Defining Flow Sensitivity in Data Flow Problems

Thomas J. Marlowe*
Barbara G. Ryder†
Michael G. Burke‡

July 17, 1995

Abstract

Since Banning first introduced *flow sensitivity* in 1978, the term has been used to indicate hard or complex data flow problems, but there is no consensus as to its precise meaning. We look at Banning's original uses of the term and some interpretations they have generated. Then we consider the multiplicity of meanings in more recent interprocedural analyses, categorizing a number of data flow problems. We also classify several recent interprocedural approximation techniques with respect to properties related to sensitivity and discuss additional data flow problem properties. Finally, we propose a definition for flow sensitivity that appears to capture much of the original intent and current use.

1 Introduction

The distinction between *flow sensitive* and *flow insensitive* data flow problems was introduced by Banning [Ban78, Ban79]. Although no uniform definition has ever been given, the terms have been used as shorthand in expressing the locality or summarizability of information, the applicability of algorithms, and the difficulty of describing and computing the data flow framework. In the latter usage, sensitivity can be taken as a characterization of the *difficulty* of a data flow problem; however, without a consistent meaning for the term, it is hard to understand the relative difficulties of some problems.

*Department of Mathematics and Computer Science, Seton Hall University, South Orange NJ 07079 *(marlowe@cs.rutgers.edu)*
†Department of Computer Science, Rutgers University, Hill Center, Busch Campus, Piscataway, NJ 08855 *(ryder@cs.rutgers.edu)*
‡IBM Thomas J. Watson Research Laboratory, Yorktown Heights NJ 10598 *(burkem@watson.ibm.com)*

Overview of the paper The concept of sensitivity appears to be useful, particularly for interprocedural problems, but also not amenable to easy formalization. Proceeding from a historical perspective, we first review, in Sections 2 and 3, prior usages for intraprocedural problems, and the difficulties in formalizing them. We suggest that none of these approaches are likely to offer useful or illuminating distinctions.

We then consider interprocedural analyses, where recent formulations have used many different meanings of flow sensitivity. We state important distinctions between problem flow sensitivity and algorithm flow sensitivity, and between flow sensitivity and context sensitivity. In Section 4, we provide two common definitions of problem flow sensitivity; in Section 5, we treat flow sensitivity for approximation algorithms; and in Section 6, we discuss context sensitivity. Throughout, we show the interrelationships but fundamental independence of these classes, using example problems and algorithms from the literature.

In Section 7, we show that neither past characterizations of flow sensitivity, nor those we consider, necessarily provide information about the algebraic (e.g., monotone vs. distributive) or convergence (e.g., fast vs. k-bounded vs. unbounded) properties [MR91] of the data flow problem to be solved. We briefly discuss two other sources of complication for data flow problems, heterogeneity and bidirectionality, and how they relate to context and flow sensitivity. Finally, in Section 8, we suggest a characterization of flow sensitivity that captures much of the intuition we have about these terms, and the essence of most current use, and generalizes it to apply more broadly than just between programs and procedures.

The Data Flow Model *Data flow analysis* involves compile-time determination of the definition and use of variables by algebraic methods. This analysis is performed under the common assumption that all apparent execution paths in the program flow graph are actually *feasible* (i.e., traversable on some program execution); for interprocedural problems, however, paths are often restricted to realizable paths, so that calls match returns, at least approximately [LR91, LR92, RHS95]. An instance of a *monotone data flow framework* [Hec77, MR91] is specified by a tuple, $D = < G, L, F, M, \eta >$, where G is a rooted digraph $< V, E, \rho >$, L is usually a meet semilattice, F is a space of monotone (that is, order-preserving) functions mapping L into L, M is a mapping of edges E of G into F, and η is an element of L. If the functions of F also preserve meets, then D is called *distributive.*

Intuitively, G is the *control flow graph* [ASU86] or *call multigraph* [Hec77] with root at ρ, L a lattice of data flow solutions, M the assignment of transition functions to edges, and η the entry solution at ρ; F is usually not represented explicitly.

Often M can be thought of as specifying a system of $| V |$ equations [RP86]. In *forward* data flow problems, information flows in the direction of flow graph edges; in *backward* problems, information flows in the opposite direction. The definitions and examples in this paper are, without loss of generality, mostly in terms of forward data flow problems.

The form of a typical equation for a forward data flow problem is

$$S(X) \;=\; \bigcap_{(Y,X)\in E} f_{(Y,X)}(S(Y))$$

(where $S(\rho) \;=\; \eta$). The equation is *linear* if there are constants $A_{(Y,X)}$ and $B_{(Y,X)}$ so that

$$f_{(Y,X)}(S(Y)) \;=\; A_{(Y,X)} \otimes S(Y) \oplus B_{(Y,X)}$$

(for associative operations $\oplus$ and $\otimes$ with appropriate algebraic properties [RP86]).

For monotone problems, solution techniques find the maximum (or minimum) fixed point (MFP); for distributive problems, this will also be the *meet over all paths* (MOP) solution. A data flow framework is *k-bounded* if, intuitively, fixed point iteration requires $k+1$ iterations around a cycle to stabilize; *fast* is 2-bounded [MR91].

Standard Data Flow Problems In the rest of this paper, we will be using a number of standard data flow problems. The best known data flow problems are the four classical intraprocedural problems: Reaching Definitions, Live Uses, Available Expressions, and Very Busy Uses [ASU86]. Reaching Definitions collects the set of definitions (assignments to a variable) reaching a given program point — intuitively, reaching a given use of the variable (assuming all paths are feasible, as discussed above); dually, Live Uses collects the set of uses reached by the definitions live at a given program point, (again intuitively, reaching a given definition). Available Expressions determines for any given program point, the set of expressions a valid value for which has been computed and survives along every path to the given program point, and Very Busy Uses the set of expressions which will need to be evaluated along every terminating path from a program point, and for which the needed value can be produced at that point. Constant Propagation determines where a use of a variable can safely be replaced by a constant; Range Propagation determines a range to which the values taken on by a variable at a given use can be restricted.

The best known interprocedural problems are Formal Bound Set, MayAlias and MayBeModified$_F$ [Bur90, CK84, CK89, CK87]. Formal Bound Set asks, for all procedure formals, which formals can be reached over call chains from this given formal. The MayAlias problem determines the set of *alias pairs* (x, y) so that x and y may refer to the same storage location at a given program point; for pass-by-reference parameters and unstructured variables (and no pointers), it is sufficient to determine the pairs true at entry to each procedure. There are two other variants which have somewhat different data flow properties: the MustAlias and MayAlias Partitions problems. In the simple case, MustAlias determines alias pairs which will hold on all invocations of a given procedure. MayAlias Partitions [Mye81] determines sets of variables at a program point such that exactly that set of variables addresses the same location along some execution path. MayBeModified$_F$ for a procedure in a Fortran-like language calculates the set of variables whose values may be changed by execution of that procedure.

In addition, each of the classical problems has an interprocedural analogue. For example, Interprocedural Reaching Definitions determines the set of definitions, including definitions in other procedures, which can reach a given program point or use. Two other simple problems are also helpful for interprocedural data flow computation: MayBePreserved and MustBePreserved. The former determines the set of variables whose definition survives along some path from procedure entry to the current program point (typically program exit); the latter, the set of variables whose definition will always survive.

2 Banning's First Definition: An Algebraic Criterion

Attempting to give intuition for a new approach to interprocedural data flow, Banning [Ban79] gives three different, although related, informal definitions of flow insensitivity, specialized to the problems he is considering. These three definitions, and the variants that have arisen from them, are hard to formulate in the now-standard notation of data flow frameworks (or abstract interpretation); and the use of one or another in isolation, has led to the confusion we try to resolve here. We have nonetheless attempted to formalize and generalize them.

Banning's first definition deals with the composition of side effects for sequential and conditional execution, resulting in, correspondingly, sequential (straight-line) flow or merged flow (confluence) as shown in the graphs of Figure 1.

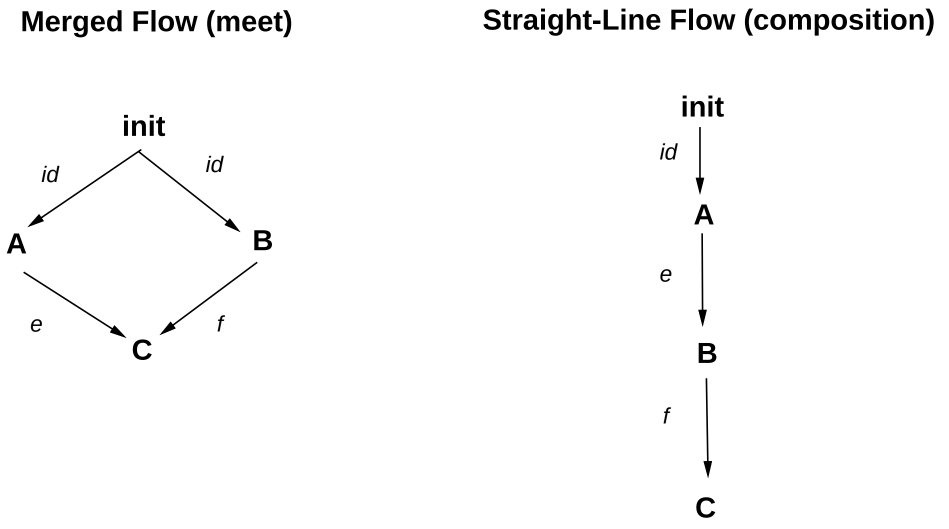

Figure 1: Meet and Composition Diagrams

In data flow framework terms, the flow function for a sequential flow is the composition of the individual flow functions; the flow function for the confluence of a set of alternatives is the semilattice meet of the flow functions. Banning seems (in this first definition) to be requiring for flow insensitivity that[1]

$$[\mathbf{B1}]\ \ (f_e \wedge f_f)(X) = (f_f \circ f_e)(X)\ \ \forall A, B, C, e, f$$

We will take the general form of [**B1**] as Banning's definition, since it is precisely defined but general, and is easily expressible in standard data flow framework terminology. Since all the forms of the definition rely on the algebraic equivalence of two functions, we term this definition the Algebraic Criterion [**AC**] for flow insensitivity. As stated, however, the

[1] Or perhaps:

$$[B1']\ f_e \wedge f_f = f_e\ op_1\ f_f;\ \ f_f \circ f_e = f_e\ op_2\ f_f;\ \ and\ op_1 = op_2.$$

In [**B1**], we can either restrict X to be η or $\top$ or $\perp$ (that is, consider only possible initial states), or allow X to have any value. In [**B1′**], we may allow op_1 and op_2 to be any operator, or restrict to just union or intersection, and so on.

criterion does not appear to be useful: only a small number of *collecting* problems[2] will be flow insensitive by this definition. Reaching Definitions, and all of the classical data flow problems, are flow sensitive by this definition, but the following intraprocedural problems are flow insensitive:

- MayBeDefined: x may be defined at vertex v if a definition of x occurs on some path along which it reaches v.
- IsNotDefined or MustBePreserved: x must be preserved at vertex v if no definition of x occurs on any path along which it can reach v.
- MustBeReferenced: The variable x occurs (in a definition or use) in every basic block on every path to v.
- TotalMustBeReferenced: The variable x occurs in every basic block.

MayBeDefined is Reaching Variables, in contrast to Reaching Definitions: a new definition of x kills the old definition, but x will still reach v.[3] MustBePreserved is essentially the complement of MayBeDefined. TotalMustBeReferenced differs from MustBeReferenced in that reachability is not involved; the solution to TotalMustBeDefined is identical for all nodes v.[4]

Clearly (referring to Figure 1), x is in MustBeReferenced at the top of C, in either diagram, if and only if it is referenced in both A and B (and is in MustBeReferenced at the bottom of **init**). Likewise (assuming that, for convenience, x is not defined at the bottom of **init**), x is in MayBeDefined at the top of C if x is defined in either A or B, and x is in MustBePreserved at the top of C if x is not defined in either A or B.

The other three problems which collect basic block properties (MayBeReferenced, MayNotBeReferenced, MustNotBeReferenced) are also easily seen to be flow insensitive, since the value at v can be determined by examination of the basic blocks reaching v, without regard to their connectivity; the corresponding node-independent versions are trivial, but likewise flow insensitive.

Consider, in contrast, the following two problems:

- MustBeDefined: The variable x must be defined at node v if a definition of x reaching v occurs on every path which reaches v.
- MayBeUndefined or MayBePreserved: The variable x may be undefined at v if there is a path to v along which no definition of x reaches.

These two problems are flow sensitive by this criterion. For example, x is contained in MustBeDefined at the top of C in the Straight-Line Flow diagram if it is defined in either A or B, but in the Merged Flow diagram only if it is defined in both A and B.

[2]That is, problems that merely collect and combine information from basic blocks, and in which no datum is ever *killed* by the presence of another.

[3]If the language has a general *undef* operation (or has sub-procedural scopes, etc.), then MayBeDefined may also be flow sensitive by this definition.

[4]That is, MustBeReferenced is *node-specific*, while TotalMustBeReferenced is *node-independent*. Since TotalMustBeReferenced supplies no local information, and can be computed without propagating information along graph edges, it might not be considered to be a data flow problem at all.

All-Paths vs. Some-Paths and May vs. Must Problems MayBeDefined is a *some-path* problem, while IsNotDefined is an *all-paths* or perhaps *no-paths* problem; that is, for MayBeDefined, there must be *some path* along which the requisite information reaches; for IsNotDefined, there is *no path* along which contrary information can reach, and *all paths* are clear of that information. For similar reasons, MayBeDefined is called a *May* problem, since information *may* reach the vertex (but there may be paths along which it does not), and IsNotDefined is a *Must* problem, since information *must* reach along every possible path.

TotalMustBeReferenced does not really fit into the first paradigm; it is completely path-independent. It can, however, be classified as a *Must* problem. The main difference between MustBeDefined and MustBeReferenced is that MustBeReferenced is a criterion on basic blocks, and not on paths; we need only collect and examine reaching basic blocks. A related point is that expressions, for example, can be made unavailable by later statements (as definitions could have been, in the presence of *undefining* constructs); references cannot be invalidated by later constructs.

[**AC**] is more natural when the flow function on an edge is defined by the source vertex (rather than by the edge or the target vertex); otherwise, the two flow functions labeled e need not in general even have the same type signature. For example, in Formal Bound Set as usually posed, the flow function involves formal parameters of both the source and the target; since the targets of the two copies of e are not identical, the flow functions for the diagrams of Figure 1 have different ranges, and so cannot be the same. (By enlarging the lattice so that all flow functions have the same domain and range — the set of all formal parameters in the program — we can cover, but not avoid the problem. The real difficulty is that f_e in the Merged Flow diagram will modify bound sets corresponding to parameters of C, and in the Straight-Line Flow will modify bound sets for B; thus their flow functions f_e will be incomparable.)

Banning [Ban78, Ban79] gives two problems as illustrative of flow insensitive data flow, Mod and Ref, which are essentially Interprocedural MayBeDefined and MayBeReferenced, respectively (Banning's problems were originally defined at call sites). It is not however clear that these problems are flow insensitive by the Algebraic Criterion.

A Modified Algebraic Criterion We can relax [**B1′**] to simply require that the combining operators for composition and meet be the same (and allow the actual results to be different). To give this a consistent meaning we restrict to the domain of flow functions of linear form,

$$f_e(X) \;=\; (A_e \otimes X) \oplus B_e$$

Typically, the A terms are *multiplicative coefficients*, and identify how the neighboring solution $S(Y)$ is preserved, while the B terms are *additive constants*, and specify local information, and how it is to be incorporated into the final solution. Thus, in some sense, the $\oplus$ operator describes how new information from several nodes in sequential flow is included in the solution, and so can be thought of as the *composition operator* in a somewhat more general sense. If flow functions have linear form, and $\otimes$ has precedence over $\oplus$, and further $\otimes$ is associative and distributive[5] over $\oplus$, then a problem is *flow insensitive* by the Modified

[5]Alternatively, $\otimes$ is the same as $\oplus$, and is associative and commutative.

Figure 2: Modified Algebraic Criterion [**MAC**]

		COMPOSITION	
		$\cap$	$\cup$
MEET	$\cap$	Flow Insensitive	Flow Sensitive
	$\cup$	Flow Sensitive	Flow Insensitive

Algebraic Criterion [**MAC**] if $\oplus$ is the meet operator and *flow sensitive* if it is not.

We can see how [**MAC**] separates problems into classes by comparing two classical problems: Reaching Definitions, which will be flow insensitive by this definition, and Available Expressions, which will be flow sensitive. For Reaching Definitions, the meet is set union, and the flow functions have the form $f_e(X) = (A_e \cap X) \cup B_e$; $f_{a \wedge b}$ is therefore $((A_a \cup A_b) \cap X) \cup (B_a \cup B_b)$. For Available Expressions, flow functions have the same form, but meet is intersection; $f_{a \wedge b}$ is thus

$$(A_a \cap A_b \cap X) \cup (A_a \cap X \cap B_b) \cup (B_a \cap A_b \cap X) \cup (B_a \cap B_b)\ ,$$

whose algebraic form is much more complicated. It is relatively simple to combine the flow constants to obtain a summary flow function for a path for a flow insensitive problem by this definition, but less simple for a flow sensitive problem. We thus get the diagram of Figure 2.

Live Uses (cell (2,2)) and MustBePreserved (cell (1,1)) are both flow insensitive by [**MAC**], while Available Expressions (cell (1,2)) and MayBePreserved (cell (2,1)) are flow sensitive. Formal Bound Set is also flow insensitive (and in cell (2,2)), since the *lozenge* operator [RMP88] has the appropriate algebraic properties.

[**MAC**] also identifies only a limited class of problems as flow insensitive, and groups as flow sensitive relatively simple problems like Available Expressions together with much harder problems. Further, it is useful only for problems in which the flow functions have relatively simple form, and depends on the syntax chosen for the functions.[6] However, in comparison to [**AC**] or a stricter interpretation of *combining operator for composition*, it allows a wider set of functions, such as Formal Bound Set, to be considered as flow insensitive, and we need not be as concerned with signatures of flow functions.

3 Banning's Second Definition: The Collecting Criterion

Banning is principally interested in collecting interprocedural flow information, which he wants to do by first processing the call multigraph, and later looking, if necessary, at the

[6] For example, if we had written the classical problem flow functions as $(A' \cup X) \cap B'$ (where $A' = B$; $B' = A \cup B$), then Available Expressions would be flow insensitive, and Reaching Definitions would be flow sensitive.

Figure 3: Collecting Criterion [**CC**]

		COMPOSITION	
		$\cap$	$\cup$
MEET	$\cap$	Flow Sensitive	Flow Sensitive
	$\cup$	Flow Sensitive	Flow Insensitive

structure of the procedures. This leads him to the following Collecting Criterion [**CC**]: A data flow problem is flow insensitive if "the side effect of the whole [structure in a program] can be determined by processing the whole and later determining the side effects of the parts, and taking their union" [Ban78]; that is, if the individual contributions of substructures to the global solution are unioned regardless of flow structure. For intraprocedural analysis, the substructures are basic blocks; [**CC**] seems to say that a problem is flow insensitive if the problem uses union for both Merged and Straight-Line Flow, and the flow functions can be computed from summary information in basic blocks.[7] Of the first four example problems on page 5, only MayBeDefined is flow insensitive by [**CC**]. For bitvector problems, [**CC**] has the effect of Figure 3.

May Problem Criterion [**CC**] is apparently responsible for the May Problem Criterion [**MPC**], a definition of flow insensitivity appearing in a number of early papers [CK84, CK89]: *May*, or *some-path*, problems are flow insensitive, *Must*, or *all-path* and *no-path*, problems are flow sensitive. This definition may also have been influenced by terminology in [Mye81], in which Myers shows the MaySummaryProblem (MayBeModified$_F$) to be flow insensitive, and the MustSummaryProblem (MustBeModified) to be flow sensitive, using the [**IC**] criterion of Section 4.

Like [**CC**], [**MPC**] identifies only MayBeDefined among the first four example problems on page 5 as flow insensitive. For the bitvector class of problems, it gives the decomposition presented in Figure 4, which contrasts with Figures 2 and 3.

Although most of the problems identified as flow insensitive by [**MAC**] are in cell (2,2), and so will also be identified as flow insensitive by [**CC**] and [**MPC**], problems in cell (1,1) (e.g., MustBePreserved) and cell (2,1) (e.g., MayBePreserved) will be identified differently by the three definitions. Both [**CC**] and [**MPC**], moreover, will not be applicable to flow frameworks in which the operators, particularly the meet operator, are not easily expressible in terms of set operations.

Also, note that one of Banning's two examples of flow sensitive problems (namely, Live Uses of Variables, which he calls Use) is a *May* problem, and so is flow insensitive by [**MPC**]. [**MPC**] can also be taken as applying to problems for which meet is union, independent of

[7] This is not the same as MAC, since in both Merged and Straight-Line Flow there is a notion of "not-after", so annotations can some order information.

Figure 4: May Problem Criterion [**MPC**]

		COMPOSITION	
		$\cap$	$\cup$
MEET	$\cap$	Flow Sensitive	Flow Sensitive
	$\cup$	Flow Insensitive	Flow Insensitive

Figure 5: Intraprocedural Sensitivity Criteria on Example Problems

Problems	**AC**	**MAC**	**CC**	**MPC**
MayBeDefined	insens	insens	insens	insens
MustBePreserved	insens	insens	sens	sens
MustBeReferenced	insens	insens	sens	sens
TotalMustBeReferenced	insens	insens	sens	sens
MustBeDefined	sens	sens	sens	sens
MayBePreserved	sens	sens	sens	insens
Reaching Defs	sens	insens	insens	insens
Live Uses	sens	insens	insens	insens
Avail	sens	sens	sens	sens

the algebraic structure of flow functions, or of the operators used for composition, so will, for example, identify Formal Bound Set as flow insensitive.

Algebraic Criteria Considered The intuition behind [**MAC**] was that mixing of intersection and union in a problem makes computation of summary flow functions from sets of edges more complicated or harder. This leads to a possible approach for defining flow sensitivity in this way, but we do not recommend it for at least two reasons: (1) we have found it difficult to state a reasonable and general criterion (more particularly since the original criterion relied on syntactic and algebraic properties of flow functions); and (2) the distinctions made by [**AC, MAC, CC, MPC**] have not turned out to be useful in practice. On the other hand, there are useful, if not widely and fully understood, distinctions which can be applied to interprocedural problems and algorithms.

Comparisons of Criteria Figure 5 shows each of our six example problems on page 5 and lists their flow sensitivity with respect to the **AC, MAC, CC** and **MPC** criteria.

4 Banning's Third Definition: The Interprocedural Criterion

For interprocedural criteria in particular, a problem is flow insensitive with respect to a given criterion, if some solution procedure satisfying the criterion yields a precise (static) solution;[8] it can, of course, have alternative algorithms which do not satisfy the criterion. Moreover, problems which do not satisfy a given criterion (and so are flow sensitive for that criterion) may have algorithms which produce approximate solutions, for which the operationally defined approximate problem does satisfy that criterion, and possibly even stronger criteria (see Section 5).

Many of the preceding definitions of flow sensitivity apply equally well to intraprocedural and interprocedural problems. However, Banning also gives an alternate (and somewhat implicit) criterion for flow insensitivity, closely related to [**CC**], principally applicable to interprocedural problems. Banning states that, if a call occurs in the evaluation of a flow insensitive data flow problem, computation of the solution can defer computation of the call site's side effects; in terms of the lattice model, this is a statement about evaluation on the call multigraph. The literature refers to these calculations as *procedure summary annotations*; for clarity, we distinguish the terms *summary* and *annotation* below.

When we discuss possible interprocedural flow sensitivity criteria, we also must acknowledge the existence of purely interprocedural problems such as Formal Bound Set and May-Alias [Bur90, CK84, CK89, CK87, Mye81]. These problems involve *no* intraprocedural information. Therefore, using the criteria of previous sections, these can be considered *trivially* flow insensitive, since their solutions depend only on declarations of procedure formal parameters, and on parameter passing patterns in calls in the program, and these facts are represented in the call multigraph of the program.

Terminology For other problems, we introduce the following terminology. Intraprocedural information will be extracted into *procedure annotations*, which can then be evaluated with incoming interprocedural data flow information into *procedure summaries*. Procedure annotations are *invariant* if they are computed once; they are *constant* if the corresponding summaries are constant. If annotations lie in a lattice of constants, then any invariant annotation is itself constant; however, a functional annotation can be invariant without being constant.

A data flow problem is *flow insensitive for a given interprocedural criterion* if we can demonstrate a precise algorithm for it, which is not flow sensitive by that criterion.

Independent of Intraprocedural Control Flow Banning's definition appears to have led Myers [Mye81] to the Interprocedural Criterion [**IC**] for flow insensitivity: a problem is flow insensitive if it uses constant intraprocedural annotations, and its solution *does not depend on the intraprocedural structure of the program.* More precisely, the procedure annotation can be extracted without considering execution paths in the procedure, as if all

[8] In our discussions in this paper we are limiting our attention to a programming model in which structured variables (e.g., arrays) are treated as scalars and there are no function-valued variables or function pointers.

its statements were in a large non-deterministic switch statement within a while loop. This clearly includes all possible intraprocedural paths.

Myers shows that MayBeModified$_F$ satisfies this criterion for flow insensitivity, and has an efficient solution, while Interprocedural Live, Avail and MustBeModified are flow sensitive and NP- or co-NP-complete. Note that [**IC**] flow insensitivity is not directly related to *May* and *Must*: although MayBeModified$_F$ is a *May* problem, and Avail and MustBeModified are *Must* problems, Interprocedural Live is a flow sensitive *May* problem by [**IC**]. Since all of our example problems in previous sections are intraprocedural (or don't involve flow at all), the [**IC**] criterion cannot be applied to them.

Myers shows further that the intractability of the three problems considered is not due to flow sensitivity, but to the need for precise MayAlias Partition information; solution of the above flow sensitive problems in the absence of aliasing is polynomial. Thus, the computational complexity of a data flow problem does not necessarily depend on its flow sensitivity; rather, problems are flow sensitive because they are more difficult to solve precisely, or require more complicated data structures or algorithms for their solution.

Note that, for the problems Myers termed flow insensitive by [**IC**], constant intraprocedural annotations are calculated only once, and may then be used throughout interprocedural propagation on the call multigraph. This characterization fits our intuitive feeling that flow insensitive problems are easily solved.

While we can apply [**IC**] to interprocedural problems, it seems too strict. [**IC**] allows consideration only of the set of statements inside any procedure, without any consideration of statement order. MayBeModified$_F$ is [**IC**]-insensitive, as are the interprocedural versions of the [**CC**]-insensitive problems, but it is hard to think of other useful problems which are.

Dependent on Intraprocedural Control Flow There are other interprocedural problems which have been called flow sensitive in the literature, but can be viewed as flow insensitive under a new criterion, [**IPC**]. These problems can be solved precisely: first, calculate intraprocedural annotations by propagating data flow information along static paths in the flow graph and second, use these invariant annotations during an interprocedural propagation of information on the call multigraph. Thus, all intraprocedural static paths can be traversed in advance of the interprocedural propagation; thereafter, the annotation associated with each procedure (i.e., a node in the call multigraph) is invariant.[9] Flow insensitive problems according to the [**IPC**] criterion *are dependent* on intraprocedural control flow, but this dependence is limited to initial extraction of intraprocedural information, at worst once per call context; this can be seen as a memoizing technique.

It is possible to use representations other than the call multigraph for interprocedural structure; most of these preserve some intraprocedural flow information. For example, the Program Summary Graph (PSG) [Cal88] preserves the control dependence and sequence between entry, calls, and exit statements, while discarding all other intraprocedural structure. For a number of interesting problems it is still possible to extract intraprocedural information in a single pass into annotations on the edges of these structures [Cal88, JPP94, LRZ93, HS94, HRB90]. Here, dependence on intraprocedural information is not necessarily completely encapsulated in the annotations, but shared between the

[9]The annotations are not, however, required to be constant.

annotations and the control flow abstraction in the representation. However, the intuition of **[IPC]** — intraprocedural summarization followed by interprocedural solution — still holds. We can extend the definitions of constant and invariant annotations to annotations on the nodes and edges of these representations, and we view these problems as **[IPC]**-insensitive for the given representation.

Recent problems which are flow insensitive by **[IPC]** include two families of interprocedural problems, one for C-like languages with only a single level of pointer dereferencing and the other for Fortran-like programs without call-by-reference aliasing. The former include Pointer-induced MayAlias [LR92] and MayBeModified$_C$ [LRZ93]. The latter include Interprocedural Kill [Cal88], Interprocedural Reaching Definitions [HS94] and Interprocedural Copy Constant Propagation [RHS95, DGS95].

Comparison Consider the utility of these various flow sensitivity measures. **[IC]** is quite limiting in that the procedural annotations must be, in fact, *constant*, not dependent on intraprocedural flow, and calculated only once. Only MayBeModified$_F$ is an obvious example of this category. On the other hand, **[IPC]** allows invariant annotations to include information, calculated once, about the control flow within a procedure. These criteria and the problems they classify are compared in Figure 6.

While **[IC]** and **[IPC]** are orthogonal to our previous criteria, they appear to state an important distinction. For **[IPC]** in particular, the problems it identifies as flow sensitive will tend to be hard problems, difficult to specify, formalize and implement. Adding them to the data flow analysis phase of a compiler will tend to add complexity to the description and implementation. **[IPC]** is weaker than **[IC]**; in the absence of aliasing, MustBeModified is flow sensitive for **[IC]**, but flow insensitive for **[IPC]**.

There is an additional factor besides the dependence on representation. Often, the annotations derived in the intraprocedural phase are tuples of constant values, which are then used as parameters in the interprocedural flow functions. For Formal Bound Set, for example, the lattice is the set of subsets of pairs of variables in which the second component is a formal parameter. The annotation for each procedure is a set of (local/global, call_parameter) bindings A and a set of (entry_parameter, call_parameter) bindings B; the flow function takes the bindings passed in by successors X, and returns $A \cup B \circ X$, where $\circ$ represents composition.[10] In this case, each of A and B are elements of the solution lattice, but in Interprocedural Reaching Definitions, the constant information includes MustBePreserved information about each procedure. This information is most readily encoded as a set of variables not occurring in the procedure, which is not an element of the solution lattice (2^D, where D is the set of definitions in the program). Yet we can consider Interprocedural Reaching Definitions, at least in the absence of aliasing, as an **[IPC]**-insensitive problem [MR90].

Once we accept annotations which are not in the original solution lattice, however, it is difficult to know what forms to allow, although in practice the choice is usually clear. In principle, we could argue that any *invariant* value, even an invariant function, could be considered an invariant annotation. However, we could then in principle encode the

[10] If only formal-to-formal bindings are considered, and not all call chains, the sets A and B will be identical.

Figure 6: Interprocedural Sensitivity Criteria

	Independent of Intraprocedural Control Flow	Dependent of Intraprocedural Control Flow
No Annotations	Formal Bound Set Call-By-Reference MayAlias	–
Invariant Annotations	Fortran MOD **[IC]**	1-level Pointer-induced MayAlias; Fortran Reaching Defs, Use, Kill (all without aliasing) Copy Constant Propagation; **[IPC]**

entire abstract semantics of a procedure into a single invariant function (with arguments the values of its callees or callers, according as the analysis is backward or forward), where interprocedural flow functions are just invocations of *apply*. In this case, any data flow problem satisfies [**IPC**]; this clearly defeats the intent of the [**IPC**] criterion! Thus, use of [**IPC**] includes, at least implicitly, a specification of a function space from which annotations will be permitted.

5 Interprocedural Approximation Algorithms

To this point, we have only discussed the flow sensitivity of data flow problems that are amenable to precise solution. There are, however, complex problems for which only approximate solutions have been devised; these problems often are provably difficult (i.e., NP-hard or unsolvable), so that an efficient solution would seem to have to be an approximate one. We would like to compare some of these approximation techniques in terms of our flow sensitivity properties, such as whether they use intraprocedural annotations that are dependent on or independent of control flow, and whether they use invariant or changing annotations. These techniques have been termed *flow insensitive* or *flow sensitive* in the literature, but we see that in our framework, these terms cannot be applied to approximation algorithms; therefore, instead we will use the terms *alg-flow insensitive, (AFI)* and *alg-flow sensitive, (AFS)*.

This distinction for algorithms must be contrasted to the [**IC**] and [**IPC**] criteria of Section 4. If the *AFI/AFS* distinction were to be applied to the obvious precise algorithms for these problems, an [**IC**]-insensitive problem algorithm would be *AFI*, whereas an [**IPC**]-insensitive problem algorithm would be *AFS*, since [**IPC**] annotations and representations can take account of intraprocedural flow. On the other hand, a precise *AFI* algorithm need not correspond to an [**IC**]-insensitive problem, since the *AFI* algorithm is permitted to recompute procedure summaries, although it cannot consider intraprocedural flow when it does. Consider the use of a limited functional model of annotations for each call multigraph node. These functional annotations can be calculated from intraprocedural information, with (for *AFS*) or without (for *AFI*) considering intraprocedural flow of control. The latter results in procedure summary information independent of internal procedure structure, but not invariant across many inputs. This is to be contrasted to the constant summary information of the [**IC**] criterion.

Examples of *AFS* and *AFI* Examples of approximation methods we can categorize in this manner are the Pointer-induced MayAlias algorithms of [BCCH94, LR92], the Def-Use analysis of C-like programs [PLR94] and MayBeModified$_C$ [LRZ93].[11] The Burke *et.al.* algorithm [BCCH94] uses a functional annotation for each procedure computed without consideration of intraprocedural program structure. Between each interprocedural data flow propagation pass, there is an evaluation of each of these procedural annotations, to calculate possible new intraprocedural information to be associated with each node in the call multigraph. We say that this algorithm is *AFI* because the procedure annotations are independent of intraprocedural flow. Here, analysis consists of three phases: (1) intraprocedural annotations are collected as the call multigraph is constructed, (2) the interprocedural data flow problem is solved using them, (3) solution to the interprocedural problem is used within procedures, possibly yielding new summaries from the annotations. There is a circular dependence between interprocedural analysis and the analysis of individual procedures, since steps (2) and (3) can interact, because the summaries are variable throughout the algorithm (i.e., different summaries may result from different information input to a procedure annotation).[12]

The Pande *et. al.* algorithm [PLR94] also uses a functional annotation for each procedure, but this summary incorporates information obtained from propagation on intraprocedural static paths. Thus, we say that this algorithm is *AFS* in that the procedure annotations depend on intraprocedural flow. The approximation algorithms for Pointer-induced MayAlias and MayBeModified$_C$ in [LR92, LRZ93] and the optimized algorithm using Kill information presented in [BCCH94] can be similarly classified as *AFS*. All of these algorithms can use memoization (i.e., table lookup) to avoid unnecessary recalculation of their annotations on repeated inputs. These algorithms compute a single constant annotation for each procedure, with portions of the annotation function computed in a lazy fashion, as discussed for [**IPC**].

Levels of Sensitivity Although we have given a binary sensitivity classification for approximation algorithms, it is possible, at least informally, to contrast the level of sensitivity of algorithms for the same or closely related problems, according to the degree of intraprocedural flow information captured in the annotations or representations used by the algorithm. Recall that representations range from the call multigraph, which contains no intraprocedural flow information, through representations like the PSG [Cal88], which preserve some

[11] All of these algorithms handle multiple levels of indirection in pointer operations.

[12] Consider describing these algorithms by regular expressions over the numbered algorithm phases. Then the *AFI* algorithm discussed here has this structure: $1\ \{2\ 3\}^+$ whereas the flow insensitive problems by [**IPC**] are solvable by algorithms which have this structure: $1\ 2^+\ 3$. The latter can be seen to embody an idea similar to separate compilation for efficient analysis; it preserves the significant property that procedure annotations can be extracted by either a single visit preceding interprocedural propagation, or alternatively by an on-demand (non-repeating) calculation during interprocedural propagation.

flow information, to representations like the supergraph [Mye81], which in essence inlines procedure flow graphs, either by adding call and return edges, or by actually inlining dynamically, although necessarily eventually identifying recursive instances, and handling them specially. Annotations range from collection functions (as in *AFI* algorithms) through the sort of annotations used in typical **[IPC]** problems to arbitrary abstract semantic functions, which also capture all the intraprocedural flow.

We can call an approximation algorithm *pure-AFS* if it preserves all intraprocedural flow, either by using a supergraph-like representation, or arbitrary abstract functions as annotations. In a *pure-AFS* algorithm, any evaluation of the value of the tentative interprocedural solution for a procedure essentially revisits the procedure (often with special handling for recursion to avoid non-termination). Examples of this are the algorithms for Pointer-induced MayAlias relations in recursive structures [HN90] or in the presence of function pointers [EGH94].

6 Context Sensitivity

The term *context sensitivity* in the literature [EGH94] refers to the use of the calling context of a procedure (e.g., call stack, call history, path history) by an interprocedural analysis, which was first discussed in [Mye81, SP81]. This should not be confused with the ideas of flow sensitivity in Sections 4 and 5; Myers made the distinction between context and flow sensitivity in discussing the complexity of flow sensitive problems (see Section 4). Indeed, these are orthogonal properties, as we show below. Context sensitivity is also implied by *polyvariant* analysis in functional languages; polyvariance is a general term which describes an analysis using partial-evaluation-like constant propagation to eliminate infeasible intraprocedural branches and distinguishing between different call sites [JGS93].

Any algorithm is *context sensitive* or *insensitive* according as it uses call history information or not. This means that both precise and approximate algorithms can be categorized as context sensitive or insensitive.[13] A context sensitive algorithm need not use the full call stack abstraction. While both Myers and Sharir-Pnueli use the full call stack, the latter suggests using the tail of the stack as an abstraction, an adaptation used by Choi *et.al* for an algorithm for Pointer-induced MayAlias [CBC93, MLR$^+$93]. The Landi-Ryder algorithm for Pointer-induced MayAlias [LR92] uses a different approximation of context, *reaching alias* information at each call site.

The table in Figure 7 shows the orthogonality of these two properties by naming problems with algorithms that fall in each of the entries. The entry in (1,2) position is the *ProcLastCalled* problem. The solution to this problem yields that procedure which was last called with respect to each program point in the program. Thus, solutions will be either the procedure containing that program point or some procedure called while that procedure is on the call stack. The key observation is that at a return from a procedure P, the solution to ProcLastCalled can contain P itself or procedures called directly or transitively in P, and these calls could have been the last procedures called independent of the call site, given the

[13] By analogy, we can term a problem context insensitive if it has a context insensitive precise algorithm, but this doesn't appear to be useful.

usual assumptions of static analysis.[14] Thus, the solution of this problem is independent of context. (Note that, in ProcLastCalled, information can flow up out of calls, but not down into them. Clearly, any problem in which information can flow only upward, or only downward, will necessarily be context insensitive; conversely, two-way flow will almost certainly make a problem context sensitive.)

Further, consider approximate algorithms for pointer-induced aliasing. Landi-Ryder [LR92] is *AFS* and context sensitive, whereas Burke *et.al* [BCCH94] is *AFI* and context insensitive. Ruf presents two algorithms for this problem: one context sensitive and one context insensitive [Ruf95]; both are *AFS*. Another approach to this problem is given in [WL95]. This algorithm is *AFS* and context sensitive, but uses still another call stack abstraction, namely an aliasing pattern or template at a call site. Considering aliases produced by call-by-reference parameter passing, Myers' May Alias Partitions algorithm is context sensitive and *AFI*.

An algorithm for an interprocedural problem can be termed *call-site specific* if, for different calls to the same procedure, differing information at the call, (the return for backward problems), may result in differing interprocedural effects at the return, (call). Some measure of call-site specificity can be achieved even in a context insensitive algorithm. For example, jump functions in approximate Interprocedural Constant Propagation algorithms [CCKT86, GT93] are context insensitive (but *AFS*) intraprocedural summaries, but *pass-through* and *linear* jump functions, among others, will result in different constant information at return sites, as a result of interprocedural effects. Thus, these approximate Interprocedural Constant Propagation algorithms are *AFS*, but context insensitive.

A newer approach to interprocedural constant propagation is presented in [CH95]. There are two algorithms: the first is *AFI*, context insensitive, since it propagates information only downward through calls in the call multigraph, rather than also propagating back through returns. The second, also context insensitive, calls the AFI algorithm and then refines the AFI solution by recomputing in a single topological-order propagation through the call graph. This refinement phase considers intraprocedural flow within each procedure to get better information at call sites, and thence at procedure entry, but does not itself compute any procedure annotations. Rather it uses the AFI annotations for back edges only, to handle call cycles. The composite algorithm does not fit into our flow sensitivity model, and cannot be classified as either AFI or AFS.

Therefore, for interprocedural data flow algorithms, alg-flow sensitivity and context sensitivity are truly independent (although not exhaustive) dimensions. An [**IPC**]-insensitive problem may have algorithms which are either context sensitive or insensitive, but almost all [**IPC**]-sensitive algorithms will be solved by call-site specific, context sensitive algorithms.

7 Other Data Flow Problem Properties

There are other properties which complicate both definition and computation of data flow problems.

[14]We do change these rules if we allow pruning of infeasible paths through constant propagation, using perhaps, parameter values [WZ85].

Figure 7: Algorithmic Flow Sensitivity versus Context Sensitivity

	Alg-Flow Insensitive	Alg-Flow Sensitive
Context Insensitive	Formal Bound Set Call-by-Reference MayAlias Pointer-induced MayAlias [BCCH94]	ProcLastCalled
Context Sensitive	MayAlias Partitions [Mye81]	Pointer-induced Aliasing [LR92] Constant Propagation

Heterogeneous Flow Functions Callahan [Cal88] defines a pair of interprocedural data flow problems, Kill and Use, which have the properties (1) that they require examination of code within procedures beyond just initial annotations, and thus are flow sensitive in the sense of [**IPC**], and (2) that their flow functions are non-uniform, with the algebraic form of the functions, and even of the meet operator, depending on a classification of the nodes and edges of the flow graph. In particular, for Kill, evaluation at all nodes except return nodes uses a join operator, but return nodes require a meet.

In [Mar89], Marlowe suggested that any problem in which there is a classification of nodes or edges into several types, such that the form of flow function depends on the classification of its underlying edge, or of its two incident vertices, can reasonably be considered to be flow sensitive. We now view this edge classification as an orthogonal dimension to alg-flow sensitivity. Recent work discusses how to transform these *multisource* data flow problems into a *k-tuple* framework formulation [MMR95]. This model is especially useful for data flow analysis of explicitly parallel programs, where there may be control, synchronization, and call and return edges [MR93, GS93, CH92].

Bidirectionality Bidirectional data flow problems were introduced by Morel and Renvoise for Partial Redundancy [MR79, Dha91] (a generalization of Available Expressions); the approach has been extended to other problems [DP93]. In [KRS92], it is shown how to solve Partial Redundancy as a sequence of unidirectional problems. Since bidirectional problems typically involve non-trivial equations at both node entry and exit, and often different meets, we can view these as a subclass of the heterogeneous problems mentioned above. Discussions in [MMR95] also show how bidirectional problems can be transformed into k-tuple frameworks.

Convergence Properties and Flow Sensitivity Because of the tight restrictions it places on the equation sets of problems, flow insensitivity for [**AC**] implies 1-semiboundedness, and in fact distributivity and rapidity (for problems which are in any way bounded). Semiboundedness follows since any path can be reduced to a meet, and computation due to cycles terminates from idempotence of meet; distributivity, since in effect, all flow is straight-line flow. Similarly, [**CC**] may also constrain the complexity, and certainly the algebraic structure, of flow insensitive problems.

However, neither [**MAC**] nor [**MPC**] imply any boundedness properties. Likewise, neither [**IC**] nor [**IPC**] implies any boundedness properties, since MayBeModified$_F$ is flow insensitive, but is not a priori *k-bounded* for any fixed, instance-independent k, because Formal Bound Set is not so bounded.

8 Flow Sensitivity as a Continuum

We have shown that there are useful criteria for flow sensitivity that can be gleaned from existing definitions. One distinction that must be made is between the classification of a *problem* and the classification of approximate *solution procedures* for it. Flow sensitive problems often admit safe *AFI* approximate algorithms of varying precision.

The interprocedural criteria [**IC**] and [**IPC**] capture the possibility of summarizing procedure information in an annotation to avoid revisiting the internals of the procedure. This concept can be generalized to any hierarchical decomposition; the decomposition in interprocedural analysis is (program, procedure, block/statement); other decompositions, such as (program, module, procedure), or (procedure, block, statement), may be interesting in other situations.

Factors influencing the precision of an approximate solution are the precision of information used, particularly alias information, and the granularity of the representation. Whenever the flow graph or its data flow has a stratification or hierarchical decomposition, we can look at the data flow problem as a problem on the induced graph of clusters at some level of granularity.

Consider a Granularity Criterion [**GC**]: Let **D** be a data flow problem for a class of entities **P** (for example, programs). Let $\mathcal{A}$ be a representation for instances P of **P** at some level of granularity, and $\mathcal{F}$ be a space of functions. Data flow problem **D** for **P** is *flow sensitive at granularity* $\mathcal{A}$ *for function space* $\mathcal{F}$ if D cannot be solved precisely using flow functions from $\mathcal{F}$ as summary annotations on nodes and edges of granularity $\mathcal{A}$, but, for some instance P, computation of the solution requires examination and annotation of P at some level of granularity finer than $\mathcal{A}$.

The most natural clustering is that of program code into basic blocks, and most data flow problems are flow insensitive at that level; flow within basic blocks is uninteresting and can be summarized as an annotation on the block. But even at this level there are flow sensitive problems such as Intraprocedural Dynamic Structure MayAlias, or Pointer-induced MayAlias (with more than one level of indirection). In solving either of these with constant annotations, there are examples in which the code within basic blocks must be considered statement-by-statement, with effects depending on the incoming information. It is at best difficult, and at worst impossible, to compute the solution using only the basic-block flow graph; rather, for a given solution on entry to a block, the individual statements in the block have to be examined in order, to determine the solution at the exit of the block.

9 Conclusions

We have surveyed the existing interpretations of flow sensitivity derived from Banning's original uses of the term, and found problems with each of them. We have classified many existing data flow problems with respect to each of them. We then proposed two interprocedural flow insensitivity definitions (i.e., [**IC**] and [**IPC**]), each of which captures part of Banning's original intuition and that of subsequent usage. We showed how they express useful distinctions in the difficulty of problems (as distinguished from computational complexity – we have seen that flow sensitive problems can be tractable, although many of them will in fact be hard problems in the sense of complexity). We have discussed context sensitivity and distinguished it from flow sensitivity. We also explored some data flow problem properties, namely heterogeneity and bidirectionality, which add difficulty to obtaining a solution. Finally, we have proposed a new definition for flow sensitivity which attempts to generalize the interprocedural criteria, and better capture the association among the details of problem formulation, the locality of use of information, and the difficulty of precise problem solution.

Acknowledgements: We are grateful to Hemant Pande and Sean Zhang for their helpful comments on this manuscript, and to John Reppy for discussions on polyvariance.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Ban78] J. B. Banning. *A Method for Determining the Side Effects of Procedure Calls.* PhD thesis, Stanford Linear Accelerator Center, Stanford U., Stanford, CA 94305, August 1978.

[Ban79] J. B. Banning. An efficent way to find the side effects of procedure calls and the aliases of variables. *Conf. Rec. Sixth ACM Symp. on Principles of Programming Languages*, pages 29–41, January 1979.

[BCCH94] Michael Burke, Paul Carini, J-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250. Springer-Verlag, August 1994.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Trans. on Programming Languages and Systems*, 12(3):341–395, July 1990.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, 23(7):47–56, July 1988.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CCKT86] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 20–24, June 1986.

[CH92] J-H Chow and W. L. Harrison III. Compile-time analysis of parallel programs that share memory. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, January 1992.

[CH95] Paul Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 23–31, June 1995. SIGPLAN Notices, volume 30, number 6.

[CK84] K. Cooper and K. W. Kennedy. Efficient computation of flow insensitive interprocedural summary information. *Proc. SIGPLAN'84 Symp. on Compiler Construction*, pages 247–258, June 1984. SIGPLAN Notices, Vol 19, No 6.

[CK87] K. Cooper and K. Kennedy. Complexity of interprocedural side-effect analysis. Computer Science Department Technical Report TR87-61, Rice University, October 1987.

[CK89] K. D. Cooper and K. W. Kennedy. Fast interprocedural alias analysis. *Conf. Rec. Sixteenth ACM Symp. on Principles of Programming Languages*, pages 49–59, January 1989. Austin, Texas.

[DGS95] E. Duesterwald, R. Gupta, and M-L Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of the Twenty-second Annual ACM Symposium on Principles of Programming Languages*, January 1995.

[Dha91] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Trans. on Programming Languages and Systems*, 13(2):291–294, April 1991.

[DP93] D. M. Dhamdhere and H. Patil. An efficient algorithm for bidirectional data flow analysis using edge placement technique. *ACM Trans. on Programming Languages and Systems*, 15(2):312–336, 1993.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–257, June 1994. Published as SIGPLAN Notices, 29 (6).

[GS93] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of Conference on Principles and Practices of Parallel Programming*, pages 159–168, May 1993.

[GT93] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study in jump function implementation. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, June 1993.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transaction on Parallel and Distributed Systems*, 1990.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[HS94] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[JGS93] Neil Jones, Karsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[JPP94] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, June 1994. published as SIGPLAN Notices, Vol 29, Number 6.

[KRS92] J. Knoop, O. Ruething, and B. Steffen. Lazy code motion. In *Proc. SIGPLAN'92 Symp. on Programming Language Design and Implementation*, pages 224–234, June 1992. Published as SIGPLAN Notices Vol. 27, No. 7.

[LR91] W. L. Landi and B. G. Ryder. Pointer-induced aliasing: A problem taxonomy. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, January 1991.

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for pointer aliasing. In *Proc. SIGPLAN'92 Symp. on Programming Language Design and Implementation*, pages 235–248, June 1992. Published as SIGPLAN Notices Vol. 27, No. 7.

[LRZ93] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[Mar89] T. J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Dept. of Computer Science, Rutgers U., October 1989.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[MMR95] Stephen P. Masticola, Thomas J. Marlowe, and Barbara G. Ryder. Lattice frameworks for multi-source and bidirectional data flow analysis problems. Technical Report LCSR-TR-241, Department of Computer Science, Rutgers University, April 1995.

[MR79] E. Morel and C. Renvoise. Global optimization by supression of partial redundancies. *Comm. ACM*, 22(2):96–103, February 1979.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[MR91] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28(2):121–164, 1991.

[MR93] S. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of Conference on Principles and Practices of Parallel Programming*, pages 129–138, May 1993. published as ACM SIGPLAN Notices, May 1993.

[Mye81] E. W. Myers. A precise interprocedural data flow algorithm. *Conf. Rec. Eighth ACM Symp. on Principles of Programming Languages*, pages 219–230, January 1981.

[PLR94] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-second Annual ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[RMP88] B. G. Ryder, T. J. Marlowe, and M. C. Paull. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming*, 11:1–15, 1988.

[RP86] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *Computing Surveys*, 18(3):277–316, September 1986.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995. (to appear).

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[WL95] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. also available as SIGPLAN Notices, 30(6).

[WZ85] M. Wegman and F. Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, January 1985.