList* t;
		T* `lh`;
		T* `rh`;
			Head of the variables list
			Temporary
			Left-hand operand
			Right-hand operand

Aliasing question	Are `p_i` and `p_j` may- (resp. must-) alias at node n_{check}?	Are `lh` and `rh` may- (resp. must-) alias at node n'_{check}?	Edge $\langle n_{check}, n'_{check}\rangle$ is labeled by a `nop` in P

Fig. 5. The data types and the variables used in the transformation of an arbitrary program P into a program B which uses only 6 variables. Program P has v pointer variables.

`at_n_x` and `at_n_y` do not participate in the simulation. They function as "place holders" that can be assigned the values of `at_n`$\rightarrow$`x` and `at_n`$\rightarrow$`y` whenever `at_n` points to a state record. Because `at_n_x` and `at_n_y` are always assigned as a unit, they preserve both may- and must- aliasing information regarding `at_n`$\rightarrow$`x` and `at_n`$\rightarrow$`y` in program Q, and thus, also regarding `x` and `y` in program P.

4.2 Bounding the Number of Fields

In this section, we show that precise (3,3)-partially-flow-sensitive may-alias and must-alias analysis is undecidable even when the number of fields is bounded. This result is not implied by Theorem 4.1 because the number of fields used by the transformation in Section 4.1 is proportional to the number of variables in the transformed program P. Specifically, the `PState` record has a `p`-field for every variable `p` in program P.

The main idea is to simulate a program P, which uses an unbounded number of variables, using a program B, which uses only a bounded number of variables. The simulation ensures that the *flow-sensitive* aliasing solution to program B yields the *flow-sensitive* aliasing solution to program P. Applying the transformation of Section 4.1 to program B achieves the desired result.

In this section, we assume that the program has only 1 user defined type, namely T. This does not limit the generality of our result for two reasons. First, it is trivial to convert any program that uses k user defined types $T_1, \ldots, T_k$ into a program that uses a single type T which contains all of their fields.[2] Second, to obtain a lower bound, it suffices to apply the transformation to programs that use a single type which has 2 recursive fields: determining may- and must- aliasing for these programs is undecidable [Ramalingam 1994].

The crux of the simulation is an encoding of the program variables by a linked list. The list nodes are of type `VarList` (see Figure 5). Every node has 2 fields: `n`, a successor field, and `px`, a data field. The value of a variable x_i is encoded by the `px`-field of the $(i-1)$-th list element. The transformation produces a program which uses (only) the 4 variables shown in Figure 5.

We transform the control flow graph of program P into that of B using the following procedure:

[2]Without loss of generality, we can assume that fields have unique names.

(1) B starts by constructing the list which encodes P's variables. This is done by a code sequence (*i.e.*, a chain of edges labeled by primitive statements) which repeats the following statements v times:[3]

$$\texttt{t=hd};\ \texttt{hd=alloc VarList};\ \texttt{hd}\rightarrow\texttt{n=t};$$

Note that in the memory state the results after the execution of the above code sequence `hd` points to a list with v nodes. The `px`-field of every list node has the value *null*, which is the value a variable should have when the program starts.

(2) Every edge e is replaced by a code sequence which simulates the statement $st = M_P(e)$. Specifically, for a statement st with a left-hand operand `p_l` and right-hand operand `p_r`, we generate the following code sequence:

`rh` $= getVarVal(r)$;	Retrieves the current value of `p_r`, as encoded in the list, into `rh` .
`lh` $= getVarVal(l)$;	Retrieves the current value of `p_l`, as encoded in the list, into `lh`.
$st\|_{\texttt{lh/p\_l,rh/p\_r}}$	Same operation as st, but with `lh` and `rh` replacing the left-hand operand and the right-hand operand, respectively.
`t→px=lh`	Stores the current value of `x` in the list of variables (optional).

We use $getVarVal(d)$, where $1 \leq d \leq v$, as shorthand for a code sequence that retrieves the value of variable `p_d` from the variable list, *i.e.*,

$z = getVarVal(d)$:	`t=hd`	initializes lookup
	`t=t→n`	Repeated $d-1$ times.
	...	
	`z=t→px`	Stores the encoded value of `p_d` into `z`.

In case st assigns a value to `p_l` (which is the usual case) we add the last statement, `t→px=lh`. Note that prior to its execution, `t` points to the `VarList` node that encodes the value of `p_l`.

(3) The edge $e_{check} = \langle n_{check}, n'_{check}\rangle$, the only edge which originates from the CFG node in which we ask the aliasing question in P regarding `p_i` and `p_j` (see Figure 5), is replaced by the following code sequence:

`lh` $= getVarVal(i)$;	Sets `p_i` to its current (*i.e.*, encoded) value.
`rh` $= getVarVal(j)$;	Sets `p_j` to its current (*i.e.*, encoded) value.
`// lh == rh ?`	

This code sequence stores the current values of the variables `p_i` and `p_j`, as stored in the list, into `lh` and `rh`, respectively. The alias question in B is asked right after this code sequence.

Clearly, program B simulates the execution of program P. Thus, the following theorem is immediate.

THEOREM 4.3. *The flow-sensitive aliasing solution for B coincides with the flow-sensitive aliasing solution for P.*

Applying the above transformation to an arbitrary program P, which has a single user defined type T that has 2 recursive fields, results in a program B which has 4 variables and

[3]Recall that program P has v variables.

4 fields (T's 2 fields and `VarList`'s 2 fields.) Applying the transformation of Section 4.1 to program B results in a program Q which has 8 fields (program B's 4 fields and one field for every one of its variables). Recall, however, that Q uses 3 user defined types: T, which has 2 recursive fields; `PState`, which has 2 fields of type pointer to `VarList` and 2 fields of type pointer to T; and `VarList`, which has 1 field of type pointer to T and 1 recursive field. Obviously, we can replace these 3 data types by a single data type which has 4 recursive fields. Furthermore, the aliasing solution to (the modified) program Q also yields the aliasing solution to program P. Thus, the following corollary is immediate.

COROLLARY 4.4. *Precise $(3, 3)$-partially-flow-sensitive may-alias and must-alias analyses are undecidable in the presence of dynamic memory allocation for programs with 4 fields.*

5. REDUCTION WITHOUT DYNAMIC ALLOCATION

In this section, we consider programs that do not use dynamic memory allocation. In this case, we present a transformation, similar in spirit to the one given in Section 4, that does not use dynamic memory allocation. Flow-sensitive may-alias analysis is PSPACE-complete for pointer programs with records and recursive fields [Landi 1992a; Muth and Debray 2000]. Our reduction shows that a $(5, 2)$-partially-flow-sensitive may-alias analysis is as hard as a flow-sensitive analysis. Obviously, it cannot be harder. Thus, we can establish that $(5, 2)$-partially-flow-sensitive may-alias analysis is PSPACE-complete.

Again, we present the reduction in two stages: Section 5.1 gives a transformation that uses an unbounded number of fields and Section 5.2 bounds their number.

5.1 A Reduction with an Unbounded Number of Fields

A program consists of type definitions, variable declarations and a CFG, just as in Section 4. However, the program does not use a heap or dynamic memory allocation. Instead, record variables can be declared and have their address assigned to pointer variables. Specifically, we use the statement `p=&rec` which assigns the address of the record variable `rec` to the pointer variable `p`.

The main idea behind the transformation of a program P into a block-partitioned program Q such that the block-flow-sensitive solution for Q yields the flow-sensitive solution for P is similar to the idea behind the transformation in Section 4.1. However, instead of using a heap-allocated record to store the values of P's variables, we use P's variables. This eliminates the use of dynamic memory allocation in the transformation. However, this introduces a few problems in the reduction.

We noted in Section 4.1 that execution along an arbitrary sequence ξ simulates multiple executions along one or more realizable paths in the CFG. There was no interference between these simulations (in the original transformation) because they operate on different state records from which disjoint parts of the heap were reachable. However, since the current transformation uses P's variables, this is no longer true; the executions of blocks in arbitrary order will have an effect on each other.

We address this problem by ensuring that an execution along any sequence ξ of edges in Q's CFG will result in a null-dereference unless it "corresponds" to a path in the program P. We noted that with the original transformation, it was possible to follow a path α in Q, and then start following a new path β, and to then resume execution along path α. We will avoid this possibility with the new transformation by simulating the progress of a program counter that may only point to a single program location. This will ensure that the program never resumes an interrupted execution. Again, the precise treatment of null-valued pointer dereferences will play a key role.

Before we formally define the transformation, we illustrate it using the CFG fragment shown in Figure 6(a). This CFG fragment consists of an edge e_1 whose target is a branch node with two successor edges e_2 and e_3, where the edges are labeled with statements st_1, st_2, and st_3, respectively. The transformation generates special *action-records* $\mathtt{at}_1$,$\mathtt{at}_2$, and $\mathtt{at}_3$ for each one of the edges e_1,e_2, and e_3, respectively. The action records are depicted in Figure 6(c-b). These records have fields named $\mathtt{stmt}_1$, $\mathtt{stmt}_2$, $\mathtt{stmt}_3$, and `next` (and possibly other fields, depending on the rest of the program). The relationship between an edge e_i and its corresponding action-record, $\mathtt{at}_i$, is encoded by creating a self reference at $\mathtt{at}_i$ using the $\mathtt{stmt}_i$-field. Lines 1,4, and 5 in Figure 6(d) show the code that is generated by the transformation to create these self references. (Every line of the code comprises a single block.)

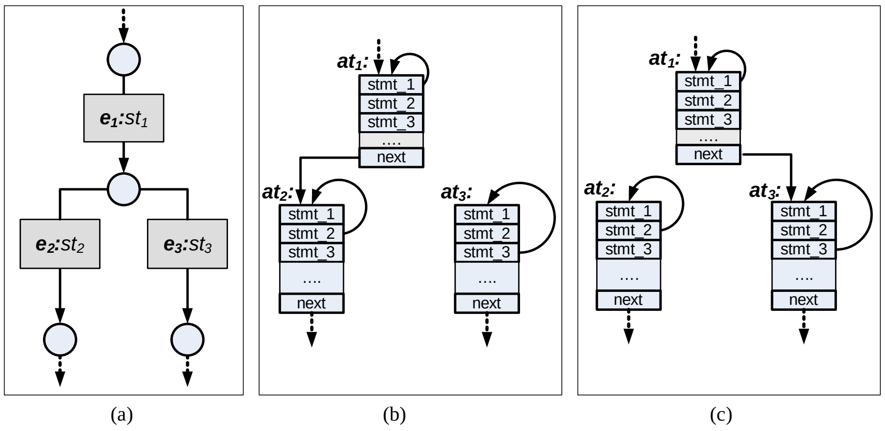
The fact that in the CFG fragment (only) edges e_2 and e_3 can be executed following edge e_1 is encoded by having the `next`-field of the action-record $\mathtt{at}_1$ point to one of the action records corresponding to one of these edges. Lines 2 and 3 in Figure 6(d) show the code that is generated by the transformation to update the `next` field of the $\mathtt{at}_1$ action-record. Figure 6(b) and Figure 6(c) depict the state of the action records $\mathtt{at}_1$,$\mathtt{at}_2$, and $\mathtt{at}_3$ after the execution of lines 1,4,5 followed by line 2 or line 3, respectively. (Recall that in a block-sensitive execution lines can be executed in an arbitrary order.)

Figure 6(e) shows how action records are utilized in the simulation of the code fragment shown in Figure 6(a). Lines 6, 7, and 8 are used to simulate statements st_1, st_2, and st_3, respectively. The pointer variable `pc` acts as the program counter. It points to the *current* action-record: the action-record corresponding to the edge which is labeled by the next statement to be executed. Every line of code (*i.e.*, block) begins with a *guard* which traverses the self reference and then sets `pc` to the `next`-field of the current action-record, effectively advancing the program counter. For example, executing line 6 when the `next`-field of action-record $\mathtt{at}_1$ points to $\mathtt{at}_2$ (resp. $\mathtt{at}_3$), as depicted in Figure 6(b) (resp. Figure 6(c)), leads to the execution of st_1 followed by the execution of st_2 (resp. st_3). Note, however, that an attempt to execute either line 7 or line 8 when `pc` points to the action-record $\mathtt{at}_1$ results in a null-dereference. This demonstrates how the guard ensures the orderly execution of statements.

The transformation of a program P into program Q is given in Figure 7. In the following, we assume without loss of generality that the original program P starts with an initialization section in which all pointer variables and all fields of all record variables are nullified. In addition, we assume that P's CFG has no sink nodes.[4] We also assume that there is a single aliasing query that we are interested in, at a specific program point, n_{check}, which is the source of a single `nop`-labeled edge $e_{check} = \langle n_{check}, n'_{check}\rangle$. The node n_{check} is not part of the initialization section.

The transformation encodes the control-flow of the original program P by representing CFG edges as records, and the connections between edges as pointers. We start by introducing an action-record for representing a CFG edge. Every edge e in P's CFG is matched with an action-record variable. The field $\mathtt{stmt}_e$ encodes this matching. The field $\mathtt{stmt}_e$

[4]The assumption that P's CFG has no sink nodes does not limit the generality of out result: any sink can be augmented with a self `nop`-labeled edge without affecting the aliasing solution.



$1:\ \texttt{pb} = \&\texttt{at}_{e_1};\ \texttt{pb}\rightarrow\texttt{stmt\_1} = \texttt{pb};$
$2:\ \texttt{pb} = \&\texttt{at}_{e_1};\ \texttt{pb}\rightarrow\texttt{next} = \&\texttt{at}_{e2};$
$3:\ \texttt{pb} = \&\texttt{at}_{e_1};\ \texttt{pb}\rightarrow\texttt{next} = \&\texttt{at}_{e3};$
$4:\ \texttt{pb} = \&\texttt{at}_{e_2};\ \texttt{pb}\rightarrow\texttt{stmt\_2} = \texttt{pb};$
$\quad\ \ldots$
$5:\ \texttt{pb} = \&\texttt{at}_{e_3};\ \texttt{pb}\rightarrow\texttt{stmt\_3} = \texttt{pb};$
$\quad\ \ldots$

(d)

$\quad\ \ldots$
$6:\ \texttt{pc} = \texttt{pc}\rightarrow\texttt{stmt\_1};\ \texttt{pc} = \texttt{pc}\rightarrow\texttt{next}; st_1;$
$7:\ \texttt{pc} = \texttt{pc}\rightarrow\texttt{stmt\_2};\ \texttt{pc} = \texttt{pc}\rightarrow\texttt{next}; st_2;$
$8:\ \texttt{pc} = \texttt{pc}\rightarrow\texttt{stmt\_3};\ \texttt{pc} = \texttt{pc}\rightarrow\texttt{next}; st_3;$
$\quad\ \ldots$

(e)

Fig. 6. Illustrating the transformation. (a) A fragment of a CFG. (b) Encoding of edge e_2 as the successor of edge e_1. (c) Encoding of edge e_3 as the successor of edge e_1. (d) Construction of the action-records. (e) Simulation of statement execution.

has a non-null value only in action record $\texttt{at}_e$, where it points back to the record $\texttt{at}_e$ in which it is contained. The action-record also uses a pointer field $next$ to point to the next action to be executed. A precise definition of the action-record is given in Figure 7.

Program Q consists of two main parts. The first part consists of the E_B and E_G edges (see Figure 7). The second part is a copy of P's CFG. The e_0 edge connects the two parts.

The first part is responsible for setting the $\texttt{stmt}_e$ and $\texttt{next}$ fields. Note that the $\texttt{stmt}_e$ field is always assigned the same value (Specifically, the $\texttt{stmt}_e$ field of record variable $\texttt{at}_e$ is always assigned the address of $\texttt{at}_e$.) The $\texttt{stmt}_e$ fields are assigned by the E_B edges. In contrast, the $\texttt{next}$ field of an action-record matching an edge e can be set to the addresses of any of the action records that match the edges following e in P's CFG. In a block-flow-sensitive execution, the assignments to the $\texttt{next}$ fields allow to create all the possible executions in P, and just these executions. The $\texttt{next}$ fields are assigned by the E_G edges.

The e_0 edge fires off the simulation of P's executions. It sets $\texttt{pc}$, the "program counter", to the address of the first edge in the CFG. Note that in a block-flow-sensitive execution, this statement can happen at any stage. However, because a program always starts with an initialization section, all the values that might have been stored in P's variables prior to the execution of the e_0 edge have been nullified.

The second part, *i.e.*, the copy of P's CFG, is responsible for executing P's statements. Every edge $e \in E_P$ is mapped to a block comprised of a transition guard and the e's original statement in P. The transition guard ensures that when $\texttt{pc}$ points to an action-

Component		**Program P**	**Program Q**	**Remark**

Type definitions		type T {T* f; ... }
		...
			type T {T* fld; ... }
			...
			type Action {
			Action* stmt_1;
			...
			Action* stmt_k;
			Action* next;
			}
				Program Q contains all type definitions of program P, plus a new type called Action that is used to represent CFG edges. An action has an stmt_i field of type pointer to Action for every edge e_i in P's CFG

Variables		T x ...	T x ... Action at$_e$ Action* pc Action* pb Action* pbn	Program Q contains all the variables in P, and a record variable of type Action for every edge $e \in E_P$; a "program counter", pc; and the "program builders", pb and pbn
CFG	Nodes	$N_P = \{n_1, \ldots, n_m\}$	$N_Q = N_P \cup N_B \cup N_G \cup \{b_{k+1}\}$ $N_B = \{b_i, b'_i \mid 1 \leq i \leq k\}$ $N_G = \left\{ g_{i,j} \;\middle\|\; \begin{array}{c} e_i, e_j \in E_P \\ e_i = \langle n_a, n_b \rangle \\ e_j = \langle n_b, n_c \rangle \end{array} \right\}$	Q's CFG is comprised of P's CFG augmented with b_i and $g_{i,j}$ nodes. There is a b_i node and a b'_i node for every edge $e_i \in E_P$. The b_{k+1} node ($k = \|E_P\|$) separates P's original CFG from the added nodes. There is a node $g_{i,j}$ for every pair of consecutive edges e_i, e_j in P. The E_G edges "guess" a path in P's CFG. The E_B edges construct actions corresponding to P's statements
	Entry	n_1	b_1	
	Edges	$E_P = \{e_1, \ldots, e_k\}$	$E_Q = E_P \cup E_B \cup E_G \cup \{e_0\}$ $E_B = \{\langle b_i, b'_i \rangle \mid 1 \leq i \leq k\}$ $E_G = \{\langle b'_i, g_{i,j} \rangle \mid g_{i,j} \in N_G\} \cup \{\langle g_{i,j}, b_{i+1} \rangle \mid g_{i,j} \in N_G\}$ $e_0 = \langle b_{k+1}, n_1 \rangle$	
	Map	M_P maps P's edges to primitive statements	M_Q maps Q's edges to blocks of primitive statements	$M_Q(e)$ is $build(e_i)$ if $e = \langle b_i, b'_i \rangle$, $nxt(e_i, e_j)$ if $e = \langle b'_i, g_{i,j} \rangle$, nop if $e = \langle g_{i,j}, b_{i+1} \rangle$), pc = &stmt$_{e_0}$ if $e = e_0$, and $block(e)$ if $e \in E_P$
Alias question		Are x and y may- (resp. must-) alias at node n_{check}?	Are pc→stmt_c→x and pc→stmt_c→y may- (resp. must-) alias ?	Edge e_{check} originates from node n_{check}

Fig. 7. A transformation of an arbitrary program P into a $(5, 2)$-block-partitioned program Q without using dynamic allocation. For an edge $e = \langle b_i, b'_i \rangle$, $build(e)$ is pb=&stmt_e$_i$; pb→stmt_i=pb. If, however, e originates from the node in which we ask the aliasing question, $build(e)$ is pb=&stmt_e$_i$; pb→stmt_i=pb; pb→check=pb. For an edge $e = \langle b'_i, g_{i,j} \rangle$, $nxt(e)$ is pb=&stmt_e$_i$; pbn=&stmt_e$_j$; pb→next=pbn. For edge $e_h = \langle n_i, n_j \rangle \in E_P$, $block(e_h)$ is pc=pc→stmt_h; pc=pc→next; $M_P(e_h)$.

record at$_{e_h}$, the only statement that can be executed is the one labeling e_h in P, *i.e.*, $M_P(e_h)$. Specifically, the transition guard of edge $e_h \in E_P$ is pc = pc→stmt_h; pc = pc→next. The guard traverses the field stmt_h before it advances pc. Because the only (possibly) non-null stmt field in at$_{e_h}$ is stmt_h, an attempt to execute any statement other than $M_P(e_h)$ when pc points to at$_{e_h}$ will lead to a null-dereference.

Clearly, every flow-sensitive execution of program P, has a corresponding block-sensitive execution of program Q. Furthermore, every block-sensitive execution of program Q corresponds to a series of flow-sensitive executions of program P, where each execution starts from a memory state in which all the pointer variables and all the pointer fields have a `null` value. Thus, the following theorem is immediate:

THEOREM 5.1. *The block-flow-sensitive aliasing solution for Q coincides with the flow-sensitive aliasing solution for P.*

Let us now measure the flow-sensitivity of the program Q. Any primitive statement requires at most 2 read operations and 1 write operation. The transition guard for an edge e can be encoded by the following 4 operations consisting of 3 read operations and 1 write operation:

$$r_1 = pc;\ r_2 = r_1 \rightarrow stmt_e;\ r_3 = r_2 \rightarrow next;\ pc = r_3$$

Every "build" edge and every "nxt" edge can be encoded using a single write operation. (Recall that getting the address of a record variable does not require a read memory access to the store.)

The aliasing question `pc`→`stmt_c`→`x == pc`→`stmt_c`→`y` can also be encoded using a total of 3 read operations and 1 write operation:

$$r_1 = pc;\ r_2 = r_1 \rightarrow stmt\_c;\ r_3 = r_2 \rightarrow x;\ r_4 = r_2 \rightarrow y;\ equal = compare(r_3, r_4)$$

The following corollary follows immediately:

COROLLARY 5.2. *Precise $(5, 2)$-partially-flow-sensitive may-alias analysis is PSPACE-complete for pointer programs that do not use dynamic memory allocation.*

Note: It is possible to ask an alternative aliasing question, one which only requires determining information regarding aliasing of variables, using a slightly more complicated transformation. The new transformation is similar to the one described at the end of Section 4.1.

We add to program Q two *pointer variables*, say `at_c_x` and `at_c_y`, of the same types as `x` and `y`, respectively. Variables `at_c_x` and `at_c_y` capture the values of `pc`→`stmt_c`→`x` and `pc`→`stmt_c`→`y`, respectively, whenever the program counter "points" to edge e_{check}. The values of `pc`→`stmt_c`→`x` and `pc`→`stmt_c`→`y` can be captured using the following code block:

$$r_1 = pc;\ r_2 = r_1 \rightarrow stmt\_c;\ r_3 = r_2 \rightarrow stmt\_c;$$
$$r_4 = x;\ r_5 = y;\ \texttt{at\_c\_x} = r_4;\ \texttt{at\_c\_y} = r_5\ .$$

This code block does *not* advance the program counter. Thus, it can be encoded using 5 reads and 2 writes. The modified aliasing question is *are `at_c_x` and `at_c_y` may- (resp. must) alias?* Note that `at_c_x` and `at_c_y` do not participate in the simulation. They function as "place holders" that can be assigned the values of `x` and `y` whenever the program counter "points" to edge e_{check}. Because they are always assigned as a unit, they preserve both may- and must- aliasing information regarding `pc`→`stmt_c`→`x` and `pc`→`stmt_c`→`y` in program Q, and thus, regarding `x` and `y` in program P.

5.2 Bounding the Number of Fields

In this section we show that precise $(5, 2)$-partially-flow-sensitive may-alias analysis is PSPACE-complete in the absence of dynamic memory allocation even when the number of fields is bounded. This result is not implied by Theorem 5.1 because the number of fields used by the transformation in Section 5.1 is proportional to the size of the transformed program P. Specifically, the `Action` record has a `stmt`-field for every edge in P's CFG.

Examining the reduction in Section 5.1, we notice that having a field for every CFG edge is, in a sense, redundant. The *only* role of the guard field `stmt_i` is to ensure that $M_P(e_i)$, the statement labeling edge $e_i = \langle n_a, n_b \rangle$, can be executed only when the program counter (`pc`) points to n_a. (This is achieved by preceding the execution of $M_P(e_i)$ with a dereference of `stmt_i`.) However, we can achieve a similar effect by using a field for every unique *statement* $st \in \{M_P(e) \mid e \in E_P\}$ in P instead of having one for every *CFG-edge*, $e \in E_P$. The guard in the block code pertaining to a CFG-edge e_i will be `pc=pc`$\rightarrow$`stmt`$_{M_P(e_i)}$; `pc= pc`$\rightarrow$`next`. This makes the number of fields required by the reduction proportional to the number of different statements in program P.

Unfortunately, the number of different statements in a program is also unbounded. It depends on the number of variables and the number of fields in the program. In the rest of this section, we show how to bound the number of statements in P that *need to be guarded* in the simulated program.

We begin by first bounding the number of fields used in P. We can assume the program has only one user defined type, namely T. This does not limit the generality of our result (see Section 4.2). Furthermore, we can assume that T has at most 2 fields: any type with k fields $f_1, \ldots, f_k$ can be represented by a list with k elements. Every list element has 2 recursive pointer fields: a successor field a data field. The value of pointer field f_k is recorded by the data field of the $k-1$ list node.

To bound the number of pointer variables, we will use the same idea as in Section 4.2 and encode the values of these variables in a list comprised of `VarList` record variables. Unfortunately, such a transformation will not do for record variables: taking the address of a variable requires specifying its name. This means that the number of different statements inherently depends on the number of record variables.

Before we describe how to overcome the aforementioned obstacle, we make some simplifying assumptions regarding program P. These assumptions do not affect the generality of our result. We assume that P has a pointer variable `p_i` for every record variable `rec_i`. Furthermore, P consists of 2 parts: P_1 followed by P_2. P_1 assigns the address of every record variable to its corresponding pointer variable. We refer to the pointer variable `p` that corresponds to record variable `rec` as the *constant record pointer* corresponding to `rec`. P_2, which is the rest of the program, never uses the address-of operator. Instead, whenever the address of a record variable is needed, the value of the corresponding constant record pointer is used. P_2 also consists of 2 parts: P_2^{init} followed by P_2^{prog}. P_2 starts its execution by a code sequence P_2^{init} which nullifies all the fields of all the record variables and all the pointer variables, except the constant record pointers. The latter are never modified by P_2. We assume that P has totally v pointer variables `p_1`,..., `p_v`.

We are now ready to describe the transformation. We encode P's variables by a linked list of `VarList` nodes in the same way we did in Section 4.2. This results in a program that utilizes the 4 *pointer* variables defined in Figure 5, the same *record* variables used by program P, and $v+1$ new record variables `vl`$_1, \ldots,$ `vl`$_{v+1}$ of type `VarList`. Record variable `vl`$_i$ represents the value of P's pointer variable `p_i`. The `vl`$_{v+1}$ node is a *dummy* node.

We transform the control flow graph of program P into that of Program B. The latter consists of 2 parts: B_1 followed by B_2. The transformation is done as follows:

(1) B_1 constructs and initializes the list encoding P's variables. It consists of a chain of edges annotated with the following code blocks. First, B_1 links the `VarList` nodes. The i-th node, $1 \leq i \leq v$, is linked using the following code block:

$$\texttt{r}_1\texttt{=\&vl}_i;\ \texttt{r}_2\texttt{=\&vl}_{i+1};\ \texttt{r}_1\texttt{→n=r}_2$$

Then, for a node `vl_i` which records the value of the constant record pointer variable corresponding to a record variable `rec`, B_1 assumes the role of P_1 and assigns the address of `rec` into the `px`-field of `vl_i` using the following code block:

$$\texttt{r}_1\texttt{=\&vl}_i;\ \texttt{r}_2\texttt{=\&rec};\ \texttt{r}_1\texttt{→px=r}_2$$

Finally, B_1 assigns to `hd` the address of $\texttt{vl}_1$, the first node in the variable list using the statement $\texttt{hd = \&vl}_1$.
Note that in the memory state which results after the execution of B_1, the variable `hd` points to a list with $v + 1$ nodes. The `px`-field of every list node has the value *null*, unless that node represents a constant record pointer. In the latter case, the `px`-field points to the corresponding record.

(2) We transform P_2 into B_2 according to stages (2) and (3) of the transformation described in Section 4.2. Note that B_2 also starts in a code sequence B_2^{init} which nullifies all the variables and all the pointer fields of all the records; followed by the rest of the program, B_2^{prog}.

Clearly, the only difference between program P and program B is that while program P can find the value of a variable directly, program B has to do it by traversing the list variables. Thus, the following theorem follows immediately:

THEOREM 5.3. *The flow-sensitive aliasing solution for B coincides with the flow-sensitive aliasing solution for P.*

We now transform program B into a $(5, 2)$-block-partitioned program Q. The code sequence in B_1 is already partitioned into $(0, 1)$ blocks. Thus, we leave it intact. We transform the CFG of B_2 according to the transformation of Section 5.1, treating the entry node to B_2 as the entry node to the program.

Note that during a block-sensitive execution of program Q, the code blocks that construct the variable list can be executed any time and any number of times. However, their effect is always the same. The transformation of Section 5.1 ensures that the orderly execution of the program statements in B_2 is faithfully simulated. Furthermore, in any execution of Q, the construction of the variable list has to be completed prior to the simulation of B_2^{prog}. The reason for this is that B_2 begins by executing B_2^{init}. There, it traverses the variable list and the `px`-fields of nodes pertaining to constant record pointers. A successful traversal ensures that the list construction is completed. Note that although list-constructing code blocks can be executed later on, they will not modify either a field or the `pc` pointer. Thus, the following theorem follows immediately.

THEOREM 5.4. *The flow-sensitive aliasing solution for Q coincides with the block-flow-sensitive aliasing solution for B.*

Let us count the number of fields used in program Q. First, program B uses 4 variables and 2 types: T and `VarList`. Both have 2 recursive fields. Clearly, we can rewrite B to use a single type that has 2 recursive fields. Transforming program B into program Q

requires at most 13 different statements that need to be guarded.[5] Consequentially, the `Action` record in program Q has 14 fields. (Recall that we have one `Action` record for every CFG edge of B_2. Every record encodes a specific statement using one of the 13 `stmt` fields. It also stores its successor using the `next` field.) Clearly, program Q can be rewritten using a single record type with 13 fields. Thus, the following corollary follows immediately:

COROLLARY 5.5. *Precise $(5, 2)$-partially-flow-sensitive may-alias analyses are PSPACE-complete for programs with 14 fields.*

6. CONCLUSIONS AND OPEN PROBLEMS

In this paper, we define a notion of partially-flow-sensitive analysis. We define a degree of flow-sensitivity based on the maximum number of memory locations read and the maximum number of memory locations written in a block which is guaranteed to be analyzed in a sequential manner. We show that precise alias analysis with a very limited flow-sensitivity is as hard as flow-sensitive alias analysis.

The numerical bounds we found are not absolute. In particular, alternative measurements lead to different bounds. For example, measuring a block by the number of statements it contains shows that precise 5-flow-sensitive may-alias and must-alias analyses are undecidable in the presence of dynamic memory allocation.[6] We chose a measurement which strikes us as being both simple and intuitive, yet sensitive enough to make certain seemingly important distinctions between blocks (see Section 2.4 and 2.6) but not too sensitive. Changing the assumption that every memory cell in the initial store contains a *null* value (see Section 2.2.1.1), may also affect our results by a constant factor.

Several interesting questions regarding partially-flow-sensitive analysis are left open. An interesting family of analyses for which we do not have lower bounds are $(k, 1)$-partially-flow-sensitive alias analyses. These analyses coincide with the standard flow-insensitive analyses as they require analysis of *single* assignments involving up to $k-1$ field dereferences while respecting the order of field dereferences in every assignment. Another open question is to find tighter lower bounds on the number of fields that a program can use. Our reductions use recursive data structures. An interesting question is whether similar bounds can be shown for programs with multi-level non-recursive pointers.

The key aspect underlying our reductions is the fact that the analysis is required to handle "null-pointer dereferences" accurately: If a particular interleaving of blocks leads to a null-pointer dereference, the analysis is not supposed to continue on with analysis along this path. This argues that an analysis which ignores possible null-pointer dereferences and keeps on analyzing paths causing such dereferences may not be subject to the complexity results of this paper. This is one possible over-approximation of the actual program executions that may be helpful in analysis design.

[5] Only the statements of program B_2 need to be guarded. According to stages (2) and (3) of the transformation described in Section 4.2, program B_2 is comprised of (a subset of) the following 13 statements: (i) 5 statements are used to manipulate the `VarList`: `t=hd`, `t=t→n`, `rh=t→px`, `lh=t→px`, `t→px=lh`; (ii) 7 statements are the original statements, with `lh` and `rh` replacing the left-hand and the right-hand operands: `noop`, `x=NULL`, `x=y`, `x=y→`f, `y→`f`=x`, where f is one of the 2 recursive fields, *i.e.*, either `px` or `n`; (iii) 1 statement is needed to `stmt_c`, the edge in which the aliasing question is asked.

[6] In Section 4.1, we show this result for $(3, 3)$-partially-flow-sensitive analyses. Note that in the number-of-statement measurement, we can gain a better bound by placing 2 non-local read operations in 1 statement. Our measurement, on the other hand, is insensitive to this sort of changes, which we consider to be a merit.

While this paper focuses only on some theoretical aspects, namely lower bounds, of analyses for block-partitioned programs, we believe that partially-flow-sensitive analyses could be a promising approach to striking a balance between scalability of flow-insensitive analyses and precision of flow-sensitive analyses. Furthermore, our reduction techniques can help in the design of new analyses. They can allow the analysis designers to focus on a restricted version of the problem and develop analyses for the restricted version (*i.e.*, develop partially-flow-sensitive analyses). Then, using our reductions, they will be able to apply their analysis to arbitrary programs. (This is similar to the way the SSA transformation allows one to achieve flow-sensitive analysis using a flow-insensitive analysis in certain contexts in the absence of pointer indirection.)

Acknowledgments. We are grateful for the helpful comments of the anonymous referees.

REFERENCES

ANDERSEN, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, Univ. of Copenhagen. (DIKU report 94/19).

CHAKARAVARTHY, V. 2003. New results on the computability and complexity of points–to analysis. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 115–125.

HORWITZ, S. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems 19,* 1 (January), 1–6.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA.

LANDI, W. 1992a. Interprocedural aliasing in the presence of pointers. Ph.D. thesis, Rutgers University.

LANDI, W. 1992b. Undecidability of static analysis. *Let. on Prog. Lang. and Syst. 1,* 4, 323–337.

MILNE, R. AND STRACHEY, C. 1977. *A Theory of Programming Language Semantics*. Halsted Press, New York, NY, USA.

MUTH, R. AND DEBRAY, S. K. 2000. On the complexity of flow-sensitive dataflow analyses. In *Symposium on Principles of Programming Languages*. 67–80.

RAMALINGAM, G. 1994. The undecidability of aliasing. *Trans. on Prog. Lang. and Syst. 16,* 5, 1467–1471.

REYNOLDS, J. 2002. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*. 55–74.

STRACHEY, C. 1966. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*, T. B. Steel, Ed. North Holland, 198–220.