SUMMARY-BASED POINTER ANALYSIS FRAMEWORK FOR MODULAR BUG FINDING

Marcio O. Buss

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2007

ABSTRACT

SUMMARY-BASED POINTER ANALYSIS FRAMEWORK FOR MODULAR BUG FINDING

Marcio O. Buss

Modern society is irreversibly dependent on computers and, consequently, on software. However, as the complexity of programs increase, so does the number of defects within them. To alleviate the problem, automated techniques are constantly used to improve software quality. Static analysis is one such approach in which violations of correctness properties are searched and reported. Static analysis has many advantages, but it is necessarily conservative because it symbolically executes the program instead of using real inputs, and it considers all possible executions simultaneously. Being conservative often means issuing false alarms, or missing real program errors.

Pointer variables are a challenging aspect of many languages that can force static analyis tools to be *overly* conservative. It is often unclear what variables are affected by pointer-manipulating expressions, and aliasing between variables is one of the banes of program analysis. To alleviate that, a common solution is to allow the programmer to provide annotations such as declaring a variable as unaliased in a given scope, or providing special constructs such as the "never-null" pointer of Cyclone. However, programmers rarely keep these annotations up-to-date.

The solution is to provide some form of *pointer analysis*, which derives useful information about pointer variables in the program. An appropriate pointer analysis equips the static tool so that it is capable of reporting more errors without risking too many false alarms.

This dissertation proposes a methodology for pointer analysis that is specially tailored for "modular bug finding." It presents a new analysis space for pointer analysis, defined by finer-grain "dimensions of precision," which allows us to explore and evaluate a variety of different algorithms to achieve better trade-offs between analysis precision and efficiency. This framework is developed around a new abstraction for computing points-to sets, the Assign-Fetch Graph, that has many interesting features. Empirical evaluation shows promising results, as some unknown errors in well-known applications were discovered.

Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

I would like to thank my advisors, Stephen A. Edwards and Alfred V. Aho, for much help, guidance, and encouragement; Daniel Brand from IBM Research, without whom this research would also have not been possible; Vugranam Sreedhar from IBM Research, for stimulating discussions and guidance; John Darringer from IBM Research, for constant encouragement. Also, I thank Bin Yao, Daniel Waddington, and Rick Buskens for early research and career directions as well as the wonderful times during my stays at Bell Laboratories Research (and ever since).

I would like to acknowledge the comments and suggestions that I received from the readers of this manuscript, Stephen A. Edwards, Alfred V. Aho, Mihalis Yannakakis, Daniel Brand, and Vugranam Sreedhar, which helped improve its contents.

I thank all friends which brought more joy to my stay in the United States. Trying to cite them by name would certainly lead to unfairness or else a ten-page list. My special thanks goes to Cristiano Pereira (in the west coast) and Diego Nehab (in the east coast) for their unrestricted friendship and solidarity, and Leandro Carvalho for the extra office space at Firestone Library. I am grateful for the unconditional support of my family, both remotely and through their visits in the United States; and last but not least, Juliana, for reasons that would not fit into this page.

This research was supported in part by the Brazilian Government Research Council (CNPq) through a student fellowship (grant number 200346/1-6), and by IBM T.J. Watson Research Center through a collaborative research effort.

To Luisa.

Chapter 1

Introduction

Modern society is irreversibly dependent on computers and, consequently, on software. Examples abound: from office work to banking, from leisure to air traffic control, communications, power grids, factories, cars, air planes, etc. Virtually everything in our daily lives is affected by software.

Yet software products today are plagued by defects. The increasing complexity of programs makes it almost impossible to deploy an error-free product. E.g., the Linux kernel grew from 2 million lines of code in 2002 to about 6 million lines in 2007 [58]; the number of lines of code in a typical GM vehicle increased from 100 thousand in 1970 to 1 million in 1990—it is estimated to grow to 100 million lines by 2010 [33]. As a striking example, the Windows 2k operating system was shipped with 63,000 defects (discovered to date) [69]. Furthermore, it has been argued from empirical evidence [53] that even bug *density* (i.e., number of defects per lines of code) is increasing with project size.

Compared to other human endeavors, the software industry is likely creating the most defective products, and it spreads its defects wherever it is used (e.g., cars, mars lander, etc). This costs money to software companies as well as to society itself. According to Djenana Campara, *Klocwork*'s CTO, a bug that costs US$ 1 to fix on the programmer's desktop will cost US$ 100 once it is incorporated into a complete program, and US$ 1000s or more if it is identified only after the software has been deployed in the field [28]. The National Institute of Standards (NIST) observes "software errors cost U.S economy $59.5 Billion annually" [56].

While software manufacturers work constantly to improve software quality, they still mostly rely on "manual" methods such as the *run, analyze results, fix, repeat* loop. Late bug detection continues to be a major problem. As mentioned, bugs found early result in significant cost savings.

Static program analysis, a solution for *automated* error detection, has been gaining renewed momentum. In the most basic sense, it means finding defects without running the code. More specifically, it means analyzing the source code and searching for violations of correctness properties. This approach is complementary, and sometimes more attractive, to more traditional methods such as *testing* [54] (dynamic analysis) or *model checking* [19, 36]. With static analysis, errors are detected early in the software development process (one can check for defects immediately after the code has been written); there is no need to generate test cases or specifications, as well as no need to run the program. Static analysis can consider all paths through the program; testing, or traditional model checking, execute only those paths that are triggered by the environment model. Static analysis tools do not require the application to be compiled or executed; bugs are found by directly analyzing the source code. Model checking or testing a system requires a crafted environment model or the generation of useful test cases.

However, static analysis is necessarily conservative because it symbolically executes the program instead of using real inputs. The downside is that static tools will inevitably signal errors in correct programs, i.e., issue "false alarms." By contrast, dynamic analysis examines particular runs of a program at a finer granularity, and can be much less conservative because they have precise information about the current execution state. For a static bug finder, being too conservative often means missing real program errors, issuing too many false alarms, or being uncertain about a program defect and choosing to remain silent to avoid them.

The presence of pointers is a challenging aspect of many languages that can force static analysis to be overly conservative. In C, pointers are often regarded as the bane of program analysis. They pose a problem to compilers and bug finding tools because it is often unclear what locations may be accessed through indirect memory references. Aliasing, i.e., two expressions referring to the same memory location, is another aspect common from using pointers that can mitigate the analysis. Without enough information, static tools are forced to make conservative assumptions such as "a pointer assignment could write to any variable in the current scope," leading to many superfluous dependencies and significantly limiting the power of the analyzer.

To alleviate that, a common solution is to allow the programmer to provide annotations to guide the analyzer, e.g., to declare a variable as unaliased in a certain scope, allowing the analyzer to aggressively infer properties in that scope without fearing any aliases. Other tools simply restrict pointer use to a minimal, systematic way [6], provide special constructs such as the "never-null" pointer of Cyclone [45], or create special data types that can never be aliased [25].

However, experience has shown [34] that programmers are reluctant to provide any but the most minimal annotations, and when they do, the annotations are rarely synchronized with code changes. This burden is particularly pronounced when applying a tool to a legacy code base.

The solution is to provide some external checking of properties. For pointers, determining useful information requires some form of *pointer analysis*. The precision of such analysis can directly affect the utility of the tool—the more precise the analysis, the more aggressive the tool can be, leading to fewer false alarms and/or missed errors. However, some solutions are precise but prohibitively expensive, while others are fairly cheap but too approximate. A bug finder that relied on the latter would be too inaccurate, while using the former could result in a checker that is unacceptably slow. The gap between these two extremes is sparsely populated by a few disconnected, ad-hoc heuristics. This has inhibited designers from implementing effective pointer analyses into real-world error detection tools, even though researchers agree [32, 72] this could lead to considerable benefits. As an example, Table 1.1 (extracted from David Wagner's PhD thesis [71]) shows the expected reduction in false alarms from several potential improvements to his buffer overrun checker. Although the data corresponds to specific reports from analyzing sendmail 8.9.3, which may not be a true representative, it illustrates benefits expected from pointer analysis.

Table 1.1: Expected reduction in false alarms from potential improvements to David Wagner's [71] buffer overrun checker. The data corresponds to specific reports from analyzing sendmail 8.9.3.

Improved analysis	False alarms that could be eliminated
flow-sensitive	48%
flow-sensitive **with pointer analysis**	63%
flow- and context-sensitive, with linear invariants	70%
flow- and context-sensitive, **with pointer analysis** and linear invariants	95%

To that end, this dissertation aims to bridge the gap mentioned above by proposing a novel analysis space for pointer analysis in which those two extreme solutions are simply special cases of a more fundamental, underlying principle. This is achieved by re-formulating the granularity and the dimensions of pointer analysis itself, allowing finer-grain trade-offs between speed and utility, as well as finding new, previously unknown, sweet spots. The integration of such framework into an industrial bug finder, IBM's *BEAM* tool [21], is also a practical contribution of this work.

Another fundamental goal of this dissertation is to provide pointer and related analysis for error detection that is based on modular analysis instead of more traditional whole program analysis. Existing techniques do not meet the requirements of such *modular bug finding*. In this dissertation we address such requirements for the first time through a novel approach to procedure summaries (a graph summarizing the function's pointer behavior).

The main contributions of this dissertation are listed in Section 1.4. The next subsections lay out some definitions and major motivations for this work.

1.1 A note on program errors

In the remainder of this document we will refer to program defects (alternatively, errors or bugs), and it is therefore useful to clarify their meaning. We are given a program and a notion of correctness. For static analysis, this informal notion of correctness represents properties that, if violated, would cause program misbehavior. These properties are modeled as rules, such as "the program should not dereference a null-pointer," "the denominator of a division operation should not evaluate to zero," "all program variables should be properly initialized before used," "the program should not allow SQL injection," "there are some methods whose return value should not be ignored, such as String.toLowerCase(), since it returns a new string object," etc. Static analysis, therefore, does not know what the program is "supposed" to do; instead, it tries to verify that a set of rules is respected.

A generic definition for a program defect is as follows. The program has inputs and outputs. We interpret inputs as broadly as possible; they include not only numeric inputs, but also initial state of memory, version of the operating system, version of the compiler, architecture of the machine, etc. Similarly, outputs include not just numeric outputs, but also performance, usability, security, etc. Then the program has a defect if there exist inputs that cause the program to produce outputs inconsistent with the notion of correctness (i.e., violate the rules).

Extending upon these notions, we can further classify program bugs into three main categories: (1) correctness: the code seems to be doing something the developer did not intend (e.g., crash); (2) bad practice: the code violates good programming practices, such as an exception that is caught but ignored; (3) security: there are vulnerabilities to malicious code, vulnerabilities to malicious inputs such as SQL injection, etc.

Some errors within these categories may require reasonably simple techniques, such as a naïve coding guideline checker or a parser. Others necessitate complex machinery such as those provided by static analysis and other techniques. Section 1.2 discusses some different types of these later semantic bugs.

1.2 A few simple bugs

Besides forcing a static analysis tool to be conservative, pointers themselves open the possibility of a variety of programming errors. From this point of view, pointers are doubly bad—they interfere with assumptions about well-behaved variables, and they can trigger errors caused by mishandled pointers themselves.

This section and its subsections show some program errors that are due to mishandled pointers and/or are bugs that could be reported by a static tool if it was able to be less conservative.

For example, one of the easiest ways to end up with a (pointer) bug is to try to dereference the value of a uninitialized pointer. Consider the following fragment of a C program:

```
void foo()
{
  int *p, z;
  if (...) {
    p = &z;
   *p = bar();
  } else {
   *p = 0;
  }
}
```

Assuming `p` is only assigned in line 5, the statement `*p=0` at line 8 is an error—if the condition at line 4 is false, `p` will be dereferenced before it has been initialized. To catch this, a bug finder needs to statically compute the set of values a pointer can assume at runtime. Moreover, such analysis must model some degree of statement ordering, and/or distinguish different control conditions under which the statements occur. For example, in the above code one can determine that if the condition at line 4 is false, `p` will be uninitialized at line 8. A so-called flow-*insensitive* pointer analysis, which ignores the order of statements, is useless here because it ignores statement execution order. Using costly flow-*sensitive* analysis may prove too expensive in general, and thus the above bug would remain latent in the code until the `else` branch is executed.

A program that neglects to deallocate a unused block is said to have a memory leak, another type of error that can be hard to find and is often (if not always) associated with pointers. Consider the following code fragment:

```
typedef struct T {
  int *element;
  struct T *next; }
} T;
int main()
{
  T *y = NULL;
  if (...) {
    int *x = malloc(sizeof(int));
    y = f(x);
  }
  ...
}

T *f(int *p)
{
  if (...) return NULL;
  T *y = malloc(sizeof(T));
  y->element = p;
  y->next = NULL;
  return y;
}
```

This code has a memory leak—if the condition at line 17 is true, we return from function `f` without having assigned the value of parameter `p` to the location reached by `y->element` (line 19). Thus after line 10 the program loses its ability to access the memory block allocated at line 9 (note the scope of `x`), and the runtime library is not informed that the block should become available.

Similarly, pointers can be (mis)used after the object or variable they point to no longer exists or has gone out of scope. If such a pointer is dereferenced, the program may continue to run, but the results will likely be incorrect and unpredictable. Such a failure is harder to track down, as silent corruption of unrelated data may occur, leading to very subtle bugs.

Another common programming mistake arises from a function or module returning the address of a local variable (a variable that is valid only inside the module's scope), such as the code below. Since these variables are automatically deallocated from the stack after the function returns, any pointer that refers to those locations will be *dangling* once the function returns.

```
char *initialize()
{
  char string[80];
  char *p = string;
  return p;
}

void main()
{
  char *q = initialize();  /* dangling pointer */
}
```

The *aliasing* bug is another program error that may arise when two or more pointers refer to the same location. The memory may be set (or freed, in case of dynamic allocation) through one pointer and later referenced through another, which may result in subtle bugs when this behavior was not intended.

Many security holes stem from pointer errors. For example, if a pointer is used to make a virtual function call, a different address (pointing at malicious code) may be called if the virtual table was overwritten.

Obviously, static analysis tools cannot find all bugs in code, nor can they ensure there will be no bugs at ship time. No matter how precise a (pointer) analysis is, some errors are simply not amenable to static analysis. Even more importantly, static analysis is not meant to go after all flavors of bugs—the utility of a static analysis tool is tied to the number of correctness properties that have been encoded as rules such as those mentioned in Section 1.1. Moreover, properties that require reasoning about the system execution are not amenable to static checking (e.g., many protocol-specific properties such as routing loops). These cases are better addressed by model checking techniques, which excel at exploring intricate behaviors of the system and finding errors in corner cases that have not been considered by system designers. Ie., the idea is that the above bugs are always bad, whereas others may be bad only *w.r.t* a given specification. Also, some bugs are runtime-dependent and can only be found through real testing and sophisticated dynamic analysis.

1.2.1 Different goals for a bug finding tool and the roles of pointer analysis

This subsection shows two simple examples that illustrate how a tool's objective can dictate the role and importance of pointer analysis in static error detection. We discuss a nonexistent error that is nevertheless reported by FlexeLint [57], as well as an instance of a real error that can be missed due to conservatism. Consider the code fragment below.

```
 1 int main()
 2 {
 3   int *p, x;
 4
 5   p = &x;
 6   x = 0;
 7  *p = 10;
 8
 9   return 100/x;  /* no error */
10 }
```

Gimpel's FlexeLint [57] reports a division by zero in the above code even though no such error exists[1]. The indirect assignment via `*p=10` guarantees that `x` is overwritten before the `return` statement executes. Nevertheless, FlexeLint assumes `x` can still be `0` at line 10 and reports the (false) error.

[1]Easily checked by editing any of Gimpel's interactive demos at http://www.gimpel-online.com/OnlineTesting.html.

FlexeLint's goal is to report as many errors as possible, even at the expense of some spurious error reports. In contrast, a completely opposite take is to drastically reduce the number of such false alarms, even at the expense of missing some real bugs. In this case, the tool has to obtain a "proof" that a potential error can *actually occur*. In the absence of such proof, the tool will remain silent even if that causes defective code to be endorsed. The importance of reducing static analysis conservatism in this latter case is even greater. Consider the following similar program:

```
 1 int x, y, *p;
 2
 3 void bar()
 4 {
 5   x = 0;
 6 }
 7
 8 int main()
 9 {
10   p = &y;
11   ...
12   bar();
13  *p = 10;
14
15   return y/x;  /* error */
16 }
```

This code has a division by zero at line 15. However, a bug-finding tool striving to reduce false alarms needs to "prove" that statement `*p=10` does *not* write to variable `x`, i.e., that `*p` is not an *alias for* `x`; if it were, there would be no error. Stepping back and taking a conservative view, such a statement could potentially modify `x`, especially if one has no side-effect information for function `bar`. Therefore, in the absence of more precise (pointer) information about statement `*p=10`, the tool has to refrain from reporting the fault. Hence, providing a precise pointer analysis as a way of reducing conservative assumptions and provide more information is of paramount importance. This is one of the main views taken in this dissertation, which means pointer analysis is helpful by allowing the tool to report more real errors.

1.2.2 Aggregate type fields

Struct fields are another source of approximations in static analysis—writing to one field of a data structure should not interfere with reading from a different field. This feature is rarely considered in bug-finding tools (to the best of our knowledge, none of them actually do it). This dissertation is thus one of first attempts to implement field-sensitive pointer analysis in a real-world bug finding tool. Details and examples are given in Section 5.7.

1.3 Modular bug-finding and the Evidence-Based approach

This section defines and motivates the need for *modular bug finding*, and discusses its requirements on pointer and related analysis. Researchers have been devising modularization techniques for various aspects of software analysis, but modularization for pointer analysis is harder. While the focus of this dissertation is static analysis, modular analysis and its requirements apply more generally. To emphasize this, in this section we will consider both static and dynamic analysis as appropriate.

A side goal in this dissertation is to develop analyses that aim at reducing the number of false alarms, i.e., a message issued by a tool that the user chooses not to translate into a code change. This goal, along with modular bug finding, imply basic principles that will shape the design of our algorithms and representations for pointer analysis.

1.3.1 Reducing false alarms as a prime goal and the need for modular analysis

Both static and dynamic analysis suffers from false positives (i.e., incorrectly reporting a defect) and false negatives (i.e., failing to report a real error). Dynamic analysis can give a false negative if it fails to consider a particular input that triggers the error. It can give a false positive if it considers an input vector that is not valid.

Static analysis suffers from false positives and false negatives because of its inaccuracy. For example, a tool may find that a variable is initialized only under condition A, and then used under condition B. But the tool may not be able to determine whether B implies A, which would ensure the variable was initialized correctly. Faced with this uncertainty, the tool may choose to report the variable as uninitialized and risk a false positive, or it may chose to remain silent and risk a false negative.

For a tool to be useful, it must report a sufficiently high number of true positives and a sufficiently low number of false positives. However, achieving both goals is computationally expensive for both static and dynamic methods. The cost is directly proportional to the length of the paths from program entry to a fault. The longer the path, the fewer the input combinations that will exercise it during testing, since all conditions along the path would have to be simultaneously satisfied. Similarly, the longer the paths, the more expensive it is for static analysis to consider all of them accurately. Therefore attempts to reduce the number of false positives should concentrate on reducing the path length from program entry to failure site.

One approach to such goal is to replace the system test (in the dynamic analysis world) or whole program analysis (in the static) paradigms with a modular approach. Modular analysis ranges from unit testing [54] to testing (or analyzing) major subsystems. It has been reported [46] that finding a bug during such modular testing is one or two orders of magnitude cheaper than during system test. This drives the software industry towards modular analysis. However, a major impediment to modular analysis is the lack of information about which inputs of analyzed component are considered legal, as opposed to whole program analysis which can analyze the actual set of inputs into the function [75]. Reporting a bug that happens only with illegal inputs is a false positive and thus highly undesirable.

There are several solutions to the problem of determining legal inputs. One general approach is to require programmers to provide specifications, preconditions, contracts, and other kinds of annotations [54]. Some tools allow developers to mark their code with special comments or some other form of metadata to describe rules and inter-function dependencies. This additional information allows the analyzer to understand the conditions under which a bug may occur as well as expectations each function has for parameters passed in and values returned. If such information is available, then the same tools used in whole program analysis could be used for modular analysis. In practice, however, such requirements are hardly met—annotations are rarely complete, rarely kept up-to-date with code changes, and practically never machine-readable [47].

The implication for modular bug finding is that some educated guesses must instead be made about what constitutes legal or intended inputs to a function. A general approach is to try to automatically infer information from the source code of the analyzed component [7, 27, 31]. Our framework relies on a method that is guided by evidence [7] from the source code, which affects the design of pointer analysis. E.g., the presence of a test for *null* tells us the variable can assume *null* sometimes; the declaration of two separate pointer variables is evidence that they sometimes do not assume the same value; the test of a Boolean variable inside a function hints that both truth values are legal at that point. Intuitively, the idea is to infer from the source code which inputs are legal by focusing on what the programmer's intended preconditions for a procedure would be if they were written down explicitly. An error is reported only if the tool can find *evidence* that the fault can be reached during an execution that starts with legal inputs.

As an example, consider the code in Figure 1.1(a). Suppose there is no documentation for `foo`, and no information about the potential callers of `foo`. Should the statement `serial->id` be reported as an error because `serial` may be null at function entry? In other words, if `foo` fails, is it because null is a legal input, and `serial` should be checked for null before dereferencing? Or is the fault with the caller, because null should never be passed in as argument? There is no universally correct answer. It may differ from project to project depending on their reliability requirements and coding practices, or even the taste of the individual programmer. However, reporting these cases as errors would probably inundate the user with false positives.

```
int foo(struct T *serial)
{
  serial->id = count++;
  ...
}
```

(a)

```
int foo(struct T *serial)
{
  if (serial != NULL) {
    serial->hdw = DEV0;
  }
  ...
  serial->id = count++;
}
```

(b)

Figure 1.1: Should the tool warn about possibly dereferencing null on `serial->id`?

Now consider the example in Figure 1.1(b), a slight modification of Figure 1.1(a). The *if* statement suggests the programmer expected `serial` to be null, while the statement `serial->id = count++` fails if `serial` is null. In this case, the test is evidence that `serial` can be null, and most programmers would agree the potential error should be reported. In the next subsections we present additional examples where such source code evidence guides the choice of legal inputs for modular analysis, and how this affects the design of our pointer analysis for the needs of modular bug finding. More technically, the expression "modular bug finding" means that, given a function f, the decision of whether a potential error inside f can be reported while analyzing f's body is to be based on information derived from the function itself plus its callees *only*. I.e., the decision to report the bug must be independent of any existing callers, no matter what data is flowing in from the caller(s). The rationale for this choice is because the correctness of a function does depend on its specific lower level libraries, but it should not depend on any existing higher software layer—these may change when the module is reused.

1.3.2 Evidence and modular analysis for pointer analysis

Since the main focus of this thesis is on pointer analysis and its applications to modular bug detection, we will illustrate the requirements for modular analysis by examples of bugs whose existence depends on pointer aliasing (a question normally answered by pointer analysis). Specifically, the examples aim to justify why, in modular bug finding, only pointer information derived from *callees* is useful for reporting errors without risking false positives. Indeed, if information derived from callers were to be used in a modular analysis[2], that information would actually "pollute" the analysis results. For the sake of presentation, we will assume testing the code (i.e., dynamic analysis).

```
int *p, *q;  bool b;
...
int error_if_aliasing()
{
  callee();
  if (b) *q = 0;
  return 42 / *p;
}
```

Figure 1.2: A divide-by-zero error if `p==q` and `b` is true. Information derived from `callee()` can be used to report the error in case pointer analysis determines that `callee()` makes `p` and `q` to be aliased.

[2]An awkward combination, since by definition modular analysis only considers a function and its callees.

Consider Figure 1.2. Suppose a tester exercises the function `error_if_aliasing()` and gets a divide-by-zero because `p==q` at the return statement. The programmer may argue that the procedure is to be invoked only in environments where $\texttt{p} \neq \texttt{q}$, and therefore the tool has reported a false error. That argument would be contradicted if pointer analysis determined that `callee()` unconditionally makes `p==q`. The programmer could still insist that `error_if_aliasing()` is to be called only when `b` is false, which means statement `*q=0` does not execute, preserving the original value of `*p`. But that argument would be defeated because such an assumption would make the test of `b` unnecessary. Anyone looking at the body of `error_if_aliasing()` would agree that *both* truth values of `b` are *legal inputs* into the function—the upshot is, if pointer analysis determines that `callee()` unconditionally makes `p==q`, then `error_if_aliasing()` is definitely defective. The two pieces of information extracted from the code and used by the analysis are that (1) $\neg\texttt{b}$ is a legal input, due to its testing by the if-statement, and (2) `p==q` has been derived from the lower layers of software. Since both pieces of information are available at the fault site, the error can be reported without risking a false positive.

Now suppose that `callee()` has no effect on the values of `p` and `q`. This time the programmer can successfully argue that `error_if_aliasing()` is to be called only where $\texttt{p} \neq \texttt{q}$ or $\neg\texttt{b}$. No pointer aliasing information obtained from callers can be used as a counterargument. I.e., even if `p==q` whenever `error_if_aliasing()` is called in an existing program, there may be no error in the whole program as long as `b` is false whenever `error_if_aliasing()` is called—reporting it would be risking a false positive. While that would make the test of `b` unnecessary from the point of view of the *existing* whole program, that test may be needed in an unforeseen future use. In other words, the programmer is relying on an (unwritten) precondition and any division by zero would be the fault of a caller. This example illustrates that even if we have *must* information about pointer aliasing, that information can be used to issue an error only if the information comes from a *callee*.

Roughly speaking, in our framework a function is analyzed by neglecting what its callers pass in as arguments. If there is an error inside the function that is only triggered by a specific calling context (not the "intended" context) then the analysis of the function may not find the error. When the summarized information for the function is used at a given call site, the calling context is taken into account and that may trigger the error. We will then consider the fault to be with the caller at hand, which passes in illegal inputs.

```
int error_if_no_aliasing()
{
 *p = 0;
  callee();
  if (b) *q = 1;
  return 42 / *p;
}
```

Figure 1.3: Error if p and q do not alias. *May* information from the callee is used to aid the analysis.

Consider an example involving *may* and *must not* information, depicted in Figure 1.3. Assume that a tester found `error_if_no_aliasing()` failing because $p \neq q$. The programmer may argue that `error_if_no_aliasing()` is to be called only in environments where `callee()` causes `p==q`. That argument can be defeated if `callee()` never makes `p==q` (i.e., `p` must not equal `q`), and hence the error could be reported. But if pointer analysis finds out that `callee()` may possibly cause `p==q`, then `error_if_no_aliasing()` may actually be fault free (that is, to reduce the number of false positives, *may* aliasing information from a callee determines whether or not a potential error should be reported).

We see that in all these cases pointer information derived from callees can be used to decide whether a possible error should be reported or not. But information derived from callers is irrelevant in that decision. Indeed, if the information from callers were allowed in the analysis, it would actually "pollute" the useful information derived from callees, and then none of them could be used.

For these and other reasons, our overall approach to pointer-related analysis (points-to, *mod*[3], side effects) is to propagate information from callees to callers in the form of procedure summaries that are designed to satisfy the requirements of modular bug finding listed in Section 1.3.3. Summaries of analyzed functions are maintained for possible reuse, so that any function call can be replaced by a summary for that function.

The general implications of evidence-based, modular analysis on pointer analysis is that, to report a bug inside a function f, one should *not* look at f's callers, and one should assume that none of f's parameters alias each other (in general, this is true not only for parameters but for all locations from the environment). I.e., during the analysis of f there is no need to consider the situation where

[3]*Mod* analysis determines whether a given variable is modified by a given procedure.

its parameters alias—indeed, this is not desired.

The reason for the former was already mentioned: the correctness of a software layer does not depend on specific callers. The reason for the latter is that the declaration of two separate parameters is evidence that the programmer expected them not to alias in general. And that assumption cannot be overridden by any pointer information derived from specific callers. For more complicated pointer expressions, other forms of evidence may exist and it continues to be true that information from specific callers cannot override it. The situation is different than information derived from lower layers. For example, the C-library function `strcpy` is such that after `x=strcpy(y,z)` executes, the variables `x` and `y` alias. And the fact that the programmer declared two separate variables cannot override that behavior.

However, if the arguments are aliased when using the summary for our function f in a call site, and if that triggers a program error, then the caller should be reported as faulty because it passes invalid inputs into f. For this reason, when instantiating a summary into a call site (and only then), the analysis needs to restore its soundness within the *instantiated summary* by considering aliases introduced by the caller.

Contrary to existing techniques, our summarization method satisfies all of these requirements. Intuitively, our procedure summaries can be viewed as generated preconditions. If there is not enough evidence inside a procedure to report a potential problem, its summary embodies assumed intended inputs (e.g., non-aliased parameters, or such-and-such parameter is unconditionally dereferenced). Callers are then checked for passing illegal inputs, such as non-null values for parameters that are unconditionally dereferenced. And if there is not enough evidence inside the caller itself to prove that the inputs are illegal, the caller gets another summary embodying *its* legal inputs.

1.3.3 A novel summarization method for pointer and related analyses

This dissertation presents a novel method for procedure summaries as a solution to a series of issues that are crucial for modular bug finding. None of the classical techniques are suitable for our purposes, as outlined below.

- In modular bug finding we do not have access to the "upper layers" of the module under analysis, thus we cannot use Wilson's technique [75] which is based on a whole-program paradigm.

- We want to compute a single summary for each function, hence we cannot use Chatterjee's [14], Landi's [48] or Wilson's [75] techniques.
- The summarized information for a procedure must work for any unforeseen calling context, and therefore we cannot use Chatterjee's [14] or Wilson's [75] approaches.
- The information from callers cannot, in any circumstance, shape the summary of a callee.
- The true transfer function of a callee's summarized information, once instantiated at a call site, depends on the context defined by the caller.

Combining the above items becomes a challenge for pointer analysis. On one hand pointer relations inside a procedure do depend on aliasing caused by callers. On the other hand, such aliasing is unknown in the type of summary-based analysis we do. These constraints, as well as all of the above, are simultaneously embraced in our summarization method, as will be discussed in later chapters. Section 2.1.2.2 discusses in more detail why existing summarization techniques fall short of providing the required capabilities for our purposes. Furthermore, our summary graphs combine pointer and *mod* information as a single data structure, which has a series of benefits.

1.4 Main contributions

The main contributions of this dissertation are

- a methodology for procedure summaries satisfying the requirements of modular bug finding as listed in Section 1.3.3;
- a novel analysis space for pointer analysis, defined by new finer-grain dimensions of precision, which allows us to explore and evaluate a variety of analyses towards new trade-offs between precision and efficiency; and
- a new abstraction for computing points-to sets, the Assign-Fetch Graph, that has many of the attributes of an ideal representation for pointer analysis.

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides background on different elements of, and alternatives, in the design of pointer analysis systems. Chapter 3 presents our novel analysis space for pointer analysis where its dimensions of precision are re-defined around finer-grain elements such as order-sensitivity and condition-sensitivity. Chapter 4 investigates some particular pointer analysis variations enabled by our analysis space. Chapter 5 discusses our main pointer analysis abstraction, the *Assign-Fetch Graph*, in more detail. Chapter 6 presents an empirical evaluation of our framework, and Chapter 7 has some final remarks.

Chapter 2

A Review of Pointer Analysis

This chapter deviates from the discussions in Chapter 1 in order to review pointer analysis in more generality; it may be skimmed over by the experienced reader.

In its most generic form, the pointer analysis field contains all the analyses that try to statically determine useful properties about pointers in a program. Because most of these properties are undecidable, the best a pointer analysis can do is to produce a sound approximation of the result. There are at least four types of analyses that fall into this category: *points-to analysis*, *alias analysis*, *escape analysis* and *shape analysis*. A points-to analysis [29] determines what storage locations a pointer can point to. Such analysis consists of computing *points-to sets*—given two program locations, p and q, we say p points-to q if p can contain the address of q. An alias analysis [48] calculates the pairs of pointer expressions which are aliased, i.e., refer to the same storage location. For example, if x and y both point to the same location(s), we say $*x$ and $*y$ are aliases, written as $\langle *x, *y \rangle$. The specific location(s) they refer to are not usually tracked down. An escape analysis [5, 16, 18] detects the memory locations that escape a given scope, and a shape analysis [63, 74] tries to determine the shape of the data structures manipulated by the program. Such an analysis checks properties such as "if this method receives a tree, it returns a tree," usually without representing the tree itself. In this thesis, we consider points-to and alias analysis, and therefore the term pointer analysis will be used interchangeably between these variants. The next sections discuss classical definitions for pointer analysis and the traditional algorithms.

2.1 Dimensions of Precision

Computing points-to sets is traditionally centered around some characteristics or dimensions that directly affect analysis precision. The two main dimensions are flow- and context-sensitivity: flow-sensitivity determines whether the order of program statements is taken into account, whereas context-sensitivity deals with the degree of differentiation between multiple calls to the same function. Other characteristics include field-sensitivity, which deals with how distinct fields of a struct or class are handled, and (more common in other program analyses) path-sensitivity, which reasons about branch correlations. Describing pointer analysis is intimately related to such dimensions.

2.1.1 Flow-sensitivity

A *flow-sensitive* analysis [8, 13, 17, 29, 48] considers the order in which the statements of a program execute, and it is usually based on some form of iterative dataflow framework [1] to produce results at the statement level. Flow-sensitive analyses model *strong updates* (or kills), in which a later statement destroys the dataflow facts created by an earlier statement [1, 12, 29].

Flow-insensitive analyses [2, 4, 9, 20, 23, 67] view the program as a pile of unordered statements, and by their nature are similar to type systems; therefore they are generally formulated as sets of typing rules. Flow-insensitive analyses may compute a single solution that is valid for the entire program [2, 44, 67, 79] or one solution for each method [9, 42].

A flow-insensitive algorithm is usually more efficient than its flow-sensitive counterpart, and is primarily used for problems for which the latter does not provide substantially increased precision. Alternatively, a flow-insensitive analysis can be used to improve efficiency at the potential cost of precision for the class of problems for which flow-sensitive analysis would indeed yield better accuracy. Pointer analysis is a problem of this class. In pointer analysis, flow-sensitive algorithms are in general more precise, but more costly than flow-insensitive algorithms[1].

The notion of flow-insensitive/sensitive analysis is intimately related to the notion of program-point specific versus summary analysis. An analysis is program-point specific if it computes points-to information for each program point. An analysis that maintains a summary for each variable, valid for all program points of the function (or the program) is a summary analysis.

[1]Investigations of the merits and drawbacks of both approaches can be found in the literature [41, 68, 78]

A program-point specific analysis is likely to be more accurate than a summary analysis, but the information provided by the latter is *safe* (e.g., the points-to sets computed by the flow-insensitive analysis are always supersets of the sets computed by the flow-sensitive analysis). The next example illustrates both types of analyses in a code fragment.

Figure 2.1 shows a fragment of the control-flow graph of a program combined with its flow-sensitive pointer analysis solution, represented as a collection of points-to graphs. In a points-to graph, each node corresponds to a memory location loc_i (referred to as an *abstract location*), and an edge from loc_1 to loc_2 means that loc_1 "points-to" loc_2 (i.e., the address of loc_2 is one possible value loc_1 can assume). Note that different program points in the figure contain distinct points-to graphs (not necessarily connected) representing the pointer relationships that are valid after each statement.

Using traditional terminology, the first statement `y=&r` *generates* the relationship "`y` points-to `r`," represented by the topmost points-to graph in the figure. Similarly, statement `p=&x` adds the relationship "`p` points-to `x`" to the set of pointer facts. The statement `p=&q`, at the right branch of the conditional, kills the relationship $\mathtt{p} \rightsquigarrow \mathtt{x}$ reaching that statement, and generates the points-to relation $\mathtt{p} \rightsquigarrow \mathtt{q}$. Similar reasoning applies to the left branch—statement `z=p` kills relations where `z` is the source location, and makes that variable point-to all locations currently pointed to by `p`. At the confluence of the two control paths, the pointer facts coming from both branches are merged, and the subsequent statement `*p=&t` generates the relationships $\mathtt{r} \rightsquigarrow \mathtt{t}$ and $\mathtt{q} \rightsquigarrow \mathtt{t}$, since $\mathit{PointsTo}(\mathtt{p}) = \{\mathtt{r}, \mathtt{q}\}$ immediately before that statement (each element of this set comes from a different branch). Loops in the control-flow are handled by iterating pointer relations within the loop body until convergence.

Formally, the effects of a statement are modeled by its *transfer function*. Instances of transfer functions for canonical pointer statements can be found in the literature [12, 29, 30, 42]. As an example, Figure 2.2 from Emami [30] shows the transfer functions for two canonical statements usually adopted in pointer analyses. Pointer relations are represented with triples $\langle x, y, q \rangle$, where x is the source and y the destination location of the points-to edge, and q is a *may*/*must* qualifier (referred to by Emami as *Definite* or *Possible* pointer relationships).

The transfer functions mean, for instance, that after analyzing an statement `x=&y`, `x` should definitely point-to `y`. In addition, all the relations in which `x` points to a location $\mathtt{x}_1$ should be

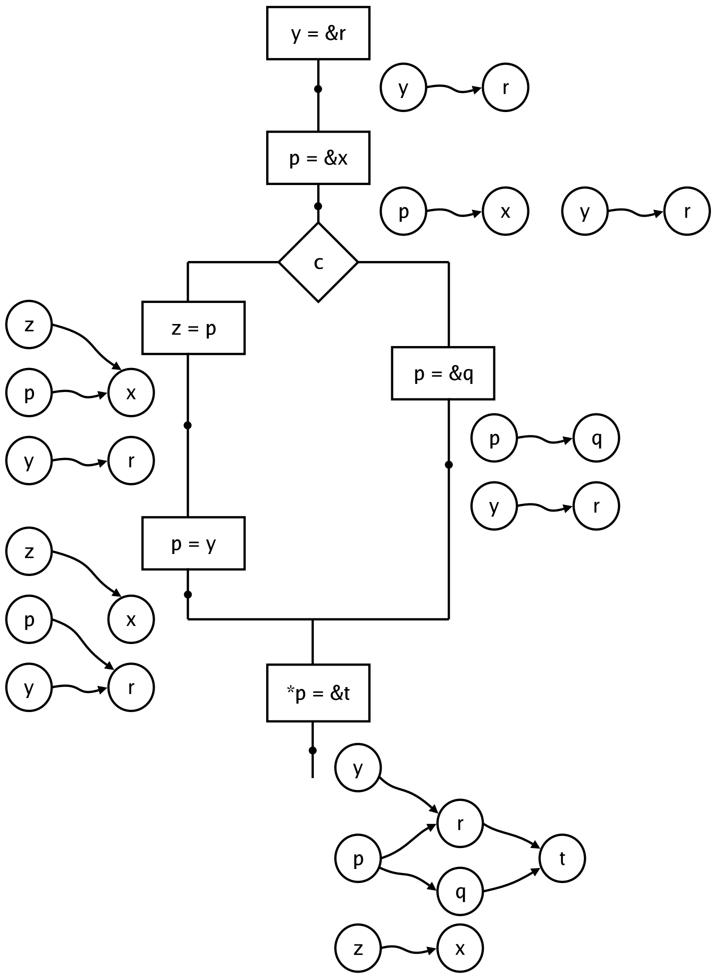


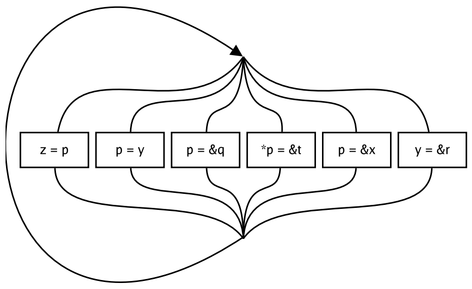
Figure 2.1: A fragment of a C program with its flow-sensitive pointer analysis solution.

x=&y:	x=y:
kill = {(x,x1,q)|(x,x1,q) $\in$ input}	kill = {(x,x1,q)|(x,x1,q) $\in$ input}
gen = {(x,y,D)}	gen = {(x,y1,q)|(y,y1,q) $\in$ input}

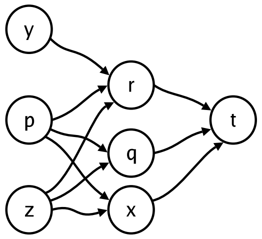
Figure 2.2: Transfer functions for two statements used in traditional pointer analysis.

removed from the points-to set being computed. The *kill* set is the same for statement `x=y`; the *gen* set for such a statement says that `x` should be made to point-to the same locations that `y` points-to, and they should inherit the same qualifier *q*.

In a flow-*insensitive* analysis, each pointer statement also adds new pointer relations to the solution, although there is no kill. A flow-insensitive analysis must, for correctness, capture any



(a)



(b)

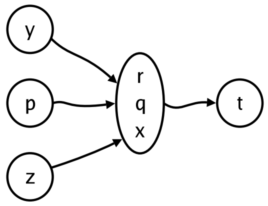


Figure 2.3: (a) A flow-insensitive view of the code. (b) Andersen's and (c) Steensgaard's pointer analyses solutions.

possible path that can be traversed by the set of statements in the function. Intuitively, it can be seen as placing the statements in a big *switch* construct that is enclosed by a loop, as in Figure 2.3(a). Pointer relations are propagated inside this loop until convergence.

There are several variations of flow-insensitive analysis; the two most well known are due to Andersen [2] and Steensgaard [67]. Figures 2.3(b) and (c) illustrate both solutions for the program of Figure 2.1. Each flow-insensitive solution is a single, larger graph containing a single node for any program location. In the case of Steensgaard's, a node can in fact represent several program locations; the interpretation for a points-to edge is that all program locations within the source node may point to any program location in the target node.

Basically, if a single location points to two different objects, Andersen will continue to track the objects separately, while Steensgaard will unify them, tracking them as a single object from that point on (the outdegree of any node in the points-to graph is at most one). This leads (recursively) to unionizing the points-to sets of these formerly distinct objects, and therefore loss of precision.

For both styles, the basic idea is to view pointer assignments as constraints, and use these constraints to generate points-to information. Andersen-style uses *inclusion* constraints so that for assignments $p{=}q$ the algorithm makes $\mathit{PointsTo}(q) \subseteq \mathit{PointsTo}(p)$. Essentially the algorithm is an immediate adaptation of the usual dataflow points-to algorithm [29] to be flow-insensitive, although the constraints are usually solved with dynamic transitive closure. Since there is no order information, the best one can say for an statement `x=&y` is that $\mathtt{y} \in \mathit{PointsTo}(\mathtt{x})$. Andersen's algorithm can be roughly expressed by the following four inference rules:

$$\frac{\texttt{x} = \&\texttt{y}}{\texttt{y} \in \mathit{PointsTo}(\texttt{x})} \qquad \frac{\texttt{x} = \texttt{y}}{\mathit{PointsTo}(\texttt{y}) \subseteq \mathit{PointsTo}(\texttt{x})}$$

$$\frac{*\texttt{x} = \texttt{y} \quad \texttt{x}_1 \in \mathit{PointsTo}(\texttt{x})}{\mathit{PointsTo}(\texttt{y}) \subseteq \mathit{PointsTo}(\texttt{x}_1)} \qquad \frac{\texttt{x} = *\texttt{y} \quad \texttt{y}_1 \in \mathit{PointsTo}(\texttt{y})}{\mathit{PointsTo}(\texttt{y}_1) \subseteq \mathit{PointsTo}(\texttt{x})}$$

Figure 2.4: Rules of inference for Andersen-style analysis.

Each statement in the program generates one of these constraints, and a fixed-point computation calculates the actual points-to sets.

Steensgaard's style is also known as a *unification*- or *equivalence*-based approach; it uses unification constraints so that for pointer assignments p=q, the algorithm makes *PointsTo*(p)=*PointsTo*(q). The algorithm views the problem of computing points-to sets as trying to assign synthetic types to each reference—so it points to objects of specified type. A type is then defined recursively as pointing to another type. Hence, the analysis proceeds as a type inference algorithm, doing unification (e.g., for statement `x=y`, $\tau(\texttt{x}) = \tau(\texttt{y})$; so take *PointsTo*(`x`)=*PointsTo*(`y`)). This behaves as if information flowed both ways, rather than from the right- to the left-hand side of an assignment.

Some analyses use a hybrid technique of both inclusion and unification constraints. The one-level flow algorithm of Das [23] uses inclusion constraints for assignments, but unification constraints everywhere else. It is able to obtain some of the benefits of inclusion-based analysis with an analysis cost only slightly higher than the unification-based analysis.

One can notice that any of the flow-insensitive solutions of Figure 2.3 is less accurate than the flow-sensitive solution of Figure 2.1. For instance, in the flow-sensitive case `z` never points-to `r` nor `q`, and `x` never points-to `t`, yet these spurious relations occur in both flow-insensitive solutions.

Moreover, a flow-insensitive solution cannot answer questions that depend on execution order. Roughly speaking, the best a flow-insensitive analysis can do is to provide reverse information: if it says x does not point to y, we can be sure this is indeed the case. In other words, if flow-insensitive analysis is used for *mod* computation, for instance, the best one can hope for is to infer that a pointer p is not modified by a given function.

In contrast, one of the goals of our new analysis space in Chapter 3 is to derive analyses that can answer more questions without paying the high price of, say, a flow-sensitive analysis. For that, we break pointer analysis into more primitive elements and show how, by putting these elements together in different ways, we can achieve interesting trade-offs.

2.1.2 Context-sensitivity

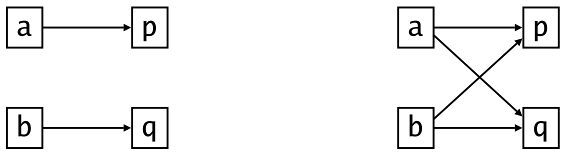
An analysis that transcends procedure boundaries is called *inter*procedural analysis. Its goal is to avoid overly conservative assumptions about the effects of procedures and the program states at call sites. An interprocedural analysis propagates values from one function to another and back, and are more complex than *intra*procedural analysis, which are limited to the scope of a single function.

For computing points-to sets, interprocedural analysis is virtually a requirement. Only extremely conservative estimates are possible by analyzing each procedure in isolation, and such estimates would certainly be useless for most pointer analysis clients. There are basically two classes of interprocedural analysis: context-*insensitive* (or monomorphic) and context-*sensitive* (or polymorphic) analysis [35]. Informally, a context-insensitive analysis does not distinguish between different calls to a function, and may allow information from one caller to mingle erroneously with information from another caller of the same function; such analysis produces just one set of results for each function regardless of how many ways a procedure may be invoked. Context-*sensitive* analysis avoids this imprecision by allowing different contexts to have different results.

As an example of the potential benefit of context-sensitive points-to analysis, consider the program in Figure 2.5 (omitting some variable declarations for clarity). If we distinguish `foo` calling procedure `id` from `bar` calling the same procedure, we discover that `a` points-to `p` and `b` points-to `q` (Figure 2.5(b)). Otherwise, we would conclude that both `a` and `b` can point to {`p`,`q`} (Figure 2.5(c)).

A straightforward approach to obtain an interprocedural but context-insensitive analysis is to combine all procedures into a single control-flow graph, adding edges for calls and returns, and adapt the intraprocedural analysis algorithm to work on this large graph. An iterative dataflow analysis using such a graph is conceptually simple, but may suffer from the problem of *unrealizable paths*. That is, values can propagate from one call site, through the called procedure, and back to a different call site. Some algorithms try to convert this approach into a context-sensitive analysis by tagging the pointer information with abstractions of the calling contexts (such as the call stack).

```
id(x) { return x; }

foo() {
    a = &p;
    a = id(a);
}

bar() {
    b = &q;
    b = id(b);
}
```



(a) (b) (c)

Figure 2.5: (a) A program, (b) context-sensitive points-to sets, (c) context-insensitive points-to sets.

In general, adding context sensitivity can be done by two classical approaches: the *call string* and the *functional* approaches. The next subsections discuss each.

2.1.2.1 Call string approach to context-sensitivity

The call string approach is based on including context information in the lattice of flow values. In such approach, the context may be defined as the top sequence on the runtime call stack; one can thus use a string to record the pending procedure calls. The idea is that dataflow information tagged with consistent call strings corresponds to the same calling context (which is being distinguished). In the example of Figure 2.5, the contents of the call stack when `id` is called is different at each call site. Thus adopting the call string approach would suffice for a context-sensitive analysis.

However, using the call stack alone might not be enough in some cases, even when there are no recursive procedures involved. Consider the example in Figure 2.6. If the analysis does not separate the two invocations of `id`, it will erroneously determine variable `a` in function `main` points-to either `i` or `j`; distinguishing them gives the correct result of `a` pointing only to variable `i`, whose value is set to zero by the last assignment in `main`. In this case, the analysis can identify that there is no division by zero at statement `1/j`; the context-insensitive result would flag it as a potential error.

Nevertheless, note the contents of the call stack immediately before invoking `id` is the *same* for both call sites. In these cases, the string to be used must be derived from constraints on which interprocedural paths are *realizable*. A realizable path is one for which function calls are matched with corresponding returns (a realizable path is not the same as a feasible path. The former involves the program's call-return structure, and the latter depends on the consistency of program predicates).

```
int *id(int *x) {
  return x;
}
int main() {
  int i,j=10;
  int *a,*b;
  a = id(&i);
  b = id(&j);
  *a = 0;
  return 1/j;
}
```

Figure 2.6: Context-sensitive analysis based on the contents of the call stack alone cannot distinguish the two invocations of `id`.

One approach to match calls and returns is based on context-free language reachability (CFL-reachability) [59]. A CFL-reachability problem [77] is not an ordinary reachability problem (e.g., transitive closure), but one in which a path is considered to connect two nodes only if the concatenation of the labels on its edges is a string in a particular context-free language. In this case, a context-free grammar akin to a balanced-parenthesis problem is defined to reproduce the call-return structure of a program's execution. Realizable paths are defined in terms of a program's *supergraph*, G^*. Such graph consists of a collection of control-flow graphs, one for each procedure, including the main function. Each control-flow graph has a unique start node and unique exit node. The other nodes represent statements and predicates in the usual way, except that each function call is represented in G^* by *two* nodes, a *call* node and a *return-site* node, and three edges, two of which are interprocedural edges; Figure 2.7 sketches the supergraph for the program in Figure 2.6.

Note that each interprocedural edge is labeled with "$(_i$" and "$)_i$," depending on its direction. Index i ranges over the number of call sites in the program. Informally, a path in G^* is a matched

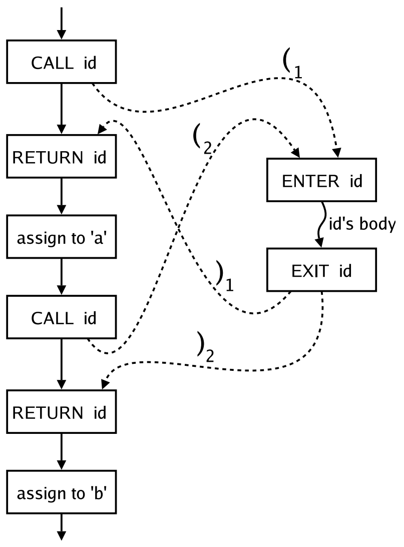


Figure 2.7: The supergraph for the program. Values are only propagated through realizable paths.

path only if there is a balance between “open” and “closed” parenthesis with the same index *i*. During analysis, values are only propagated through realizable paths. Note in Figure 2.7 that this technique avoids mixing up values from distinct call sites; the invocation of `id` with label “$(_1$” only propagates values to the topmost return site (through edge labeled “$)_1$”). In a monomorphic analysis, values flowing through paths “$(_2$” $\rightarrow$ “$)_1$” and “$(_1$” $\rightarrow$ “$)_2$” would also be considered.

Another technique to achieve context-sensitivity for the example of Figure 2.6 is to use the polymorphic analysis based on the typing constraints of Forster et al. [35]. In the example, `id` is the identity function. Because Forster assigns a type for `id` that is instantiated for each call to that function, the points-to sets are computed precisely.

Alternatively, Whaley’s [73] technique for call-string context-sensitivity is based on the notion of *cloning*. Cloning conceptually generates multiple instances of a method such that every distinct calling context invokes a *different instance* (thus preventing information from one context to flow to another). A clone of the method is built for each path through the call graph, linking each call site to its own unique clone. The context of a method invocation is thus distinguished by its call string. Such technique may require an exponential number of clones to be created, since exponentially many distinct contexts are possible. Their approach is to allow the exponential blowup to occur and then rely on Binary Decision Diagrams (BDDs) to find and exploit the commonalities across contexts. Contexts with identical information will automatically be shared at the data structure level.

2.1.2.2 Functional approach to context-sensitivity

The second main approach to context-sensitivity is the *functional* approach. It involves embedding information about program state at the call site, and using that to distinguish calls from one another. The basic idea is to use a transfer function to summarize the effects of the called procedure—parametrized summaries are created for each procedure and then used in generating the summaries of its callers. Although the functional approach is theoretically appealing, its implementation is not straightforward. This is exacerbated when the summary function cannot be expressed symbolically. In this case, one can use a variant to the functional approach known as *assumption sets* [55], adopted by most existing summarization methods. The idea of assumption sets is to use a table with mappings from context information to output flow values to represent the transfer function corresponding to the procedure body for a given input context. The context information is encoded by an assumption set, which is in essence any representation of the abstract state of the program before entering the procedure. Context-sensitivity is achieved through the selection of the right context when propagating and transferring flow values. For example, arguments to the same function often have the same types or similar aliases. This observation led to the concept of partial transfer functions, or PTFs [75], where function summaries for each input pattern are computed on the fly as they are discovered during a top-down traversal of the whole program, starting at the `main` function. This technique analyzes a function for each existing aliasing pattern of its actual parameters (determined by the callers)—for a given function f, there is one computed summary for each distinct calling context under which f is invoked in the existing program. The rationale is that the number of distinct contexts actually used in a program may be small because of commonalities among contexts. For this approach, the context is defined as the aliasing relationships among function parameters at the call site. Thus, the technique provides one procedure summary (i.e., PTF) for all contexts that share the same input/output aliasing relationships. This signifies that a procedure is not summarized for all potential and unforeseen aliases among its inputs, but only for those that occur in the existing whole program. Although its objective is to memoize summaries, the technique is a whole program analysis.

In contrast, Chatterjee et al. [14] propose a functional context-sensitive analysis for Java and C++ that uses a flow-sensitive analysis with conditional points-to relations whose validity depends on the aliasing and type information found in the calling context. Similar to Wilson and Lam [75], this technique stores multiple analysis results (i.e., multiple summaries); each result is indexed under the possible alias relations that may exist at function entry. On the other hand, summaries are built using no information "from above" (i.e., from callers). However, for a function `f` with two pointer parameters `p1` and `p2`, two initial conditions are assumed: either `p1` and `p2` point to the same location(s) when the function is called, or they don't. From these starting points, two initial summaries are generated, which may later branch into more derived summaries. Basically, this technique uses *alias contexts* to lazily enumerate *potential* aliases among parameters in order to distinguish transfer functions for different calling contexts. It is not guaranteed, however, that any unforeseen calling context will be covered by this analysis, because it relies on heuristics to decide which contexts are "relevant" and which are not. Landi and Ryder [48] propose a similar technique where a conditional may alias analysis for C is formulated, handling multiple levels of indirection, for both scalar and aggregate data types. Given a set of alias pairs that are true at function entry, they compute the set of the alias pairs that hold at function exit (i.e., *conditioned* on the alias pairs at the entry).

As will be explained in Chapter 5, our technique is a functional approach that is not based on assumption sets; it represents summary information symbolically.

2.1.3 Path-sensitivity

A common question that arises in program analysis is the following: Given a path in the program, is it feasible? Due to the conservative nature of static analysis, a large portion of paths considered by static tools are infeasible. Avoiding the exploration of such paths can help reducing the number of false alarms. A *path-sensitive* analysis only considers feasible paths through the program, whereas a *path-insensitive* analysis will examine all structurally possible paths, including those that are logically infeasible. For example, if two operations at different parts of a function are guarded by equivalent predicates, a path-sensitive analysis must only consider paths where both operations execute or neither operation executes. In such sense, a path-***in***sensitive analysis can be seen as replacing each conditional expression c in the program with a fresh, uninterpreted variable β_c, so that no two conditions are correlated after the substitution. Path-sensitive analysis can be expensive because accurately tracking every branch in the control-flow of a program in which the execution state differs along the two branch paths may result in an exponential or even infinite search space.

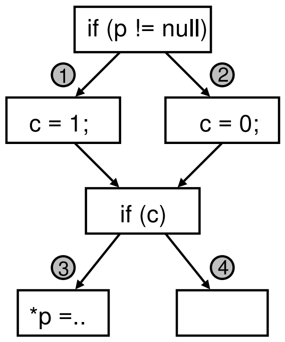


Figure 2.8: A code fragment depicted as a control-flow graph.

Consider the code in Figure 2.8. In this example, the dereference `*p` would cause a null pointer exception only on the path 2 $\rightarrow$ 3, which is not feasible during runtime—even though there are four distinct paths in the code, only two of those are *valid* paths during execution.

To capture this additional information the analysis must provide some form of path-sensitivity, although pointer analyses do not model individual paths per se. Instead, a few existing techniques [52, 76] tag pointer information with predicates to achieve a limited form of path-sensitivity. In such method, program statements are guarded by Boolean predicates expressing the conditions under which each statement may execute. Two points-to edges are related only if they occur under consistent predicates.

For other program analyses, a common technique is *symbolic path simulation* [37] which performs per-path simulation. Since it is prohibitively expensive to explore one path at a time, some approaches enumerate only a representative set of paths whose number is configurable by the user [10]. Alternatively, some analyses use properties of the program to reduce the number of paths that need to be enumerated. ESP [24] tracks values by maintaining a symbolic state of the program. At the merge point in the control flow, if two symbolic states have the same values with regard to some property, they are merged together much like the join operation in dataflow analysis. This selective merging may reduce the number of paths to be explored to a tractable number.

Most path-sensitive analyses use some form of theorem proving to check for consistency of program predicates through a program path. Some tools use expensive theorem provers [3, 10, 40] while others (including BEAM) employ a lightweight prover. In our framework for pointer analysis, we rely on BEAM's theorem prover to provide path-sensitivity in the form of guards, similar to the techniques mentioned above.

Chapter 3

Finer-grain Analysis Dimensions

This chapter constitutes one of the main contributions of this dissertation—a novel analysis space for pointer analysis defined by finer-grain dimensions of precision, allowing us to explore and evaluate a variety of analyses to obtain better trade-offs between precision and efficiency. The following sections introduce the elements needed for such analysis space and then elaborate on a framework to implement it.

3.1 A “cube” idea

Traditional analysis dimensions, specially flow-sensitivity, are restrictive when it comes to exploring trade-offs; their coarse-grain nature lacks flexibility, and as a consequence numerous sweet spots are hindered. Finer-grain dimensions provide more possibilities for finding the right spot, as well as a new way to reason about pointer analysis in general. In this chapter, we present a new look into the dimensions of pointer analysis by decomposing it into the primitive elements of order-, condition-, and kill-sensitivities. Then, we present several ways to recombine those elements to obtain multiple analysis variations as well as to uncover several sweet spots in the space of pointer analysis that have not been discovered so far. Exploring this analysis space is a step towards bridging the gap between cheap but approximate and precise but expensive solutions, providing a different perspective on these trade-offs.

The essence of our analysis space is shown in Figure 3.1, depicting three dimensions of precision for designing pointer analyses. Each analysis variation is a point in this 3-D space.

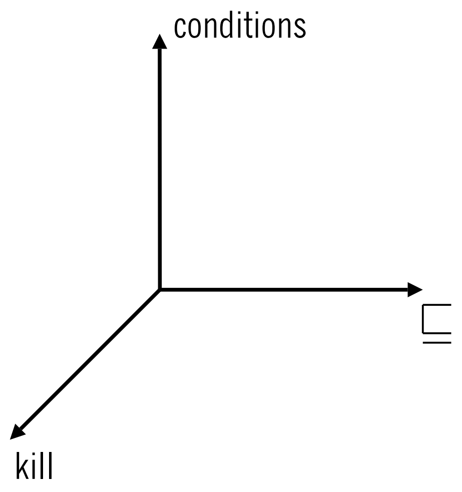


Figure 3.1: Three fine-grain dimensions of precision for pointer analysis.

The horizontal axis, labeled "$\sqsubseteq$," represents the degree of statement ordering that is respected by the analysis. For example, a traditional flow-insensitive analysis, which completely ignores the order of statements, would be located at the origin of this axis.[1]

The vertical axis corresponds to the *condition*-sensitivity of the analysis, i.e., how the predicates in the program are taken into account. Once more, a classical flow-insensitive analysis is placed at the origin of this axis, as it does not separate mutually exclusive statements σ_1 and σ_2 (e.g., because they belong to opposite branches of an `if`-statement). A traditional flow-sensitive analysis considers this information while propagating dataflow facts through the control-flow graph, and would correctly make σ_1 and σ_2 independent; a path-sensitive analysis goes beyond disassociating immediate exclusive branches only—it attempts to correlate distinct conditional expressions.

Finally, the third axis, labeled "kill," refers to handling strong updates, i.e., whether a statement is allowed to destroy the dataflow facts created by another statement. Besides this binary choice, intermediate levels of "kill" can be defined—e.g., strong updates can be allowed for scalar variables but neglected if the location being written is the result of a pointer dereference, etc.

Combining different values for each dimension gives different analysis variations. Some of these variations lead to interesting sweet spots, as discussed later in this chapter and in Chapter 4.

[1]For this type of analysis, it is irrelevant whether a given statement σ_1:`x=&y` occurs *after* another statement σ_2:`*x=z` in the code. The dereference of `x` in σ_2 still reads the value set by σ_1, even though σ_1 does not affect σ_2 at runtime.

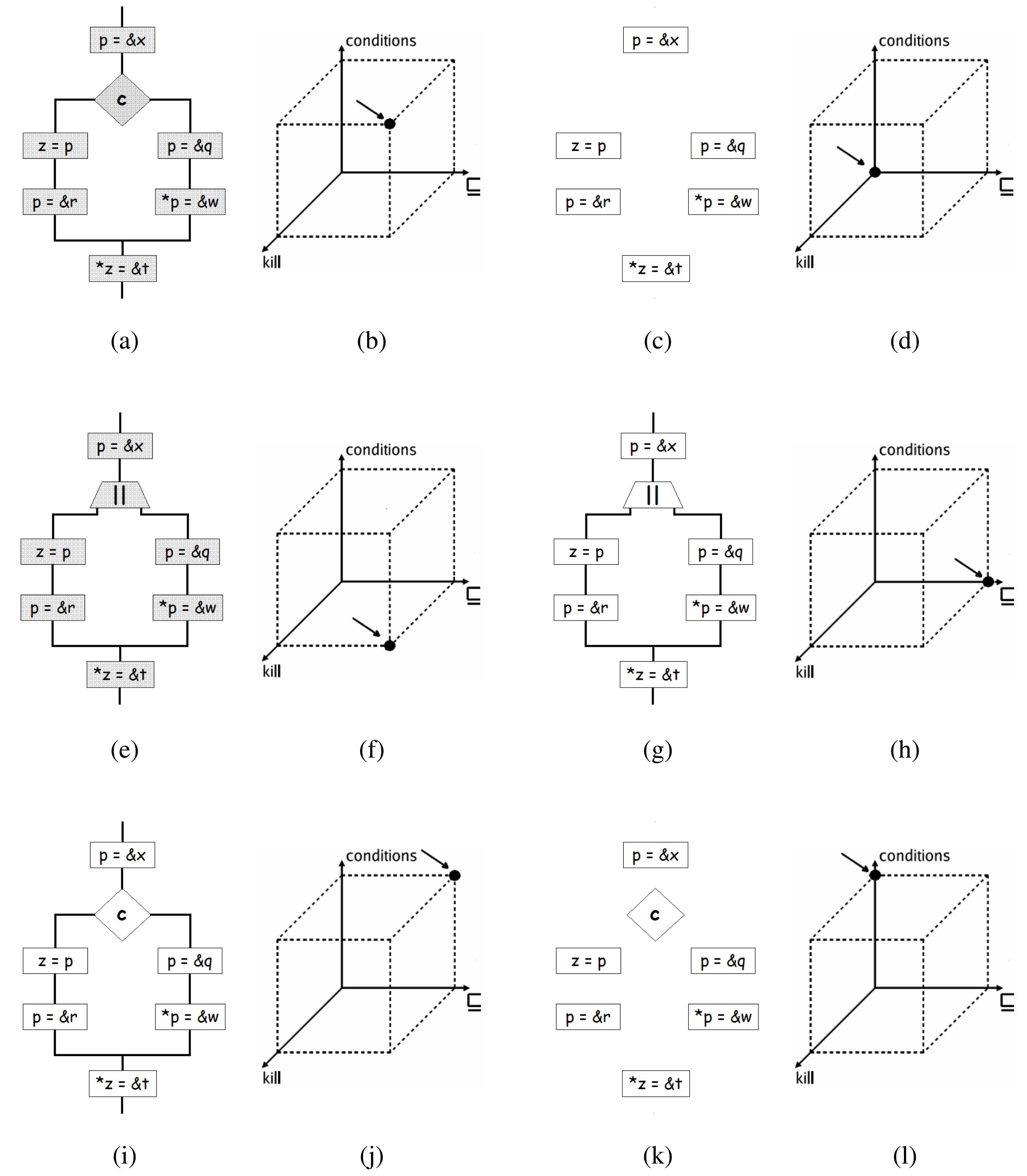


Figure 3.2: Each analysis variation can be seen as ignoring different aspects of the code, or alternatively, modifying the program's original semantics. In (a) the program is considered exactly as given (filled rectangles express strong updates); this gives the point in (b). In (c) everything is neglected except the program statements. The resulting analysis point is shown in (d). Ignoring the conditions is expressed by replacing the test by a "parallel" construct in (e) and (g). The semantics for (i) does not consider strong updates; the resulting analysis is shown in (j). Ignoring statement ordering and kill, but considering statement guards gives (k) and (l).

Given a program *P*, each pointer analysis variation can be also seen as ignoring different aspects of the code or, alternatively, modifying *P*'s original semantics. Intuitively, the more elements are ignored, the cheaper the analysis. Figure 3.2 illustrates several such points of view (control-flow graphs are used to represent fragments of programs). We use a filled rectangle to express the fact that strong updates are considered; empty rectangles mean the opposite.

Figure 3.2(a) shows a program fragment. An analysis that fully considers statement order, strong updates, and branch correlations, strives at analyzing the program exactly as it is given. In our 3-D space, such analysis is located at the point highlighted in Figure 3.2(b), and it is the most expensive. On the other hand, ignoring all three elements is akin to restructuring the program as shown in Figure 3.2(c), (i.e., a "bag" of statements), and corresponds to the point illustrated in Figure 3.2(d). Interestingly, such analysis is *not* the least expensive, contradicting the intuition.

Embracing strong updates and statement ordering, but neglecting conditionals, can be viewed as replacing tests with fictitious "parallel" constructs such as illustrated in Figure 3.2(e), and represents the point in Figure 3.2(f). This means that the relative order of the statements in opposite branches of an `if` statement is non-deterministic; within the true and false branches, however, the program order is observed.

From the scenario in Figure 3.2(e), eliminating strong updates (Figure 3.2(g)) leads to the point in Figure 3.2(h); restoring conditional expressions results in Figure 3.2(i) and 3.2(j), and ignoring statement ordering once again gives Figure 3.2(k) and 3.2(l), which can be viewed as a order-insensitive, kill-insensitive but condition-sensitive analysis (each statement has its guard).

Each of these analysis variations gives a different points-to solution for the same input program. Assuming we build a summary solution instead of a program-point specific solution, Figure 3.3 shows the analysis results for the same six points in the analysis space. Each solution represents the set of valid dataflow facts that summarize the procedure. The specific notion of *valid* changes according to the analysis variation being considered, and will be formalized in Section 3.2.

In Figure 3.3(d), a traditional Andersen-style [2] flow-insensitive solution is obtained by ignoring everything but the program statements themselves. At the other extreme, the more precise graph in Figure 3.2(b) is achieved when all elements are considered. For instance, the dereference of `p` at statement `*p=&w` cannot read the address of `r` because `p=&r` occurs in the opposite branch. Also, statement `p=&x` is killed by `p=&q` prior to statement `*p=&w`, and therefore `q` is the only location

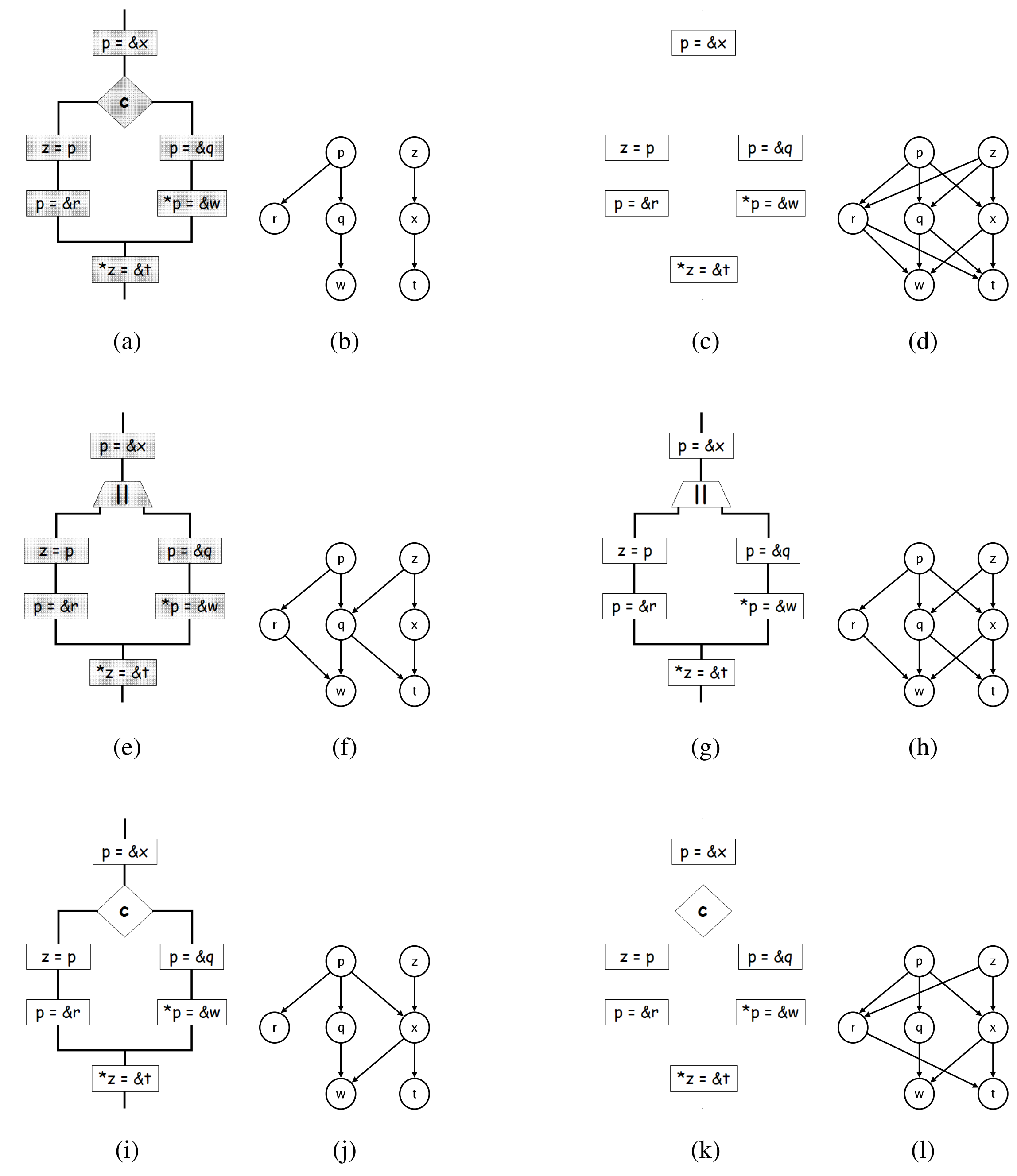


Figure 3.3: Points-to summaries for the six analysis variations of Figure 3.2.

that can be set to point to w.

When program conditions are neglected (Figure 3.3(e)), the fictitious parallel construct makes the two branches conceptually execute simultaneously, with a non-deterministic synchronization except for the control flow join. This means *p in statement *p=&w can read either q or r, and thus both points-to relations $q \rightsquigarrow w$ and $r \rightsquigarrow w$ become valid. Similarly, because the right-hand side in z=p can read the value set by p=&q, statement *z=&t also sets q to point-to t. The resulting graph is shown in Figure 3.3(f).

By further removing strong updates (Figure 3.3(g)), the deference of p in *p=&w can also refer to the address of x, and make that variable to point-to w as well. The solution for Figure 3.3(g) is shown in Figure 3.3(h).

In Figure 3.3(i), only kill information is removed from the program's original semantics from Figure 3.3(a), resulting in the graph of Figure 3.3(j). In Figure 3.3(k), statement ordering and strong updates are ignored, but predicates can be used to disassociate mutually exclusive operations. The result is depicted in Figure 3.3(l).

Note that choosing one of these analysis variations over another may impact the outcome of a client analysis. Assume that a false-positive-suppressing tool needs to "prove" (similar to Section 1.2.1) that *z=&t does not write to variable q. Then, only solutions (b), (j), and (l)—i.e., their respective analysis variations—would be effective. In this case, the reason is because these analyses consider the conditions in the code and hence are able to separate mutually exclusive branches. In contrast, the other variations include the spurious information that z points-to q.

Similarly, suppose we need to show that statement *z=&t does not write to variable r. Then, solutions (b), (f), (h), and (j) are effective—this time because statement order is taken into account by each of them. Without order information we have to assume that z may point-to r and thus statement *z=t potentially affects that variable.

Fundamentally, the general mechanism to compute points-to sets is the same for any of these analysis variations (including the classical ones): for each pointer dereference in the code, we need to determine which pointer assignments could have set its value. Different analyses answer this questions differently. The exact answer is undecidable in general, and thus any effective analysis is an over-approximation. In the 3-D analysis space, different approximations can be obtained depending on which analysis variation is considered. To compute such solutions, we show a general mechanism that is built upon a graph representation we have dubbed the Assign-Fetch Graph (AFG), which allows us varying levels of precision by allowing different assignment/dereference matchings. Details are given in Section 3.2.2.

To conclude this high-level overview, Figure 3.4 presents the potential placement of traditional pointer analyses in our 3-D analysis space.

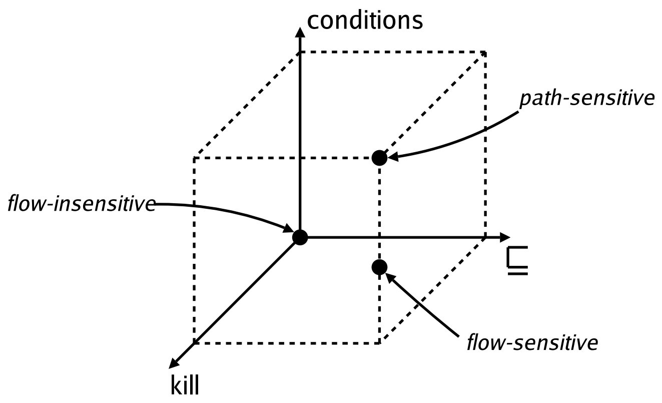


Figure 3.4: Points for traditional pointer analyses.

A traditional *flow-insensitive* analysis corresponds to the origin of the analysis space, since (1) the program is considered as a bag of statements; (2) conditions in the code are simply assumed nonexistent; (3) only weak updates are modeled. A *flow-sensitive* analysis considers strong updates and statement execution order, and can be seen as replacing each conditional expression by a fresh variable—mutually exclusive statements on opposite branches of an `if`-statement do not interfere with each other, although distinct conditionals are always unrelated (we informally depict this by adding some degree to the vertical axis). A (guarded form of) *path-sensitive* pointer analysis is placed at the corner of the "cube" in which all three characteristics are fully considered. As we go towards that corner starting from any point, accuracy always increases. Interestingly, we will show that there are points in this analysis space that do not adhere to the intuitive trade-off "increased precision, decreased efficiency." We will demonstrate cases where the increased accuracy is obtained *faster* than a baseline analysis that was assumed to be cheaper.

3.2 Basic analysis framework

This section introduces the framework upon which the analysis space of Section 3.1 unfolds. We describe in Sections 3.2.1 and 3.2.2 two main data structures, the *Flow Graph* and the *Assign-Fetch Graph* (AFG), which are manipulated by the analysis. Chapter 5 will provide a detailed treatment of the latter.

Classical pointer analysis techniques develop their algorithms and representations around higher level structures, such as the control-flow graph of a function. They commonly use the following four canonical statements: `x=&y`, `x=y`, `x=*y`, and `*x=y`, and a collection of inference rules covers any syntactically correct program. To break up pointer analysis into more primitive elements, our framework is based on a different view:

- a functional (dataflow) representation of the program, called the Flow Graph, is used as the input for the analysis. The flow graph is BEAM's intermediate representation of programs.
- loops in the code are substituted by tail-recursive procedures, and hence the loop body is replaced by a function call. Thus, each procedure is translated into one *or more* flow graphs (one for each loop), each of them being an acyclic representation.
- the control structure of a procedure is manipulated as a partial order "$\sqsubseteq$" on its statements.
- conditions imposed by `if` statements are mapped into guards under which operations execute—every relevant operation on the AFG is guarded by a boolean expression.

In our pointer analysis, we deal with an abstract definition of a procedure as a partially ordered set $(\Sigma, \sqsubseteq)$ of statements $\Sigma = \{\sigma_1, \sigma_2, \ldots\}$, where each statement σ_i has a guard g_i representing the control predicate under which σ_i executes. Also, instead of defining the analysis in terms of the usual canonical statements, the representation deals with individual memory accesses: assignments and memory dereferences (fetches).

Figure 3.5 illustrates the general flow from source code to the Flow Graph and AFG representations, and then we describe each in more detail.

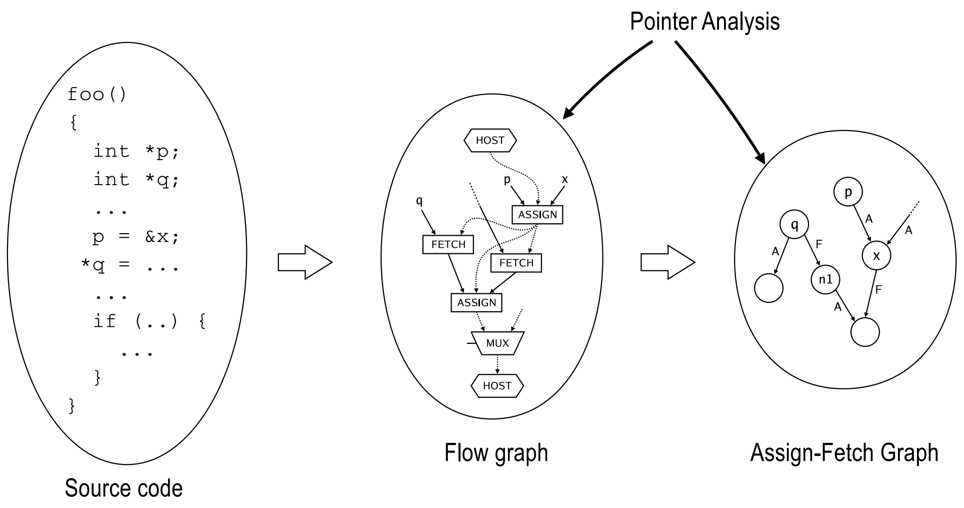


Figure 3.5: The source code for a function, its flow graph, and AFG.

3.2.1 Flow graph

The flow graph is the intermediate representation of programs developed at IBM that is used by the BEAM tool. As mentioned in Section 3.1, each function in the source program is first translated into one (or more) graphs that have a dataflow, functional style.[2] The purpose of this translation is to get the input program into the most suitable form for logical and program analysis. This section overviews the basic structure of this representation; its full treatment is not publicly available at the moment.

There is a classical way of drawing an algorithm as a graph: the flowchart. In a flowchart the majority of nodes have an operator assigned to them, an incoming edge and one or more outgoing edges. To execute a flowchart, one begins at the start node, execute the operators and follow the arrows, making choices according to the results of *if* operators, possibly making loops, until reaching the end node. A flow graph is akin to a flowchart in that its edges carry states of memory and its nodes may update the input state of memory M to produce the output state of memory M'. However, the semantic rules of executing a flow graph are different. For instance, in a flow graph one can simultaneously follow more than one path, *delaying* the "if..then" choice till the paths merge. Multiplexor nodes (muxes) are used for this purpose, where data and control inputs are used to select and propagate memory states.

[2]At a high level, they can be seen as a "hardware circuit" characterization of a function.

Formally, the *Flow Graph* is a directed acyclic graph where each node (referred to as a “gate”) represents a particular operation to be performed on the input edges, such as storing a value in memory. The output edges of a gate are grouped into *nets*. Figure 3.6 shows a generic representation of a flow graph node (we will interchangeably refer to “nodes” and “gates” when no confusion arises).

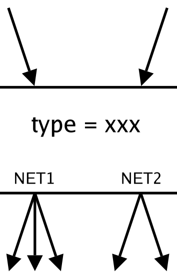


Figure 3.6: A generic flow-graph gate.

A flow graph is executed just in the same way as a logical circuit with *not*, *and*, and *or* gates. Only, (i) instead of 0/1 signals, some edges carry the complete *state of memory*, which conceptually includes the values of all global variables and the values of all variables local to the function. Edges that carry states of memory are called *control* edges, shown as dotted lines in the figure that follow. All other edges are referred to as *data* edges, represented by solid lines. All *output* control edges that belong to the same net carry the same state of memory; and (ii) instead of Boolean *and*, *or*, and *not*, the gates perform certain operations on the input states of memory so that when passing through a gate the state of memory might change according to the semantics of the gate. We interleave the relevant details of a flow graph with an example.

Figure 3.7(a) shows a C function, omitting variable declarations, while Figure 3.7(b) illustrates a simplified flow graph representing the code. There are four types of nodes in this graph: *host* nodes, representing entry and exit points of a procedure, *assign* nodes, representing changes of memory, *fetch* nodes, representing reading from memory, and *location* nodes, representing program variables. In particular:

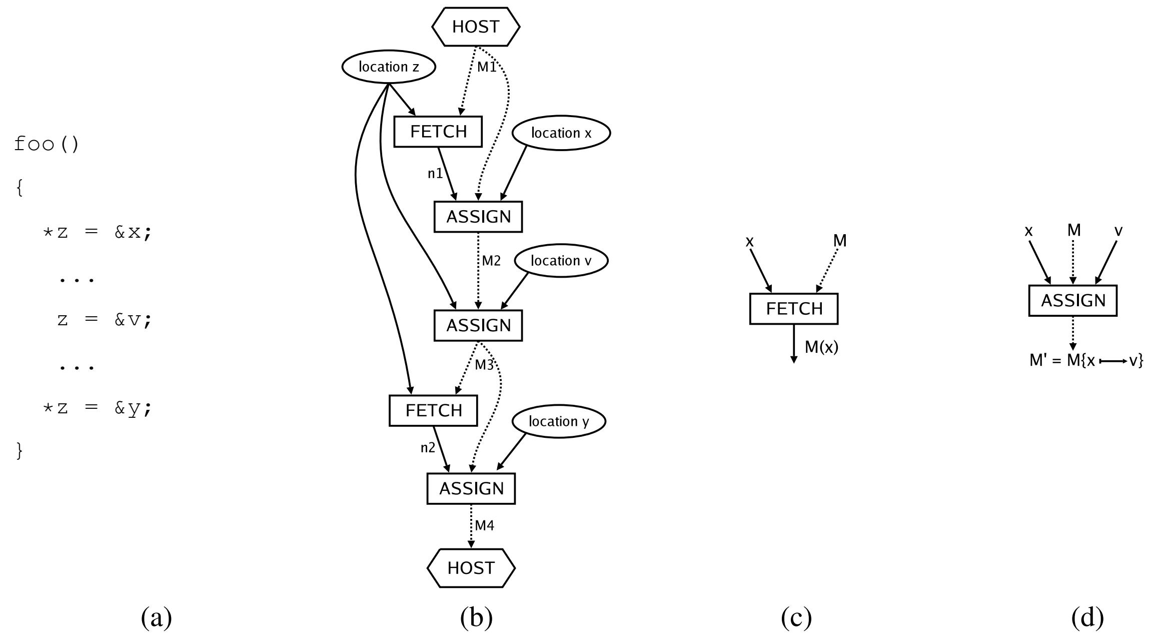
- *host*: a flow graph contains two host nodes: the *top* host, representing the entry point of the procedure, and the *bottom* host, representing the exit point. Multiple exits from a function can be handled in the usual way, *i.e.*, by introducing a unique exit point and re-directing to it the multiple exits. The top host must only have output edges, and the bottom host has only input edges. The output *control net* of the top host carries the state of memory at the time of the function call (M_1 in the figure).

- *fetch*: a fetch gate has one input control edge, carrying the current state of memory, one input data edge, representing the address of memory to be fetched, and one output data net, representing the value being fetched.
- *assign*: this gate has two input data edges, representing respectively an address and a value to be stored at that address, one input control edge and one output control net. The state of memory on the output control net is derived from the state of memory on the input control edge by executing the assignment.
- *location*: each location node represents a program variable, and is associated with a constant address such that the addresses of different location nodes do not overlap with each other.

Figures 3.7(c) and 3.7(d) show the formal meanings of FETCH and ASSIGN gates, respectively. In this figure, M represents a memory state, $M(x)$ is the value stored at address x while in state M, and $M' = M\{x \mapsto v\}$ is the new memory state obtained from M by storing value v at address x.

The graph in Figure 3.7(b) can be seen as as a low-level dismantling of the statements in Figure 3.7(a). An assignment statement is converted into an assign gate whose input data edges designate the left and right hand side expressions. A fetch gate is inserted to access the *value* of a variable (in general, to access the value of an arbitrary address). For instance, the topmost fetch gate in Figure 3.7(b) represents the pointer dereference `*z` in the first statement of `foo`. The fetched value is represented by the *data net* labeled n_1. Each net in the flow graph of a function is given a unique name.

Since the flow graph is simply an intermediate representation for the function, n_1 represents an unknown value—in Figure 3.7(b), fetching z at memory state M_1 yields the *unknown initial value* of z at function entry. To that initial value, the topmost assign gate writes the address of x to produce the memory state M_2. Then, assigning the address of v to variable z results in the memory state M_3. From there, z is fetched and the resulting location is assigned the address of y. The output memory state, M_4, is the memory state before the function returns.



```
foo()
{
  *z = &x;
   ...
   z = &v;
   ...
  *z = &y;
}
```

Figure 3.7: (a) a fragment of a C program omitting variable declarations, (b) the simplified flow graph representing the program in (a), (c)–(d) the semantic interpretation of *fetch* and *assign* gates. M is a memory state, $M(X)$ is the value stored at address X in state M, and $M' = M\{X \mapsto v\}$ is the new memory state M' obtained from M by storing value v at address X.

Figure 3.8 lists the four canonical statements used in pointer analysis and their respective flow graph representations.

Consider the first canonical statement `x=&y`. Both the *l*-value "`x`" and the *r*-value "`&y`" are locations, and the statement is modeled by a single assignment as shown in Figure 3.8(a). Note that this statement does not read or change the contents of `y`.

By contrast, since the statement `x=y` does read the value of `y`, a fetch gate is introduced to read these contents, which are represented by n_1 in Figure 3.8(b). The assign gate in the same figure writes the contents of y to location x. Similar reasoning can be applied to construct the other two graphs in the figure.

The address being written to by an assign gate is referred as the "address net" of that gate. Similarly, the "value net" represents the *rhs* of the assignment. Also, the address net of a fetch gate corresponds to the address being fetched, and the value net is the output of the gate. Figure 3.9

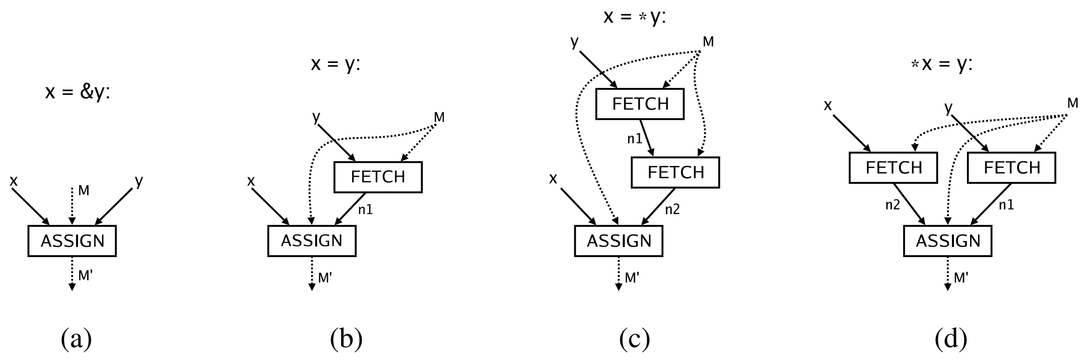


Figure 3.8: The conventional canonical statements and their respective flow graph representations. M is a memory state, and M' is a new memory state obtained after a memory change.

illustrates these notations using several scenarios. In general, a net can be connected to more than one gate, e.g., the output net of location z in Figure 3.7. The right side of Figure 3.9 illustrates a case where the *value* net of a fetch also serves as the *address* net for an assign. In the same figure, location x is both the value net for one assignment and the address net for another.

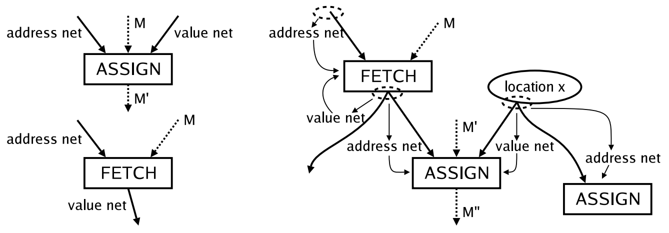


Figure 3.9: The *address net* for an assign gate is the address being written to. The *value net* represents the right-hand side of the assignment. For a fetch gate, the *address net* is the address being fetched, and the *value net* is the output of the gate.

Given Figure 3.8, it is straightforward to check that the flow graph in Figure 3.7(b) is a valid representation of the program in Figure 3.7(a). One missing detail regards the program's execution order. In the original code, this is given by the control structure of the function, generally represented by its control flow graph; roughly, the interpretation for the flow graph is that of a topological ordering of the gates (memory state and data edges taken into account). By construction, the existence of two statements *i* and *j* such that *i* appears strictly before *j* in the program text will at least induce a memory state edge between the last low-level action of *i* and the first low-level action of

j. Assuming the contrary, i would not have any effect in the program state, and could therefore be removed.

3.2.2 Assign-Fetch Graph

This section introduces the main data structure of our pointer analysis, and one of the contributions of this dissertation. It will be explained in the context of BEAM's flow graph, although it can be made independent of such.

The *Assign-Fetch Graph* (AFG) for a function f is initially derived from f's flow graph G, and it concisely represents the assignments and memory dereferences in G. The basic goal of pointer analysis is to produce a transformed AFG that summarizes the pointer behavior of the function. Depending on the value for each analysis dimension, a different "summary AFG" is obtained.

Unlike a points-to graph, whose nodes represent pointer variables and whose edges represent points-to relations, the nodes in our AFG represent locations and values and edges represent reads and writes to memory. Pointer analysis amounts to matching pointer dereferences ("fetch edges") to pointer assignments ("assign edges"). The AFG allows varying levels of precision by allowing different matchings; a more selective matching (i.e., a more precise analysis variation within the cube) may require more resources.

The AFG has a number of other features that makes it superior, specially for modular bug finding, than traditional points-to graphs. For example, it allows us to infer points-to, unconstrained alias, and *mod* analysis queries on the same representation; a single AFG can summarize a procedure for any unforeseen calling context; from the AFG one can determine whether a pointer has been dereferenced within a function; etc.

Our AFG is designed to let us compute what values a program might read from memory while it is executing; since the program itself must have written these values, pointer analysis can be thought of as an attempt to understand which writes could be seen by each read. One approximation is that each write to a location can be seen by every read of that location, but this is usually an overapproximation: a read and write may occur in different branches of a conditional, or a write might occur in sequence after a read. Our analysis space enables us to approximate these relationships differently, producing different pointer analysis algorithms; the AFG is the main data structure behind it.

In our implementation, AFG *nodes* correspond to flow graph *nets*, and AFG *edges* map to flow graph *gates*. An assign edge represents a write to memory and a fetch edge represents reading from memory. Figure 3.10 shows two basic rules used to convert flow graph operations into corresponding AFG representations. The address and value nets of the assign and fetch gates are represented by l_1 and l_2, respectively. Assign and fetch gates in the flow graph become assign and fetch *edges* in the AFG (labeled 'A' and 'F,' respectively).

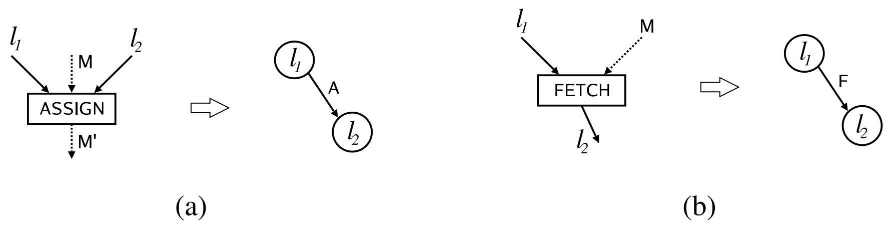


Figure 3.10: Rules to construct the AFG from the flow graph (a) assign rule (b) fetch rule.

Each (relevant) net in the flow graph has a corresponding node in the function's AFG. This is illustrated by the long dashed arrows in Figure 3.11, which corresponds to the AFG for the flow graph previously shown in Figure 3.7(b). The following subsection illustrates a simple example of pointer analysis using the AFG.

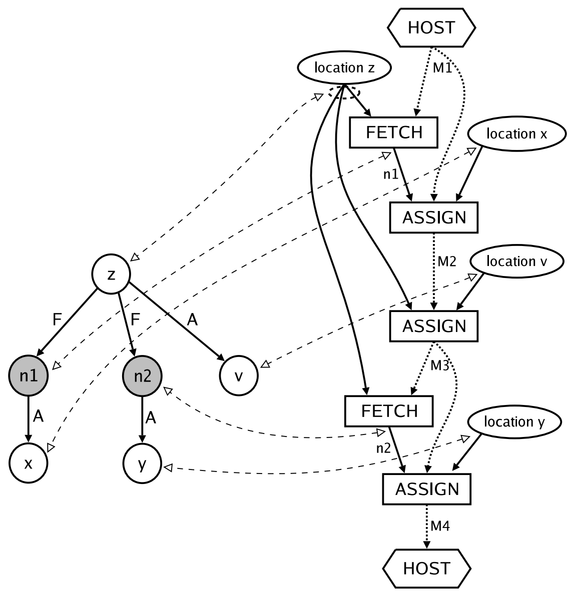


Figure 3.11: The AFG for the flow graph of Figure 3.7(a).

3.2.2.1 An example

Figure 3.13(b) depicts the AFG for the C code in Figure 3.13(a). The first statement in Figure 3.13(a), `*z=&x`, stores the address of the global variable `x` at the address in `z`. We represent `z` with the location node `z`, the dereference of `z` with the fetch edge F_1, the address read from `z` with the *fetch node* n_1, the address of `x` with the location node `x`, and the write to `*z` with the assign edge A_2.

In our figures, we shade each fetch node as a reminder that we do not know its value when we construct the graph. The basic question for pointer analysis then becomes, *given a fetch, which assigns should it match*? The AFG abstraction allows us to answer this question differently depending on speed/precision trade-offs.

Answering this question is the goal of the *resolution phase*, which adds *alias edges* from fetch nodes to location nodes to indicate what values could be fetched. Each fetch of the same variable in a procedure generates a distinct fetch node, allowing the AFG to represent variables that take on different values at different times.

```
 x = &y;   // A1
.. = x;    // F2
```

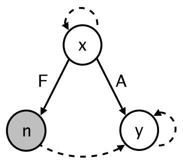
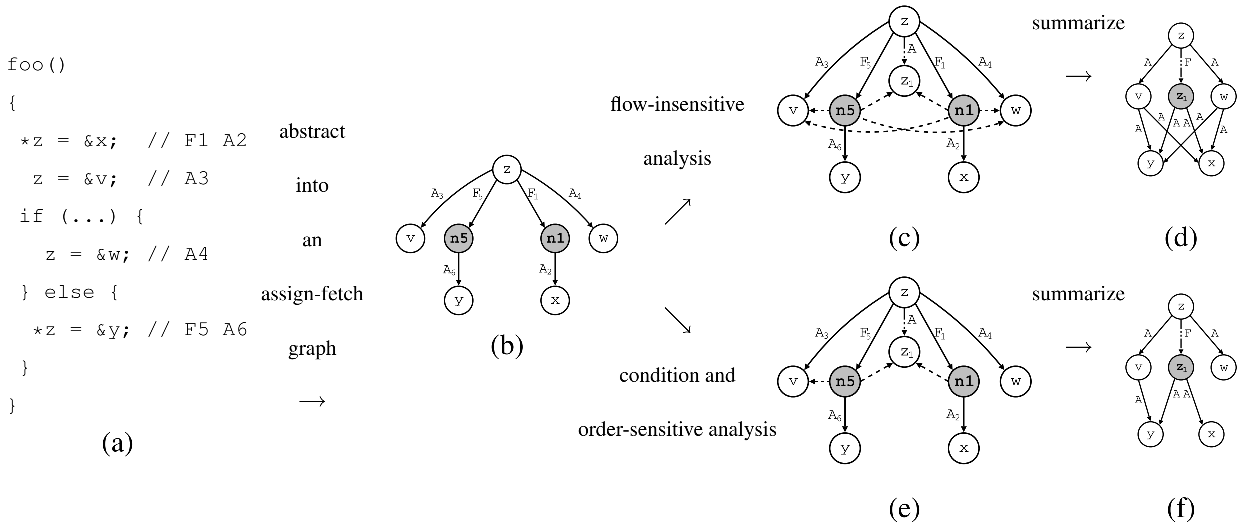


Figure 3.12: The simple case: `x` is assigned in A_1 and fetched in F_2; an *alias edge* indicates that *n* can be an alias for `y`. Self-loops represent trivial aliasing of locations.

Figure 3.12 shows the simplest alias case: we add a (dashed) alias edge from *n* to `y` to indicate fetching `x` (F_2) after assigning it the address of `y` (A_1) returns the address of `y`. The self-loops indicate, e.g., the address of `x` is itself (trivial aliasing of locations nodes). We omit them in all other figures.

Adding aliases from fetch to location nodes produces a *resolved* AFG. Using different analysis variations generate different resolved AFGs, such as Figures 3.13(c) and 3.13(e). The former is a traditional flow-insensitive view of the program, where a fetch from a location matches any assignment to the same location; the latter is a more precise result obtained when considering statement ordering and mutually exclusive operations, which leads to a smaller number of fetch/assign matchings. E.g., because they appear in separate branches of a conditional, `*z=&y` and `z=&w` are mutually exclusive, so fetch F_5 cannot see assign A_4 and there is no alias edge from n_5 to `w` in Figure 3.13(e). Also, the first fetch of `z` in the code (F_1) can only see the (unknown) initial value of `z` coming from the environment (represented by $\texttt{z}_1$). Thus, n_1 to $\texttt{z}_1$ is the only alias edge.



```
foo()
{
 *z = &x;  // F1 A2
  z = &v;  // A3
 if (...) {
   z = &w; // A4
 } else {
  *z = &y; // F5 A6
 }
}
```

Figure 3.13: A simple illustration of pointer-analysis using AFGs. A procedure (a) is first abstracted as an assign-fetch graph (b), whose nodes represent addresses and values and whose edges represent memory operations. Several approximations are possible for the set of alias edges: a flow-insensitive analysis (c), where potential aliases are calculated ignoring statement order to produce a summary (d); considering execution order and mutually exclusive operations (e) to produce a more accurate summary (f).

Figure 3.14 shows AFG fragments for the four canonical statements. For `x=&y`, we represent the lvalue `x` and the rvalue `&y` as location nodes and connect them with an assign edge indicating `x` points to the memory location for `y`. For `x=y`, a fetch node n_1 is introduced to represent the contents of `y`, which are written to `x` through an assign edge. The expression `*y` in the right-hand side of an assignment requires two pointer dereferences (Figure 3.14(c)), whereas `*x` in the left-hand side is converted in to a single fetch. The assign edges in these cases should be clear.

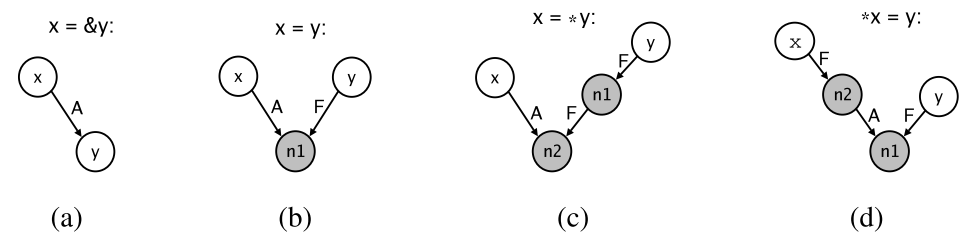


Figure 3.14: The canonical statements and their respective Assign-Fetch Graph representations.

3.2.2.2 Pointer Alias analysis

In alias analysis, executing statement `p=&r` creates the alias relation $\langle$`*p, r`$\rangle$, meaning `*p` is *an alias for* `r`. Computing points-to sets using the AFG amounts to determining the locations for which a fetch node could be an alias. We represent such relations by adding directed *alias edges* to the AFG; each alias edge corresponds to a potential alias relation in the code. In Figure 3.12, the dashed edge (n, `y`) indicates the alias relation $\langle$`*x, y`$\rangle$. Each (non-trivial) alias edge starts at a fetch node and terminates at a location node.

The central goal of pointer analysis is to determine a small set of alias edges that includes every possible one (i.e., remains sound). While the most conservative over-approximation is to proclaim that every fetch node aliases every location, such a gross overapproximation would not be very helpful. Instead, the goal is to produce analysis results that are not hard to compute and that have as few aliasing relations as possible (namely, the minimum number of alias edges that give a sound result). In the following sections, we show how using different analysis variations within the 3-D space produce different minimal sets of alias edges for a given AFG. Each such set corresponds to a different points-to solution, with distinct precision levels.

3.2.3 Determining aliases

Determining aliases between fetch nodes and location nodes is the main step in pointer analysis on AFGs. In this subsection, we discuss a general rule of inference that will be specialized to create different analyses with varying levels of precision; that the AFG lends itself to such variants is one of its key strengths. A second strength is its ability to summarize procedures so as to fulfill the requirements of modular bug finding listed in Section 1.3.3. This will be detailed in Chapter 5.

Let `x`, `y`,... denote *location* nodes, $n_1, n_2, ...$ denote *fetch* nodes; and $\alpha, \beta, ...$ denote arbitrary nodes. We write $al(\alpha)$ to indicate the set of nodes that α can be an alias for. In Figure 3.12, $al(n) = \{\mathtt{y}\}$, $al(\mathtt{y}) = \{\mathtt{y}\}$ and $al(\mathtt{x}) = \{\mathtt{x}\}$.

An alias edge from a node α to a node $\mathtt{x}$ indicates $\mathtt{x} \in al(\alpha)$; an alias edge's target is always a location. We assume variables are distinct, so a location node only aliases itself: $al(\mathtt{x}) = \{\mathtt{x}\}$.

A fetch node n can be an alias for many locations; computing them is the main purpose of any analysis. Because a fetch can only return a value that the program wrote to memory, any alias of a fetch node must be the target of an assign edge (we model the initialization of global variables with assign edges).

We write $affects(\sigma_A, \sigma_F)$ to indicate the assign edge σ_A could write a value that fetch edge σ_F could read. This relation can be many-to-many: one assignment could be seen by many fetches, and a fetch might see many assignments.

The relation *affects* is an "oracle" that, given a pair assign/fetch, determines whether or not the pointer assignment σ_A is seen by the fetch σ_F. As with any other useful property about computer programs (Rice's theorem), *affects* is not effectively computable, and therefore any pointer analysis must be an approximation of such relation. Different approximations result in different sets of alias edges for the same initial AFG, thus leading to different pointer analysis solutions. The more accurate the approximation, the more accurate the analysis. For a sound analysis, it should always be an *over*-approximation, i.e., the approximation should always be true when the relation is, but not necessarily vice versa.

When an assignment affects a fetch, the fetch can return anything written by the assignment, so aliases for the fetch must include all aliases of the assignment's right-hand side. Put formally,

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad affects(\sigma_A, \sigma_F)}{al(\beta) \subseteq al(n)} \qquad [\text{ALIAS}]$$

where $\gamma \xrightarrow{A} \beta$ indicates an assign edge from γ to β and $\alpha \xrightarrow{F} n$ indicates a fetch edge from α to n. The solution to pointer analysis is the minimal set of alias edges that satisfies this rule.

Consider the previous example in Figure 3.3. The assignment `p=&r` affects the fetch of `p` in statement `*p=&w` if condition `c` is neglected (e.g., Figure 3.3(c)), but this does not happen in case we consider that the two statements are mutually exclusive.

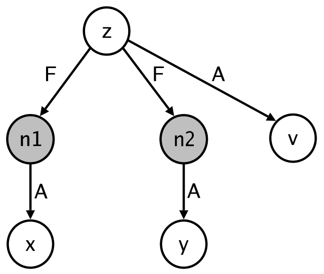
Unfolding the 3-D analysis space of Section 3.1 corresponds to providing different approximations for the *affects* relation. The next subsection presents an approximation that gives the origin of the analysis space, and the subsequent sections show how the three axis unfold from there.

3.2.4 Space origin—Flow-Insensitive Analysis

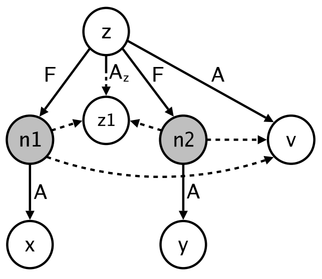
Section 3.2.2 described pointer analysis of arbitrary precision that depends on the uncomputable *affects* relation. In this section, we describe one possible approximation to *affects* that gives the origin of the analysis space, which as mentioned corresponds to an Andersen-style flow-insensitive analysis. We also describe the main steps of any analysis: building an initial AFG, adding alias edges to obtain a resolved AFG, and removing information that's not visible to callers to produce the summary AFG. Consider the function in Figure 3.15(a). Assume all variables mentioned in the code are globals.

```
foo()
{
  *z = &x;
   ...
   z = &v;
   ...
  *z = &y;
}
```

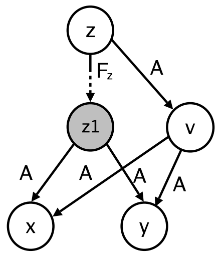
(a)



(b)



(c)



(d)

Figure 3.15: (a) A function and (b) its initial AFG, (c) its resolved AFG, (d) its summary AFG for a flow-insensitive analysis.

We first generate the initial AFG for `foo` depicted in Figure 3.15(b). Nodes and edges are created for each pointer-related statement in the procedure following the patterns of Figure 3.14.

The next step is to determine aliases between fetch nodes and location nodes to obtain the resolved AFG, shown in Figure 3.15(c). In this example such aliases were added by considering no order, condition or kill information; hence determining aliases merely involves multiple instances of the simplest case in Figure 3.12. For example, alias edge (n_1, $\mathtt{v}$) is obtained due to operation edges $\mathtt{z} \xrightarrow{F} n_1$ and $\mathtt{z} \xrightarrow{A} \mathtt{v}$, respectively fetching from and assigning to variable `z`. Alias edge (n_2, $\mathtt{v}$) is derived similarly, due to edges $\mathtt{z} \xrightarrow{F} n_2$ and $\mathtt{z} \xrightarrow{A} \mathtt{v}$.

Node $\mathtt{z}_1$ in Figure 3.15(c) corresponds to the *initial value* of `z`, lazily created by the analysis to represent the environment initialization. This is represented by the assign edge labeled $A_\mathtt{z}$ that sinks at $\mathtt{z}_1$ (note that $\mathtt{z}_1$ is a location node). Accordingly, alias edges (n_1, $\mathtt{z}_1$) and (n_2, $\mathtt{z}_1$) are added to express the fact that dereferencing `z` also yields its initial value at function entry.

The final step is to produce the summary AFG depicted in Figure 3.15(d). Any information that would be invisible to a potential caller is deleted from the resolved AFG, including local and temporary computations, as well as all fetch nodes. Before deleting such nodes, we transfer their effects to nodes that will remain. In general, if an assignment is made to a fetch node n, and n can be an alias for a location node n', [3] the assignment is equivalent to one to n'. In Figure 3.15(d), four assign edges are derived. For instance, n_1 in Figure 3.15(c) is an alias for both `v` and $\mathtt{z}_1$, and there is an assignment edge $n_1 \xrightarrow{A} \mathtt{x}$. Therefore, assign edges $\mathtt{v} \xrightarrow{A} \mathtt{x}$ and $\mathtt{z}_1 \xrightarrow{A} \mathtt{x}$ are added in Figure 3.15(d). The same rationale follows for n_2, generating assign edges $\mathtt{v} \xrightarrow{A} \mathtt{y}$ and $\mathtt{z}_1 \xrightarrow{A} \mathtt{y}$. Finally, fetch nodes n_1 and n_2 are removed and node $\mathtt{z}_1$ is "demoted" to a fetch node to indicate the dereference of `z`; accordingly, the "initial value edge" labeled $A_\mathtt{z}$ in Figure 3.15(c) is demoted to a fetch edge $F_\mathtt{z}$ in Figure 3.15(d). The graph in Figure 3.15(d) summarizes the effects of the procedure on pointers, including the fact that variable `z` is dereferenced and the resulting value is assigned the addresses of both `x` and `y`.

The computation of alias edges in this example has, as its only premise, the fact that the locations fetched and assigned are aliases. For instance, the alias edge (n_1, $\mathtt{v}$) is created because `z` is a (vacuous) alias for `z`. Define the predicate *aliases* as

$$aliases(\alpha,\gamma) \;\Leftrightarrow\; \alpha = \gamma \vee al(\alpha) \cap al(\gamma) \neq \emptyset.$$

[3] n' proper location node or n' initial value node not yet demoted, see Section 5.1.

This says nodes α and γ are aliases for the same thing if they are identical or if they are aliases for some common location node. This relationship is a flow-insensitive approximation of the exact *affects*(σ_A, σ_F), so [ALIAS] can be approximated by

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad aliases(\alpha, \gamma)}{al(\beta) \subseteq al(n)} \qquad \text{[ORIGIN-ALIAS]}$$

Because this rule is recursive (the premise refers to the *aliases* relation, which depends on *al*, which is to be computed), finding the minimal resolved AFG requires computing a fixed point. Our implementation uses the usual worklist algorithm that iterates to convergence (Chapter 5).

Figure 3.16 illustrates the [ORIGIN-ALIAS] rule graphically. Existing alias relationships are shown with thin dashed lines and the rule generates the edges in bold. Figure 3.16(d) is the most generic case; Figure 3.16(a) is the special case when α is an arbitrary node and $\beta = \mathtt{y}$; Figure 3.16(b) is the special case when $\alpha = n_0$, $\gamma = \mathtt{x}$ and $\beta = \mathtt{y}$; and Figure 3.16(c) is the special case when α is an arbitrary node and β is a fetch node n_1; assume $al(n_1) = \{\mathtt{y}\}$ for this figure.

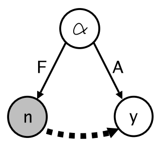
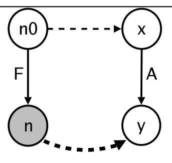
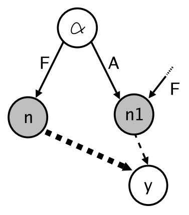
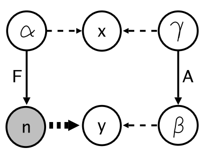
	Fetch from a node	Fetch from a fetch node
$al(\mathtt{y}) = \{\mathtt{y}\} \subseteq al(n)$ in (a) and (b)	 (a)	 (b)
$al(n_1) = \{\mathtt{y}\} \subseteq al(n)$ in (c)	 (c)	 (d)

Figure 3.16: Four cases of applying the [ORIGIN-ALIAS] rule. A node is fetched in (a) and (c) ($\alpha = \gamma$); the result of a fetch is itself fetched in (b); (d) is the general case.

3.2.5 Unfolding the Partial Order Axis

The [ORIGIN-ALIAS] rule ignores the fact that assignments and fetches in a program happen in sequence: later assignments cannot be seen by earlier fetches. This leads to overly approximate results, arriving at more aliases than actually possible. For example, in Figure 3.15, the first dereference of `z` can only see the initial value of `z`, since it is the first statement in the procedure. Thus, the only possible alias edge from n_1 is to $\texttt{z}_1$. The [ORIGIN-ALIAS] rule also introduced the alias edge from n_1 to `v`.

This section refines the *affects* relation by considering the (partial) order of the statements in the program[4]. This gives a different answer to the question "which assignments should a fetch match?"—it reduces the number of such matches resulting in more accurate analysis results.

Figure 3.17 illustrates the matching of fetch and assign operations when considering the statements partial order. Figure 3.17(a) is a fragment of the control-flow graph of a program showing some assignments and fetches. For simplicity, assume that these operations are performed on the same program variable, so that *aliases* is always true. For the sake of presentation *only*, dashed arrows have been added to the control-flow graph—a dashed arrow from a fetch F_i to an assign A_j indicates that F_i resolves to A_j. Figures 3.17(a) and 3.17(b) represent the analysis results when the [ORIGIN-ALIAS] rule is applied.

Figure 3.17(c) represents the same program fragment of Figure 3.17(a), but the dashed arrows indicate the matching of fetches and assignments *when execution ordering is considered*. For instance, note that fetch F_i will only resolve to assignments (omitted) occurring before the code fragment shown, and assignment A_n does not affect any of the fetches depicted since it happens later.

Let us abuse our notation slightly and define a binary relation "$\sqsubseteq$" between two operations. Namely, $\sigma_A \sqsubseteq \sigma_F$ holds in case the assign σ_A occurs before fetch σ_F. Then, the refined *affects* relation which unfolds the partial order axis of our analysis space is defined as

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad aliases(\alpha,\gamma) \qquad \sigma_A \not\sqsupseteq \sigma_F}{al(\beta) \subseteq al(n)} \quad \text{[PO-ALIAS]}$$

Informally, in addition to α and γ being aliases, the [PO-ALIAS] rule also requires that the assign σ_A does not occur after the fetch σ_F in the execution order. The idea of "unfolding" the

[4]Formally, the partial order is w.r.t. the atomic operations of assign and fetch.

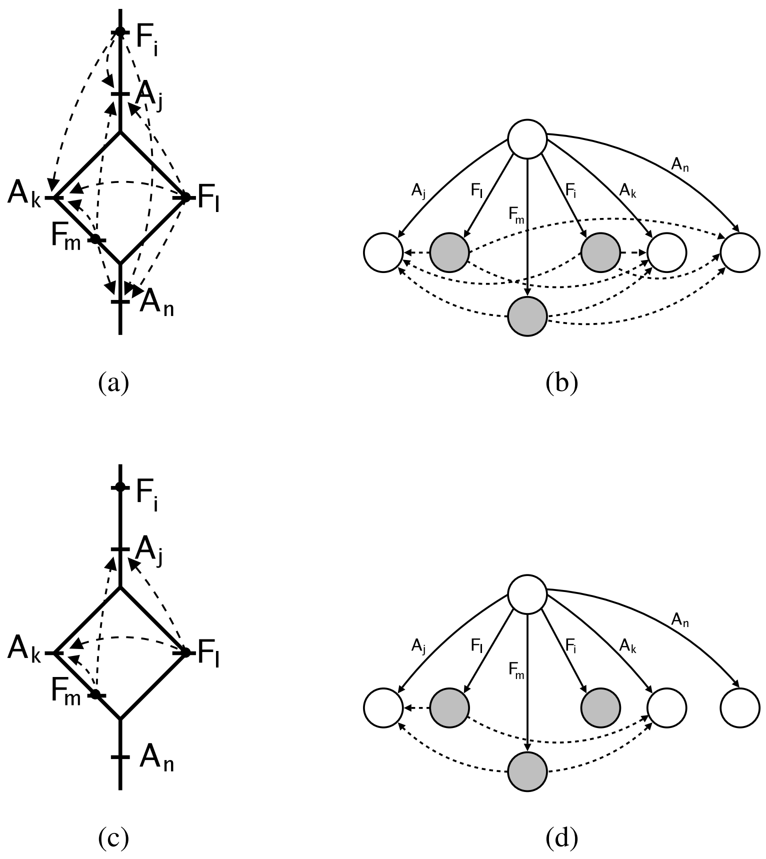


Figure 3.17: Ordering information reduces the number of alias edges. (a) and (c) are fragments of the control flow graph of a program. (b) The resolved AFG using the [ORIGIN-ALIAS] inference rule. (d) The resolved AFG when the [PO-ALIAS] rule is applied.

horizontal axis is that different approximations to the execution order can be given: from a "flat" set of statements, to a quick-to-compute total order, to a complex partial order taking into account conditionals and data, etc. Each of these approximations give a different answer regarding the relative order of a pair of statements—the tighter the approximation, the more accurate the results. Chapter 4 examines some approximations that lead to interesting algorithms for pointer analysis.

Figure 3.17(d) shows the result of applying the [PO-ALIAS] inference rule to the control-flow graph fragment, where the program's "true" partial order is considered. E.g., F_m matches A_k and A_j because both assignments precede it; F_i does not match any of A_j, A_k, and A_n because all of these assignments happen after F_i. Figure 3.2(h) showed the placement of this point in our analysis space.

Note that the [PO-ALIAS] rule does not require the fetch to occur *after* the assignment, or $\sigma_A \sqsubseteq \sigma_F$, because a given pair of statements might be not comparable under the partial order. This is the case with F_l and A_k in Figure 3.17(c). In order to obtain an answer from the "$\sqsubseteq$" relation, [PO-ALIAS] requires instead that $\sigma_A \not\sqsupseteq \sigma_F$, which is true in case σ_A and σ_F are not comparable. As mentioned in Section 3.1, we can interpret the [PO-ALIAS] rule by replacing conditional expressions with fictitious parallel constructs—from this point of view, there is no well-defined order between F_l and A_k in our example. To be able to fully disassociate F_l from A_k we will need to consider the conditions under which they occur (Section 3.2.6).

Also note that the [ORIGIN-ALIAS] rule is an (over)approximation of the [PO-ALIAS] rule where "$\sigma_A \not\sqsupseteq \sigma_F$" is proclaimed true for any given pair σ_A and σ_F. Intuitively, this means that the fixed-point obtained by [PO-ALIAS] is smaller than that of [ORIGIN-ALIAS]. Given an initial AFG, the iterative process of adding alias edges until convergence terminates earlier for [PO-ALIAS], which means a more precise, and in some cases *more efficient*, analysis.

To compute the relation "$\not\sqsupseteq$" for the [PO-ALIAS] rule, statement ordering information is obtained from the flow graph data structure presented in Section 3.2.1. There are basically three ways to implement this: (1) access the flow graph and the AFG data structures simultaneously during analysis, (2) augment the analysis framework with an additional data structure whose sole purpose is to keep order information, or (3) augment the AFG such that it is the only data structure needed after a setup phase. We discuss the latter in Chapter 4.

3.2.6 Unfolding the Conditions Axis

In this subsection, we enhance our analysis space by using program conditions as guards stating under what conditions a fetch or assign occurs in the AFG. Guards contribute to the precision of the analysis; for example, a fetch is not matched to an assign if they have guards that cannot simultaneously hold. Guards also specify under what conditions an *alias* occurs; in this case, the guard is associated with the alias edge, and is derived from the corresponding pair(s) of assign and fetch edges responsible for the aliasing.

Consider the example described in the previous section. Figure 3.18(a) shows the control-flow graph fragment used earlier, with the difference that program conditions have been taken into account *in addition* to considering the statements partial order. Fetch F_l is not affected by assign A_k

because the two operations are independent. The corresponding point in the analysis space was depicted in Figure 3.2(j).

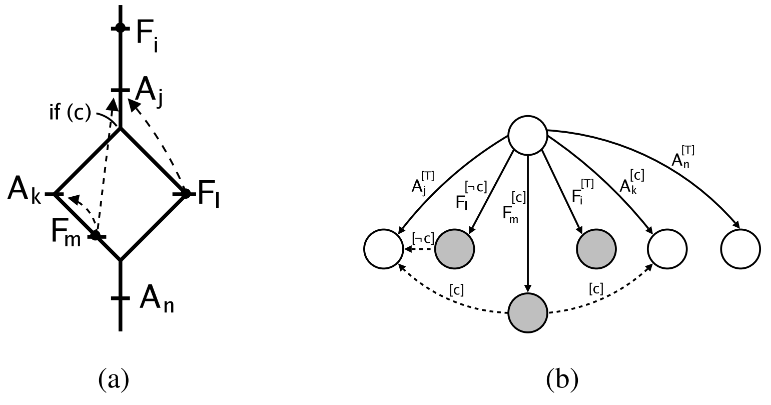


Figure 3.18: Program predicates in the form of guards reduces the number of alias edges even further. (a) The fragment of the control flow graph used in Section 3.2.5, with the difference that conditions are taken into account in addition to the program's partial order, (b) the resulting resolved AFG.

Assume that the left side of the diamond corresponds to the true branch. Note in particular the alias edge in Figure 3.18(b) derived by matching F_l and A_j. Although A_j occurs unconditionally, the guard for the alias edge is $\neg c$ because F_l only executes if c is false. Guards on alias edges are derived from guards in the operation edges, and they state under what conditions the alias will occur.

If the program's partial order is ignored, but not the guards, we move to the analysis point of Figure 3.2(l), and the analysis results are as shown in Figure 3.19.

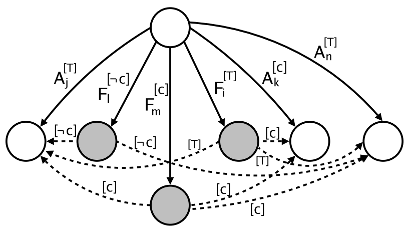


Figure 3.19: The analysis solution when considering guards but not the program's partial order.

The inference rule that includes partial order information *and* guards can be written as:

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad aliases(\alpha,\gamma) \qquad \sigma_A \not\sqsupseteq \sigma_F \qquad consistent(\sigma_A,\sigma_F)}{\forall \mathsf{y} \in al(\beta)\,,\ \ g_{\sigma_F} \wedge \mathscr{C}(\alpha,\gamma) \wedge g_{\sigma_A} \wedge \mathscr{C}(\beta,\mathsf{y}) \subseteq \mathscr{C}(n,\mathsf{y})}$$

[GUARDED-ALIAS]

where $consistent(\sigma_A, \sigma_F)$ means that the assignment σ_A occurs under a predicate which is consistent with the guard of σ_F. More precisely, the check for consistency also includes the predicates on the alias edges connecting α and γ, and β and y. The illustration in Figure 3.20 helps explaining the elements of this inference rule.

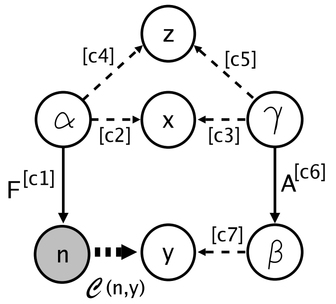


Figure 3.20: To add an alias edge, the guard $\mathscr{C}(n,\mathsf{y})$ must not evaluate to false.

This figure shows a fetch edge under a guard $g_F = c_1$ and an assign edge subject to predicate $g_A = c_6$. The source nodes for each edge, α and γ, are aliases for each other under the guard $c_2 \wedge c_3 \vee c_4 \wedge c_5$. This is expressed in the inference rule by the notation $\mathscr{C}(\alpha,\gamma)$. Similarly, the condition for which β is an alias for y, $\mathscr{C}(\beta,\mathsf{y})$, is represented by c_7. Thus, the predicate for which n becomes an alias for y is expressed by $g_F \wedge \mathscr{C}(\alpha,\gamma) \wedge g_A \wedge \mathscr{C}(\beta,\mathsf{y})$, or $c_1 \wedge (c_2 \wedge c_3 \vee c_4 \wedge c_5) \wedge c_6 \wedge c_7$. If there is more than one possible way that n can alias with y, the resulting guard contains a disjunction of these different aliasing possibilities. This is represented by the "$\subseteq$" in the rule.

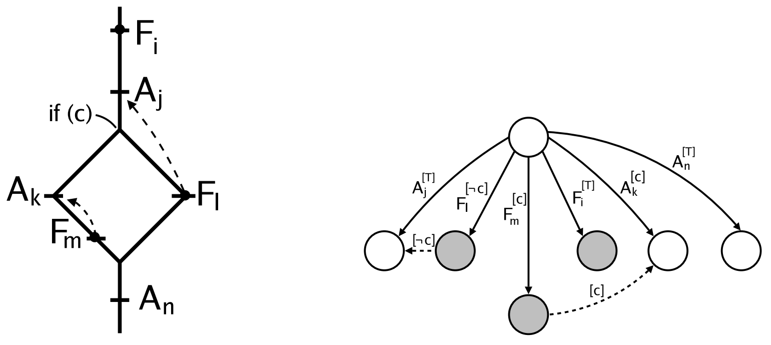
Note that an alias edge is added by the [GUARDED-ALIAS] inference rule only if the expression $g_{\sigma_F} \wedge \mathscr{C}(\alpha,\gamma) \wedge g_{\sigma_A} \wedge \mathscr{C}(\beta,\mathsf{y})$ does not evaluate to false. A false condition means no aliasing. In other words, the rule adds alias edges to the graph by assigning the conditions under which such aliasings occur. Also note that $consistent(\sigma_A, \sigma_F)$ is obtained by computing $g_{\sigma_F} \wedge \mathscr{C}(\alpha,\gamma) \wedge g_{\sigma_A}$.

As with the program's partial order, different approximations can be given to program predicates as well as to the results of the above Boolean operations. A widening [22] operator can be defined such that, e.g., the "or" of two predicates is widened to true. This is one of the mechanisms we use in our implementation, controlled by a parameter.

3.2.7 The kill Axis

This section presents an integrated analysis model, incorporating the kill axis with the other two coordinates, completing the analysis space introduced in Section 3.1. The model presented here is a complex one, in which to determine whether a pointer assignment has been killed by the time the pointer is read, the analysis has to rely on the predicates in the code as well as statement ordering. From this perspective, the kill axis is not orthogonal to the other two—indeed, without ordering information we cannot ascertain whether a statement σ_1 kills the dataflow facts generated by another statement σ_2. In Chapter 4 a simplified (and less precise) modeling for strong updates is presented in which only execution order is considered.

The most conservative answer is to say that no assignment is ever killed. This is the fallback answer in case ordering information is unavailable or imprecise. However, the ability to rule out some of the assignments with respect to a given fetch increases the precision of the analysis. Continuing with our running example, Figures 3.21(a) and 3.21(b) show the fetch/assign matchings and alias edges generated in case the analysis is able to determine that assignment A_j is killed prior to fetch F_m. Compare this solution with the graph in Figure 3.17(b).



(a) (b)

Figure 3.21: Being able to determine kill information reduces the number of aliases even further In Figure 3.18(a) F_m resolved to A_j.

Traditionally, strong updates are intimately related to program-point specific analyses, although a summary analysis can model such destructive updates and simply consider the set of relations at the exit of the procedure as the function summary. The latter is the approach described here.

Let the initial AFG G for a procedure P to have N nodes and M assignments. Recall from Section 3.2 that the relation *al* for a given node α was the set of nodes that α could be an alias for. Each element of $al(\alpha)$ was depicted as an alias edge. In the generalized case presented here, $al(\alpha)$ is a vector of size N, where each cell of this vector corresponds to a distinct node of G. The contents of a cell in $al(\alpha)$ are the control predicates under which α represents the node corresponding to the cell. Thus, a cell containing f (false) means that α is not an alias for that node. Of course, the vector $al(\alpha)$ when α is a location node has a single cell containing t (true) and all the remaining cells containing f, since a location node can only be an alias for itself.

Moreover, let each assign edge $\mathsf{A}_i \in G$ to have a NxM characteristic matrix K_i. Each row in this matrix corresponds to a distinct node of G, and each column corresponds to one of the M assignments. The *i-th* column on matrix K_i is the very assignment represented by edge A_i, and it is treated specially.

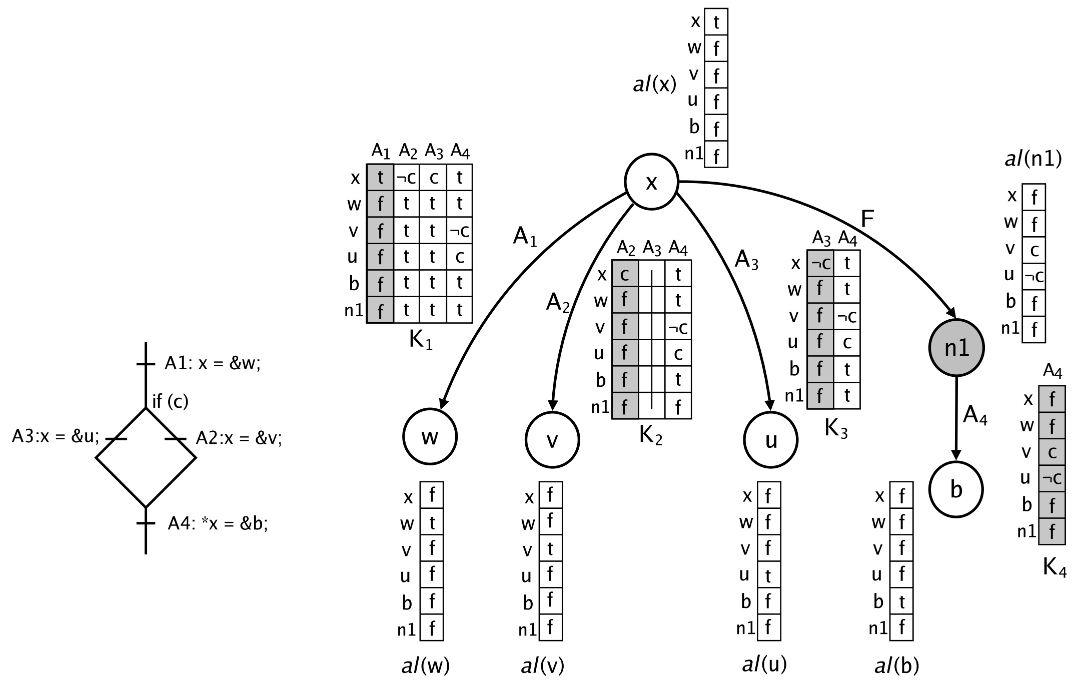
To illustrate, Figure 3.22(b) shows the resolved AFG for the code in Figure 3.22(a). Instead of adding alias edges to the AFG, the resolution phase calculates the matrices K_i and vectors *al*, and is composed by two assign rules and one fetch rule, as explained later.

The indices on the assign edges are given *w.r.t.* the order on the statements in the code, which form the lattice in Figure 3.22(a). An arbitrary choice is made between true and false branches. Note that each matrix K_i actually starts with the *i-th* column. This is because all assignments l such that $l < i$ cannot affect assignment i, and thus the irrelevant columns can be discarded.

The interpretation for the matrices K_i is as follows. The special *i-th* column (shaded in each matrix) contains, for each row j, the conditions that must hold for assignment A_i to assign to node j. Although an assign edge that has a location node as its source can only assign to that location, an assign edge with a fetch node as its source can represent more than one actual assignment to memory. In that case, each distinct assignment may occur under a particular control condition.

For example, the leftmost, topmost cell for matrix K_1 in Figure 3.22(b) says that variable `x` is unconditionally assigned at statement A_1. Of course, no other location can be assigned by A_1, and the remaining cells of the first column are all false.

The remaining columns of K_1 refer to the "survival" of assignment A_1. Precisely, each column $i > 1$ refers to the conditions that must hold for A_1 to survive assignment A_i, $\forall\, \mathsf{A}_i \sqsupset A_1$. For example, because A_2 executes only if c is true, assignment A_1 is *not* clobbered by assignment A_2 only if the



(a) (b)

Figure 3.22: (a) A code fragment. (b) Its resolved AFG.

execution takes the other branch, i.e., only if c is false. Thus, the matrix cell on row x, column A_2 on matrix K_1 is marked $\neg c$. A similar situation happens with A_3, and the corresponding cell (row x, column A_3) is marked with c. Assignment A_4 does not write to variable x, and thus the (vacuous) condition for A_1 to survive A_4 is t.

Note that column A_2 on matrix K_1 is the negation of column A_2 from matrix K_2; the same happens with columns A_3 and A_4 on matrix K_1 with respect to matrices K_3 and K_4, respectively.

The interesting case occurs at matrix K_4, which corresponds to assignment A_4 whose source node is n_1. Note that A_4 assigns to variable v in case c is true, and it writes to u in case c is false—n_1 is an alias for v if c but an alias for u if $\neg c$. In the graph of Figure 3.22(b), the assign edge labeled A_4 represents both assignments. Such information is extracted from $al(n_1)$, which is used in the computation of K_4. The formal rules and definitions to derive the above facts are shown in Figure 3.23.

The notation $\sigma_i : \alpha \xrightarrow{A_i^{K_i}} \beta$ means an assignment statement σ_i, represented in the AFG by assign edge A_i from node α to node β, whose characteristic matrix is K_i. The symbol $K_i[\sigma_i]$ in the inference

$$\frac{\sigma_i : \alpha \xrightarrow{A_i^{K_i}} \beta}{K_i[\sigma_i] = guard(\sigma_i) \wedge al(\alpha)} \quad [\text{ASSIGN}]$$

(a)

$$\frac{\sigma_i : \alpha \xrightarrow{A_i^{K_i}} \beta \sqsubset \sigma_j : \gamma \xrightarrow{A_j^{K_j}} \delta}{K_i[\sigma_j] = \neg K_j[\sigma_j]} \quad [\text{KILL}]$$

(b)

$$\frac{\sigma_i : \alpha \xrightarrow{A_i^{K_i}} \beta \not\sqsupseteq \sigma_j : \gamma \xrightarrow{F} n_1}{[K_{i\,\sigma_i}^{\sigma_j}.al(\gamma)] \wedge al(\beta) \subseteq al(n_1)} \quad [\text{FETCH}]$$

(c)

Figure 3.23: (a) The assign rule (b) The kill rule (c) The fetch rule.

rules refers to the column labeled σ_i in matrix K_i, and the expression $guard(\sigma_i) \wedge al(\alpha)$ returns a vector of size N in which $al(\alpha)$ is *and*'ed with the (scalar) guard under which σ_i occurs. We slightly abuse the notation and use "σ_i" to mean both an assignment statement and the label for a column in a given matrix—the meaning should be clear from the context.

The inference rule of Figure 3.23(a) is used to compute the special column σ_i in matrix K_i for all assignments σ_i. The straightforward interpretation for this rule is as follows. To compute the conditions under which $\sigma_i : \alpha \xrightarrow{A} \beta$ assigns to a node γ in the graph, we need to compute the conditions for which α is an alias for γ and combine this result with the guard for σ_i.

Writing $\sigma_i : \alpha \xrightarrow{A_i^{K_i}} \beta \sqsubset \sigma_j : \gamma \xrightarrow{A_j^{K_j}} \delta$ means that assignment σ_i with matrix K_i is comparable and smaller, *w.r.t.* the program's partial order, than assignment σ_j with matrix K_j. The notation $K_i[\sigma_j] = \neg K_j[\sigma_j]$ means the negated form of column σ_j of matrix K_j is copied to column σ_j of K_i.

The inference rule of Figure 3.23(b) is used to compute the predicates for which an earlier assignment σ_i is *not* clobbered by a later assignment σ_j (thus the negation in the conclusion of the rule). This rule can only be applied if σ_i and σ_j are comparable under the program's partial order. Basically, this rule computes the remaining, non-special, columns in all the matrices, such as columns A_2, A_3 and A_4 in matrix K_1 of Figure 3.22(b). Note that column A_3 in matrix K_2 has been invalidated, given that assignments σ_2 and σ_3 are independent and not comparable.

In the rule of Figure 3.23(c), ${K_i}_{\sigma_i}^{\sigma_j}$ is defined as follows:

$${K_i}_{\sigma_i}^{\sigma_j} = \bigwedge_{\sigma_i \sqsubseteq \sigma_k \sqsubseteq \sigma_j} K_i[\sigma_k]$$

and it means *and*'ing together all columns σ_k from labels σ_i through σ_j on matrix K_i such that σ_k is comparable with both σ_i and σ_j. The result of this operation is a single column (a vector of size N).

The "multiplication" operator in $[{K_i}_{\sigma_i}^{\sigma_j}.al(\gamma)]$ produces a single "scalar formula" that is then *and*'ed with every cell in $al(\beta)$. This results in a vector of size N that is *or*'ed with every cell in $al(n_1)$, generating the updated $al(n_1)$. The symbol $\not\sqsupseteq$ means the fetch has to occur *after* the assignment.

The rule in Figure 3.23(c) matches a fetch edge only to those assignments that reach the fetch. Assignments that are killed before reaching the fetch are implicitly excluded as a result of computing ${K_i}_{\sigma_i}^{\sigma_j}$. For example, the assignment `x=&w` is killed in both branches of the code in Figure 3.22(a). Therefore, fetching `x` at statement `*x=&b` cannot return `w`, but only $\{$`v`,`u`$\}$. Indeed, by applying the inference rule of Figure 3.23(c) with assignment $\mathtt{x} \xrightarrow{A_1^{K_1}} \mathtt{w}$ and fetch $\mathtt{x} \xrightarrow{F} n_1$ we get:

$$\begin{array}{ccccccccc}
 & K_1[A_1] & & K_1[A_2] & & K_1[A_3] & & \\
 & \mathsf{t} & \wedge & \neg\mathsf{c} & \wedge & \mathsf{c} & & \mathsf{f} \\
 & \mathsf{f} & \wedge & \mathsf{t} & \wedge & \mathsf{t} & & \mathsf{f} \\
{K_1}_{A_1}^{A_3} = & \mathsf{f} & \wedge & \mathsf{t} & \wedge & \mathsf{t} & = & \mathsf{f} \\
 & \mathsf{f} & \wedge & \mathsf{t} & \wedge & \mathsf{t} & & \mathsf{f} \\
 & \mathsf{f} & \wedge & \mathsf{t} & \wedge & \mathsf{t} & & \mathsf{f} \\
 & \mathsf{f} & \wedge & \mathsf{t} & \wedge & \mathsf{t} & & \mathsf{f}
\end{array}$$

Therefore,

$$\begin{array}{cccccccccc}
 & {K_1}_{A_1}^{A_3} & & al(\mathsf{x}) & & & & al(\mathsf{w}) & & al(\mathsf{n}_1)[\mathsf{w}] \\
\mathsf{x} & \mathsf{f} & & \mathsf{t} & & & & \mathsf{f} & & \mathsf{f} \\
\mathsf{w} & \mathsf{f} & & \mathsf{f} & & & & \mathsf{t} & & \mathsf{f} \\
\mathsf{v} & \mathsf{f} & . & \mathsf{f} & = & \mathsf{f},\mathsf{f} & \wedge & \mathsf{f} & = & \mathsf{f} \\
\mathsf{u} & \mathsf{f} & & \mathsf{f} & & & & \mathsf{f} & & \mathsf{f} \\
\mathsf{b} & \mathsf{f} & & \mathsf{f} & & & & \mathsf{f} & & \mathsf{f} \\
\mathsf{n}_1 & \mathsf{f} & & \mathsf{f} & & & & \mathsf{f} & & \mathsf{f}
\end{array}$$

Which means that `*x` is not an alias for `w` after the control-flow join (fetch edge $(\mathsf{x},\mathsf{n}_1)$ occurs after the confluence of both branches). The notation $al(\mathsf{n}_1)[\mathsf{w}]$ is used to indicate the partial result for $al(n_1)$ obtained when applying the inference rule of Figure 3.23(c) with $\beta = \mathtt{w}$. In other words, it indicates the "contribution" of $al(\mathtt{w})$ for computing $al(n_1)$. Note the solution obtained for $al(n_1)$ can be made program-point specific although a single graph is used.

Also, by matching fetch $\mathtt{x} \xrightarrow{F} n_1$ with assignments A_2 and A_3 we get:

$$\begin{array}{cccc}
 & al(\mathsf{n}_1)[\mathsf{v}] & & al(\mathsf{n}_1)[\mathsf{u}] \\
\mathsf{x} & \mathsf{f} & & \mathsf{f} \\
\mathsf{w} & \mathsf{f} & & \mathsf{f} \\
\mathsf{v} & \mathsf{c} & \text{and} & \mathsf{f} \\
\mathsf{u} & \mathsf{f} & & \neg\mathsf{c} \\
\mathsf{b} & \mathsf{f} & & \mathsf{f} \\
\mathsf{n}_1 & \mathsf{f} & & \mathsf{f}
\end{array}$$

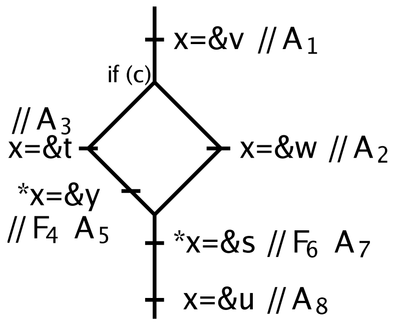
respectively. Thus, $al(n_1)$ is:

$$\begin{array}{cccccccc}
 & al(\mathsf{n}_1)[\mathsf{w}] & & al(\mathsf{n}_1)[\mathsf{v}] & & al(\mathsf{n}_1)[\mathsf{u}] & & al(\mathsf{n}_1) \\
\mathsf{x} & \mathsf{f} & \vee & \mathsf{f} & \vee & \mathsf{f} & & \mathsf{f} \\
\mathsf{w} & \mathsf{f} & \vee & \mathsf{f} & \vee & \mathsf{f} & & \mathsf{f} \\
\mathsf{v} & \mathsf{f} & \vee & \mathsf{c} & \vee & \mathsf{f} & = & \mathsf{c} \\
\mathsf{u} & \mathsf{f} & \vee & \mathsf{f} & \vee & \neg\mathsf{c} & & \neg\mathsf{c} \\
\mathsf{b} & \mathsf{f} & \vee & \mathsf{f} & \vee & \mathsf{f} & & \mathsf{f} \\
\mathsf{n}_1 & \mathsf{f} & \vee & \mathsf{f} & \vee & \mathsf{f} & & \mathsf{f}
\end{array}$$

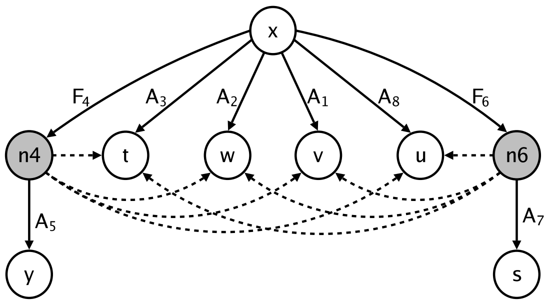
which says that fetching `x` after the control-flow join can only lead to either `v` or `u`, not `w`.

3.2.8 Putting the pieces together

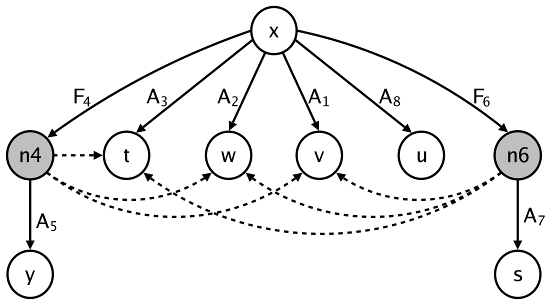
To put the pieces together, Figure 3.24 shows a code fragment and four different sets of alias edges added to the AFG; each corresponds to the solution for a particular analysis variation. In



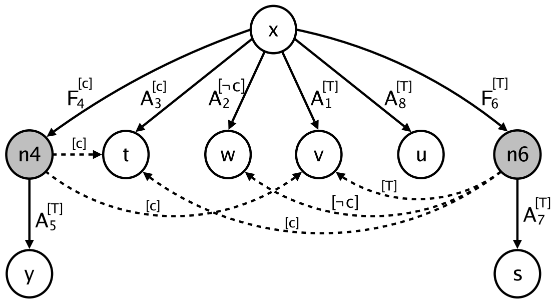
(a)



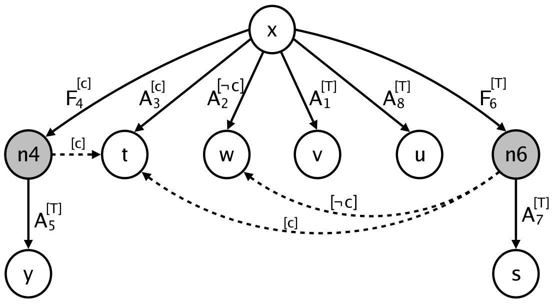
(b)



(c)

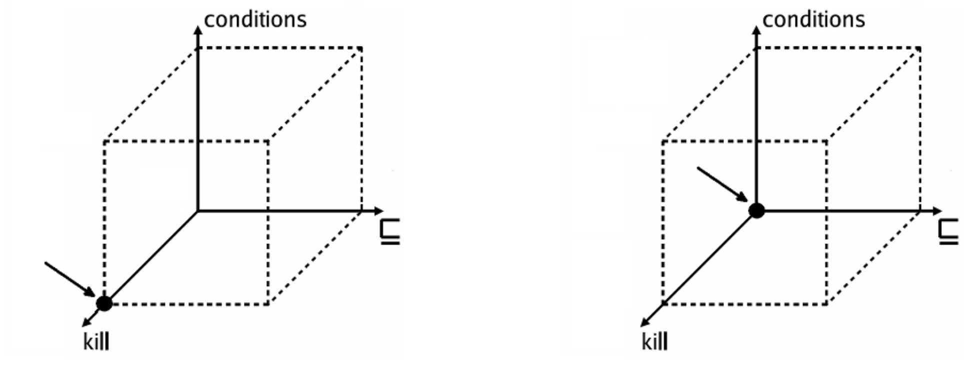


(d)



(e)

Figure 3.24: (a) A code fragment. The analysis results for (b) origin, (c) when partial order is considered, (d) including conditions, (e) kill information is added.



(a) (b)

Figure 3.25: The final solution for analysis variations (a) and (b) are equivalent, since to avoid the fall back answer regarding strong updates it is imperative that statement ordering be considered.

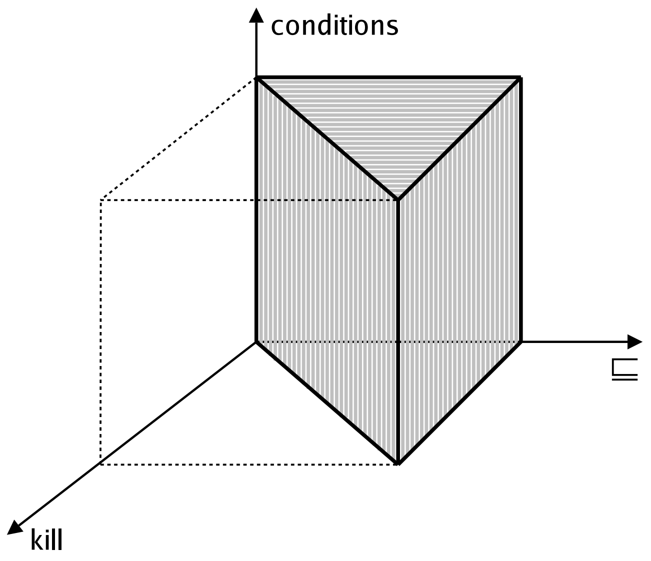


Figure 3.26: Space of solutions within the analysis space.

Figure 3.24(b) the [ORIGIN-ALIAS] rule is used. Many spurious alias edges are derived. In Figure 3.24(c), the [PO-ALIAS] rule is applied—because statement ordering is considered, a more precise result is obtained. In Figure 3.24(d), guards are added leading to more precision, and in Figure 3.24(e) all the elements, including strong updates, are taken into account. Compare that result with the graph in (b).

Space of solutions

A final observation on the analysis space proposed in this chapter is that the space of *distinct solutions* is a subset of the space of analysis. As mentioned earlier, the unfolding of certain dimensions may depend on the specific values attributed to others. For instance, modeling strong updates requires statement ordering. This means that the final *solution* for the analysis variations in Figures 3.25(a) and 3.25(b) map to the same point in the space of solutions. Such space of solutions is represented by the shaded solid in Figure 3.26.

Note this should not be taken for the space of analysis. Namely, a particular point in the latter indicates which characteristics from the program's semantics are being considered and which are ignored. But the solution obtained by two of these points may be equivalent, such as Figures 3.25(a) and 3.25(b).

Chapter 4

Investigating some sweet spots

This chapter investigates some analysis variations in our analysis space that yield significant benefits over traditional pointer analysis. In addition to being novel, these sweet-spots provide new insights into pointer analysis as a whole.

4.1 Flow-aware analysis

Our 3-D analysis space and AFG data structure enable a variety of new pointer analysis algorithms. One of them is what we call *flow-aware* analysis [11]. The basic idea is to approximate the procedure's execution by a total order on the statements—i.e., to provide a total order approximation to the [PO-ALIAS] inference rule of Section 3.2.5. This is a quick-to-compute sound approximation that can significantly improve *both* the precision and the efficiency of the analysis over the [ORIGIN-ALIAS] rule (and as a consequence over Andersen-style flow-insensitive analysis). Such a symbiosis is rare in static analysis; most often gains in one aspect involve costs in another.

Totally ordering the statements in a procedure means ignoring mutual exclusivity between conditional branches, data-dependent infeasible paths, etc., but it is inexpensive and produces a significant improvement. To illustrate, Figure 4.1 shows the matchings between fetches and assigns when considering (a) no statement ordering, (b) the "true" partial order in the code fragment, and (c) a total order approximation where the code for the right branch comes before the left branch. The total order constrains which assignments are visible to each read: a fetch σ_F resolves to an assign σ_A only if the assign occurs strictly before the fetch. This reduces the number of spurious aliases.

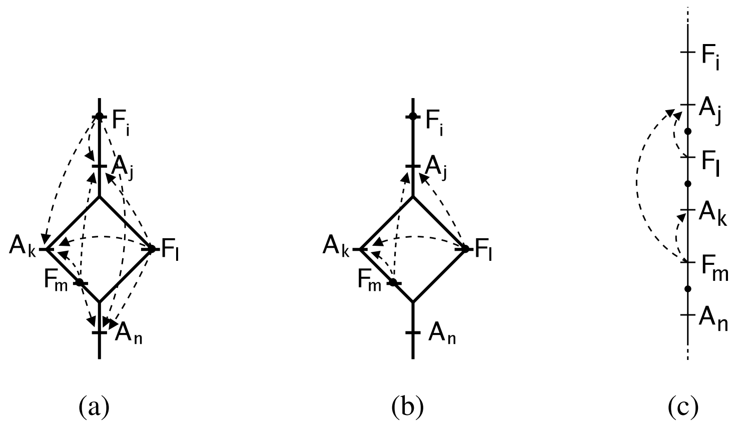


Figure 4.1: Matchings obtained when (a) no statement ordering is considered (b) the "true" partial order is used (c) a total order approximation where the right branch comes before the left branch.

A total order approximation has also the benefit of partially disassociating the two branches of an `if` statement. E.g., in Figure 4.1(c) the assignment A_k is not visible to fetch F_l, a correct behavior. Of course, this is affected by which branch comes first in the total order—the latter branch will always see the earlier one.

The control-flow graph fragments in Figure 4.2 further illustrate the motivation for flow-aware analysis. Figure 4.2(a) is the simplest case: the assignment A_j runs before the fetch F_i, so A_j can affect F_i, i.e., the relation *affects*(A_j, F_i) holds. However, a fetch that runs before an assignment, such as in Figure 4.2(b), cannot be affected by the assignment. A flow-insensitive analysis treats these two cases identically, whereas the flow-aware analysis does not build an alias edge in the second case.

Conditionals add complexity. In Figure 4.2(c), fetch F_i should resolve to assign A_j since the latter occurs strictly before the former, whereas *affects*(A_k, F_i) is false. Finally, *affects*(A_m, F_i) and *affects*(A_m, F_l) are both false.

The situation in Figure 4.2(d) is slightly different. Although F_l occurs (non-strictly) after A_m, A_m cannot affect F_l because they are mutually exclusive: the expression c controls both conditionals. However, A_k does affect F_l because A_k comes before F_l along a feasible path. As mentioned in Chapter 3, *affects* is always an approximation since path-feasibility is undecidable in general.

Figure 4.2(e) shows one possible total order for the control-flow graph of Figure 4.2(d): we numbered statements in the left branch of each conditional before its right branch.

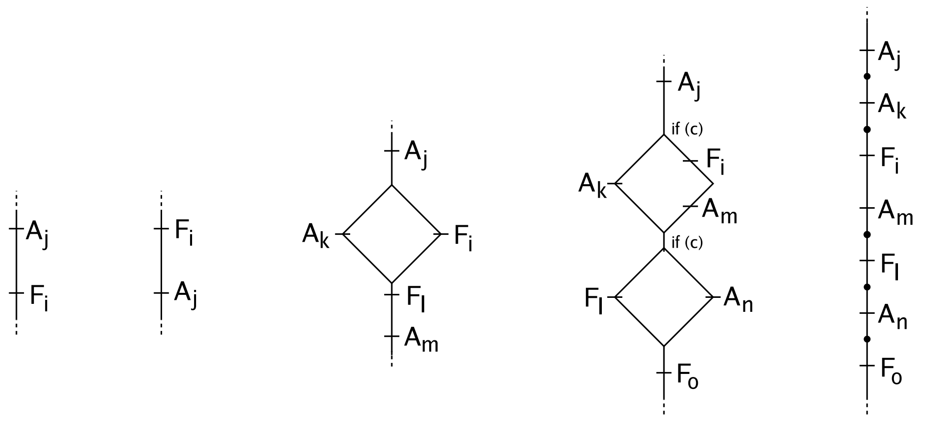


Figure 4.2: Motivation for flow-aware analysis. (a) An assignment before a fetch can affect the fetch, but (b) an assignment after a fetch cannot. The presence of conditionals (c, d) further constrain which assignments are visible to each fetch. A possible approximation for the execution order of (d) is total order of (e).

Let $affects_0(\sigma_A,\sigma_F)$ be true when the assignment σ_A occurs before the fetch σ_F in the total order. This approximation can produce spurious results. For example, in the linearization of Figure 4.2(d) in Figure 4.2(e), $affects_0(A_k,F_i)$ and $affects_0(A_m,F_l)$ are true, yet $affects(A_k,F_i)$ and $affects(A_m,F_l)$, the exact relations in Figure 4.2(d), are false. Thus, $affects_0$ allows a fetch edge to resolve to extra assignments, but it is a sound solution with substantially improved precision over flow-insensitive analysis. The [ALIAS] rule for flow-aware analysis becomes

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad aliases(\alpha,\gamma) \qquad affects_0(\sigma_A,\sigma_F)}{al(\beta) \subseteq al(n)} \; [\text{FLOW-AWARE}]$$

This rule leads to faster convergence with fewer alias edges when compared to [ORIGIN-ALIAS].

Figure 4.3 places the flow-aware analysis in our analysis space. Any pair of statements becomes comparable under the imposed total order, including those incomparable under the true partial order; this implies the value we assigned to the horizontal axis. Some value is also given to the vertical axis because of the partial separation between `if`-branches discussed above.

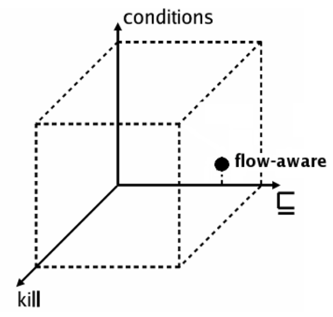


Figure 4.3: The flow-aware analysis uses a total order approximation, not the program's "true" partial order, and it has the benefit of partially disassociating operations that occur in separate branches of a conditional—thus we assign some value to the vertical axis.

4.1.1 Implementation

Our AFG data structure is ideal for flow-aware analysis. To implement such analysis, we simply augment each edge σ in the AFG with an index $rank(\sigma)$ from a topological sort of the statements in the procedure (recall each procedure is acyclic after our pre-processing). Then $affects_0(\sigma_A, \sigma_F)$ can be implemented as $rank(\sigma_A) < rank(\sigma_F)$. We write these ranks as subscripts on F's and A's.

As an example, consider the code in Figure 4.4(a); the total order is given by the text in comments. The initial AFG is shown in Figure 4.4(b), and Figure 4.4(c) shows the resolved AFG obtained by applying the [FLOW-AWARE] rule. Fetch F_2 only matches assign A_1, given that A_4 occurs after F_2 and therefore does not affect it. As a consequence, when F_5 resolves to A_3 only one alias edge is added. The equivalent points-to set obtained from Figure 4.4(c) is shown in Figure 4.4(d). This figure shows the flow-aware and traditional flow-insensitive points-to sets superimposed—the two edges in bold are spurious relations avoided by the flow-aware analysis.

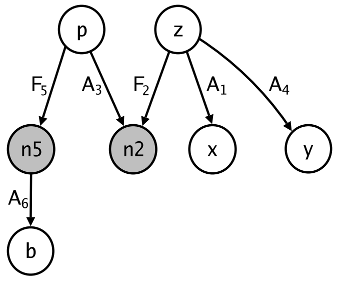
4.1.2 Interprocedural order propagation

This section illustrates how flow-aware analysis manipulates ordering information across procedures and how spurious pointer relations that would otherwise span multiple functions are avoided. For simplicity, initial values are ignored here.

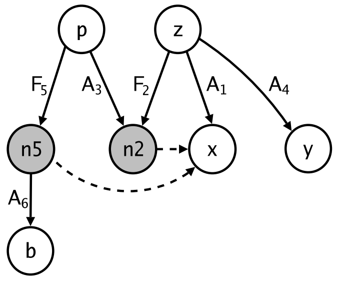
Performing flow-aware analysis across procedure calls requires us to order statements on both sides of a call site. To get this right, we "shift" the indices of the callee by the maximum index that occurs in the caller before the call site, then increase the indices in the caller that appear after the

```
foo()
{
  z = &x;      // A1
  if (c) {
    p = z;     // F2 A3
  }
  z = &y;      // A4
  if (c) {
   *p = &b;    // F5 A6
  }
}
```

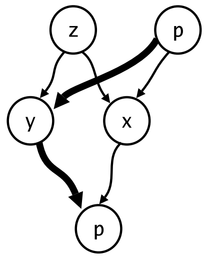
(a)



(b)



(c)



(d)

Figure 4.4: (a) A function and (b) its initial AFG, (c) its resolved AFG using the [FLOW-AWARE] rule, (d) the flow-aware and flow-insensitive points-to sets superimposed—the two edges highlighted are spurious dataflow facts avoided by the former but included in the latter.

call (based on the maximum index within the callee's summary). This means that an assignment occurring after the function call cannot be seen by a fetch occurring before or within the called function. Alternatively, a fetch occurring before the call site does not read a value assigned later by the callee. Figure 4.5 illustrates this.

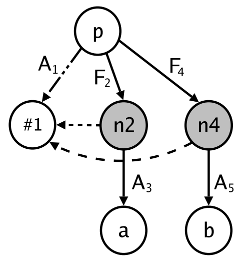
In Figure 4.5(b), `x`'s value is read by `q=x` in `bar()` then modified by `f()` at the call site `f(&x)`. When statement `*q=&w` executes, the original value of `x`, `&v`, is set to point to `w`. By ignoring order information, an interprocedural flow-insensitive analysis would pessimistically include `a` and `b` as values that could be read by `q=x`. Our flow-aware analysis avoids such spurious relations. Figures 4.5(c) and 4.5(d) show the resolved and summary AFGs for function `f`. Nodes labeled "#1" in Figure 4.5(c) and "`f#1`" in Figure 4.5(d) represents the initial value for parame-

```
f(int *p)
{
  *p = &a;
  ...
  *p = &b;
}
```

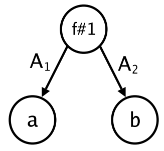
(a)

```
bar()
{
   x = &v; // A1
   ...
   q = x;  // F2 A3
   ...
   f(&x);  // A4 A5
   ...
  *q = &w; // F6 A7
}
```

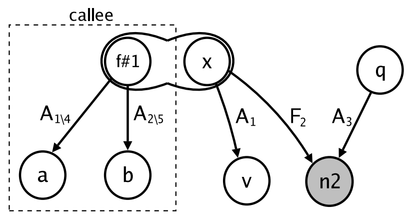
(b)



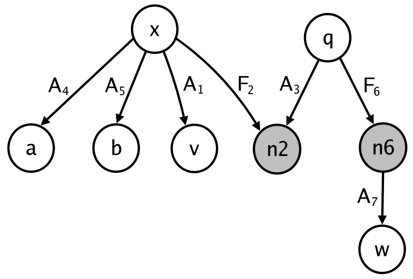
(c)



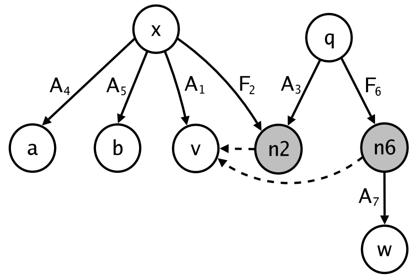
(d)



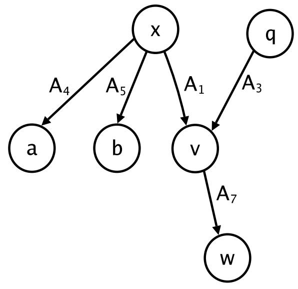
(e)



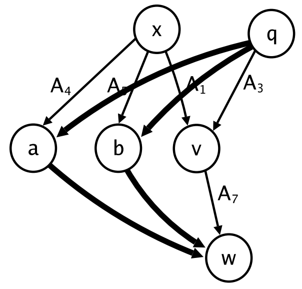
(f)



(g)



(h)



(i)

Figure 4.5: Propagating flow-aware ordering across procedure calls. (a) Function `f()` is called by (b) `bar()`. The resolved (c) and summary (d) AFG for `f()`, which is instantiated at the call site `f(&x)` (e): the indices in the summary are updated and the remaining statements in `bar()` are processed, generating the initial AFG for `bar()` (f). Flow-aware analysis is performed (g) by considering ordering across procedures. The flow-aware points-to set (h) is more precise than the flow-insensitive (i).

ter `p`, which is merged with the argument node at the call site (modeling parameters and summary instantiation is explained in detail in Section 5.2).

We sort the edges in a function summary and number them starting from 1, being careful to preserve the relative order among statements. Figure 4.5(e) shows how the summary for `f` is instantiated at the call site `f(&x)`. Statements before the call are labeled A_1, F_2, and A_3.

To place a callee's statements in the total order, we add the highest index before the call to every statement in the callee's summary when we instantiate it. In Figure 4.5(e), this index is 3, so we label callee's statements $A_{1\backslash 4}$ and $A_{2\backslash 5}$ to indicate A_1 and A_2 will become A_4 and A_5. Processing the remaining statements after the function call gives the initial AFG for `bar()` in Figure 4.5(f).

Figure 4.5(g) shows the result after flow-aware analysis. Note the fetch of `x` in `q=x` (F_2) resolves to A_1, the only assignment occurring before that fetch. When F_6 matches A_3 a single alias edge is added. Figure 4.5(h) is the corresponding points-to sets, which is more precise than the flow-insensitive result in Figure 4.5(i).

4.1.3 Loops and Recursive Procedures

This section describes how loops and recursive procedures are handled in general, emphasizing the case for flow-aware analysis. We convert loops into tail-recursive procedures and iteratively analyze (such) recursive procedures until we reach a fixed-point. The first time a recursive procedure is analyzed, we do not have a summary for it, so we only consider the other statements in the procedure. This gives a better summary for the procedure, which we then instantiate at recursive call sites and summarize again.

It may appear this procedure may not terminate, but this is not the case. It turns out the number of edges and arcs that can be added is bounded. The number of heap nodes is bounded because of the heap naming scheme we adopt (Section 5.8). The number of fetch edges is bounded because the final summary allows at most one fetch edge out of any node, and there is a parametrizable limit on the length of any chain of fetch edges. Finally, we prohibit duplicate assign edges. Together, these constraints bound the summary and guarantee convergence. If duplicate assignments between a pair of nodes is allowed, such as in Figure 4.6(e), the comparison between two summaries must only consider whether $x \xrightarrow{A} y$ exists and not the number of such edges (details given in Chapter 5).

Figure 4.6 illustrates summarizing a simple *for* loop. We transform the function in Figure 4.6(a) into the tail-recursive procedure in Figure 4.6(b). The transformation assumes an Algol/Pascal-style scope where a procedure can be defined inside another procedure, and variables declared in between are local to the outer procedure and global to the inner one. To illustrate, we nested the definition of `Loop` inside `bar` to emphasize that it has access to `bar`'s local variables.

Figure 4.6(c) is a simplified control-flow graph for this code. On the left is the structure of the loop; on the right is a linearized version of the `Loop` procedure that assumes flow-aware analysis placing the *then* branch of the *if* before the *else*.

Figure 4.6(d) is the first summary of `Loop`—the assignment `p=&y` is hidden from `z`. We now have a summary of `Loop`, which is used in the second iteration of the analysis giving Figure 4.6(e). Edges with subscripts 1, 2, and 3 correspond to the loop body statements within the function. Instantiating the earlier summary adds edges F_4, A_5, and A_6 (the indices are shifted as in Section 4.1.2). This time, fetch edge F_4 matches assignment A_3, and `z` will point to `y` as a result; edges F_4 and A_3 belong to different iterations of the original loop.

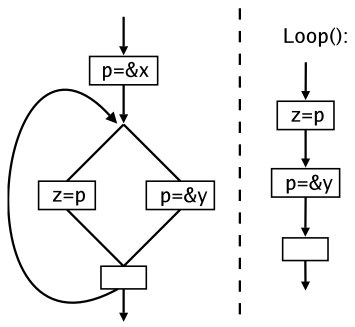
Figure 4.6(f) is the summary of Figure 4.6(e) and also the fixed-point—the final summary for function `Loop()`. Some edges have two numerical labels because they are the result of merging multiple edges. This means they represent the interval for which the operation is valid. For example, A_6^3 represents the merge of A_3 and A_6.

Figure 4.6(g) shows the graph for `bar` after we inserted the summary for `Loop`. The fetch of `p` resolves to `p=&x` since the assignment occurs before the fetch (i.e., we check that the subscript index of the fetch is greater than the superscript on the assign, in case one exists, or the subscript otherwise). Finally, Figure 4.6(h) shows the summary for `bar`, which notes that the global variable `p` is fetched.

Note in Figure 4.6(g) that F_5^2 does not resolve to A_7^4, although $5 > 4$. This happens because A_1 is the only *new* fact that needs to be considered when instantiating the summary of `Loop()` into `bar()`; all the other edges were already present in Figure 4.6(f), and F_4 had already been resolved to A_3 in Figure 4.6(e). Our algorithm is such that only the addition of new aliases or new facts trigger the inference rule for resolution. This incremental nature is detailed in Section 5.5.

```
int *p, x;
void bar() {
  int *z, y;
  p = &x;
  for (i=0; i<n; i++) {
    if (...)
      z = p;
    else
      p = &y;
  }
}
```

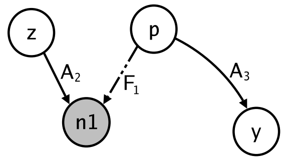
(a)



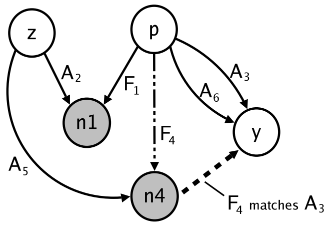
(c)

```
int *p, x;
void bar() {
  int *z, y;

  void Loop() {
    if (i < n) {
      if (...)
        z = p;
      else
        p = &y;
      i++;
      Loop();
    }
  }

  p = &x;
  i=0; Loop();
}
```

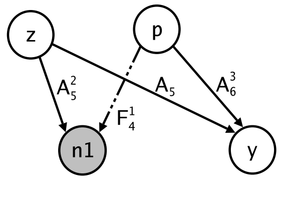
(b)



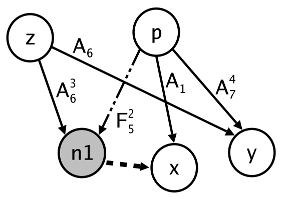
(d)



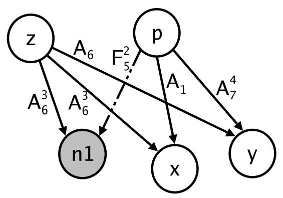
(e)



(f)



(g)



(h)

Figure 4.6: Handling loops and recursive functions.

4.2 Flow-Branch-Aware analysis

The flow-aware analysis of last section provides a total order approximation to the program's partial order, implemented through a simple labeling algorithm. We observed that a clever extension to such algorithm gives a true partial order relation to the [PO-ALIAS] inference rule, leading to another interesting pointer analysis algorithm.

The basic idea is to calculate two total orders on the statements. One total order always starts with the left branch of any conditional, and the other always starts with the right branch. Then one statement can reach another only if it precedes it in both total orders. Furthermore, if statement p precedes q in one total order, and q precedes p in the other, then they are in fact mutually exclusive. This means this analysis fully disassociates the two branches of a conditional without using guards.

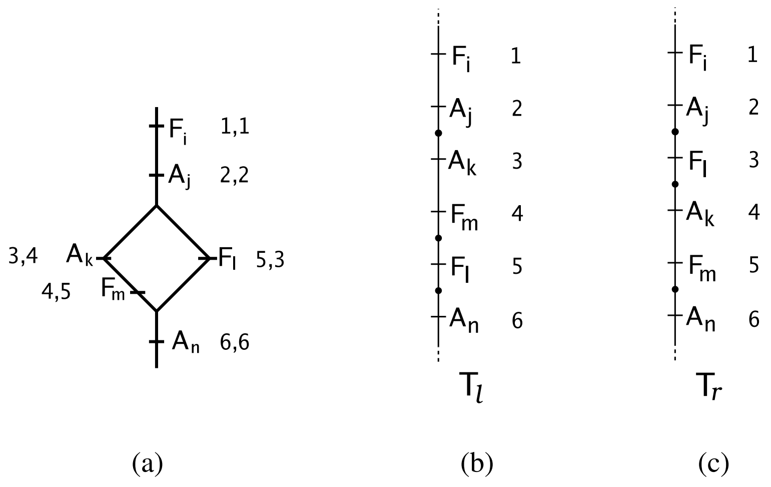


Figure 4.7: (a) a fragment of a control-flow graph (b)—(c) the two total orders for it. T_l starts with the left branch and T_r starts with the right branch. An assignment reaches a fetch only if the assignment precedes the fetch in both total orders. This is the case for A_j and F_l, A_j and F_m, and A_k and F_m.

Figure 4.7(a) is the fragment of a control-flow graph, and Figures 4.7(b) and 4.7(c) are the two total orders for it, labeled T_l and T_r. The numbers on the right represent the ordering. Let $\mathit{affects}_1(\sigma_A, \sigma_F)$ be true whenever the assignment σ_A precedes the fetch σ_F in both total orders. In Figure 4.7(a) $\mathit{affects}_1(A_j, F_l)$, $\mathit{affects}_1(A_j, F_m)$, and $\mathit{affects}_1(A_k, F_m)$ hold. Fetch F_l does not resolve to assign A_k, or $\neg\mathit{affects}_1(A_k, F_l)$, because their relative order is swapped from T_l to T_r .

The [ALIAS] rule for flow-branch-aware analysis becomes

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad \mathit{aliases}(\alpha,\gamma) \qquad \mathit{affects}_1(\sigma_A,\sigma_F)}{\mathit{al}(\beta) \subseteq \mathit{al}(n)} \;[\text{FLOW-BRANCH-AWARE}]$$

This rule reduces even further the number of fetch/assign matchings, resulting in a more precise analysis. Furthermore, it can also be more efficient than both [ORIGIN-ALIAS] and [FLOW-AWARE] in some cases: the faster convergence may overrun the extra computation required to build and maintain the more precise order information.

We implement flow-branch-aware analysis by keeping two ranks on each edge, one rank for each total order (named rank_l and rank_r). This is depicted in Figure 4.7(a) by two numbers separated by a comma. The relation $\mathit{affects}_1(\sigma_A,\sigma_F)$ can be computed by

$$\mathit{rank}_l(\sigma_A) < \mathit{rank}_l(\sigma_F) \wedge \mathit{rank}_r(\sigma_A) < \mathit{rank}_r(\sigma_F)$$

For example, $\mathit{affects}_1(A_k,F_m)$ holds because $3 < 4 \wedge 4 < 5$. Similarly, $\mathit{affects}_1(A_j,F_l)$ is true since $2 < 5 \wedge 2 < 3$. On the other hand, $\neg\mathit{affects}_1(A_k,F_l)$ because $3 < 5$ but $4 \not< 3$.

When doing interprocedural analysis, the callee summary records the highest index for both total orders, and the "shifting" described previously is performed on each order separately.

4.2.1 Why does it work?

The reason why two total orders work is not because each if-statement has two branches, but because control-flow graphs of programs can be imbedded in two dimensions. The algorithm still works for a generic `switch` statement modeled by multiple branches. The required generalization is that one total order always starts with the leftmost branch and moves rightward, while the other does the opposite. Still, only two ranks need to be stored on each edge of the AFG, and the relation $\mathit{affects}_1$ remains the same. The basic idea is given in Figure 4.8.

Intuitively, the number of ranks needed is equal to the number of dimensions required to imbed the partial order. Imagining a "unidimensional" control-flow graph (i.e., a straight line code), then only one index is sufficient to compare the relative order between any two operations. For planar graphs, two indices are required, and so on. An inductive argument goes as follows. Assume we have a control-flow graph G with n dimensions that has been properly labeled with n ranks per

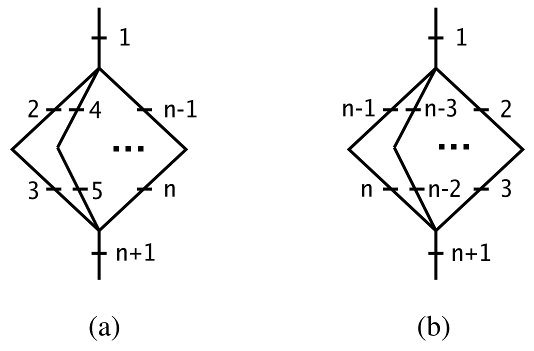


Figure 4.8: The two total orders strategy works for `switch` statements modeled by multiple branches. The required generalization is that one total order always starts with the leftmost branch and moves rightward, while the other does the opposite.

operation. If we extend G such that an extra dimension is required, then all operations need to be ranked with respect to this $n+1$-th dimension.

Practically, the only restriction to the flow-branch-aware analysis is that the procedure's control-flow graph be planar—it is hard to imagine a program written in a high-level language that would not satisfy this requirement.

As an example of flow-branch-aware analysis, Figure 4.9(a) shows a fragment of a control-flow graph, while Figures 4.9(b) and 4.9(c) show the two total orders for it. The resolved AFG using the [FLOW-BRANCH-AWARE] rule is illustrated in Figure 4.9(d), and its summary shown in Figure 4.9(e) (initial values are ignored for simplicity). Compare this graph with the summaries obtained by the flow-insensitive (Figure 4.9(f)) and flow-aware (Figure 4.9(g)) analyses (assume that the latter uses T_r as the code linearization).

Figure 4.10 informally places the flow-branch-aware analysis in our analysis space. A higher degree of "conditions" is given because this analysis fully disassociates mutually exclusive operations within an `if` or `switch` statement.

4.3 Unconditional kill

A bottleneck of some analysis variations is generating unnecessary initial values due to lack of kill information. I.e., when instantiating a callee's summary at a call site, a (demoted) fetch edge from

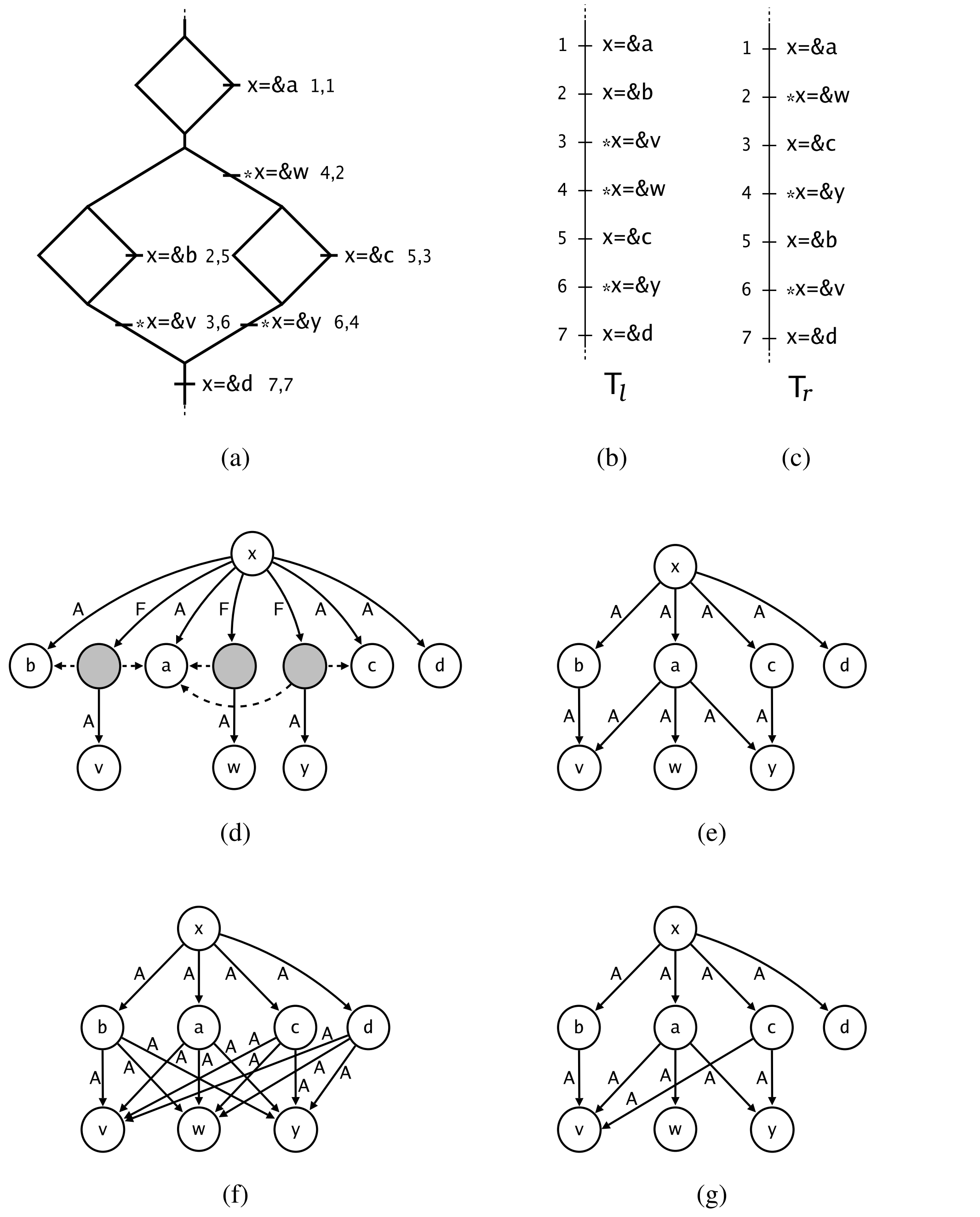


Figure 4.9: An example of flow-branch-aware analysis. (a) a fragment of a program's control-flow graph. (b)—(c) the two total orders for it. (d) the resolved AFG for the code using the [FLOW-BRANCH-AWARE] rule. (e) the corresponding summary AFG. Compare this summary with the flow-insensitive (f) and flow-aware (g) for the same code (assume the latter uses the T_r order).

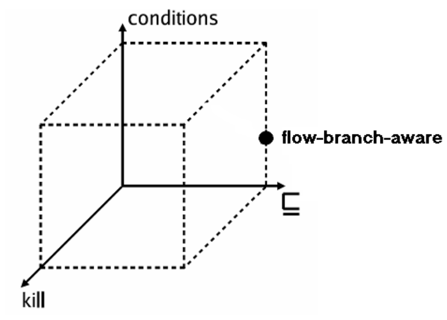


Figure 4.10: The placement for flow-branch-aware analysis.

the summary may replicate itself as the unknown initial value for some variable in the caller, and the process may get repeated in a grand-caller, and so on, each time making the analysis less precise. This may be unnecessary in many cases, since a variable that is initialized at function entry does not need a fictitious initial value. The idea of the unconditional kill is to avoid creating unnecessary initial values by looking for variable initializations (i.e., unconditional assignments). Figure 4.11 illustrates an example.

Figures 4.11(a), (b), and (c) show the code for three procedures, `zzz()`, `bar()`, and `foo()`. Assume all variables mentioned are globals. Figure 4.11(d) is the resolved AFG for `foo()` assuming, e.g., [FLOW-AWARE] rule, and Figure 4.11(e) is the final summary. Note `p` has an initial value modeled through the demoted fetch edge $\mathtt{p} \xrightarrow{F_2} \mathtt{p}_1$.

Using `foo`'s summary at the call site inside `bar` gives the resolved AFG in Figure 4.11(f). Because strong updates are ignored, the analysis replicates the fetch of `p` as the initial value $\mathtt{p} \xrightarrow{A_3} \mathtt{p}_1$, which is demoted to $\mathtt{p} \xrightarrow{F_3} \mathtt{p}_1$ in Figure 4.11(g).

Note however that statement `p=&x` inside `bar` initializes `p` before calling `foo`. I.e., any value for `p` coming from a caller of `bar` is unconditionally killed by `p=&x`.

By noting this, the more precise graphs in Figures 4.11(h) and 4.11(i) can instead be generated as the resolved and summary AFGs for `bar`. The idea is to mark nodes that have been unconditionally assigned prior to the first time they are fetched. For these cases, fictitious initial values are not created. Note this requires the use of some statement ordering (e.g., flow-aware, flow-branch-aware, etc.), since flow-insensitive analysis is not suitable for properties that depend on execution order.

The unconditional kill is a simple yet useful approximation on the kill axis. For instance, Figures 4.11(j) and 4.11(k) show the resolved and summary graphs for `zzz` when strong updates are

```
zzz()
{
  p = &w;
  ...
  bar();
}
```

(a)

```
bar()
{
  p = &x;
  ...
  foo();
}
```

(b)

```
foo()
{
  if (...) {
    p = &y;
  }

  *p = &z;
}
```

(c)

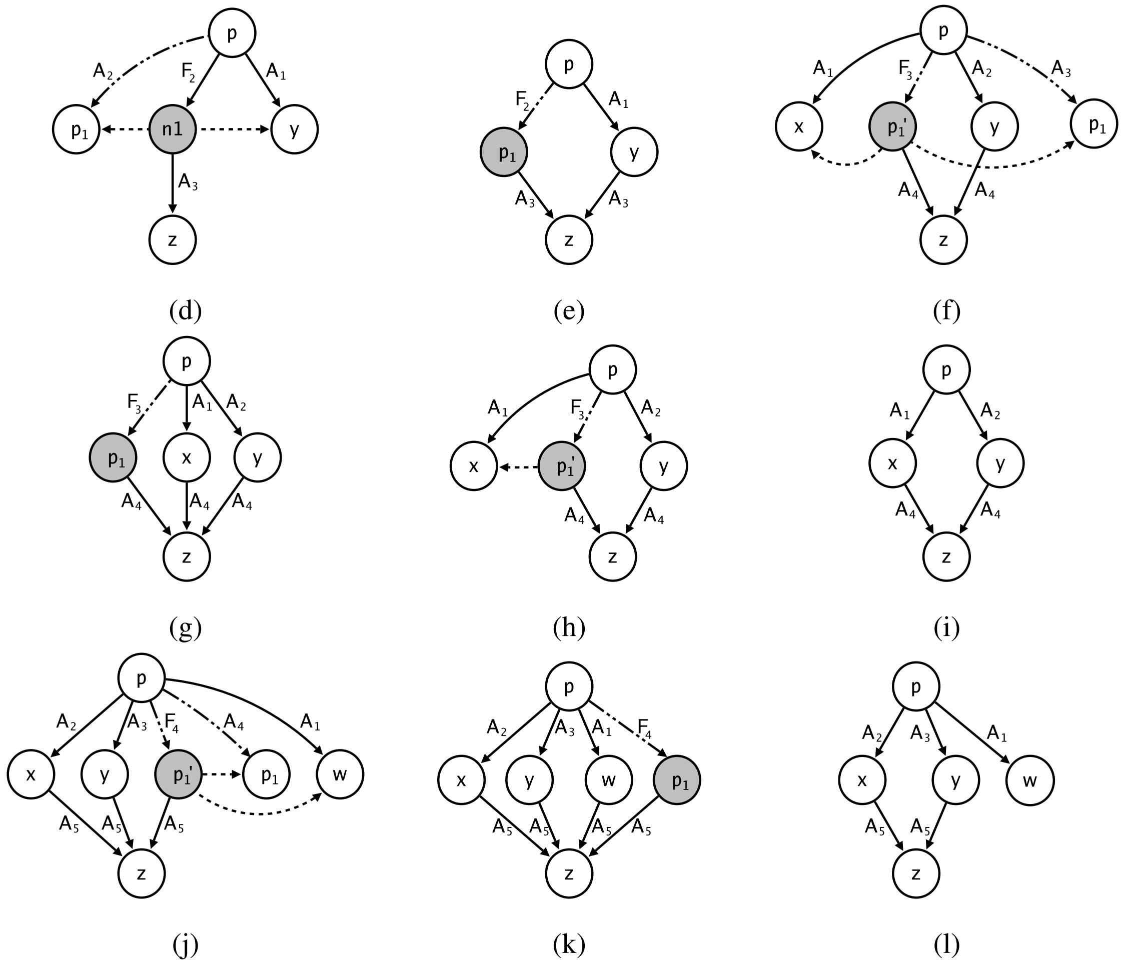


Figure 4.11: A variable that is initialized at function entry does not require an unknown initial value modeled—p is unconditionally assigned at bar's entry, so the fetch of p originated at foo does not propagate past bar. This results in the more precise summary (l) of zzz instead of the graph in (k).

ignored throughout the analysis. Figure 4.11(l) shows the summary obtained when the approximation is considered, a cleaner result.

4.4 T, F, "?" approximation for conditions

In analyzing procedures, it is generally interesting to distinguish among side-effects that always happen, side-effects that can never happen, and side-effects that may happen under some (unknown) condition. E.g., for *mod* calculation it is very useful to distinguish whether a variable *must*, *may* (regardless of what specific condition) or *won't* be modified by a function call. It is generally useful to know when a pointer (e.g., function parameter) is unconditionally dereferenced upon function entry.

Similarly, we can approximate the predicates of our analysis to *True*, *False*, and *"?"* guards, the latter representing some unknown condition. An undefined edge is interpreted as the predicate *False*, which is added for logical completeness. If both *may* and *must* information are recorded, then the false information need not be included. But in an application where *may* information is too numerous, it may be more efficient to list *must* and *won't* information.

This approximation simplifies the analysis at the cost of spurious relations, but facilitates propagating conditions inter-procedurally. I.e., we can limit path-sensitivity to within function bodies, and then convert the function's final summary to use the trivial guards (T and "?"). As a result, no non-trivial guards are propagated between functions.

Chapter 5

The Assign-Fetch Graph in Detail

This chapter examines the Assign-Fetch Graph introduced in Chapter 3 in more detail, including the manipulation of procedure parameters, summary instantiation, field-sensitivity, heap modeling, etc., and how procedure summaries based on the AFG satisfy the requirements of modular bug finding listed in Section 1.3.3. It also formalizes various concepts and terminology that were introduced informally earlier.

5.1 General definitions

As mentioned before, the pointer analysis framework presented in this thesis is interprocedural; it constructs summaries to model the (pointer) effects of a procedure at a call site (the functional approach to context-sensitivity).

The input into the analysis is a program we assume has been partitioned into procedures. Each procedure is treated as consisting of three types of operations: assignments, fetches, and function calls. In this section, we will ignore pointer arithmetic by considering all elements of an array to be one location. Similarly, fields of structures and classes are merged so an expression such as `p->next` is treated like `*p`—field-sensitivity is the subject of Section 5.7. We model return values with a special node in the graph. Assume no heap locations for now; these are handled in Section 5.8.

To analyze procedures, we represent them initially using an assign-fetch graph. As explained earlier, an AFG represents the assignments and memory dereferences of a function rather than points-to information. Unlike a points-to graph (denotational) whose shape depends on a function's context (i.e., aliases present when the function is called) the AFG is "context-agnostic" (operational) and therefore can be used in any context.

Following the philosophy of Section 1.3.2, when we analyze a procedure we assume none of its parameters (in general, locations from the environment) are aliased. Any bug report within a function f is bound by that assumption—if some two aliased arguments[1] would cause a program error, then the caller would be reported as faulty because it passes invalid inputs into f (the flip-side is that no error exists as long as no one passes such aliased arguments). When (and if) the summarized information for f is used at a call site, a context with aliases causes the instantiated summary to be incrementally updated (Section 5.5).

The output of the analysis is a summary for each procedure in the form of a *summary* AFG. Summarizing can be seen as a partial evaluation of an abstract code which specializes what is not affected by its inputs. The remained part is evaluated when the inputs are given, specially if the actual inputs differ from the assumed ones (e.g., non-aliased arguments)—a kind of delayed computation.

Computing points-to sets on the AFG is divided into three parts: (1) building the *initial* AFG for the function by traversing its statements, including calls to other procedures; (2) computing the *resolved* AFG by adding alias edges to the initial AFG; (3) removing unnecessary information from the resolved AFG to derive the *summary* AFG.

A formal classification of AFG elements based on some common characteristics will be useful later; we refer to the example on Figure 5.1 to illustrate these concepts.

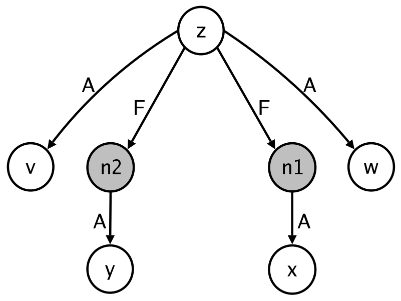
A *location node* is simplest: it represents the address of global or stacked variables, or heap-resident data. In Figure 5.1(b), `v`, `w`, `x`, `y`, and `z` are each location nodes that represent the address of distinct global variables.

Fetch nodes, shaded gray, represent the results of reading a value from memory. Since in general the result of such a fetch depends on the behavior of the program at runtime, we think of the "value" of a fetch node as being unknown. Resolving the possible values returned by each fetch is the bulk of the analysis. In Figure 5.1(b), n_1 and n_2 are fetch nodes. By definition, each fetch node has exactly one incoming *fetch edge*.

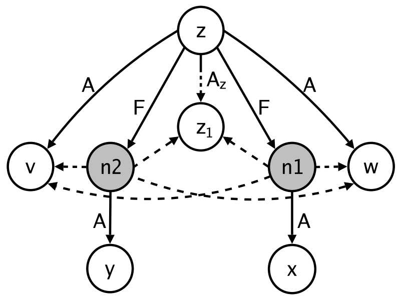
[1]Arguments are sometimes referred to by the term actual parameters.

```
foo()
{
  *z = &x;
   z = &v;
   if (...) {
     z = &w;
   }
   else {
    *z = &y;
   }
}
```

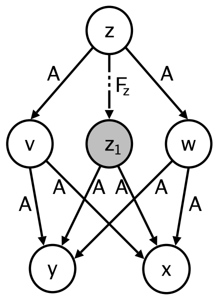
(a)



(b)



(c)



(d)

Figure 5.1: (a) A function and (b) its initial AFG, (c) its resolved AFG using [ORIGIN-ALIAS] rules, (d) its summary AFG.

Interface nodes are location nodes that correspond to information that crosses the caller-callee interface of a procedure. They are the global variables, parameters values, and heap locations allocated inside the callee. Nodes for local (automatic) variables are therefore never interface nodes. In Figure 5.1, nodes v, w, x, y, and z are all interface nodes, assuming they all represent global variables.

Initial value nodes are placeholders for (chains of) values of global variables and parameters supplied by the environment and represent chains of dereferences of unbounded length that start at interface nodes. Of course, only a finite number of initial values will be needed, and therefore an actual implementation will generate the initial values on demand. In our implementation, we add them lazily in the process of analyzing the assign-fetch graph. At that point, we think of them as location nodes (e.g., z_1 in Figure 5.1(c)). After we summarize the results of an analysis, we demote them to fetch nodes (e.g., z_1 in Figure 5.1(d)); an initial value created for an interface node z implies that z has been fetched within the function's body. Note that we only add those initial value nodes that are actually needed by the procedure, e.g., even though the variable x appears in Figure 5.1(a), we do not construct an initial value node for it because the procedure never reads its value.

There are two type of *operation edges*. A *fetch edge*, which we label with an "F," represents a memory read operation. Its source node represents the address being read and its target is a fetch node that represents whatever value(s) is stored at that location. An *assign edge*, labeled "A," represents a memory write operation. Its source node represents the target address and its destination represents the value being written.

Alias edges, which we draw as dashed lines, represent aliasing information among nodes and can be read "can be an alias for." Each location node has an implicit self-loop alias edge that represents the fact that the address of each variable is unique. Proper alias edges always leave fetch nodes and terminate at location nodes that the fetch node can alias. For example, the alias edge from n_2 to v in Figure 5.1(c) means the alias relation $\langle *\mathtt{z}, \mathtt{v}\rangle$.

Finally, *initial value edges* represent environment initialization. Like initial value nodes, we think of them one way (as assign edges) when we are analyzing the AFG and another way (as fetch edges) after we summarize the graph. We draw them with dashed/dotted lines, e.g., from z to z_1 in Figures 5.1(c) and 5.1(d).

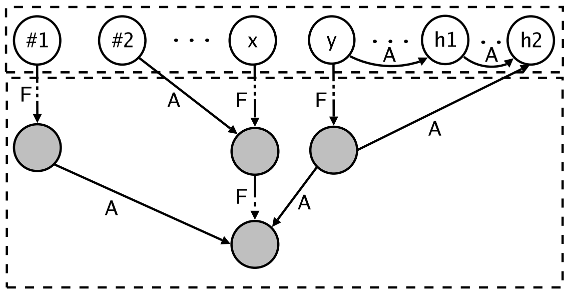


Figure 5.2: A generic procedure's summary.

Using the above terminology, the *summary* AFG for a procedure has only interface nodes and (demoted) initial value nodes. Each interface node may have a chain of initial values represented by a chain of (demoted) fetch edges; these are the only fetches in the summary since all others are removed as part of the summarization process. Since each interface node is assumed to have at most one (direct) initial value, each node in the summary has at most one outgoing fetch edge. Assign edges may connect any two pair of nodes in the summary.

A generic summary graph where interface nodes are drawn in a horizontal line instead of the more intuitive "tree-like" form is given in Figure 5.2. When using the summary at a call site, the interface layer is used to paste together the callee-caller boundary. Nodes labeled #1 and #2 are parameter values, explained in the next section.

5.2 Modeling parameters

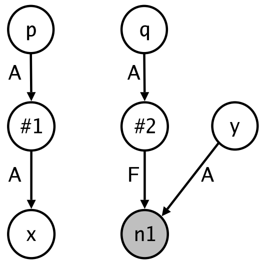
We treat procedure parameters almost like global variables, except for two things: (1) while we assume each comes from the environment, a caller will always initialize formal parameters so we add explicit initial values for them in the form of location nodes (labeled "#*i*" in our figures). In contrast, we only add an initial value for a global variable if it is fetched by the procedure; (2) initial values for global variables are demoted from location to fetch node status during summarization; initial values for parameters keep their location node status and are merged with argument nodes during summary instantiation.

Since formal parameters (not their initial values) are local variables, i.e., stacked and discarded when a procedure returns, we remove their nodes during the summarization process.

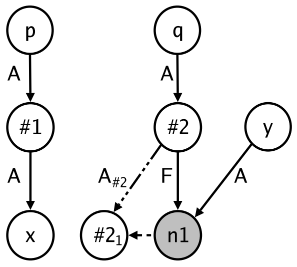
Figure 5.3 illustrates the summarization of a simple procedure with two parameters. More

```
f(p,q)
{
  *p = &x;
   y = *q;
}
```

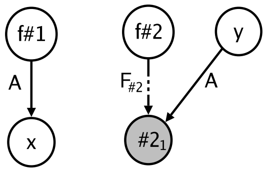
(a)



(b)



(c)



(d)

Figure 5.3: (a) A procedure with arguments, (b) its initial, (c) resolved and (d) its summary AFGs.

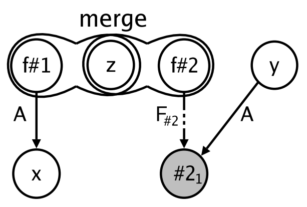
precisely, given a function f with formal parameter list $p_1, \ldots, p_k$, an assign edge $p_i \xrightarrow{A} \#i$ is created for each parameter p_i. In the example, two nodes are created for the formal parameters `p` and `q`, which are assigned initial values `#1` and `#2` representing their initial caller-passed values. This plus adding nodes and edges for the two statements in the procedure gives the initial AFG for function `f` in Figure 5.3(b).

Since formal parameters are initialized by "#*i*" nodes, the AFG representation for, e.g., `*p=&x` in Figure 5.3(b) does not include a fetch edge; `*p` directly yields its initial value, #1.

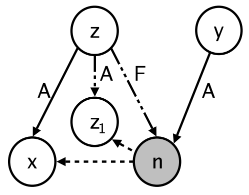
In summarizing this (alias-free) procedure, we remove the nodes for the formal parameters `p` and `q` and rename their initial values to include the procedure's name. Also, fetch edge $\#2 \xrightarrow{F} n_1$ in Figure 5.3(b) generates an initial value for node #2, which in Figures 5.3(c) and 5.3(d) is labeled $\#2_1$. Figure 5.3(d) is the final summary. Note parameters are assumed non-aliased at function entry.

```
g()
{
  f(&z, &z);
}
```

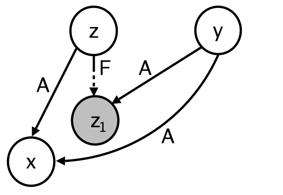
(a)



(b)



(c)



(d)

Figure 5.4: (a) A procedure `g`, which calls the procedure `f` from Figure 5.3, (b) its initial AFG, (c) after resolving, and (d) its final summary.

5.3 Modeling procedure calls

When building the initial AFG for a procedure, a call to a function is replaced by the callee's summary. Instantiating a summary involves merging global variables shared by both and connecting formal parameters to actual parameters. Figure 5.4 illustrates calling procedure `f` from Figure 5.3. The address of global variable `z` is passed to both `p` and `q`, so when we copy the summary of `f` from Figure 5.3(d), we mark the nodes for the initial values of `p` and `q`, `f#1` and `f#2`, to be merged with `z`.

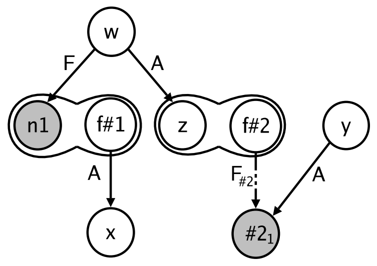
We perform the same process for each global variable: its node in the callee is merged with its node in the caller. This is vacuous in Figure 5.4 since `g` does not touch globals `x` or `y`.

Once each callee's summary has been instantiated and all remaining statements considered, we compute the caller's summary. In Figure 5.4(c), we added an initial value node for global `z` and used flow-insensitive analysis to add alias edges from from *n* to `x` and $\texttt{z}_1$. Figure 5.4(d) is the summary. We removed fetch node *n*; its aliases now manifest themselves as the assign edges from `y`. A caller of `g` knows that `z` is dereferenced somewhere down the line by looking at `g`'s summary.

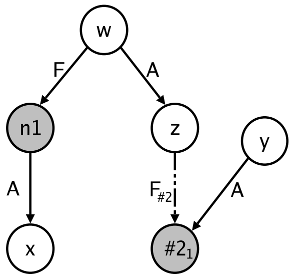
This example illustrates how a summary is agnostic about parameter aliasing and can be used in any context. The summary for `f` does not consider the case when its two parameters are aliased. However when `g` calls `f` we correctly ascertain that `y` may point to `x`, which is only possible if

```
g()
{
  int *w = &z;
  f(w, &z);
}
```

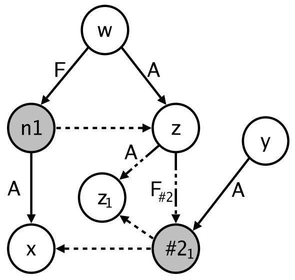
(a)



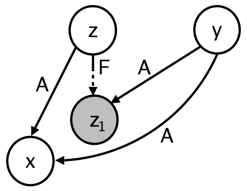
(b)



(c)



(d)



(e)

Figure 5.5: (a) A semantically equivalent code of g from Figure 5.4; (b) the parameter passing (c) its initial AFG, (d) after resolving, and (e) its final summary, which is equivalent to the summary from Figure 5.4(d).

the two arguments are aliased at the call site. Existing solutions either use information from the environment while building a summary, or build multiple summaries for each function, one for each possible environment; these solutions are unfit for our purposes. In our tool, the analysis of f considers the pointer information as in Figure 5.3(d). The analysis of g uses Figure 5.4(d). According to the requirements listed in Section 1.3.3, note that we "start fresh" each time a new summary is constructed instead of flowing information from callers while building the summary.

Note that node merging such as used in this section always works, even when the actual parameter is an arbitrary expression. Consider for instance the code in Figure 5.5(a), which is semantically equivalent to the code in Figure 5.4(a). Note w is a local variable for function g, and therefore is not allowed to have an initial value edge added by the analysis.

Similar to f(&z,&z) in Figure 5.4(a), the function call f(w,&z) in Figure 5.5(a) has both arguments aliased at the call site. The first argument reads the value of w, and the second "hard-wires" the address of z. Accordingly, there is a fetch of w in Figure 5.5(b)—the fetch node represents the first argument. The second argument is the location node for z itself. As with the previous example, nodes for formal parameters are merged with their corresponding arguments, as indicated in Figure 5.5b—n_1 is marked to be merged with `f#1` and `z` is marked to be merged with `f#2`.

Figure 5.5(c) is the resulting graph after node merging, and Figure 5.5(d) shows the resolved AFG. The aliasing introduced at the call site `f(w,&z)`, assumed nonexistent while summarizing `f`, is recovered by the analysis in Figure 5.5(d). The final summary for `g` is shown in Figure 5.5(e), which is equivalent to Figure 5.4(d).

Figure 5.6 shows another example of function summarization for three procedures `f`, `g`, and `h` (assume [ORIGIN-ALIAS] rule). The summaries assume no aliasing among locations from the environment; aliasing between `f`'s parameters `r` and `t` is then introduced by `f`'s caller, `g`. Aliasing among `g`'s parameters is also introduced by its caller, `h`. These aliasing are incrementally recovered by the analysis when using summaries at call sites; the final summary for `h`, for example, (Figure 5.6(k)) indicates that both `z` and the initial value of `x` point-to both `y` and `w`, which occurs only if `r`, `s`, and `t` are all aliased. Figures 5.6(a), 5.6(b) and 5.6(c) are the code for the three procedures; Figure 5.6(d) is the initial AFG for `f` and Figure 5.6(e) is its summary; Figures 5.6(f), 5.6(g) and 5.6(h) show the summarization of `g`: Figure 5.6(f) uses `f`'s summary at the call site `f(p,q,p)`, which results in the initial AFG for `g` in Figure 5.6(g); `g`'s summary is Figure 5.6(h). The summarization of `h` is depicted in Figures 5.6(i), 5.6(j) and 5.6(k), which uses the summary of `g`.

5.4 Return values

Return values are represented by a special node in the Assign-Fetch Graph (unless a function unconditionally returns a single location node, such as `return &x`, in which case the node for variable `x` is also the return node). In general, because a function may return from different places in its body, a temporary node (labeled *ret* in our figures) is created; such node is assigned as many times as there are `return` statements within the function. Then, a fetch edge is inserted to read the contents of *ret*—the corresponding fetch node is the value returned by the procedure. Because it resolves to a set of nodes, this node encapsulates all the returned values.

To illustrate, the code in Figure 5.7(a) shows two procedures, `f` and `g`; the former returns `*x` and the address of `b`, both `x` and `b` global variables. Return nodes are drawn as double circles.

When building the initial AFG for `f` in Figure 5.7(b), the temporary node *ret* is assigned the

```
h()
{
  g(&x, &x);
}
```

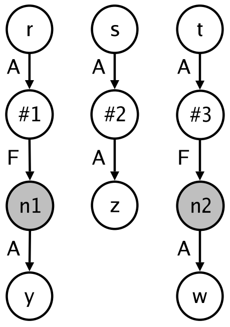
(a)

```
g(p,q)
{
  f(p,q,p);
}
```

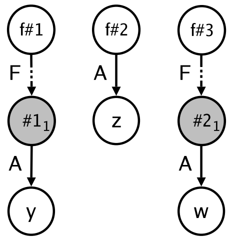
(b)

```
f(r,s,t)
{
  **r = &y;
   *s = &z;
  **t = &w;
}
```

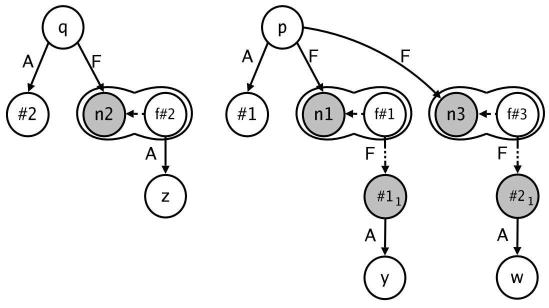
(c)



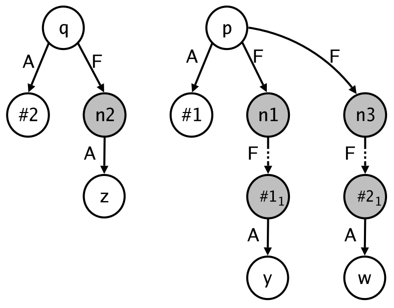
(d)



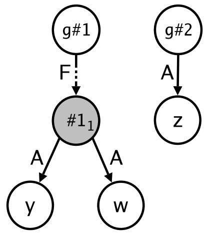
(e)



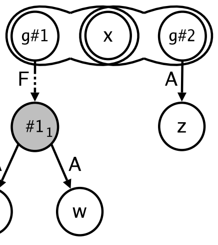
(f)



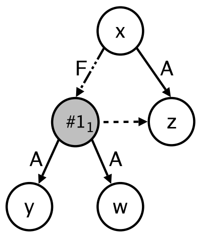
(g)



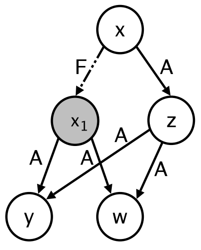
(h)



(i)



(j)

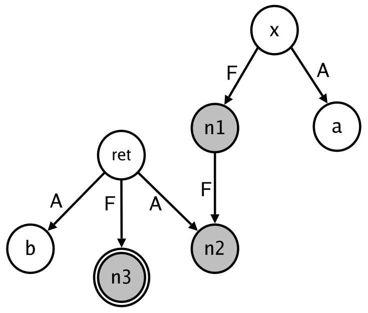


(k)

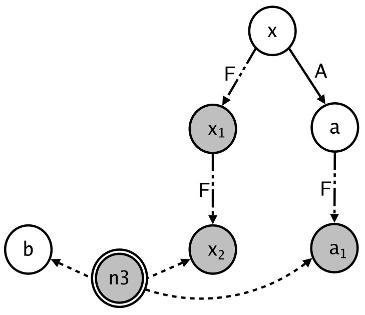
Figure 5.6: An additional example of function summarization.

```
int ***x, **a, *b;        int **s, k;
int *f()                  void g()
{                         {
  if (...) {                s = f();
    x = &a;                *s = &k;
  }                       }
  else {
    return &b;
  }
  return *x;
}
```

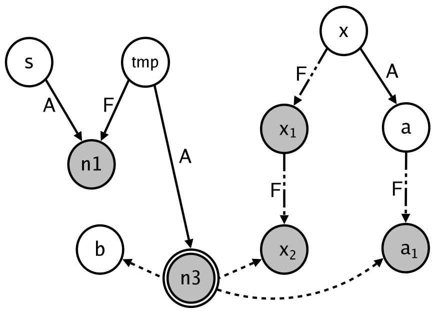
(a)



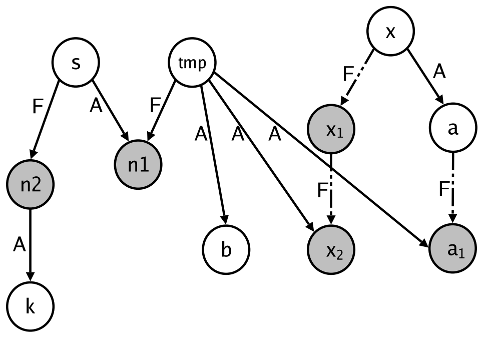
(b)



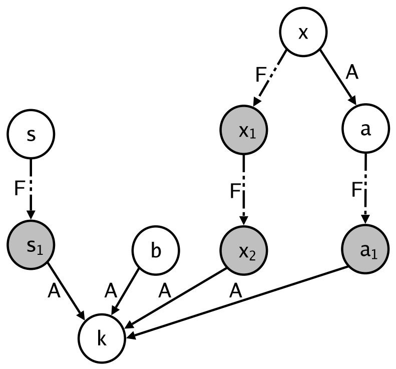
(c)



(d)



(e)



(f)

Figure 5.7: (a) Two procedures f and g. (b) The initial AFG for f; the return node is depicted as a double circle and a node labeled *ret* is used for temporary computation (c) The summary AFG for f; alias edges from the return node are allowed. (d) Applying f's summary into g. (e) The initial AFG for g. (f) The summary AFG for g.

address of `b` (the first return statement in Figure 5.7(a)) and node n_2, representing the second read of `x` in the statement `return *x`. Fetching location *ret* gives the double-circled node labeled n_3, representing the values returned by the procedure.

Assume the analysis of `f` uses the [ORIGIN-ALIAS] rule. The resulting summary AFG in Figure 5.7(c) has the return node resolving to `b`, $\mathtt{x}_2$ and $\mathtt{a}_1$ ($\mathtt{x}_2$ is a node in the chain of initial values for `x`; $\mathtt{a}_1$ is the initial value for `a`, induced by edge $n_1 \xrightarrow{F} n_2$ in Figure 5.7(b) after $\mathtt{x} \xrightarrow{F} n_1$ is resolved to $\mathtt{x} \xrightarrow{A} \mathtt{a}$).

Two things should be noted: (1) alias edges are present in the *summary* AFG, which was not the case so far; (2) alias edges can connect two *fetch* nodes together (e.g., the return node n_3 and $\mathtt{x}_2$). These exceptions are allowed for return nodes because, after instantiating the summary for a function in a call site, the first thing to do is to remove the return node. This process is depicted in Figures 5.7(d) and 5.7(e), where the statement `s=f()` is converted into two statements `tmp=f()` and `s=tmp`. First, a temporary node labeled *tmp* is created and then assigned the return node from the callee's summary to model `tmp=f()`. Then, two new edges are inserted to model `s=tmp`—a fetch to access *tmp*'s contents and an assign to write these contents into `s`. Any further manipulation of the returned value is done through the fetch node. This is done to allow us a uniform treatment of function calls anywhere in the program.

Second, the return node is removed from Figure 5.7(d). Before we remove such node, its adjacent operation edges are transfered to nodes in $al(n_3)$. This is seen in Figure 5.7(e) where three assign edges replace the edge $tmp \xrightarrow{A} n_3$. This uses the same rationale applied when removing fetch nodes to produce the summary AFG for a function[2]. Figure 5.7(e) also includes the operation edges for the other statement in `g`, i.e., `*s=&k`, and it is the initial AFG for that function.

The analysis proceeds as usual, and resolving the graph in Figure 5.7(e) gives the summary of `g` in Figure 5.7(f).

5.5 Worklist Implementation of [ALIAS] rules

In this section, we illustrate a worklist-based implementation of the [ORIGIN-ALIAS] inference rule through the example in Figure 5.8. Our algorithm maintains a set of pairs $\langle \sigma_A, \sigma_F \rangle$ that are known

[2]Note this means an initial AFG cannot contain proper alias edges.

to satisfy the premise of the rule. Technically, we maintain a worklist of pairs of the form $\langle \sigma_F, S_{\sigma_F} \rangle$ where S_{σ_F} is a *set* of assignments. Intuitively, instead of considering pairs $\langle \sigma_A,\ \sigma_F \rangle$ separately, we group together all assignments that should resolve to σ_F in the current iteration of the algorithm. We will refer to pairs $\langle \sigma_A,\ \sigma_F \rangle$ and $\langle \sigma_F, S_{\sigma_F} \rangle$ interchangeably; the meaning should be clear from context.

We initialize the worklist with all pairs of edges $\langle \sigma_A,\ \sigma_F \rangle$ that share their source nodes. The algorithm is incremental in the sense that addition of new aliases triggers additions to the worklist.

Consider the code for `bar()` in Figure 5.8(a). Its initial AFG is shown in Figure 5.8(b). For the sake of presentation, operation edges have been subscripted. The worklist starts containing exactly two elements: $\langle F_3, \{A_1, A_{\mathtt{z}}\} \rangle$ and $\langle F_6, \{A_1, A_{\mathtt{z}}\} \rangle$. The assign edge labeled $A_{\mathtt{z}}$ is the initial value edge for interface node `z`.

For the initial AFG of `bar()`, the algorithm adds alias edges as follows. The first element, $\langle F_3, \{A_1, A_{\mathtt{z}}\} \rangle$, is taken from the worklist, and alias edges (n_3,x) and (n_3,$\mathtt{z}_1$) are added. This is shown in Figure 5.8(c); the newly added alias edges are highlighted in bold. Doing so requires three actions: first, we add the pair $\langle F_4, \{A_2\} \rangle$ to the worklist because n_3 is now an alias for `x`, and therefore F_4 should resolve to A_2—an application of the rule in Figure 3.16(b). Second, because $al(n_3) = \{\mathtt{x}, \mathtt{z}_1\}$ at this point, fetching n_3 through edge F_4 means indirectly fetching both `x` and $\mathtt{z}_1$. Our algorithm therefore creates two initial values: $\mathtt{x}_1$ and $\mathtt{z}_2$, and adds the assign edges $\mathtt{x} \xrightarrow{A} \mathtt{x}_1$ and $\mathtt{z}_1 \xrightarrow{A} \mathtt{z}_2$. In Figure 5.8(c) these edges are labeled $A_{\mathtt{x}}$ and $A_{\mathtt{z}_1}$ (highlighted). Third, the worklist is augmented with $\langle F_4, \{A_{\mathtt{x}}, A_{\mathtt{z}_1}\} \rangle$ for the same reason it was augmented with $\langle F_4, \{A_2\} \rangle$: applications of the rule in Figure 3.16(b). Since we group together common elements based on σ_F, this means the new worklist element is in fact $\langle F_4, \{A_2, A_{\mathtt{x}}, A_{\mathtt{z}_1}\} \rangle$.

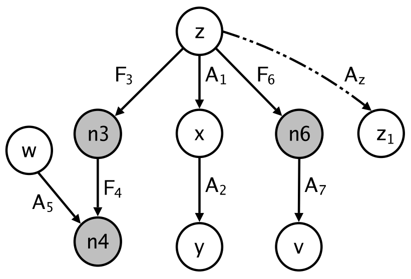
Assume the worklist is augmented at its head; $\langle F_4, \{A_2, A_{\mathtt{x}}, A_{\mathtt{z}_1}\} \rangle$ is hence the next to be processed. This adds alias edges (n_4, `y`), (n_4, $\mathtt{x}_1$) and (n_4, $\mathtt{z}_2$), depicted in Figure 5.8(d).

Next, $\langle F_6, \{A_1, A_{\mathtt{z}}\} \rangle$ is processed and alias edges (n_6, `x`) and (n_6, $\mathtt{z}_1$) are added (Figure 5.8(e)). The *new aliasing* discovered between n_6 and `x` requires us to return F_4 to the worklist, this time paired up with A_7—i.e, $\langle F_4, \{A_7\} \rangle$, an instance of Figure 3.16(d) with $\alpha = n_3$, $\gamma = n_6$, $n = n_4$, and $\beta =$ `y`, which is replaced by `v` in the example. This happens because $aliases(n_3, n_6)$ is true, i.e., $al(n_3) \cap al(n_6) \neq \emptyset$. Specifically, $\mathtt{x} \in al(n_3), al(n_6)$ (as well as $\mathtt{z}_1 \in al(n_3), al(n_6)$).

Processing $\langle F_4, \{A_7\} \rangle$ adds the alias edge (n_4, `v`) (Figure 5.8(f)) completing the resolution phase.

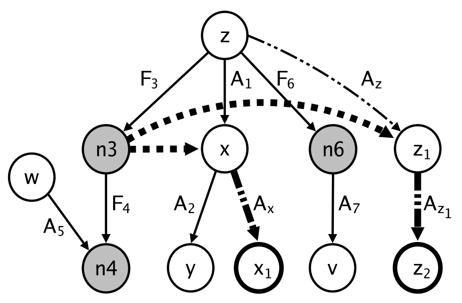
```
bar()
{
  z = &x; // A1
  x = &y; // A2
  w = *z; // F3 F4 A5
 *z = &v; // F6 A7
}
```

(a)



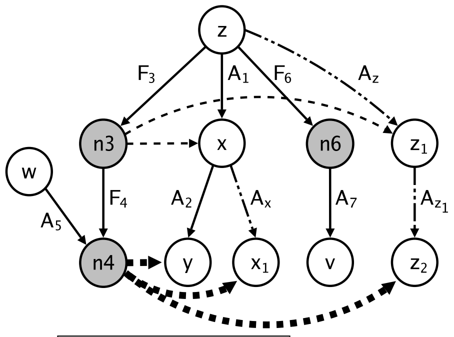
$\langle F_3, \{A_1, A_z\}\rangle$; $\langle F_6, \{A_1, A_z\}\rangle$

(b)



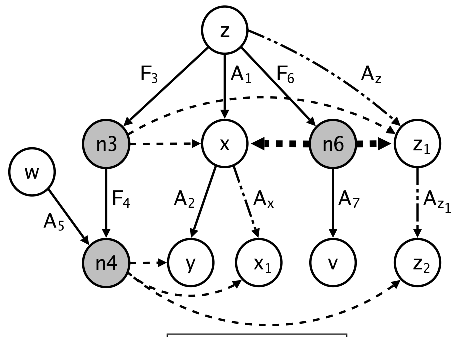
$\boxed{\langle F_3, \{A_1, A_z\}\rangle}$; $\langle F_6, \{A_1, A_z\}\rangle$

(c)



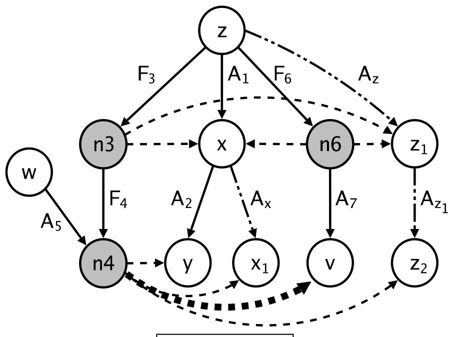
$\boxed{\langle F_4, \{A_2, A_x, A_{z_1}\}\rangle}$; $\langle F_6, \{A_1, A_z\}\rangle$

(d)



$\boxed{\langle F_6, \{A_1, A_z\}\rangle}$

(e)



$\boxed{\langle F_4, \{A_7\}\rangle}$

(f)

Figure 5.8: The worklist algorithm step-by-step. The boxed tuples correspond to the worklist element being processed.

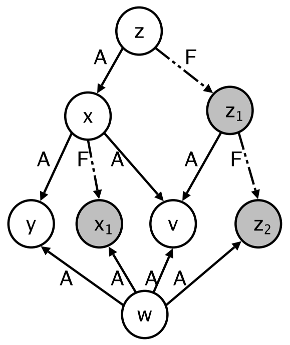


Figure 5.9: Summary AFG for the code in Figure 5.8.

Because $al(n_3) \cap al(n_6) \neq \emptyset$, both n_3 and n_6 are two aliases for at least one common location node $\mathtt{x}$. Therefore, any assignments to or dereferences of n_3 and n_6 indirectly affect each other.

We returned F_4 to the worklist because we need to arrive at a fixed point for the [ORIGIN-ALIAS] rule. In the graph of Figure 5.8(b), note that a solution with $al(n_3) = al(n_4) = al(n_6) = \emptyset$ does not satisfy the rule. A valid solution must include an alias edge, e.g., from n_3 to $\mathtt{x}$ for two reasons: there is a fetch edge $\mathtt{z} \xrightarrow{F} n_3$ and an assign edge $\mathtt{z} \xrightarrow{A} \mathtt{x}$, and $aliases(\mathtt{z}, \mathtt{z})$ is true. A similar argument demands the other alias edges in Figure 5.8(f).

We produce the summary AFG in Figure 5.9 by adding assignment edges around each fetch node based on its aliases, deleting all fetch nodes, and demoting assignment edges $\mathtt{x} \xrightarrow{A} \mathtt{x}_1$, $\mathtt{z} \xrightarrow{A} \mathtt{z}_1$ and $\mathtt{z}_1 \xrightarrow{A} \mathtt{z}_2$ to fetch edges, turning nodes $\mathtt{x}_1$, $\mathtt{z}_1$ and $\mathtt{z}_2$ into fetch nodes.

Note that flow-aware and flow-branch-aware analysis can be implemented by a simple variant of this worklist algorithm. Namely, a pair (σ_A, σ_F) is inserted into the worklist only if $affects_0(\sigma_A, \sigma_F)$ (in the case of flow-aware) or $affects_1(\sigma_A, \sigma_F)$ (flow-branch-aware) is true. For example, analyzing the above example follows almost the same steps as those described. However, the addition of alias edge (n_6, $\mathtt{x}$) does not trigger the insertion of $\langle F_4, \{A_7\}\rangle$ into the worklist in any of the more precise analysis. Indeed, both $affects_0(A_7, F_4)$ and $affects_1(A_7, F_4)$ are false.

Our algorithm adds elements to the worklist when *new aliases* are discovered by the analysis. This means newly-added alias edges will trigger the inference rule. Figure 5.10 shows a general case: a new alias edge $(n_i, \mathtt{x})$ is highlighted—due to this new alias, the worklist is augmented with $\langle F_1, \{A_5, A_6\}\rangle$, $\langle F_2, \{A_4\}\rangle$ and $\langle F_3, \{A_4\}\rangle$. These three elements are the only new facts that need to be considered because of the new aliasing. For this incremental procedure, node merging at summary instantiation can be seen as preceded by an alias edge from the callee node to the corresponding caller node.

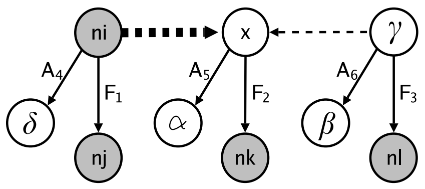


Figure 5.10: With the addition of new alias (n_i, x), the worklist is augmented with $\langle F_1, \{A_5,A_6\}\rangle$, $\langle F_2, \{A_4\}\rangle$ and $\langle F_3, \{A_4\}\rangle$.

Let $L(\mathtt{x}) = \{\varphi | \mathtt{x} \in al(\varphi)\}$ be the inverse of the *al* function. Since the range of *al* is sets of location nodes, L is only defined for location nodes. In Figure 5.10, $L(\mathtt{x}) = \{n_i, \mathtt{x}, \gamma\}$ (γ not necessarily distinct from x). When we add a new alias $(n_i, \mathtt{x})$, we add to the worklist fetch edges of the form (φ, n_j) where $\varphi \in L(\mathtt{x})$. In Figure 5.10, such set of edges corresponds to F_1, F_2 and F_3. The sets of assignments that pair up with each fetch edge (φ, n_j) are defined as follows. If $\varphi = n_i$, all $\eta \xrightarrow{A} \mu$ s.t. $\eta \in L(\mathtt{x}) \backslash n_i$. In Figure 5.10, this means $\langle F_1, \{A_5,A_6\}\rangle$. If $\varphi \neq n_i$, all $n_i \xrightarrow{A} \delta$. This means the other two worklist elements. Note that if no $n_i \xrightarrow{A} \delta$ exists, then both F_2 and F_3 in Figure 5.10 are not affected by the new alias $(n_i, \mathtt{x})$.

In the example of Figure 5.8, the new alias discovered between n_6 and x while processing $\langle F_6, \{A_1,A_\mathtt{z}\}\rangle$ augments the worklist with $\langle F_4, \{A_7\}\rangle$ (an instance of Figure 5.10 with $n_i = n_6$, x=x, and $\gamma = n_3$).

Note that F_4 corresponds to the second dereference of variable `z` in the statement `w=*z`. In traditional incremental techniques, the addition of a new alias would trigger the (re)analysis of at least one entire statement [64, 70]. In our finer-grain algorithm, it only triggers the local (re)analysis of *sub-expressions* within statements.

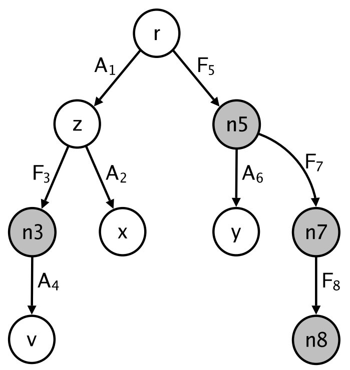
Recall that the initial AFG for a function has, as sub-graphs, the instantiated summaries for its callees. The incremental procedure illustrated above guarantees that a pair of fetch and assign edges that come from the same summary (an "internal pair") can never be (re)matched together unless an alias, assumed nonexistent while building the callee summary, is introduced at the call site. This is directly implemented by the process in Figure 5.10—only a *new* alias can trigger a pair $\langle \sigma_A, \sigma_F\rangle$ into the worklist. This guarantees that a function summary is not re-analyzed at each call site it is used—only local, incremental adjustments are made to account for eventual aliasing among environment locations previously assumed nonexistent.

5.5.1 Caching optimization

This section describes a simple optimization designed to avoid re-inserting into the worklist a pair of assign/fetch edges that has already been paired up in some prior iteration. Assume the initial AFG in Figure 5.11(a) along with the contents of the worklist before the analysis starts. For simplicity, we omit initial values. Figure 5.11(b) shows the first worklist element, $\langle F_3, \{A_2\}\rangle$, being analyzed. This causes alias edge (n_3,x) to be added.

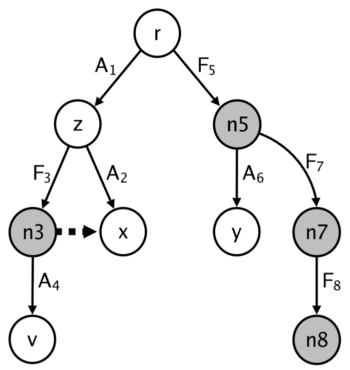
Figure 5.11(c) shows the matching of $\langle F_5, \{A_1\}\rangle$. Alias edge ($n_5$,z) is added, and the new aliasing causes two new pairs into the worklist: $\langle F_7, \{A_2\}\rangle$ and $\langle F_3, \{A_6\}\rangle$. Figure 5.11(d) illustrates $\langle F_7, \{A_6, A_2\}\rangle$ being handled. The new aliasing discovered between n_7 and x causes $\langle F_8, \{A_4\}\rangle$ to be added to the worklist; in Figure 5.11(e), this fetch/assign pair is matched up resulting in alias edge (n_8,v). Finally, in Figure 5.11(f) F_3 and A_6 are resolved, creating the aliasing between n_3 and y. According to Figure 5.10, this newly discovered aliasing would cause $\langle F_8, \{A_4\}\rangle$ to be (re)inserted into the worklist. This is however unnecessary: fetch edge F_8 and assign A_4 have already been resolved, and nothing can change with $al(v)$ in between; hence, there is no need to re-match F_8 and A_4. By keeping a cache of such fetch/assign pairs, the analysis can avoid doing redundant work and thus terminate faster.

A technicality: if the target node of A_4 was instead a fetch node n_i, and new aliases were added to such node since the last time A_4 was matched with F_8, then $\langle F_8, \{A_4\}\rangle$ would need to be re-evaluated (with respect to these new aliases) upon the addition of (n_3, y). Another solution is to allow alias edges to terminate at *fetch* nodes, e.g., from n_8 to n_i, similarly to what was allowed to return nodes in Section 5.4. Then, the first time F_8 and A_4 are resolved, we can rely on these "pseudo"-aliasing between fetch nodes to adjust new aliases—i.e., apply a transitive rule such that (n_i,n_j) and (n_j,x) implies (n_i, x).



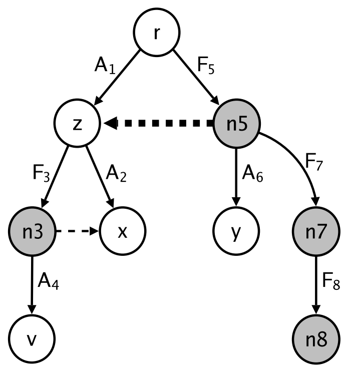
$\langle F_3, \{A_2\}\rangle$; $\langle F_5, \{A_1\}\rangle$; $\langle F_7, \{A_6\}\rangle$

(a)



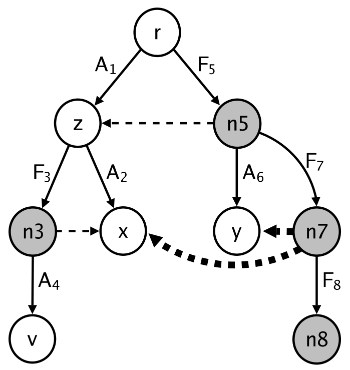
$\boxed{\langle F_3, \{A_2\}\rangle}$; $\langle F_5, \{A_1\}\rangle$; $\langle F_7, \{A_6\}\rangle$

(b)



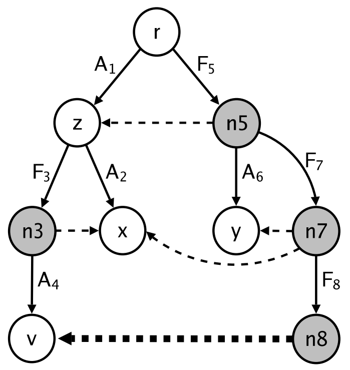
$\boxed{\langle F_5, \{A_1\}\rangle}$; $\langle F_7, \{A_6, A_2\}\rangle$; $\langle F_3, \{A_6\}\rangle$

(c)



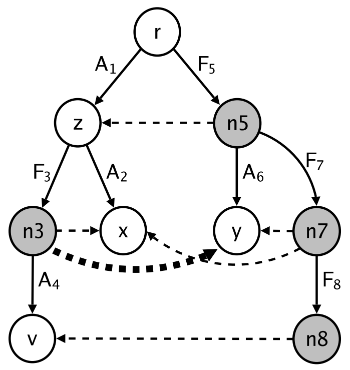
$\boxed{\langle F_7, \{A_6, A_2\}\rangle}$; $\langle F_8, \{A_4\}\rangle$; $\langle F_3, \{A_6\}\rangle$

(d)



$\boxed{\langle F_8, \{A_4\}\rangle}$; $\langle F_3, \{A_6\}\rangle$

(e)



$\boxed{\langle F_3, \{A_6\}\rangle}$; $\cancel{\langle F_8, \{A_4\}\rangle}$;

(f)

Figure 5.11: Because of caching, $\langle F_8, \{A_4\}\rangle$ does not need to be added to the worklist in (f) even though the new alias in bold would trigger such addition according to Figure 5.10.

5.6 A flow-aware example and ordering aliases

To increase the accuracy of our flow-aware analysis, we also maintain ordering information on the alias edges in the AFG. This information indicates at what point in the procedure's execution the alias is created. We use such information to make later aliases invisible to earlier fetches, much as we do for later assignments.

Reconsider the resolved graph in Figure 5.8(f). The initial value edge for z—assignment edge $A_{\mathtt{z}}$—is demoted into a fetch edge in Figure 5.9. This fetch edge condenses both F_3 and F_6 from Figure 5.8(f), given that either (or both) of them would set the initial value for z. To record this fact in the final summary, initial value edges are annotated with the interval that they represent—in Figure 5.9, the fetch edge $\mathtt{z} \xrightarrow{F} \mathtt{z}_1$—would be labeled as F_6^3. Operation edges can be seen as representing a unitary interval, e.g., A_1^1. We will refer to the superscript and subscript as *min* and *max* indices.

```
f(r,s,t)
{
  **r = &y;
   *s = &z;
  **t = &w;
}
```

(a)

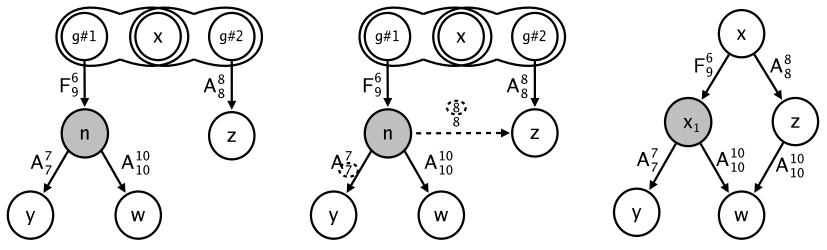
```
g(p,q)
{
  f(p,q,p);
}
```

(b)

```
h()
{
  g(&x,&x);
}
```

(c)

Figure 5.12: The code for procedures f, g and h.



(a) (b) (c)

Figure 5.13: (a) Initial, (b) resolved and (c) summary AFG for h.

Figure 5.13 illustrates alias ordering. Figure 5.13(a) represents the initial AFG for function h in Figure 5.12(c), obtained after the summaries for f and g have been computed. This is similar to the flow-insensitive example of Figure 5.6 which uses the same three functions. Figure 5.13(a) shows the call to g that occurs in the body of h.

The resolved AFG for h (Figure 5.13(b)) shows the alias edge derived by the resolution phase. Like the other edges in the graph, it has been annotated with two indices. Such indices are obtained from the assign edge participating in the resolution, here A_8^8.

Indices for alias edges are used when re-directing assign edges in the resolved AFG to form the summary AFG. To remove a fetch node n, we re-direct its incoming and outgoing assign edges according to $al(n)$. In the graph of Figure 5.13(b), this would mean creating two new assign edges: $\mathtt{z} \xrightarrow{A_7^7} \mathtt{y}$ and $\mathtt{z} \xrightarrow{A_{10}^{10}} \mathtt{w}$. However, edge $\mathtt{z} \xrightarrow{A_7^7} \mathtt{y}$ is a spurious pointer relation that emerges because multiple dereferences have been encapsulated into a single dereference (i.e., multiple fetches have been condensed into F_9^6 much as F_6^3 represents $A_{\mathtt{z}}$ in Figure 5.8(d)). Indeed, a quick look at Figure 5.12 verifies that z does not point to y because A_8^8, representing the assignment in `*s=&z`, occurs between A_7^7 and A_{10}^{10}, representing the assignments in `**r=&y` and `**t=&w`, respectively.

We avoid the spurious edge $\mathtt{z} \xrightarrow{A_7^7} \mathtt{y}$ by noting the *min* index for the resolution edge (n_1, z) is greater than the *max* index for A_7^7 (both encircled in Figure 5.13(b)). Intuitively, this means the aliasing between n and z in Figure 5.13(b) is created only after A_7^7 executes, and therefore $\mathtt{z} \xrightarrow{A_7^7} \mathtt{y}$ is invalid. Maintaining this additional ordering information increases the accuracy of our flow-aware analysis.

5.7 Field Sensitivity

There are two types of analyses according to how they handle fields. A field-*sensitive* analysis distinguishes between the different fields of a structure; a field-*insensitive* analysis collapses a structure into a single variable, so that statements `x.f1=&a` and `x.f2=&b` are treated as `x=&a` and `x=&b`.

This section gives an overview of field-sensitivity in pointer analysis, and then describes how we model struct fields and handle type casting on the AFG. Type casting allows a program to access an object as if it has a type different from its declared type, which complicates the design of a field-sensitive pointer analysis.

In addition, we present a new hybrid approach to field-sensitivity which allows us to have a tunable analysis depending on available resources.

5.7.1 Field-sensitive vs. -insensitive analyses

The example in Figure 5.14 illustrates a bug found in systems code that depends on a field-sensitive analysis to be caught. The real code has been simplified for the sake of presentation.

```
 1  graph_construction()
 2  {
 3   graph *G = new graph;
 4    ...
 5    delete G->correspondence;
 6
 7    if (condition) {
 8      graph_cleanup(G);
 9    }
10    ...
11  }
12
13  graph_cleanup(graph *g)
14  {
15    ...
16    if (g->correspondence) {
17      delete g->correspondence;
18    }
19    ...
20  }
```

Figure 5.14: A program error that depends on field-sensitive analysis to be discovered.

A double memory deallocation occurs if `condition` at line 7 is true—the "`correspondence`" field for `G` is deleted at line 5 and then checked at line 16. Because the field still points at a valid memory address, the condition "`if (g->correspondence)`" is true, and the `delete` statement at line 17 is executed causing the error.

It is not hard to detect such error if a summary-based, field-sensitive analysis is used. The summary for function `graph_cleanup()` contains the information that field `correspondence` of formal parameter `g` may have been deallocated. When the summary is instantiated in the caller, `graph_construction()`, the same field of `G` (the argument) has already been deallocated, a double-free.

Nevertheless, most pointer analysis algorithms, specially for C, are field-insensitive, and thus a write into one field is treated as writing to all the fields in the structure. Consider the following code fragment:

```
struct T {int *f1; int *f2; } x;
int a, b, *p;
if (...) {
  x.f1 = &a;
} else {
  x.f2 = &b;
}
p = x.f1;
```

A field-insensitive analysis treats these three assignments as shown below:

```
if (...) {
  x = &a;
} else {
  x = &b;
}
p = x;
```

Consequently, a points-to fact of the form *s points-to t* is interpreted as "*any field of s may point to any field of t*." The advantage of this approach is that it is more efficient specially in terms of memory space. A drawback is that it can produce imprecise points-to information. For example, for the code fragment above, the field-insensitive method identifies `p`'s points-to set as {`a`,`b`}, while in fact `p` only points to `a`. This loss of precision can have a negative impact on pointer clients.

In contrast, one can treat each field of a structure as a separate object to obtain a field-sensitive analysis. In such analysis, a store to a field via a pointer `p` only affects the objects pointed to by `p` and only in that field. In the above code fragment, `x.f1` and `x.f2` are two distinct locations, and `p`'s points-to set is correctly identified as {`a`}. The disadvantage of this method is that the graphs used to represent pointer relationships are larger, impacting both memory usage and analysis time.

Our goal with regard to field-sensitivity is to distinguish between different fields within a structure but not necessarily between different elements of an array. For that purpose, we represent struct fields by using offsets and sizes (i.e., strides). A field in a structure is identified by its offset in bits from the beginning of the structure (called the base address). A structure's stride corresponds to its total size in bits. Although strides are often used for arrays, we generalize the term to signify a structure's size because our approach is to first assume that all program variables are seen as unbounded arrays of their declared type, and later map each array's cells back to a single location.

To compute offsets and strides, we adopt an implementation dependent memory layout that is parametrizable. Type casts and unions are handled within such mechanism as explained in Section 5.7.5; a type cast from type T_1 to type T_2 is handled by calculating the greatest common divisor (*gcd*) between the strides of both types.

The next sections present the new graph elements and algorithms needed to include field-sensitivity in our analysis. Assume the usual sizes for pointers and scalar variables, i.e., 4 bytes for integer, 4 bytes for a pointer, etc.

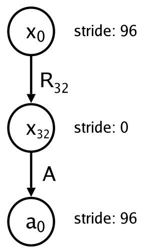
5.7.2 New graph elements

This section discusses the new AFG elements needed to handle fields. There is one additional edge type, called an *offset edge*, which represents a field access starting from a base address. In addition, each node contains an offset value—when a node's offset is not zero, the node is the target of an offset edge. In such case, we call the node an *offset node*. In contrast, a node with zero offset corresponds to a base address. Each node also carries a stride value. To illustrate, a simple case is shown in Figure 5.15.

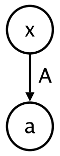
The single statement in Figure 5.15(a) generates the (field-sensitive) AFG in Figure 5.15(b). The offset edge labeled R_{32} corresponds to the field access in the expression `x.left`. The subscripts in nodes $\mathtt{x}_0$, $\mathtt{x}_{32}$, and $\mathtt{a}_0$ correspond to their offset values. I.e., $\mathtt{x}_{32}$ is the memory address that results from taking an offset of 32 bits from the base address of variable `x`, represented by $\mathtt{x}_0$. The stride value for each node is written outside the node.

```
struct T {          // Stride=96
  int       data;   // offset 0
  struct T *left;   // offset 32
  struct T *right;  // offset 64
};

struct T a;
struct T x;

void foo()
{
  x.left = &a;
}
```

(a)



(b)



(c)

Figure 5.15: (a) A function and (b) the new graph elements used to represent struct fields. (c) is the field-insensitive view of the code.

In contrast, the graph in Figure 5.15(c) is the field-*insensitive* AFG for the same code (since offset and stride values are unnecessary, they are not used to label nodes).

Note that $\mathtt{x}_0$ coincides with the field called `data` in `struct T`. I.e., an statement like `x.data=&b` generates an assign edge from node $\mathtt{x}_0$ to a location node `b`, with no need for an intermediate offset edge. Figure 5.16 illustrates the general structure of a field-sensitive graph.

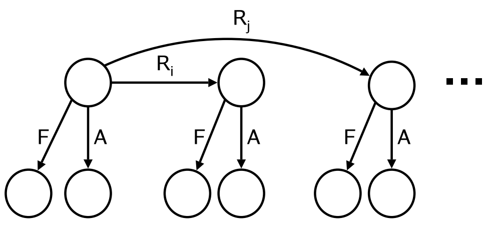


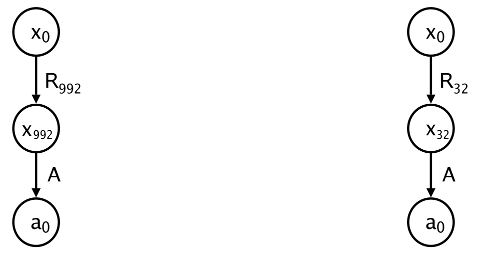
Figure 5.16: The general structure of a field-sensitive graph.

In general, offset edges always start at a base address, i.e., a chain of offset edges is converted into a single edge whose offset value is the sum of the individual components of the chain.

Figure 5.17 illustrates another example of field-sensitive graphs. The code in Figure 5.17(a) uses the same declaration of `struct T` from Figure 5.15.

```
struct T a;
struct T x[100];

void foo()
{
  x[10].left = &a;
}
```

(a)



(b) (c)

Figure 5.17: (a) A function, (b)–(c) field-sensitive AFGs.

In this example, the true offset value in the expression `x[10].left` is 992—i.e., (10×96)+32 bits from the base address `x`. Therefore, Figure 5.17(b) is the most accurate representation of function `foo`. However, an enumerative and exact approach to this would invalidate any termination guarantees. As an approximation, the representation of objects is extended to allow an object to wrap around onto itself. All accesses to an offset *f* are converted to accesses to *f mod s*, where *s* is the size (stride) of the base address. This is shown in Figure 5.17(c), and it means expressions `x.left`, `x[10].left`, and `x[i].left` are all equivalent, i.e., they all access an offset of 32 bits from node $\mathtt{x}_0$.

Some additional, self-explanatory examples of offset edges are shown in Figure 5.18.

5.7.3 "Fetch-" and "assign-" offset edges and the resolution phase

The examples of offset edges given so far were obtained via the dot operator '.' as in `x.left`. In general, such cases have a base address that is known at compile time, e.g., `x`. This means that both the base address node $\mathtt{x}_0$ and the offset node $\mathtt{x}_{32}$ are location nodes, i.e., they correspond to concrete memory addresses.

In contrast, the arrow operator '`p->`', or alternatively '`(*p).`' creates offset edges whose sink nodes represent unknown addresses. Such offset edges behave like fetch edges, and need to be resolved during analysis.

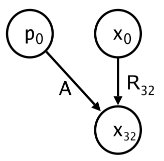
Consider the example in Figure 5.19. There are two unknowns to be determined in the AFG of Figure 5.19(b), i.e.,`*p` and `(*p).left`, represented respectively by *n*1 and *n*2. Now consider

```
struct T **p;
struct T x[100];

void foo()
{
  ...
  p = &(x[i].left);
}
```

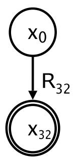
(a)

```
struct T x[100];

struct T ** foo()
{
  return &(x[5].left);
}
```

(b)



(c)

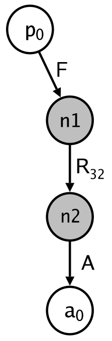


(d)

Figure 5.18: (a)–(b) Two functions, (b)–(c) their respective field-sensitive AFGs.

```
struct T  a;
struct T *p;

void foo()
{
  p->left = &a;
}
```

(a)



(b)

Figure 5.19: (a) A code fragment, (b) the offset edge labeled R$_{32}$ behaves like a fetch.

instantiating this graph into function `bar` as in Figure 5.20.

The fetch edge out of `p` gets resolved with the assign edge $\mathtt{p} \xrightarrow{A} \mathtt{x}_0$; this means the offset edge $n1 \xrightarrow{R_{32}} n2$ takes an offset from (base address) node $\mathtt{x}_0$, depicted by the edge labeled $\mathtt{x}_0 \xrightarrow{R_{32}} \mathtt{x}_{32}$. From the point of view of the analysis (i.e., the resolution phase inference rules) the former behaves like a fetch, and the later behaves like an assign. We refer to these different flavors of offset edges

```
struct T *p;
struct T  x;

void bar()
{
  p = &x;
  foo();
}
```

(a)

$$\begin{tikzcd} & p_0 \arrow[dl, "A"'] \arrow[dr, "F"] & \\ x_0 \arrow[d, very thick, "R_{32}"] & & n1 \arrow[ll, dashed] \arrow[d, "R_{32}"] \\ x_{32} & & n2 \arrow[ll, dashed] \arrow[d, "A"] \\ & & a_0 \end{tikzcd}$$

(b)

Figure 5.20: (a) Function `bar` calling function `foo` from Figure 5.19, (b) the fetch offset edge induces an assign offset edge (highlighted).

as fetch-offset and assign-offset edges.

It is not hard to incorporate field-sensitivity in our analysis through the use of such offset edges. As alluded to in Figure 5.20, the resolution phase has to handle offset edges in addition to regular fetches and assigns. The requirements to match a fetch-offset with an assign-offset are the same as in the inference rules presented in Chapter 3, with the additional condition that the offset values of the edges must be the same (modulo stride values).

Note that our field-sensitive analysis allows us to model expressions such as `&x.left` and `&info->var`, which explicit takes the address of a field. As far as we know, this is not addressed by other techniques.

Figures 5.21 and 5.22 show two additional examples of field-sensitive resolution.

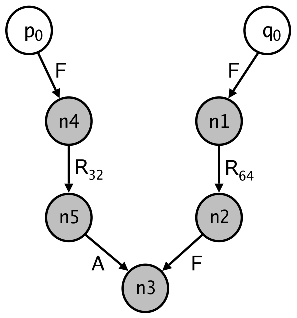
5.7.4 Strides, effective offsets and R_0 edges

We assign a stride value for each node, and thus the effective offset of an offset edge is the given offset modulo its source node's (i.e., base address's) stride. If the given offset is an exact multiple of the stride, the effective offset will be zero. To have a sound analysis, we allow a (temporary) offset edge with zero offset as shown in Figure 5.23.

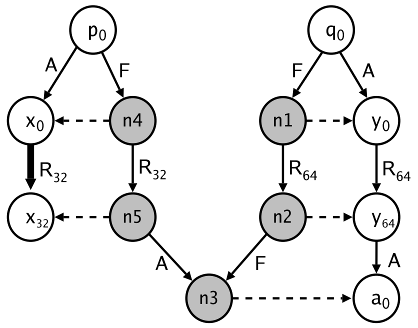
In this figure, the statement `p->right=&a`, following statement `p=&(x.left)`, takes an offset of 64 bits from the address of `x.left`. This is represented by the light gray edge labeled R_{64} in Figure 5.23(b). This in turn corresponds to taking an offset of 96 bits from the actual base

```
struct T *p, *q;
struct T  x,  y;
struct T  a;

void foo()
{
  p->left = q->right;
}

void bar()
{
  y.right = &a;
  p = &x;
  q = &y;
  foo();
}
```

(a)



(b)

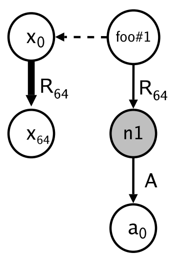


(c)

Figure 5.21: (a) A program fragment with two functions. (b) summary AFG for function `foo`. (c) resolved AFG for function `bar`.

```
struct T  x;
struct T  a;

void foo(struct T *p)
{
  p->right = &a;
}

void bar()
{
  foo(&x);
}
```

(a)



(b)

Figure 5.22: (a) A program fragment (b) the summary AFG for function `foo` being instantiated at the call site within `bar`. An offset edge out of node $\mathtt{x}_0$ is "induced" by the offset edge $foo\#1 \xrightarrow{R_{64}} n1$.

address, $\mathtt{x}_0$, and is represented by the offset edge labeled R_{96}. However, since the stride of node $\mathtt{x}_0$ is also 96, the effective offset for such edge is (96 mod 96)=0.

Nevertheless, the node labeled $n2$ in the figure needs to be resolved to a concrete address (which would be $\mathtt{x}_{96}$ in case of a bigger stride). To account for that, we create a temporary offset edge with zero offset as illustrated in Figure 5.23(c). During the resolution phase, the edge $n1 \xrightarrow{R_{64}} n2$ resolves to $\mathtt{x}_0 \xrightarrow{R_0} \mathtt{x}_0$, as depicted by the alias edge between $n2$ and $\mathtt{x}_0$ in Figure 5.23(c). The summary AFG for this example is shown in Figure 5.23(d) (for simplicity, we have kept the labels n_1 and n_2 on the fetch nodes).

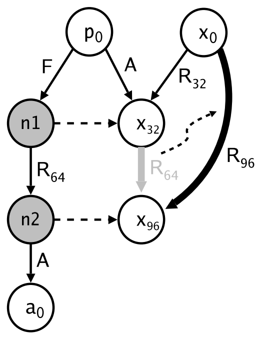
5.7.5 Type casting and strides

In a programming language like C, casting allows an object to be accessed as if it had a type different from its declared type. For structure types, casting further allows a different layout pattern to be used in place of the structure's declared pattern. This makes the design of a field-sensitive pointer analysis algorithm more challenging. In this section, we discuss how our AFG-based analysis handles casting by manipulating strides and offsets of nodes and edges.

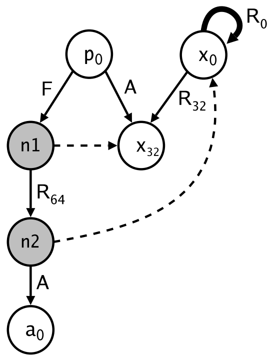
The basic idea is to update a node's stride whenever it has been the subject of casting. Initially,

```
struct T *p;
struct T x[];
struct T a;

p = &(x.left);
p->right = &a;
```

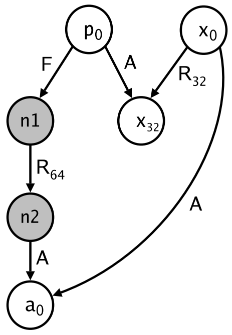
(a)



(b)



(c)



(d)

Figure 5.23: An example showing the computation of effective offset values and the need for R_0 edges. In (b) the offset edge $n1 \xrightarrow{R_{64}} n2$ would generally create an offset edge $x_{32} \xrightarrow{R_{64}} x_{96}$, but because x_{32}'s base address is x_0, the resulting edge is in fact $x_0 \xrightarrow{R_{96}} x_{96}$. (c) because x_0's stride is 96, the effective offset for edge R_{96} is actually 0, and a self-loop, zero offset is created to replace R_{96}. Then, edge $n1 \xrightarrow{R_{64}} n2$ matches edge $x_0 \xrightarrow{R_0} x_0$ during resolution, generating the alias edge (n2, x_0). (d) shows that the second assignment in (a) is considered in the final summary.

the node's stride is the size of its declared type. When the the node is casted to another type, we update its stride by computing the greatest common divisor between the current and the new strides. This may require that all the node's (outgoing) offset edges be also updated, since the computation offset modulo new stride may change. An example is illustrated in Figures 5.24 and 5.25.

Figure 5.25(b) shows the initial AFG for function `main` in Figure 5.24. In particular, $\mathtt{x}_0$'s initial stride is the size of its declared type, i.e., 96. The offset value of the edge representing `x.left` is thus (32 mod 96) = 32, depicted by $\mathtt{x}_0 \xrightarrow{R_{32}} \mathtt{x}_{32}$.

Assume that the fetch edge $\mathtt{p}_0 \xrightarrow{F} n1$ gets resolved, making $n1$ an alias for $\mathtt{x}_0$ (Figure 5.25(c)). Note that such fetch corresponds to `y=&(*p).c`, which follows statement `p=(struct T2 *)&x` where `x` is cast. This means that $\mathtt{x}_0$'s stride needs to be updated to reflect the fact that it has been accessed via different types. We do this by computing the *gcd* as indicated in Figure 5.25(c), where *gcd*(96,48)=48. Structurally, the alias edge between $n1$, whose stride is 48, and $\mathtt{x}_0$, whose stride is 96, triggers the update.

The new aliasing discovered between $n1$ and $\mathtt{x}_0$ also makes the offset edge labeled R_{40} to induce an offset of (40 mod 48) = 40 bits from $\mathtt{x}_0$. This is indicated by the edge labeled $\mathtt{x}_0 \xrightarrow{R_{40}} \mathtt{x}_{40}$ highlighted in bold.

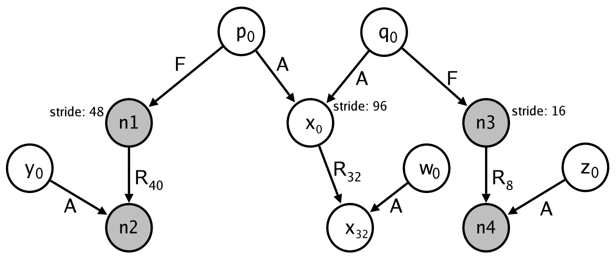
A similar situation occurs when fetch edge $\mathtt{q}_0 \xrightarrow{F} n3$, representing `*q` in statement `z=&(*q).e`, is resolved with $\mathtt{q}_0 \xrightarrow{A} \mathtt{x}_0$ (Figure 5.25(d)). Because $\mathtt{x}_0$ is now casted as `struct T3`, whose size is 16, its new stride is *gcd*(48,16) = 16, as indicated in the figure. The implication in this case is that the offset edges out of $\mathtt{x}_0$ need to be updated since (32 mod 16) = 0 and (40 mod 16) = 8. Such updates are depicted in Figure 5.25(e), where two new offset edges R_0 and R_8 replace R_{32} and R_{40}, respectively.

The rationale for the gcd approach can be explained via Figure 5.26, which shows the memory layout for unbounded arrays of types `T1`, `T2` and `T3`. One of our assumptions is that a node becomes "dirty" whenever it has been the subject of casting, since its declared type becomes somewhat irrelevant. Because different struct types have different sizes and layout patterns, we try to preserve a common access pattern that would make sense for all casted types. For instance, accessing field `c` of `struct T2` at statement `y=&(*p).c`, which we will re-write as `x.c`, is akin to accessing `x.e` within `struct T3`, as indicated in the figure.

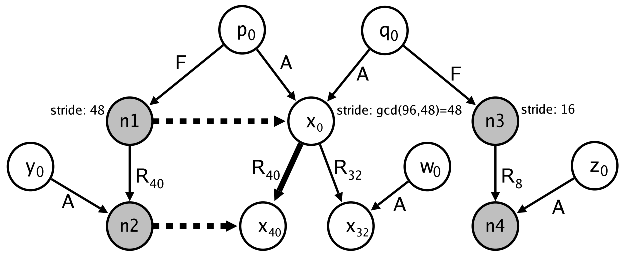
Figure 5.27 shows two possible summary AFGs for the program in Figure 5.24. Figure 5.27(a)

```
struct T1 {
  int        data;
  struct T1 *left;
  struct T1 *right;
};

struct T2 {
  int  a;
  char b;
  char c;
};

struct T3 {
  char d;
  char e;
};

struct T1  x;
struct T1 *w;
struct T2 *p;
struct T2 *y;
struct T3 *q;
struct T3 *z;

int main()
{
  p = (struct T2 *)&x;
  y = &(*p).c;   /* As if we did x.c with x of type T2 (x40)   */

  q = (struct T3 *)&x;
  z = &(*q).e;   /* As if we did x.e with x of type T3  (x8)   */

  w = &(x.left); /* x.left with x is of original type T1 (x32) */
}
```

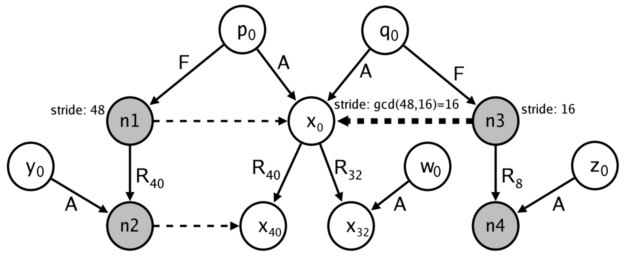
Figure 5.24: An example illustrating the manipulation of type casting (continues in Figure 5.25).



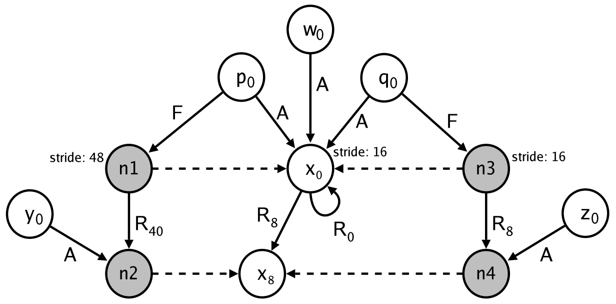
(b)



(c)



(d)



(e)

Figure 5.25: Continued from Figure 5.24. (b) the initial AFG for function `main`; the stride values for nodes $n1$, x_0 and $n3$ are indicated. (c) The alias edge between $n1$ and x_0 enforces an stride update for x_0 (indicated in the figure). Offset edge $n1 \xrightarrow{R_{40}} n2$ induces edge $\mathrm{x}_0 \xrightarrow{R_{40}} \mathrm{x}_{40}$. (d) alias edge ($n3$, x_0) causes another stride update for x_0, this time setting its value to 16. (e) the newest stride for x_0 causes its outgoing offset edges to be revised since (32 mod 16) = 0 and (40 mod 16)=8.

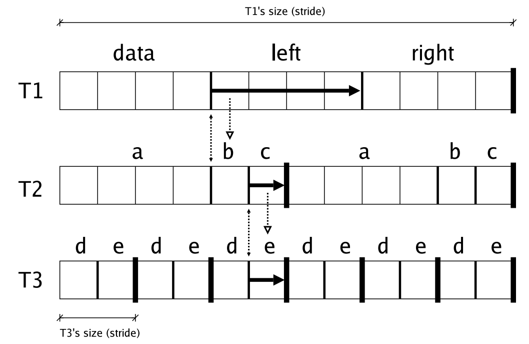


Figure 5.26: The memory layouts for the structure types T1, T2 and T3 from Figure 5.24.

is the graph obtained when casting is only partially considered (i.e., a node's stride is not updated as a result of casting), and Figure 5.27(b) shows the result obtained when casting is fully considered. In Figure 5.27(a) none of y_0, z_0 and w_0 point-to the same objects. In Figure 5.27(b) both y_0 and z_0 point-to x_8 and w_0 point to x_0. Our implementation can generate either result depending on a user's provided parameter.

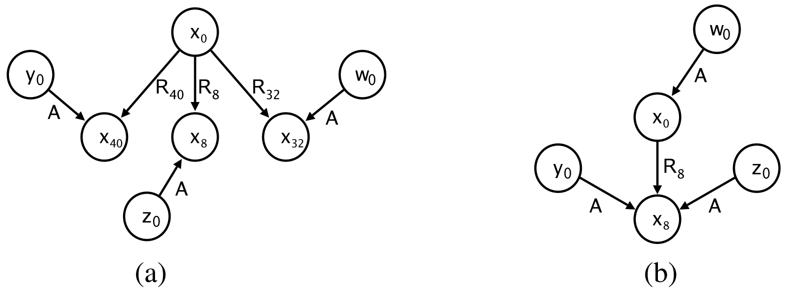


Figure 5.27: Resulting graph when type casting is (a) partially considered and (b) fully considered.

5.7.6 Tunable field-sensitivity

Most existing pointer analysis are either field-sensitive or field-insensitive. In this dissertation, we allow a hybrid approach where some pointers are tracked field-sensitively while others may get collapsed into a single location. The motivation is to keep the graphs' sizes under control while providing field-sensitivity as much as possible.

The basic idea is to establish a limit on the number of outgoing offset edges that a node is allowed to have. If that limit is exceeded, the node and its corresponding fields are collapsed together, subsequently treating the node field-insensitively. The limit is a user's provided parameter.

The implementation of such rule is straightforward: by setting a node's stride to 1, we guarantee that any offset edge out of the node has zero offset, since (k *mod* 1)=0 for any k. This is implemented as a temporary self-loop as discussed in Section 5.7.4. This means that the node is effectively treated field-insensitively, while the rest of the graph is treated field-sensitively. The above also happens when the gcd of two stride values in a type casting is 1.

5.7.7 Chains of offset edges

In Section 5.7.2 we mentioned that offset edges cannot form chains. There is one possible exception to this rule in that offset edges whose sink nodes represent unknown addresses may form *temporary* chains. This is more than a pragmatic issue specific to our implementation. A bottom-up summary-based analysis that wants to track struct fields accurately has to deal with elements (e.g., nodes) representing unknown values that map to fields; such non-concrete addresses cause uncertainty about which locations are accessed through a succession of offset-taken operations, which lead to chains (of edges, in our case). Figure 5.28 illustrates one such example.

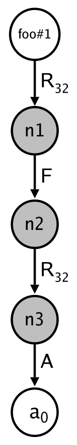
Figure 5.28(a) shows the code for a recursive procedure, `foo`, while Figure 5.28(b) shows the summary AFG for the procedure after the first iteration of the analysis. The recursive call does not have any effect at this point because there's no previous summary computed for `foo`.

Figure 5.28(c) shows the summary graph for `foo` being constructed during the second iteration of the algorithm. This time, the summary computed in Figure 5.28(b) is used at the recursive call to produce the new summary. Note that there's a chain of two offset edges—both labeled R_{32}—starting at the node labeled `foo`#1 (the edges highlighted in bold). This is caused by the recursive nature of `foo`, which takes an offset of 32 bits from a location which is itself an offset from some unknown address.

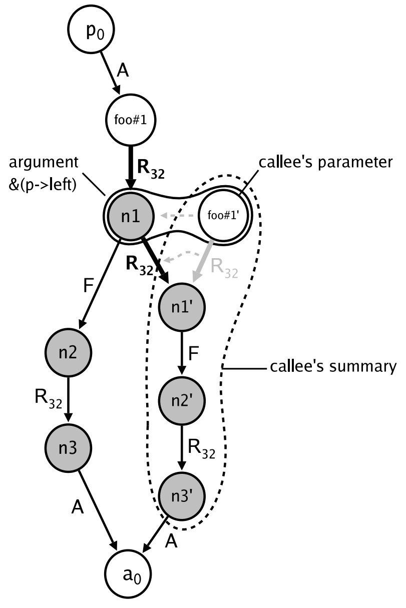
Assume that Figure 5.28(c) results in the final summary for `foo`, and that a caller invokes `foo(&x)`. The resulting summary instantiation is shown in Figure 5.28(d). Similar to Figure 5.23, instead of generating an edge $\mathtt{x}_{32} \xrightarrow{R_{32}} \mathtt{x}_{64}$ (in light gray), the second offset edge in the chain induces an edge $\mathtt{x}_{0} \xrightarrow{R_{64}} \mathtt{x}_{64}$. That is, after a concrete base address has been determined ($\mathtt{x}_0$), the succession of offset-taken operations is converted into a single operation as shown in the figure.

```
void foo(struct T * p)
{
  p->left->left = &a;
  ...
  foo(&(p->left));
}
```

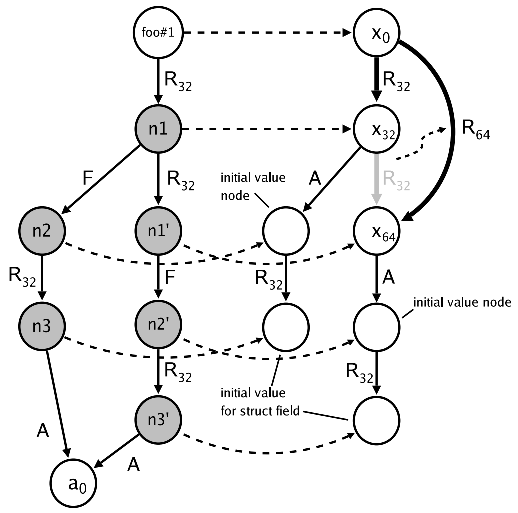
(a)



(b)



(c)



(d)

Figure 5.28: (a) A recursive function to be summarized. (b) The summary AFG after the first iteration of the fixed-point computation (c) During the second iteration, a chain of offset edges is created (highlighted in bold). (d) Assuming that (c) is the final summary for `foo`, and that it is called as `foo(&x)`, the chain of offset edges is converted into a single edge when a concrete base address, `x`, is identified.

5.8 Heap Modeling

This section describes how heap locations are modeled in our analysis, discussing two naming schemes for heap allocated objects that have been implemented.

Most analyses model the heap (a potentially infinite structure) by using a graph of bounded size [13, 18, 51, 60, 62, 73, 75]. Of course, using graph nodes to represent memory locations is only possible when pointers point to named objects. For *anonymous* objects in a heap storage returned by allocators such as `malloc` in C, a *naming scheme* is needed. A number of naming schemes have been proposed [13, 17, 61], and the main difference is their ability to distinguish among objects created at different invocations of the `malloc` function, or else, their ability to distinguish different *instances* of objects created at the *same* `malloc` call site. Three major methods can be identified:

- One name. Assign only one name to the entire heap space. The obvious consequence of such method is that all heap-oriented pointers are all aliased together.
- Line numbers, or allocation site. Roughly, a memory block is named after the line number of the allocating statement. This is a common scheme used by some existing industrial tools. Such naming scheme allows different memory blocks allocated in different statements to be disassociated, a significant improvement over using only one name. However, it cannot differentiate instances of memory blocks allocated by the same statement.
- Calling paths. To improve the precision of the analysis, one can name dynamically allocated objects by their calling paths in addition to their statement line numbers. The calling path is the sequence of call sites from the `main` function to the malloc invocation. A limit on the length of a calling path can be established in order to make the analysis feasible. When a calling path is partially used, the call sites are often selected backwardly, instead of starting from the program's entry point.

Alternatively, some analyses do not use any graph at all to model the heap: they manipulate sets of pairs of aliased *access paths* (i.e., chains of field dereferences starting from pointer variables that lead to the same object) [15, 26]. This does not require a naming scheme.

However, storing an explicit graph requires less memory space than manipulating its possible sets of aliases access paths [15] (the number of paths is larger then the number of edges). Also, a graph representation of the memory contents offers a natural understanding of the resident data structures. In our framework we adopt the explicit graph model.

Because we must bound the graph's size, we need to merge nodes together. The choice of which nodes to merge is intimately connected to the naming scheme adopted. One technique [13] uses the allocation site model. This means that all objects created at the same `malloc` call site are merged together. A variant approach is *k-limiting* [17, 39, 43, 47, 49, 61]: it uses a bound, *k*, on the maximum acyclic path length in the heap. If all elements in this path are allocated by the same call site, then the bound *k* basically sets the number of instances of the heap object that are allowed to have distinct names. Nodes that would cause the limit to be exceeded are merged together.

The basic goal of a useful naming scheme is to merge nodes that are have similar attributes during program's execution; merging unrelated nodes results in analysis imprecision. The next subsections presents two naming schemes and corresponding merging strategies we have developed towards this goal, both based on variants of the calling path method.

5.8.1 Naming scheme 1

In this section, we describe a naming scheme that is based on the place of the allocating statement combined with a variant of the *k*-limiting approach. Similar to Choi et. al. [17], the naming method qualifies anonymous objects with an additional string that captures call path information. In contrast to Choi et. al., two qualified names are "compatible" if and only if they share a common prefix *and* a common suffix. When this happens, we say one of the nodes is a "mirror" of the other.

The naming strategy works as follows. The summary for a call to malloc is a fresh, anonymous node marked as "heap." When a summary for a function that calls malloc is instantiated at a call site, heap nodes within this summary are prefixed with the caller's name indexed by a call site index (each procedure numbers its function calls $1,..,n$). When the summary for the caller is itself instantiated into a grand-caller, each heap node is given another prefix (the name of the grand-caller), and so on. Call stack information is recorded by such strategy in a bottom-up manner. (Mutually) recursive functions will cause the names of heap nodes to be prefixed indefinitely. A limit *k* is then used to set the maximum number of times the *same function name* can appear in a node's name. When a node exceeds the limit, it is merged with its mirror, if one exists, or its name is re-assigned based on the name its mirror would have. Note that a node can exceed the limit only when it is prefixed with the name of a caller. This happens when the node already has k appearances of such caller's name; we will refer to such offending caller function as the *leading function*.

The remaining details are simpler to explain through an example, where $k = 2$ will be adopted. Figure 5.29(a) shows three mutually recursive functions, `f`, `h`, and `z`. Assume the analysis starts by `f` and continues in the following order: `h`, `z`, `f`, `h`, `z`, and so on, until until the summary for each function converges. In this and the following figures, heap nodes will be drawn as lightly shaded circles.

Figure 5.29(b) shows the summary graph for `f` during the first iteration of the analysis. The two function calls within `f`'s body result in empty summary instantiations since none of the callees has a summary yet. The summary for `f` is thus equivalent to a function `f(){ return malloc(..);}`. This is indicated by the heap node in Figure 5.29(b). The name $\texttt{f}_1$ is obtained when the summary for malloc, an anonymous heap node, is instantiated at the call site `*p=malloc(..)`. The index 1 is because such call site is the first within `f`.

Figures 5.29(c) and 5.29(d) show the summaries for `h` and `z`, respectively, after the first iteration is completed. When instantiating `f`'s summary within `h` and `z`, the heap node returned by `f` is given the prefixes $\texttt{h}_1$ and $\texttt{z}_1$, respectively.

The second iteration of the analysis is shown in Figure 5.29(e), (f), (g), and (h). The first two illustrate the initial and summary AFGs for `f`. At this time, both `h` and `z` have summaries computed in the previous iteration, and they are instantiated at the respective call sites as depicted in Figure 5.29(e). The names for the two heap nodes from the callees are prefixed with $\texttt{f}_2$ and $\texttt{f}_3$, respectively. In addition, `f` itself calls malloc generating a second "incarnation" of node $\texttt{f}_1$. The summary AFG for `f`, depicted in Figure 5.29(f), has all nodes within the limit 2. Such graph is then used to produce the summaries for `h` and `z` in Figures 5.29(g) and 5.29(h), which completes the second iteration.

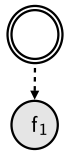
Figure 5.30 shows the analysis of `f` during the third iteration. In Figure 5.30(a), the summaries for `h` and `z` from Figures 5.29(g) and 5.29(h) are instantiated, and an additional incarnation of $\texttt{f}_1$ results from malloc. In Figure 5.30(b), a tentative summary for `f` is computed after the resolution and summarization phases finish. Note that all the nodes in the bottom of this figure exceed the limit of 2—`f` appears three times in each. We thus have to merge these nodes with their respective mirrors. Intuitively, the mirror m of a node n (n created at the i-th iteration of the analysis) is an

```
int *f()
{
  int *p = malloc(..);
  h(p);
  z(p);
  return p;
}
```

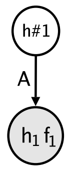
```
void h(p)
{
  *p = f();
}
```

```
void z(p)
{
  *p = f();
}
```

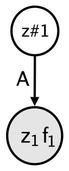
(a)



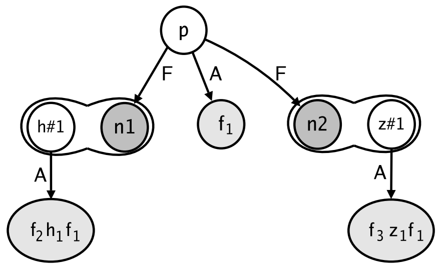
(b)



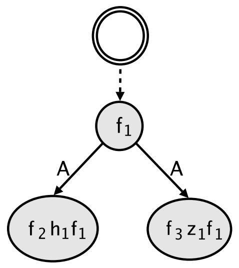
(c)



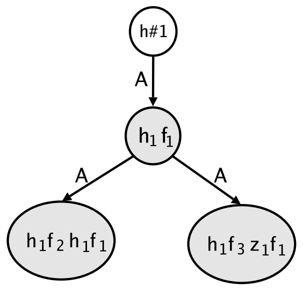
(d)



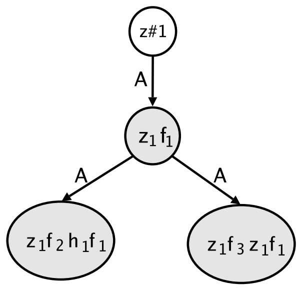
(e)



(f)



(g)



(h)

Figure 5.29: The analysis with heap nodes using naming scheme 1. (a) the code for three functions, `f`, `h`, and `z`; (b),(c) and (d) the summary AFGs for `f`, `h`, and `z`, respectively, after the first iteration of the analysis; (e) the initial AFG for `f` for the second iteration—the previous summaries for `h` and `z` are instantiated; (f) the summary for `f` at the second iteration; (g)—(h) the summary AFGs for `h` and `z` after the second iteration.

incarnation of n that is created at a later iteration. The mirror m re-traces part of the call path taken by the node n during summary propagation; this is captured by common prefix and suffix strings, whose lengths depend on k. Such strategy strives at merging nodes only if they have similar attributes in the program.

Take for instance node $f_3z_1f_2h_1f_1$ in Figure 5.30(b). Its mirror is named $f_3z_1f_1$. The computation of such mirror goes as follows. First, we compute what we call the k-prefix of the node's name. Such prefix depends on both k and the name of the leading function, in our example `f`. Precisely, we allow the k-prefix to have $k-1$ appearances of the leading function, i.e., the prefix extends from the beginning of the string all the way to where the k-th appearance of the leading function occurs, exclusive. In our example with $k = 2$ the prefix is f_3z_1. Then, we skip everything including the k-th appearance of the leading function up until the next occurrence of that name (if any). In our example, we skip f_2h_1. Finally, we concatenate the prefix with the remaining suffix, f_1, to generate the string $f_3z_1f_1$. A node with such a name is searched (through a hash table) and then merged with the offending node; the resulting node keeps the shorter name $f_3z_1f_1$.

These merging is indicated in Figure 5.30(b). Both nodes $f_2h_1\xcancel{f_2h_1}f_1$ and $f_2h_1\xcancel{f_3z_1}f_1$ have the same mirror $f_2h_1f_1$. The same happens for nodes $f_3z_1\xcancel{f_2h_1}f_1$ and $f_3z_1\xcancel{f_3z_1}f_1$ whose mirror is $f_3z_1f_1$. The resulting graph is depicted in Figure 5.30(c).

Note the underlying shape of the data structure is preserved by this naming scheme/merging strategy. Also, apart from the self-loop edges in Figure 5.30(c), the final summary for `f` in this figure is identical to the one obtained at the end of the second iteration (Figure 5.29(f)). The analysis will converge at the next iteration, and the summaries for `f`, `h`, and `z` will all have the structure in Figure 5.30(c).

Note that depending upon which function (within a set of mutually recursive functions) the analysis starts, it may be possible that no mirror exists for a given node. In that case, the node itself has its name re-assigned, and it becomes the otherwise missing mirror in a later iteration.

Also observe that if the k-limit is 1 we can never have a node with 3 occurrences of the same function name—the node exceeds the limit whenever it has 2 instances of the leading name, and it is thus merged with its mirror at that point. Thus, any heap node in the graph, at any given point, can either have its name below the limit, equal to the limit, or offending the limit by *one* unit, in which case it is merged with its mirror. For instance, if $k = 1$ and the node's name is $f_3z_1f_1$, then its mirror's is f_1. Indeed, allowing $k-1=0$ occurrences of f makes the k-prefix f_3z_1 to be discarded, along with the *empty* string that spans from the last character of the prefix all the way to the next occurrence of f, resulting in the suffix f_1.

In general, by choosing to merge only nodes that have a common suffix, we do not merge nodes that would (or could) be given different type declarations.

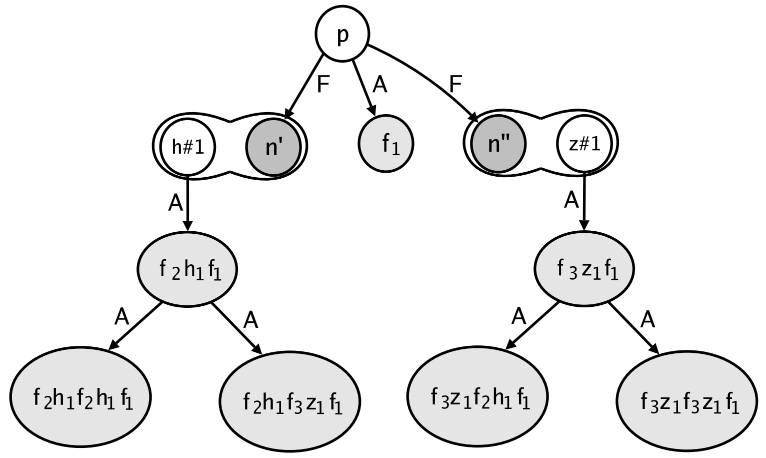
5.8.2 Naming scheme 2

The naming scheme presented in this section is more aggressive in its strategy to merge nodes. Its objective is to alleviate the potential growth in the number of nodes that may be caused by the previous method. Its drawback lies in the fact that unrelated nodes may get merged together, compromising the analysis precision. Basically, instead of limiting the number of occurrences of the same function name, this naming scheme limits the *absolute number* of function names in the string representing a node's name.

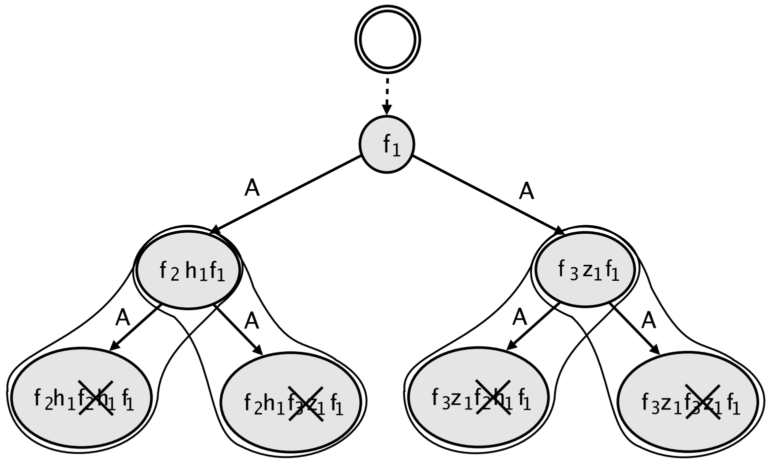
Two limiting values are adopted here. The *stack limit* constrains the absolute size of the call stack. The *merge depth* indicates how aggressive node merging will be. It establishes the size of a common suffix that two nodes must have in order to be merged together. The size is in terms of absolute number of functions. The larger the merge depth, the less aggressive the merging, since less nodes are likely to share a larger suffix.

The basic process of attaching prefixes to heap nodes when instantiating them at call sites is the same as in the previous section. However, when a node n exceeds the *stack limit*, a suffix s with size merge depth (in number of functions) is extracted from n's name. Then, all nodes in the graph such that (1) have a name with size at least merge depth, and (2) have an identical suffix s, are merged with n. Define $M(\alpha, \beta, \textit{merge depth})$ as a function that, given a heap node α exceeding the stack limit, returns true in case β obeys the two conditions above. In that case, α and β are merged together. It may happen that no other node in the graph shares the suffix s with α. In this case, α's name is re-assigned such that it stays within the stack limit. The new name is constructed by concatenating α's leading function name, indexed by a fictitious call site of 0, with s.

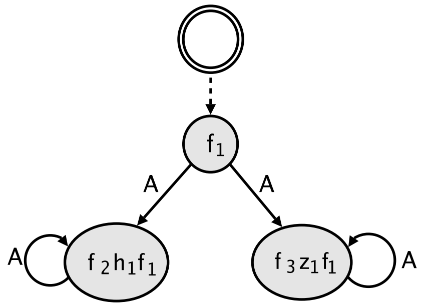
Reusing the code for functions `f`, `h`, and `z` of last section, Figure 5.31 shows the graphs obtained when a stack limit of 3 and a merge depth of 2 is used. The first five graphs in Figure 5.29 are identical to the ones obtained by the new merging scheme. Figure 5.31 shows where they differ.



(a)



(b)



(c)

Figure 5.30: The summarization of `f` during the third iteration of the analysis. (a) instantiating the summaries for `h` and `z`; (b) the resulting graph after resolution and clean-up phases—all nodes in the bottom row exceed the limit of 2 because *f* appears three times in each; (c) the resulting graph after the node merging indicated in (b) takes place.

For example, when the summary for h is built during the second iteration of the analysis (Figure 5.31(a)), two nodes exceed the stack limit: $h_1f_2h_1f_1$ and $h_1f_3z_1f_1$ (both strings contain 4 function names). For the first node, the suffix s mentioned above is h_1f_1, and there is a node in the graph with a large enough name sharing this suffix. The merging is indicated in Figure 5.31(a). For the second offending node, the suffix s is z_1f_1. Since no other node in the graph has a common suffix, $h_1f_3z_1f_1$ becomes $h_0z_1f_1$. The resulting graph is shown in Figure 5.31(b). Figures 5.31(c) and 5.31(d) show the respective situation for z's summary.

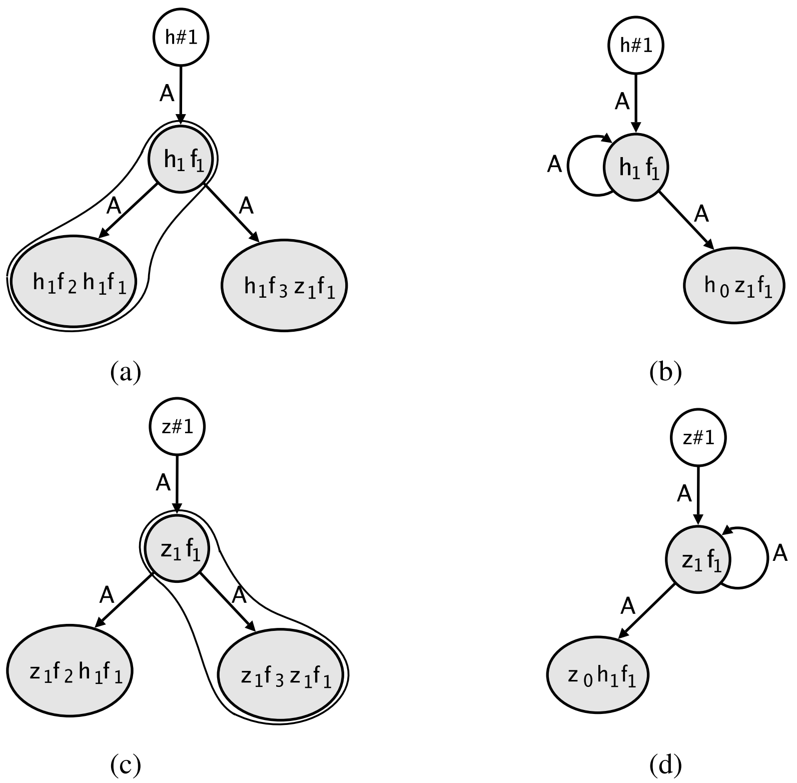


Figure 5.31: The second naming scheme merges nodes based on absolute number of functions in a string. (a) node $h_1f_2h_1f_1$ exceeds the assumed stack limit of 3. It is then merged with h_1f_1 which shares a suffix of size (merge depth) 2; (b) the resulting graph after the merging; (c)—(d) the respective situation for function z.

Figure 5.32 corresponds to the third iteration of the analysis, and resembles what we had encountered in Figure 5.30. In Figure 5.32(a), the summaries for h and z from Figures 5.31(b) and 5.31(d) are instantiated at the call sites h(p) and z(p). Figure 5.32(b) shows a tentative final summary for f. Two additional nodes exceed the stack limit, and are thus merged with two other nodes in the graph as indicated (both $M(f_2h_0z_1f_1, f_3z_1f_1, 2)$ and $M(f_3z_0h_1f_1, f_2h_1f_1, 2)$ hold). The final summary for `f` is depicted in Figure 5.32(c). Observe it does not maintain the data structure's shape as in Figure 5.30(c).

5.8.3 The call graph, *a.k.a* automata, view

This section evaluates the above naming schemes by using some elements from automata theory. Basically, the general mechanism for both naming schemes can be viewed as heap nodes propagating over the program's call graph, as indicated in Figure 5.33. This figure investigates the first naming scheme with *k-limit*=2, and it shows a strongly connected component of a call graph with three functions, `f`, `h`, `l`. In addition, procedure `alloc` is a heap storage allocator. Labels on call graph edges indicate call site indices. Assume the SCC is iterated in the following order: `f`, `h`, `l`, `f`, `h`, `l`, and so on, until convergence.

Figure 5.33(a) illustrates the set of heap nodes created during the first analysis iteration, named f_1, h_1f_1, $l_1h_1f_1$, and $l_2h_1f_1$.

As implied by the figure, we can depict heap nodes as walking (backwards) along call graph edges. Each new edge traversed prefixes a node's name with the edge's source node (i.e., the caller) indexed with the edge's label. This reproduces the bottom-up characteristic of the analysis where summaries for callees are instantiated inside callers. Heap nodes are replicated at call graph nodes with multiple predecessors, as is the case for function `h`.

The second iteration of the analysis is shown in Figure 5.33(b). When `f` is analyzed, heap nodes coming from both successors in the call graph account for f_1, $f_2l_1h_1$, and $f_2l_2h_1f_1$. The latter two originate at function l from Figure 5.33(a). These three nodes percolate along the edges of the automata to result in the heap locations indicated in the figure.

By the third time `f` is visited (Figure 5.33(c)), four nodes exceed the *k-limit* (i.e., the maximum number of occurrences of the same function name). The merging strategy based on common prefixes and suffixes (Section 5.8.1) collapses the two sets of three nodes into two heap locations as indicated. The result for function `f` is identical to the one obtained in Figure 5.33(b), and therefore the analysis converges (Figure 5.33(d)).

The basis for this naming scheme is akin to string generation in a finite automaton. Strings in the automaton's language are the names for heap locations. Informally, the mirror string S_m for a string

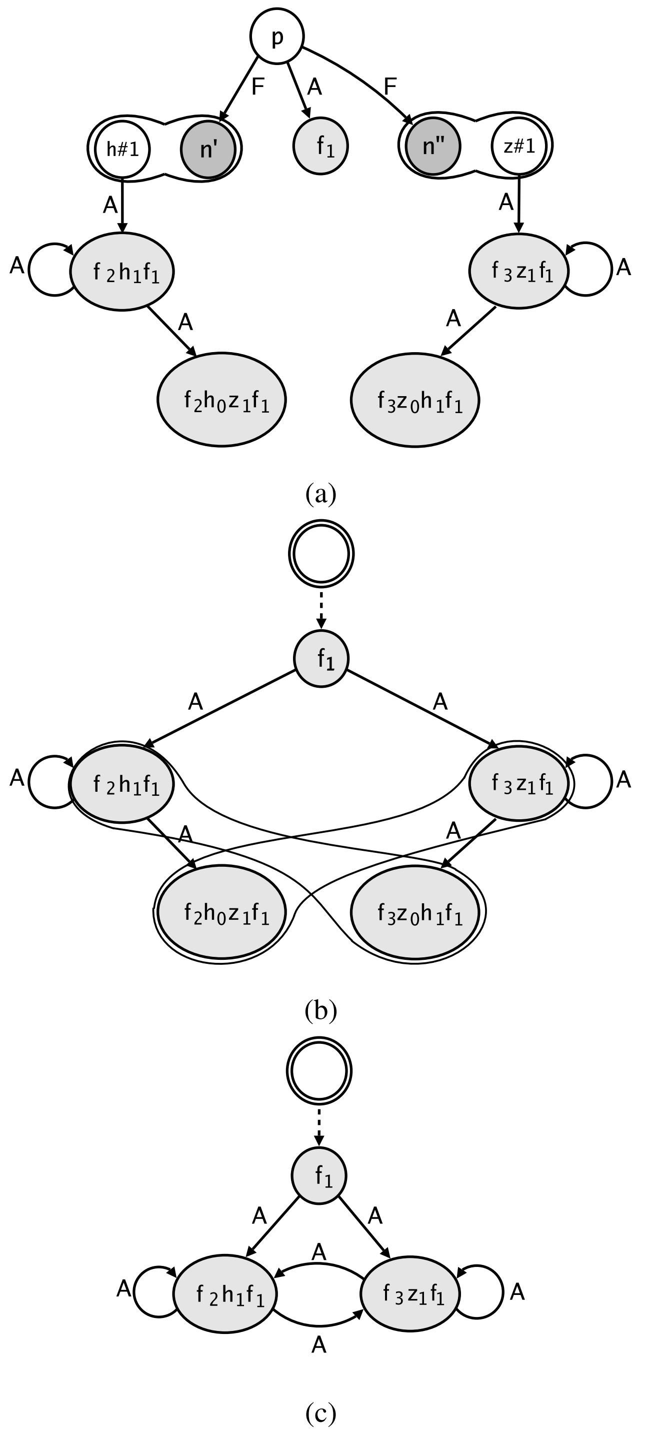


Figure 5.32: The third iteration of the analysis for function `f`. (a) the summaries for `h` and `z` from Figure 5.31 are instantiated; (b) the tentative summary for `f` has two nodes that exceed the stack limit; (c) the final summary for `f` after the node merging indicated in (b) are performed. Note it does not maintain the data structure's shape.

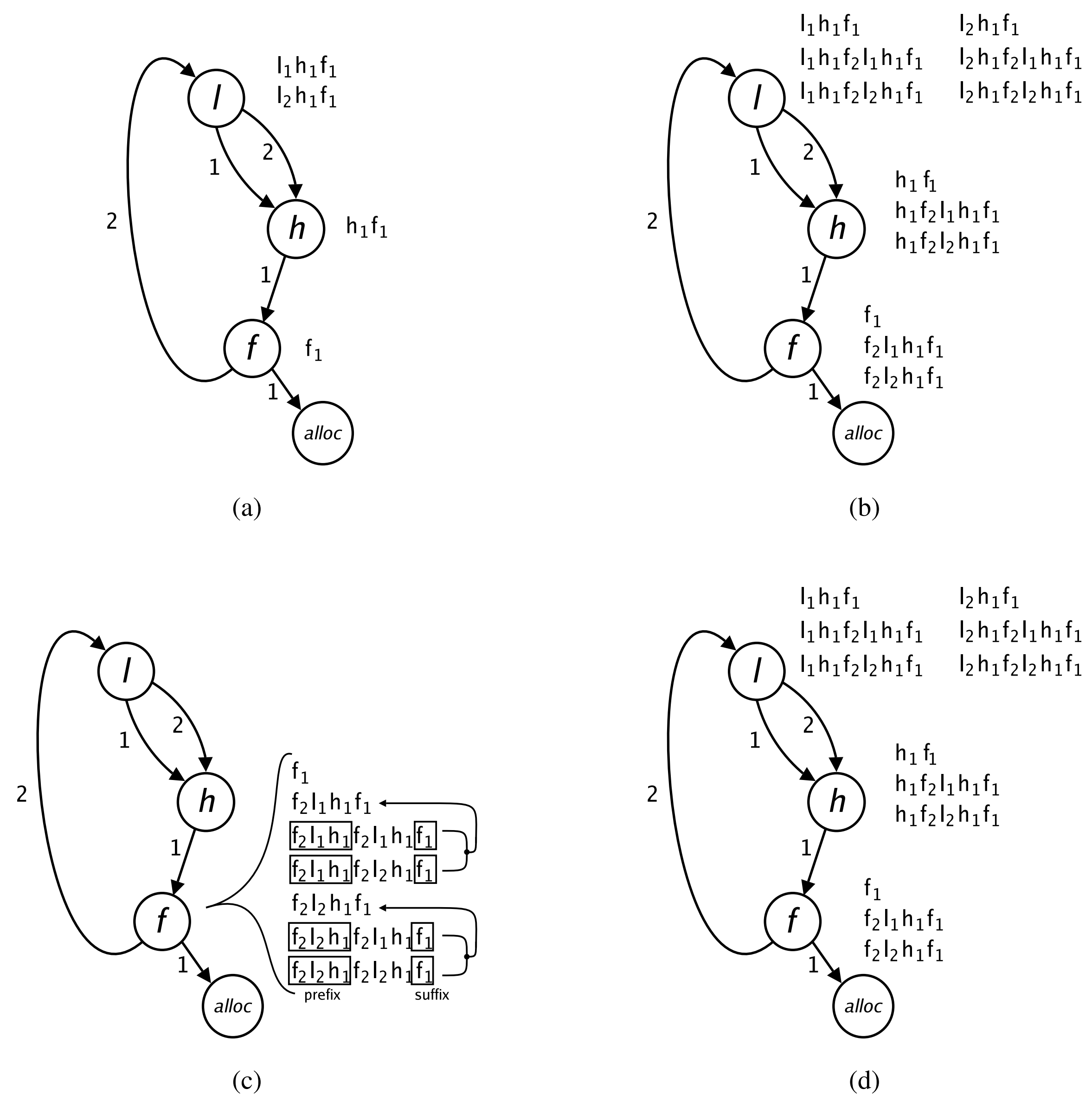


Figure 5.33: The naming schemes can be viewed as string generation in a finite automaton; this shows the first naming scheme. (a) a call graph with four functions; `alloc` is a heap storage allocator. Heap nodes percolate up the graph as indicated on the right of each node; (b) The result of the analysis after its second iteration; (c) analysis of `f` during the third iteration: offending heap nodes are merged as indicated; (d) the final number of heap nodes in each function.

S_n is obtained by eliminating from the latter a substring that corresponds to a cycle in the automaton. The limit k is a measure of how many cycles a name can have. In the example, observe that $f_2l_1h_1f_1$

is the mirror for both $f_2 l_1 h_1 \boxed{f_2 l_1 h_1} f_1$ and $f_2 l_1 h_1 \boxed{f_2 l_2 h_1} f_1$; the corresponding cycles removed from each string are enclosed in boxes. Such strategy guarantees that two nodes are merged only when they correspond to distinct incarnations of the same object, and therefore collapsing them does not incur in excessive loss in analysis precision. This is a novel merging scheme used for the first time in this thesis.

As a comparison, Figure 5.34 shows the set of heap locations obtained when the second naming scheme (Section 5.8.2) is adopted. Figures 5.34(a) and 5.34(b) assume a stack limit of 3 and a merge depth of 2, and Figures 5.34(c) and 5.34(d) use stack limit 4 and merge depth 3. In both cases, the convergence is achieved in the second iteration of the fixpoint computation, shown respectively in Figures 5.34(b) and 5.34(d). This can be observed from the number of heap locations for function `l`, which stabilizes from the (respective) previous iterations.

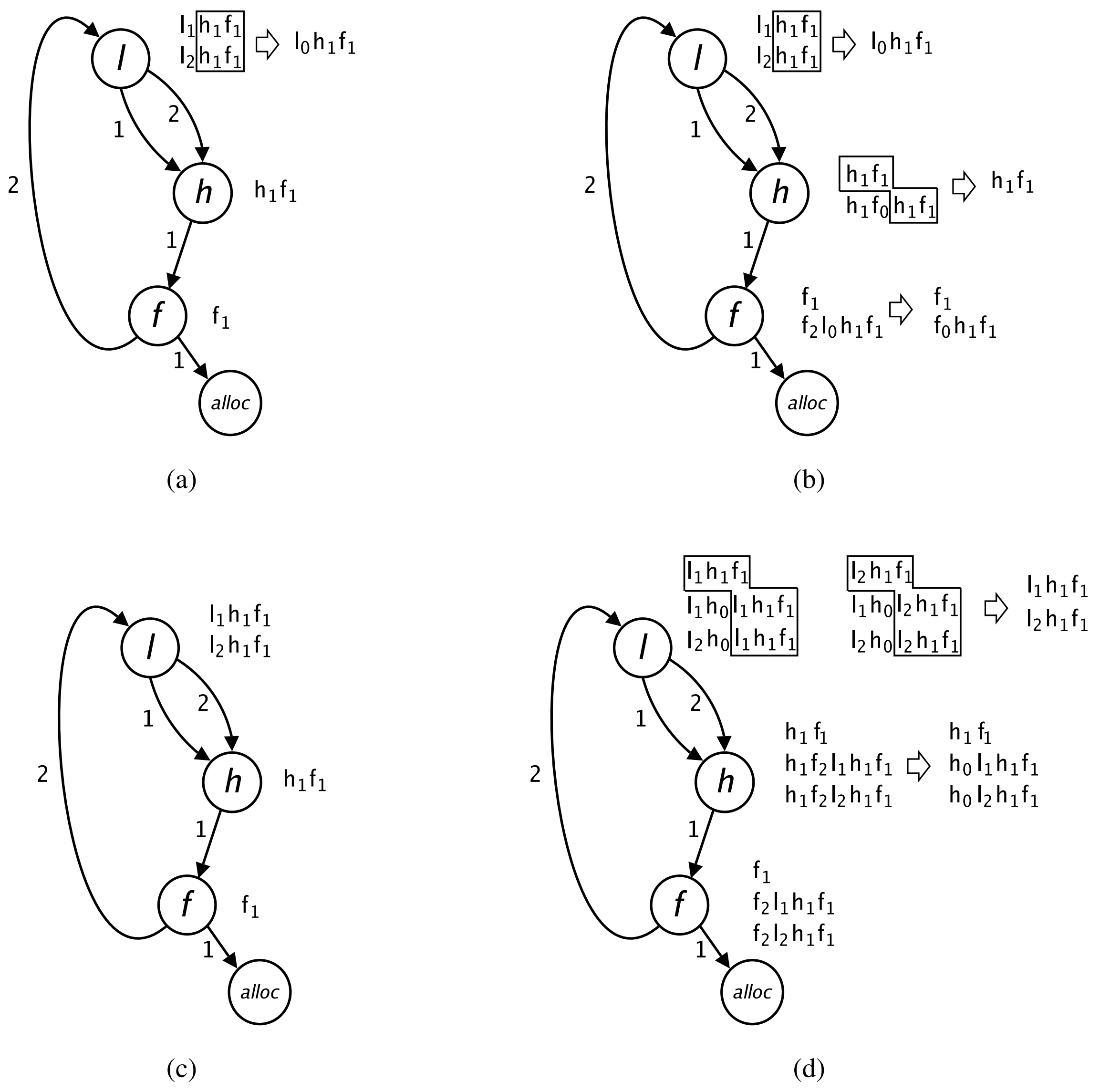


Figure 5.34: The second naming scheme results in less heap nodes because a more aggressive merging scheme is utilized. (a)—(b) a stack limit of 3 and merge depth of 2 is used; (c)—(d) stack limit equals 4 and the merge depth is 3.

Chapter 6

Empirical Studies

This chapter presents an empirical evaluation of our pointer analysis framework. Its basic goal is to provide two types of experimental data: (1) analysis metrics, and (2) (pointer) bugs we found when analyzing several benchmark applications. Some of these projects receive regular source code checking, both manual and automated, and thus finding errors in a number of them indicates the utility of our analysis.

Tables 6.1 and 6.2 summarize the benchmarks we have evaluated. We divided them into two groups: "whole applications" (Table 6.1) corresponds to complete programs; "Linux kernel modules" (Table 6.2) corresponds to complete *modules* within the latest stable version of the Linux O.S. kernel (2.6.23 as of December 2007).

The fourth column of Table 6.1 gives the number of lines of code, the fifth column represents the number of procedures in the program, and the sixth column corresponds to the total number of procedures after loops have been transformed into tail recursive functions. In Table 6.2, the second column specifies which module of the kernel the benchmark belongs to.

6.1 Analysis metrics

We first evaluate a basic metric in our analysis framework: the time it takes to run pointer analysis when considering the origin of our analysis space (i.e., flow-insensitive analysis). Table 6.3 lists such runtimes. We have divided this table into two sections, one for whole application benchmarks and the other for Linux Kernel modules. For each section, the table lists the analysis times for

Table 6.1: Benchmarks I: whole applications.

Benchmark	**Version**	**Description**	**Lines**	**Source funcs**	**Internal funcs**
balsa	2.3.13	E-mail client	110.0k	2659	4682
bftpd	2.0.3	Ftp file server	4.9	145	245
bison	2.1	GNU parser generator	25.4	700	1297
black-hole	1.0.9	Spam prevention	18.0	87	290
cfingerd	1.4.3	Configurable ”finger” daemon	4.5	68	123
compress	1.3	Compression software	2.2	30	66
firestorm	0.5.4	Network firewall	8.0	229	330
gzip	1.2.4	Compression software	8.3	126	331
identd	1.3	TCP identification protocol daemon	0.3	21	40
ispell	3.1	Spell checker	10.1	117	337
lhttpd	0.1	Http server and content management	0.8	20	40
make	3.81	Application building system	22.1	309	853
mingetty	1.07	Minimalist ”getty” program	0.4	24	43
muh	2.05d	IRC bouncing tool	5.2	75	107
pcre	7.1	Regular expressions interpreter	15.4	63	300
pgp4pine	1.76	Integrate PGP into pine mail reader	4.2	72	146
polymorph	0.4.0	Filename converter (”unixizer”)	1.0	19	31
stunnel	3.26	Universal SSL tunnel	3.9	93	134
tar	1.15.1	File archiver	32.7	651	1145
trollftpd	1.26	Ftp file server daemon	2.8	52	102

field-insensitive analysis (i.e., not modeling struct fields) as well as field-sensitive analysis. Given a benchmark, the field-sensitive analysis is often slower than the field-insensitive—modeling fields imply more nodes in the graph, more edges to resolve, and consequently a higher runtime. Interestingly, in a number of cases the field-sensitive analysis turns out to be faster than the field-insensitive. The additional overhead needed to distinguish fields is compensated by a more selective analysis, with less spurious aliases to be analyzed and propagated.

In general, analysis times are short enough to make our technique practical for bug finding, but vary widely depending on the size of the input program and other characteristics. An important

Table 6.2: Benchmarks II: Linux 2.6.23 kernel modules.

Benchmark	**Module**	**Lines**	**Source funcs**	**Internal funcs**
algos	I2C	1.7k	52	75
amso1100	Network	8.2	234	304
atm	I2C	5.7	166	209
bluetooth	Bluetooth	8.4	291	358
busses	I2C	20.3	434	520
chips	I2C	7.1	115	131
core	USB	15.8	527	659
cxgb3	Network	9.3	307	420
gadget	USB	41.9	693	872
host	USB	25.6	629	878
ieee1394	Firewire	24.0	512	735
inficore	Network	25.4	659	840
irq	Kernel	2.3	86	105
kernel	Kernel	64.5	2451	3254
misc	USB	16.8	523	665
mlx4	Network	4.3	149	178
mthca	Network	15.0	442	597
power	Kernel	4.8	220	302
serial	USB	42.1	911	1205
storage	USB	12.2	274	366
video	Video	84.0	1134	1667

characteristic is the shape of the program's call graph—an application with big clusters of functions, all calling each other makes propagating summaries a more challenging task, since a global fixpoint is necessary for the analysis to converge (i.e., all the summary AFGs for all the functions in the strongly-connected component have to stabilize). Arguably, some of the runtimes shown in Table 6.3 would not be practical for compilation. A bug finding tool, however, often runs overnight checking code written during the day. For that purpose, the runtime of pointer analysis is often at least one order of magnitude faster than the total time it takes to find errors.

Table 6.3: Analysis times for the analysis space origin.

Whole applications			Linux Kernel		
Benchmark	**Field Insensitive**	**Field Sensitive**	**Benchmark**	**Field Insensitive**	**Field Sensitive**
balsa	42.71s	76.28s	algos	0.46s	1.57s
bftpd	3.53	2.94	amso1100	1.47	1.56
bison	51.15	98.55	atm	1.58	3.54
blackhole	11.21	7.59	bluetooth	1.17	4.13
cfingerd	2.56	3.72	busses	7.28	12.30
compress	0.68	1.09	chips	2.01	2.78
firestorm	1.28	2.03	core	5.59	9.57
gzip	1.35	1.04	cxgb3	2.41	3.52
identd	0.12	0.17	gadget	18.86	41.75
ispell	8.02	9.65	host	16.63	32.12
lhttpd	1.48	1.32	ieee1394	34.85	26.88
make	123.21	284.59	inficore	7.31	10.29
mingetty	0.36	0.51	irq	0.44	1.43
muh	2.77	2.99	kernel	27.45	44.68
pcre	19.40	24.09	misc	6.91	8.76
pgp4pine	2.85	3.38	mlx4	0.90	0.91
polymorph	0.52	0.72	mthca	3.70	4.71
stunnel	1.49	1.74	power	1.52	2.34
tar	37.07	86.34	serial	12.73	14.70
trollftpd	1.84	1.68	storage	4.78	6.74
			video	31.19	78.59

6.1.1 Speedup

Our second set of data illustrate how analysis times vary when different analysis parameters are considered. A myriad of numbers can be obtained by combining virtually a dozen parameters we have defined. The results contained in this section are just a sample of our ability to evaluate these countless combinations. In this section we have exercised three most important parameters which correspond to some possible values for the three axis in our analysis space.

For these experiments, we ran the analysis with different values of order-, condition- and kill-sensitivities, for both field-sensitive and field-insensitive variants. For order, we borrowed the approximations defined for the flow-aware and flow-branch-aware analyses. The approximation described in Section 4.4 was adopted in which conditions are tracked as precisely as possible within function bodies and later approximated to T,F, and "?" when summaries are computed. Similarly, the approximation discussed in Section 4.3 was used for the kill dimension, namely, only relations that are unconditionally killed were taken into account when kill is switched on.

Tables 6.4 and 6.5 show speedup results. The interpretation for each of these tables is as follows. Each column corresponds to a different analysis variation; the contents of a column are the speedup (in %) of the corresponding analysis variation over its *baseline* analysis, whose runtime was listed Table 6.3. As before, the results are grouped by field-sensitivity (Table 6.4 corresponding to field-insensitive analysis and Table 6.5 showing the results for field-sensitive analysis.)

Invariably, a condition-sensitive analysis is slower than using no conditions; handling conditions involve invoking the theorem prover, which is relatively slow compared to the other modules of the analysis. As seen in all tables, speedups for condition-sensitive columns are all negative. The magnitude of the slowdown varies between $\sim$1% and $\sim$120%, meaning over 2x slower. These values also vary widely depending on which field-sensitivity is chosen.

Consider for example the program blackhole. In a field-insensitive analysis we have achieved speedups of $\sim$8% and experienced slowdowns of $\sim$50%. When fields are modeled, speedups[1] of up to $\sim$40% are obtained, whereas the slowdowns stay within $\sim$20%. Having order information when fields are present has a bigger impact for several of the benchmarks.

[1]Obviously, the speedup is computed over the appropriate baseline analysis.

Table 6.4: Speed up over baseline analysis—field-insensitive (values in %).

	Whole applications									**Linux Kernel**							
	Order [Flow Aware]				**Order [Flow Branch Aware]**					**Order [Flow Aware]**				**Order [Flow Branch Aware]**			
Bench	**Cond. Insens.**		**Cond. Sens.**		**Cond. Insens.**		**Cond. Sens.**		**Bench**	**Cond. Insens.**		**Cond. Sens.**		**Cond. Insens.**		**Cond. Sens.**	
	No Kill	**Kill**	**No Kill**	**Kill**	**No Kill**	**Kill**	**No Kill**	**Kill**		**No Kill**	**Kill**	**No Kill**	**Kill**	**No Kill**	**Kill**	**No Kill**	**Kill**
balsa	14.60	23.45	-39.74	-37.21	15.72	21.21	-37.30	-35.90	algos	6.93	7.89	-17.33	-15.70	7.50	8.80	-18.33	-18.43
bftpd	9.50	8.58	-22.08	-19.95	10.23	11.90	-34.42	-25.61	amso1100	8.39	5.53	-5.29	-7.53	3.13	3.93	-4.82	-7.46
bison	55.69	50.01	-88.42	-88.74	52.43	50.34	-88.32	-88.46	atm	0.01	6.29	-10.09	-7.77	1.87	5.44	-13.20	-7.90
blackhole	6.97	4.20	-52.49	-52.53	7.55	6.67	-52.66	-52.12	bluetooth	7.38	7.80	-5.42	-2.43	6.20	9.85	-6.90	-4.52
cfingerd	12.20	13.06	-20.03	-18.57	14.02	16.56	-19.79	-19.77	busses	14.77	16.86	-8.54	-6.55	15.93	16.91	-4.85	-3.37
compress	3.64	4.58	-10.78	-9.50	5.95	6.56	-12.86	-9.83	chips	8.29	9.25	-8.21	-8.78	10.80	13.64	-15.59	-11.72
firestorm	4.29	4.83	-15.69	-16.46	5.19	5.25	-28.18	-13.46	core	5.76	6.40	-12.07	-10.55	7.46	7.87	-11.14	-9.27
gzip	8.68	9.00	-17.32	-19.88	8.90	10.14	-19.87	-18.07	cxgb3	6.06	7.10	-22.27	-22.25	7.39	8.26	-22.97	-23.13
identd	4.45	6.40	-15.02	-10.94	3.80	4.58	-25.74	-13.64	gadget	30.69	34.47	-53.28	-51.08	29.88	37.20	-56.67	-54.75
ispell	50.25	53.08	-35.27	-32.78	50.12	50.77	-35.86	-35.43	host	35.04	32.67	-32.61	-35.86	35.07	38.89	-43.78	-40.15
lhttpd	0.68	0.47	-5.62	-0.85	3.74	9.22	-7.43	-9.88	ieee1394	7.98	6.06	-39.98	-39.84	7.67	7.14	-40.09	-39.58
make	66.64	64.20	-90.04	-90.02	71.44	71.09	-90.03	-89.92	inficore	7.27	5.37	-5.01	-6.30	6.59	7.42	-1.83	-1.88
mingetty	0.76	0.33	-8.26	-11.11	1.47	2.57	-13.29	-9.66	irq	1.27	2.20	-17.18	-19.82	3.45	3.59	-32.35	-16.41
muh	25.05	22.41	-14.52	-12.91	21.07	18.35	-16.10	-17.12	kernel	14.28	16.20	-14.86	-13.01	15.05	18.02	-15.79	-10.24
pcre	83.07	84.83	-93.57	-93.51	81.67	82.76	-93.52	-93.43	misc	6.59	8.93	-11.70	-7.70	7.23	6.61	-9.94	-8.50
pgp4pine	4.97	4.80	-26.30	-27.83	4.16	3.19	-27.66	-25.26	mlx4	1.87	0.48	-20.22	-16.55	2.42	2.33	-20.03	-16.90
polymorph	1.79	0.36	-23.01	-12.91	3.68	3.33	-20.17	-23.22	mthca	12.51	13.92	-9.19	-6.96	13.10	13.11	-9.28	-8.64
stunnel	3.92	2.97	-17.50	-16.66	4.95	3.17	-17.16	-18.69	power	8.35	8.62	-25.68	-23.71	9.74	8.26	-31.69	-31.45
tar	43.71	42.48	-50.64	-52.23	41.28	40.35	-50.21	-51.08	serial	33.88	32.64	-22.49	-22.15	33.45	32.34	-19.55	-27.04
trollftpd	0.13	0.46	-21.31	-17.86	1.57	2.94	-21.17	-19.35	storage	15.94	16.39	-17.27	-14.24	15.93	14.11	-18.89	-26.42
									video	37.52	34.93	-39.64	-40.31	38.90	31.56	-42.65	-44.82

Table 6.5: Speed up over baseline analysis—field-sensitive (values in %).

	Whole applications									**Linux Kernel**							
	Order [Flow Aware]				**Order [Flow Branch Aware]**					**Order [Flow Aware]**				**Order [Flow Branch Aware]**			
Bench	**Cond. Insens.**		**Cond. Sens.**		**Cond. Insens.**		**Cond. Sens.**		**Bench**	**Cond. Insens.**		**Cond. Sens.**		**Cond. Insens.**		**Cond. Sens.**	
	No Kill	**Kill**	**No Kill**	**Kill**	**No Kill**	**Kill**	**No Kill**	**Kill**		**No Kill**	**Kill**	**No Kill**	**Kill**	**No Kill**	**Kill**	**No Kill**	**Kill**
balsa	20.28	19.77	-43.92	-35.22	14.70	12.70	-37.35	-35.10	algos	5.89	6.32	-16.74	-17.99	6.21	5.83	-18.96	-17.58
bftpd	15.18	17.46	-26.68	-26.40	21.15	22.33	-25.31	-22.72	amso1100	2.31	3.42	-1.49	-5.80	2.41	2.50	-0.52	-2.33
bison	157.97	160.19	-63.42	-64.67	163.21	157.41	-63.42	-64.86	atm	14.61	8.54	-16.18	-15.07	15.69	12.74	-16.98	-15.89
blackhole	43.21	37.39	-22.73	-22.16	44.25	36.91	-23.27	-22.74	bluetooth	4.80	5.50	-9.95	-13.70	6.36	7.39	-8.16	-8.54
cfingerd	21.71	21.35	-15.35	-15.97	19.52	20.05	-22.14	-19.31	busses	23.45	22.71	-4.52	-4.26	24.04	22.94	-7.01	-2.08
compress	1.83	1.16	-9.21	-12.14	3.68	3.79	-14.23	-19.15	chips	11.68	12.14	-9.86	-8.84	12.36	14.50	-6.52	-7.25
firestorm	17.42	16.21	-7.34	-1.57	20.62	16.35	-26.94	-20.19	core	13.54	9.75	-11.01	-8.09	12.32	14.31	-12.72	-12.93
gzip	6.25	5.59	-18.88	-18.62	5.11	6.14	-17.63	-18.29	cxgb3	3.48	4.60	-21.48	-20.44	5.23	6.98	-20.99	-20.36
identd	9.87	3.55	-1.61	-2.84	13.13	12.31	-1.46	-0.72	gadget	38.59	41.02	-16.44	-16.45	40.48	39.61	-21.23	-18.54
ispell	45.51	42.35	-37.75	-37.16	41.07	47.83	-38.31	-37.08	host	26.20	24.91	-47.28	-44.19	28.60	25.43	-42.93	-41.68
lhttpd	2.75	2.62	-28.00	-19.11	3.58	3.54	-19.69	-21.82	ieee1394	12.61	11.57	-28.99	-27.16	11.86	13.16	-31.19	-30.22
make	103.41	114.35	-79.52	-79.43	111.91	116.35	-78.13	-78.12	inficore	73.81	74.22	-60.83	-54.12	75.94	77.68	-62.74	-59.57
mingetty	6.76	7.12	-10.45	-7.19	3.27	7.18	-24.62	-15.56	irq	5.75	6.51	-17.08	-17.48	5.97	5.46	-19.41	-18.14
muh	36.26	38.95	-62.80	-61.28	41.63	40.60	-72.66	-73.78	kernel	22.26	24.73	-124.62	-117.65	19.55	18.65	-121.36	-120.16
pcre	70.45	72.27	-93.24	-93.19	62.28	65.63	-93.16	-93.08	misc	7.90	8.85	-19.07	-19.12	11.20	14.47	-24.94	-19.53
pgp4pine	10.66	10.87	-21.28	-19.43	9.57	10.02	-22.01	-21.65	mlx4	6.51	7.17	-19.45	-17.51	7.26	8.20	-17.35	-15.70
polymorph	3.68	2.65	-15.12	-10.14	4.32	3.97	-16.06	-12.09	mthca	10.98	9.31	-3.47	-2.06	13.57	11.65	-5.51	-8.41
stunnel	4.39	4.50	-15.86	-11.24	5.25	6.33	-16.09	-14.27	power	10.52	9.74	-15.63	-12.90	11.47	10.34	-21.44	-20.89
tar	61.87	69.12	-90.38	-90.24	67.53	68.60	-90.39	-90.20	serial	15.66	13.06	-36.87	-20.85	17.56	14.77	-25.01	-24.10
trollftpd	7.80	5.29	-24.40	-21.76	8.38	6.97	-24.69	-30.40	storage	15.13	18.76	-20.29	-14.27	13.69	11.97	-16.14	-12.31
									video	41.87	42.49	-29.72	-30.11	45.04	41.80	-31.78	-33.16

6.1.2 Accuracy

Besides runtime, accuracy also differs from one analysis to another. In contrast to the previous section, accuracy can only increase by adding more elements to a given analysis (additional elements mean a refinement in the *affects* relation, and thus at most as great a fixpoint for the inference rules).

The accuracy tables (Tables 6.6 and 6.7) report increase in precision compared with the (appropriate) baseline analysis. The numbers involve the size of the summary graphs obtained by each analysis variation. We computed these numbers as follows: let R be the ratio of assign edges to the total number of nodes in the final summary graph for a procedure. If R^B is this ratio for the baseline analysis, and R^V is this ratio for an analysis variation V, then the increase in accuracy is $Q = (R^B - R^V)/R^B$. The peak accuracy reported in all columns labeled *Peak* is the highest such Q over all procedures in each benchmark; the other columns represent average increase in precision over all procedures: $(Q_1 + \ldots + Q_n)/n$.

Of course, this measure of relative precision only accounts for the size of pointer graphs, i.e., the ratio between edges and nodes in the pointer abstraction of a procedure. It does not incorporate the fact that, for instance, some analysis variations provide conditions attached to assign and fetch edges, or that some solutions tell us ordering information, which are both very useful assets. For example, the flow-insensitive and flow-aware solutions for a given program may end up having the same number of edges, but having the order information from the flow-aware solution is a big advantage. Similarly, being able to distinguish what happens under so-and-so condition by taking advantage of the predicates attached to the graph edges is key in certain cases.

As was the case with speedup, the presence of fields can have an impact on relative accuracy, although we did not observe big swings. For instance, ieee1394 changes from $\sim$20% average gain to $\sim$40% when we go from field-insensitive to a field-sensitive analysis. Also observe that "average" is a risky concept: it does not mean that all summary graphs within the benchmark get reduced—note the peak accuracy in some cases is $\sim$1000%, certainly raising the average value.

Table 6.6: Accuracy gain over baseline analysis—field-insensitive (values in %).

Whole applications										Linux Kernel									
	Order [Flow Aware]				Order [Flow Branch Aware]						Order [Flow Aware]				Order [Flow Branch Aware]				
Bench	Cond. Insens.		Cond. Sens.		Cond. Insens.		Cond. Sens.		Peak	Bench	Cond. Insens.		Cond. Sens.		Cond. Insens.		Cond. Sens.		Peak
	No Kill	Kill	No Kill	Kill	No Kill	Kill	No Kill	Kill			No Kill	Kill	No Kill	Kill	No Kill	Kill	No Kill	Kill	
balsa	10.95	11.94	12.01	12.17	13.01	13.10	14.52	14.69	350	algos	1.23	1.26	1.31	1.35	1.23	1.26	1.31	1.35	45
bftpd	1.09	1.09	1.09	1.09	1.09	1.09	1.09	1.09	50	amso1100	8.14	8.17	9.64	9.66	10.39	10.83	10.39	10.83	67
bison	27.34	28.70	29.24	30.60	29.20	30.56	29.20	30.56	881	atm	3.89	3.89	3.89	3.89	3.89	3.89	3.89	3.89	92
blackhole	7.36	7.36	7.36	7.36	7.36	7.36	7.36	7.36	215	bluetooth	6.47	7.53	7.28	8.53	6.47	7.53	7.28	8.53	78
cfingerd	1.07	1.07	1.07	1.07	1.07	1.07	1.07	1.07	64	busses	25.68	27.45	25.68	27.45	25.68	27.45	25.68	27.45	427
compress	1.05	4.44	1.05	4.44	1.05	4.44	1.05	4.44	88	chips	2.51	2.51	2.51	2.51	2.51	2.51	2.51	2.51	33
firestorm	9.42	9.42	9.42	9.42	9.42	9.42	9.42	9.42	193	core	10.78	10.78	10.78	10.78	10.78	10.78	10.78	10.78	50
gzip	6.04	6.14	7.23	7.55	6.04	6.14	7.23	7.55	57	cxgb3	14.29	14.29	14.29	14.29	14.29	14.29	14.29	14.29	107
identd	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	29	gadget	70.09	70.09	73.66	73.66	73.66	73.66	73.66	73.66	1075
ispell	10.99	12.97	11.44	13.42	11.44	13.42	11.44	13.42	250	host	15.53	15.53	15.53	15.53	15.53	15.53	15.53	15.53	102
lhttpd	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	43	ieee1394	27.59	27.26	28.69	28.36	28.80	28.46	28.80	28.46	900
make	11.48	12.27	12.89	13.69	12.99	13.79	13.21	14.01	656	inficore	18.67	19.61	18.67	19.61	18.67	19.61	18.67	19.61	322
mingetty	0.00	0.62	0.00	0.62	0.00	0.62	0.00	0.62	11	irq	1.71	1.71	1.71	1.71	1.71	1.71	1.71	1.71	58
muh	7.31	7.99	7.31	7.99	7.31	7.99	7.31	7.99	69	kernel	13.84	15.68	16.50	17.82	13.84	15.68	16.50	17.82	541
pcre	15.53	16.26	15.53	16.26	15.61	16.34	15.61	16.34	122	misc	4.20	4.20	4.20	4.20	4.20	4.20	4.20	4.20	32
pgp4pine	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	30	mlx4	5.01	5.16	5.01	5.16	5.01	5.16	5.01	5.16	70
polymorph	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32	17	mthca	8.81	8.92	8.81	8.92	8.81	8.92	8.81	8.92	85
stunnel	10.33	11.04	10.33	11.04	10.33	11.04	10.33	11.04	76	power	4.33	5.27	4.33	5.27	4.33	5.27	4.33	5.27	72
tar	6.56	7.77	6.71	7.91	6.71	7.91	6.71	7.91	350	serial	26.69	27.17	26.69	27.17	26.69	27.17	26.69	27.17	204
trollftpd	2.94	2.94	2.94	2.94	2.94	2.94	2.94	2.94	103	storage	5.07	5.87	5.07	5.87	5.07	5.87	5.07	5.87	42
										video	21.75	27.26	23.36	28.96	21.75	27.26	23.36	28.96	728

Table 6.7: Accuracy gain over baseline analysis—field-sensitive (values in %).

Whole applications										Linux Kernel									
	Order [Flow Aware]				Order [Flow Branch Aware]						Order [Flow Aware]				Order [Flow Branch Aware]				
Bench	Cond. Insens.		Cond. Sens.		Cond. Insens.		Cond. Sens.		Peak	Bench	Cond. Insens.		Cond. Sens.		Cond. Insens.		Cond. Sens.		Peak
	No Kill	Kill	No Kill	Kill	No Kill	Kill	No Kill	Kill			No Kill	Kill	No Kill	Kill	No Kill	Kill	No Kill	Kill	
balsa	18.50	18.75	18.50	18.75	18.50	18.75	18.50	18.75	647	algos	1.45	1.67	1.52	1.68	1.45	1.67	1.52	1.68	49
bftpd	2.67	2.67	2.67	2.67	2.67	2.67	2.67	2.67	60	amso1100	10.84	11.43	11.95	12.03	12.71	13.94	12.71	13.94	78
bison	29.44	37.11	31.84	39.55	31.84	39.55	31.84	39.55	993	atm	3.89	3.89	3.89	3.89	3.89	3.89	3.89	3.89	92
blackhole	7.43	7.43	7.43	7.43	7.43	7.43	7.43	7.43	259	bluetooth	7.21	8.49	8.18	9.53	7.21	8.49	8.18	9.53	83
cfingerd	1.07	1.07	1.07	1.07	1.07	1.07	1.07	1.07	64	busses	21.36	24.68	21.36	24.68	21.36	24.68	21.36	24.68	519
compress	1.05	4.44	1.05	4.44	1.05	4.44	1.05	4.44	88	chips	3.49	3.49	3.49	3.49	3.49	3.49	3.49	3.49	57
firestorm	10.77	10.77	10.77	10.77	10.77	10.77	10.77	10.77	207	core	15.65	15.65	15.65	15.65	15.65	15.65	15.65	15.65	72
gzip	7.61	7.38	8.43	8.53	7.61	7.38	8.43	8.53	71	cxgb3	18.60	18.60	18.60	18.60	18.60	18.60	18.60	18.60	273
identd	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	29	gadget	94.98	94.98	115.69	115.69	115.69	115.69	115.69	115.69	2283
ispell	11.97	13.96	12.42	14.41	12.42	14.41	12.42	14.41	293	host	14.27	15.89	15.96	16.01	14.27	15.89	15.96	16.01	129
lhttpd	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	43	ieee1394	41.68	41.68	44.44	44.44	44.44	44.44	44.44	44.44	1368
make	13.65	14.33	14.29	14.98	14.06	14.75	14.32	15.01	890	inficore	16.51	17.23	16.51	17.23	16.51	17.23	16.51	17.23	387
mingetty	0.00	0.62	0.00	0.62	0.00	0.62	0.00	0.62	11	irq	1.71	1.71	1.71	1.71	1.71	1.71	1.71	1.71	58
muh	9.10	10.80	9.10	10.80	9.10	10.80	9.10	10.80	91	kernel	15.27	16.93	17.32	17.99	15.27	16.93	17.32	17.99	835
pcre	17.51	17.90	18.26	18.74	17.51	17.90	18.26	18.74	146	misc	3.98	4.17	5.62	6.03	3.98	4.17	5.62	6.03	61
pgp4pine	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	30	mlx4	5.01	5.16	5.01	5.16	5.01	5.16	5.01	5.16	70
polymorph	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32	17	mthca	6.48	7.20	6.48	7.20	6.48	7.20	6.48	7.20	85
stunnel	9.46	10.05	9.46	10.05	9.46	10.05	9.46	10.05	80	power	3.18	3.92	3.18	3.92	3.18	3.92	3.18	3.92	68
tar	28.98	29.50	29.20	29.75	29.20	29.75	29.20	29.75	1388	serial	28.27	28.94	28.27	28.94	28.27	28.94	28.27	28.94	217
trollftpd	2.94	2.94	2.94	2.94	2.94	2.94	2.94	2.94	103	storage	6.26	6.80	6.26	6.80	6.26	6.80	6.26	6.80	51
										video	18.63	20.22	19.40	21.12	18.63	20.22	19.40	21.12	814

6.2 Bug reports

This subsection presents statistics and descriptions of pointer bugs we have found in some of the benchmarks we analyzed. Our tables report the numbers obtained when running our most precise analysis. Later we discuss a set of representative errors across the applications and the analysis elements that help to safely report them.

Table 6.8: Bugs found on whole applications.

Benchmark	# of reports	# of bugs	Type
balsa	8	2	null pointer dereference, passing null argument
bftpd	2	2	memory leak, using garbage value from malloc
bison	5	0	
black-hole	13	3	memory leak
cfingerd	7	1	memory leaks when things fails elsewhere, file leak
compress	0	0	
fireStorm	8	0	
gzip	4	0	
identd	1	0	
ispell	5	1	null pointer dereference
lhttpd	0	0	
make	14	1	null pointer dereference
mingetty	0	0	
muh	2	0	
pcre	8	2	pointer to local var exposed, null pointer dereference
pgp4pine	19	3	null pointer dereference
polymorph	3	0	
stunnel	6	1	file leak
tar	9	0	
trollftpd	15	3	null argument passing, double allocation
Total	129	21	

At first we were surprised to find relatively simple errors in some of the benchmarks, e.g., the Linux kernel. Our impression was that this application was well tested and only the most intricate bugs would remain uncovered. However, we learned that the drivers section of the kernel, for example, does not share this expectation. By manually inspecting users reports we found out that runtime errors are common when playing with USB devices, network cards, etc. Below is an excerpt of a common report pattern, referred to as an “Oops”:

```
ieee1394: Initialized config rom entry ‘ip1394’
ohci1394: fw-host0: OHCI-1394 1.1 (PCI): IRQ=[17]  MMIO=[e7004000-e70047ff]  Max
Packet=[4096]  IR/IT contexts=[4/8]
ohci1394: fw-host1: OHCI-1394 1.0 (PCI): IRQ=[19]  MMIO=[e7006000-e70067ff]  Max
Packet=[2048]  IR/IT contexts=[8/8]
ieee1394: Error parsing configrom for node 0-00:1023
ieee1394: Host added: ID:BUS[0-01:1023]  GUID[0001080000002d02]
eth1394: eth1: IEEE-1394 IPv4 over 1394 Ethernet (fw-host0)
BUG: unable to handle kernel NULL pointer dereference at virtual address 000003e4
 printing eip:
f8c7db91
*pde = 00000000
Oops: 0000 [#1]
PREEMPT SMP
Modules linked in: eth1394 ohci1394 ieee1394 nfsd exportfs lockd sunrpc
snd_via82xx snd_ac97_codec snd_ac97_bus snd_pcm snd_timer snd_page_alloc
snd_mpu401_uart snd_rawmidi snd lp af_packet 8139too mii loop via_agp agpgart
uhci_hcd
CPU:    0
EIP:    0060:[<f8c7db91>]    Not tainted VLI
EFLAGS: 00010202   (2.6.19-rc4 #10)
EIP is at ether1394_remove_host+0x31/0xa0 [eth1394]
eax: f680ad0c   ebx: 00000380   ecx: f678efc4   edx: f680ad0c
esi: f680ad0c   edi: f5c26000   ebp: f5c57e4c   esp: f5c57e30
ds: 007b   es: 007b   ss: 0068
Process modprobe (pid: 6822, ti=f5c56000 task=f678ea90 task.ti=f5c56000)
Stack: f8c80fa0 f5c26000 f8f2bf66 f7639d34 f8c80fa0 f5c26000 f5c26000 f5c57e70
       f8f2c1fc f5c26000 f5c26000 00000000 00000282 f8c80fa0 f5c26000 c21e0094
       f5c57e8c f8f2cb56 f8c80fa0 f5c26000 00000000 f5c26000 f5c260c4 f5c57e9c
```

The significant number of such reports we found online indicates that null pointer dereferences, for example, still haunt the users despite the alleged maturity of the software.

Table 6.9: Bugs found on the Linux kernel (latest stable version, 2.6.23).

Benchmark	**# of reports**	**# of bugs**	**Type**
algos	0	0	
amso1100	2	1	null pointer dereference
atm	0	0	
bluetooth	2	0	
busses	2	1	null pointer dereference
chips	5	0	
core	3	1	pointer to local var exposed
cxgb3	5	2	null pointer dereference
gadget	9	2	null pointer dereference
host	7	1	null pointer dereference
ieee1394	3	0	
inficore	5	1	null pointer dereference
irq	7	1	null pointer dereference
kernel	11	2	pointer to local var exposed, null pointer dereference
misc	8	2	pointer to local var exposed, null pointer dereference
mlx4	4	0	
mthca	6	2	null pointer dereference
power	0	0	
serial	4	1	null pointer dereference
storage	0	0	
video	9	2	buffer overrun, null pointer dereference
Total	92	19	

Tables 6.8 and 6.9 reports errors that are pointer related, which range from null pointer dereference, to returning from a function with some global pointer referring to one of the function's local variables, simple memory leaks and buffer overruns. The column labeled "type" is a short description of the types of errors encountered for each benchmark.

Several errors span more than one procedure, and this is not unexpected: bugs that span different functions, possibly from different original source files, are harder to find. For example, the following procedure in the amso1100 Linux driver triggers a null pointer dereference:

```
void handle_vq(struct c2_dev *c2dev, u32 mq_index)
{
  void *adapter_msg, *reply_msg;
  struct c2wr_hdr *host_msg;
  int err;
  ...
  host_msg = vq_repbuf_alloc(c2dev);
  ...
  if (!host_msg) {
    host_msg = &tmp;
    memcpy(host_msg, adapter_msg, sizeof(tmp));
    reply_msg = NULL;
  } else {
    memcpy(host_msg, adapter_msg, reply_vq->msg_size);
    reply_msg = host_msg;
  }
  ...
  err = c2_errno(reply_msg);  // DOES NOT HANDLE NULL
  ...

  if (!err) switch (req->event) {
    case IW_CM_EVENT_ESTABLISHED:
    ...
  }
}
```

This function would be correct if `c2_errno` could handle a null argument; note `reply_msg` is explicitly assigned null in the *then* branch of the *if* statement. However, as shown below, the first thing `c2_errno` does is to call a function named `c2_wr_get_result`, which unconditionally dereferences its parameter `wr`. This error manifests itself when `host_msg` receives a null value from `vq_repbuf_alloc`, which means it failed to allocate a so-called VQ Reply Buffer. Clearly the programmer did not intend to crash due to a null pointer dereference if this happened: the fact that he tests the value of `err` indicates he expected the program to continue running.

```
int c2_errno(void *reply) {
  switch (c2_wr_get_result(reply)) { ... }
}

unsigned c2_wr_get_result(void *wr) {
  return ((struct c2wr_hdr *) wr)->result;
}
```

This error can be caught because the summarized information for `c2_errno` contains the fact that its parameter `reply` is unconditionally fetched (after incorporating the summarized information from `c2_wr_get_result`). For that to happen, the analysis needs some minimum information about program conditions. Also, order information is needed within `handle_vq`, as well as the ability to disassociate between the *then* and *else* branches. Otherwise, the analysis cannot figure out the relationship between statements `reply_msg = NULL` and `reply_msg = host_msg`. Note `c2_wr_get_result` is not faulty because it does not check for `wr` before dereferencing it. The fault comes from `c2_errno` passing in an illegal input[2]. Needless to say, interprocedural analysis is a requirement, as the statements leading to the error span several procedures.

A similar error occurs in the USB driver (gadget benchmark), where a test for null indicates the possibility of such value for a pointer. The simplified code fragment is as follows:

```
int set_ether_config (struct eth_dev *dev, gfp_t gfp_flags)
{
  ...
  if (!subset_active(dev) && dev->status_ep) { // TEST
    ...
  }
  ...
  if (result < 0) {
    if (!subset_active(dev))
      usb_ep_disable (dev->status_ep); // USAGE
      ...
  }
  ...
}
```

[2]Well, this is open to debate.

As shown below, the procedure `usb_ep_disable` does not handle a null argument, yet is possible that in some cases it will receive such value. Modeling fields, at least to some extent, is an important feature for effectively finding this error, since the pointer in question is a field on a struct (`dev->status_ep`).

```
int usb_ep_disable (struct usb_ep *ep)
{
  return ep->ops->disable = ...;
}
```

Balsa also has a potential null pointer dereference in an exception case:

```
static void
handle_mdn_request(LibBalsaMessage *message)
{
  gboolean suspicious, found;
  const InternetAddressList *use_from;

  ...
  if (message->headers->reply_to)
    use_from = message->headers->reply_to;
  else if (message->headers->from)
    use_from = message->headers->from;
  else if (message->sender)
    use_from = message->sender;
  else
    use_from = NULL; // SETTING TO NULL

  ...
  suspicious =
     !libbalsa_ia_rfc2821_equal(message->headers->dispnotify_to->address,
                                use_from->address);  // DEREFERENCE

  if (!suspicious) {
    ...
  }
  ...
}
```

Perhaps the *if* statement above is not supposed to "fail." If it does, however, `use_from` is assigned null, and then later dereferenced at expression `use_from->address`. Being able to distinguish among the mutually exclusive branches of the *if*, as well as modeling some statement order, makes this error not hard to find.

A number of memory leaks were found in some of the benchmarks, all similar to the following excerpt from blackhole:

```
int send_mail_box(char *mbox, char *tmpfile, char *mailfrom, char *iprelay)
{
  FILE *tmp, *mb;
  char *buffer;
  int i;
  int eoh = 0;

  buffer = malloc(MAX_INPUT_LINE + 1);
  if(buffer == NULL)
    return 1;

  tmp = fopen(tmpfile, "r");
  if(tmp == NULL)
    return 1;

  /* Lock Mailbox */
  if(mbox_lock(mbox) != 0)
    return 1;     // RETURNING FROM FUNCTION

  ...
}
```

If the test `mbox_lock(mbox) != 0` fails, the statement that follows the test returns from the function without deallocating neither `buffer` nor closing `tmp`.

Perhaps one of the most interesting errors we have reported occurs in the USB module of the Linux kernel, and it involves order, fields, and variable aliasing. A simplified version of the code is below:

```
struct ehci_qh *
qh_make (...)
{
  ...
  if (type == PIPE_INTERRUPT) {
    ...
  } else {
    struct usb_tt *tt = urb->dev->tt;
    int            think_time;



    ..
    think_time = tt ? tt->think_time : 0;
  }

  ...

  switch (urb->dev->speed) {
    ...
    case USB_SPEED_FULL:

    if (!ehci_is_TDI(ehci) || ...)
      info2 |= urb->dev->tt->hub->devnum << 16;  // DEREFERENCING
  }                                              // urb->dev->tt
  ...
}
```

The potential error in the above code is a null pointer dereference of `urb->dev->tt` at the last statement shown. This can happen because `tt` and `urb->dev->tt` may be aliased due to statement `tt = urb->dev->tt` at the top, and failing at the null check "`tt ?`" means null is possible. Upon reporting this bug we received the following comments from the developers:

```
Looks to me  like this is a longstanding bug in the root
hub TT support. See if this patch makes that work better.

It looks  like this is an ARC-derived core, and no root
hub TT has  been set up.  Moreover, it looks  like even
```

```
the  original  patch adding  root hub TT  support (only
for the  PCI based  devboard/FPGA)  didn't actually set
up  such  a TT ...  so this bug  has been around  for a
very long time.
```

Although this is not an overly intricate error, developers have failed to find it manually or dynamic automated testing. This reinforces our intuition that simple enhancements to static analysis techniques can go a long way towards making software less faulty.

Our tables also indicate that a number of false positives were reported. Most of the time the cause is lack of information about the semantics of a callee function, or errors that would exist only if a loop executes zero times, which is unlikely in most cases. Consider the example below:

```
int
progcomp_insert (cmd, cs)
      char *cmd;
      COMPSPEC *cs;
{
  register BUCKET_CONTENTS *item;

  if (cs == NULL)
    programming_error (_("progcomp_insert: \%s: NULL COMPSPEC"), cmd);

  if (prog_completes == 0)
    progcomp_create ();

  cs->refcount++;  // DEREFERENCE
  ...
}
```

Due to its testing, the analysis assumes that `cs` can be null when it is dereferenced at statement `cs->refcount++`. However, the function `programming_error` aborts the execution whenever it is invoked, and thus no error exists. This is (arguably) different from crashing right after `malloc`, for instance, fails to allocate memory. A considerable number of programs do not handle this case well and as a result program misbehaviors may occur. The counter-argument is that if there is no available memory, the application might as well crash (although we do not agree).

Another common type of error is to return from a function with a global pointer referring to a deallocated block of memory. This is often a low-severity bug if the global pointer is re-assigned prior to its next use, but it can turn into a hard-to-find error otherwise. In pcre's function `match` the statement `md->recursive = &new_recursive` assigns the address of `new_recursive`, a local variable, to `md->recursive`, an outside pointer. Several lines later the function returns without clearing up that pointer. Similarly, Linux kernel's function `start_unregistering` executes `p->unregistering = &wait` to assign the address of a local variable to outside pointer `p->unregistering`, and then returns without prior notice.

Perhaps two of the funniest errors we found were these, respectively from the gadget USB module of the Linux kernel and blackhole's `make_match` procedure:

```
int gs_recv_packet(struct gs_dev *dev, char *packet, unsigned int size)
{
  ...

  port = dev->dev_port[0];

  if (port == NULL) {
    printk(KERN_ERR "gs_recv_packet: port=\%d, NULL port pointer\n",
           port->port_num);  // DEREFERENCE port
    return -EIO;
  ..
}

void make_match(int check)
{
  ...
  if(bh_action[bh_match_start->match].score < bh_action[check].score) {
    ...
  } else {
    pcur = (struct bh_matches *) malloc(sizeof(struct bh_matches));
    pcur = bh_match_start; // RE-ASSIGN pcur, LEAKING PREVIOUS VALUE
    ...
  }
}
```

In both cases the error is obvious, and probably indicates some careless code change (it is not likely that the original programmer would have committed them). In the first error `port` is dereferenced only when it has been tested for null and succeeded, and in the second `pcur` is allocated and immediately leaked through a re-assignment. In both cases order information is needed to determine what occurs first.

6.3 Points-to sets sizes

In this subsection we show that the summary-based feature of our analysis generates points-to sets that are smaller than those of Andersen or Steensgaard [2, 67]. We compute points-to set sizes by observing that assign edges in the AFG for a function correspond to the set of locations pointed-to by the source node of the assign edge. Of course, there are several summary AFGs in a given benchmark—one for each function in the program. To calculate our points-to set sizes in Table 6.10, we compute an average among all AFGs, as apposed to Andersen and Steensgaard which are whole program analyses and therefore incur in excessive imprecision. The table shows the number for the origin of our analysis space using no fields.

Table 6.10: Average Points-to Set Sizes for Representative Benchmarks.

Benchmark	**AFG**	**Andersen**	**Steensgaard**
compress	0.524	1.22	2.1
gzip	2.06	2.96	25.17
ispell	1.92	2.25	16.45
make	26.11	74.70	414.04
bison	1.88	1.72	20.51
tar	2.58	17.41	53.7

Chapter 7

Final Remarks

Pointer analysis is still an active area of research, as recent work illustrates [38, 50, 65, 66]. The number of papers in the subject is no fewer than a hundred, and several PhD theses have been published exclusively on pointer analysis. Being a critical component for most software analysis tools, there is a lot of interest on the problem and many researchers trying to develop ingenious solutions. Being a very difficult problem, it is unlikely that any of these attempts will solve it in general; instead, we believe the idea is to tailor the analysis to a particular application. In this thesis, we set out to try such specialization for what we call modular bug finding.

We have learned that a relevant increment in analysis precision should not require a large decrease in efficiency, and that interesting trade-offs can be obtained by looking at pointer analysis in different ways. For example, the execution-insensitive nature of some dataflow analyses makes software analysis tools to be overly conservative, causing many errors to go unreported. Providing some minimal order is cheap and can assist with information the tool can rely upon when facing uncertainty about a procedure's execution. Loops may pose a challenge to that objective, and there are basically two ways of dealing with them. Either penalize the entire function by enclosing it on a big cycle such as Figure 2.3(a), or take the loops apart and analyze them separately, converting the rest of the procedure's code into an acyclic representation. In general, algorithms for acyclic structures are simpler and more efficient, and can yield better results. Our framework is built on top of such kind of representation, and therefore it is simple and efficient. We also find it to be elegant, since the definition of a single inference that can be refined in different ways covers a large set of pointer analysis algorithms. The ability to evaluate a myriad of such variations is one of the strengths of our technique, perhaps more important than the specific results of a particular implementation itself.

We have not tried to make our analysis demand-driven, but we find our Assign-Fetch Graph representation could be adapted for such purpose since it would not require constructing nor intersecting points-to sets like most existing demand-driven approaches. The structure of the AFG could be explored to derive alias edges on the fly, and only for those operations that are relevant for a given query. This would resemble the work of Sridharan et. al. [66], in which a demand-driven points-to analysis for Java is proposed. A key to their approach is to match loads and stored on the same class field through “match” edges. Their technique does not apply for C, however, where one can explicitly take the address of a variable, the address of a field, or dereference any pointer (not only fields like Java).

Indeed, the Assign-Fetch Graph representation has a number of natural features that makes it a very attractive abstraction. Because it models pointer assignments within a function instead of points-to relations, it is used to answer MOD and other side-effect questions directly. In addition, fetch edges in a procedure’s summary, specially if annotated with some form of conditions, can provide information about which variables are dereferenced. This plus the ability to summarize a function for any possible calling context makes it ideal for bottom-up, modular analysis. It is our expectation that the AFG will be extended by others in some currently unknown but certainly interesting way.

Bibliography

[1] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools, Second Edition*. Addison-Wesley, 2007.

[2] L. O. Andersen. Program analysis and specialization for the C programming language. PhD thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

[3] Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2002.

[4] J. Banning. An efficient way to find the side-effects of procedure calls and the aliases of variables. In *6th ACM Symposium on the Principles of Programming Languages*, pages 29–41, 1979.

[5] Bruno Blanchet. Escape analysis for object oriented languages: Application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1999.

[6] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, 2003.

[7] Daniel Brand, Marcio Buss, and Vugranam Sreedhar. Evidence-based analysis and inferring preconditions for bug detection. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM)*, pages 44–53, 2007.

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

[9] M. Burke, P. Carini, J. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Lecture Notes in Computer Science, 892, Springer-Verlag, Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250, 1995.

[10] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice & Experience*, 30(7):775–802, 2000.

[11] Marcio Buss, Daniel Brand, Vugranam Sreedhar, and Stephen A. Edwards. Flexible pointer analysis using the assign-fetch graph. In *To appear in the Proceedings of the 23rd ACM Sympoium on Applied Computing (SAC), Programming Languages Track*, March 2008.

[12] Marcio Buss, Stephen A. Edwards, Bin Yao, and Daniel Waddington. Pointer analysis for source-to-source transformations. In *Proc. of the 5th IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 139–150, 2005.

[13] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN'90 Conference on Program Language Design and Implementation*, pages 296–310, 1990.

[14] Ramkrishna Chatterjee, Barbara Ryder, and William A. Landi. Relevant context inference. In *Proceedings of Principles of Programming Languagues (POPL)*, pages 133–146, 1999.

[15] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths: design implementation and evaluation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2000.

[16] Sigmund Cherem and Radu Rugina. A practical escape and effect analysis for building lightweight method summaries. In *Compiler Construction (CC)*, pages 172–186, 2007.

[17] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[18] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999.

[19] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[20] K.D. Cooper and K. Kennedy. Inter-procedural side-effect analysis in linear time. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 487–506, 1988.

[21] IBM Corporation. A software falsifier. In *International Symposium on Software Reliability Engineering*, pages 174–185, October 2000.

[22] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Principles of Programming Languagues (POPL)*, pages 238–252, 1977.

[23] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 35–46, 2000.

[24] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.

[25] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, New York, NY, USA, 2001. ACM.

[26] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *ACM SIGPLAN 94-6/94*, pages 230–241, 1994.

[27] Isil Dillig, Thomas Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 435–445, 2007.

[28] The Economist. Building a better bug-trap. Science Technology Quarterly, June 19th, 2003. Also available at http://www.klocwork.com/company/downloads/economist.html.

[29] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.

[30] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, School of Computer Science, McGill University, August 1993.

[31] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[32] Dawson Engler and Madanlal Musuvathi. Static analysis versus model checking for bug finding. In *VMCAI*, 2004.

[33] eWeek Magazine. GM to software vendors: Cut the complexity. http://www.eweek.com/article2/0,1895,1679844,00.asp.

[34] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[35] Jeffrey S. Foster, Manuel Fähndrich, and Alex Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, pages 175–198, 2000.

[36] Patrice Godefroid. Model checking for programming languages using verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[37] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 52–58, New York, NY, USA, 2005. ACM Press.

[38] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 290–299, New York, NY, USA, 2007.

[39] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):35–47, 1990.

[40] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth McMilan. Abstraction from proofs. In *Proceedings of Principles of Programming Languagues (POPL)*, pages 232–244, 2004.

[41] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the 5th International Static Analysis Symposium*, pages 57–81, 1998.

[42] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[43] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 28–40, New York, NY, USA, 1989. ACM Press.

[44] Marc Shapiro II and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of Principles of Programming Languagues (POPL)*, pages 1–14, 1997.

[45] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[46] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality (2nd Edition)*. McGraw-Hill, Inc., New York, NY, USA, 1996.

[47] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lips-like structures. In *Steven S. Muchnick and Neil D. Jones, editors. Program Flow Analysis: Theory and Applications*, pages 102–131, 1981.

[48] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 235–248, 1992.

[49] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31, New York, NY, USA, 1988. ACM Press.

[50] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 278–289, New York, NY, USA, 2007.

[51] Ondrej Lhotak. Program analysis using binary decision diagrams. PhD thesis, McGill University, January 2006.

[52] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 317–326, 2003.

[53] Subhas C. Misra and Virendra C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *ICCSA 2003, Lecture Notes in Computer Science (LNCS 2667)*, pages 724–732, 2003.

[54] G. Myers. *The Art of Software Testing, 2nd Edition*. John Wiley and Sons, 2004.

[55] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. Principles of program analysis. Springer, 1999.

[56] NIST. Software errors cost u.s. economy $59.5 billion annually. http://www.nist.gov/public_affairs/releases/n02-10.htm.

[57] Gimpel Software PC-lint/FlexeLint. Version 7.5. http://www.gimpel.com/html/flex.htm.

[58] Linux Kernel Project. http://www.kernel.org.

[59] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[60] Atanas Rountev, Ana Milanova, and Barbara Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2001.

[61] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–293, New York, NY, USA, 1988. ACM Press.

[62] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN'99 Conference on Program Language Design and Implementation*, 1999.

[63] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 16–31, 1996.

[64] Jyh shiarn Yur, Barbara Ryder, and Willim Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st International Conference on Software Engineering*, pages 442–451, 1999.

[65] Manu Sridharan. Refinement-based program analysis tools. PhD Thesis, UC Berkeley. Also available as technical Report No. UCB/EECS-2007-125, October 2007.

[66] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM.

[67] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[68] P. Stocks, B. Ryder, W. Landi, and S. Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 21–31, 1998.

[69] CNN Technology. Will bugs scare off users of new windows 2000? http://archives.cnn.com/2000/TECH/computing/02/17/windows.2000.

[70] Frederic Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 35–46, 2001.

[71] David Wagner. Static analysis and computer security: New techniques for software assurance. PhD thesis, University of California at Berkeley, December 2000.

[72] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[73] John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.

[74] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Compiler Construction (CC)*, pages 1–17, 2000.

[75] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 1–12, 1995.

[76] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of Principles of Programming Languagues (POPL)*, pages 351–363, 2005.

[77] Mihalis Yannakakis. Graph-theoretic methods is database theory. In *PODS'90: Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 230—242, 1990.

[78] S. Zhang, B. Ryder, and W. Landi. Experiments with combined analysis for pointer aliasing. In *Workshop on program analysis for software tools and engineering*, pages 11–18, 1998.

[79] Sean Zhang, Barbara Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Foundations of Software Engineering (FSE)*, pages 81–92, 1996.