McGill University
School of Computer Science

A Practical Context-Sensitive Interprocedural Analysis Framework for C Compilers

*Laurie J. Hendren**
Maryam Emami
Rakesh Ghiya
Clark Verbrugge

ACAPS Technical Memo 72
July 24, 1993

Advanced Compilers, Architectures, and Parallel Structures Group

3480 University St • Montreal • Canada • H3A 2A7

*The work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

Contents

1 Introduction and Motivation **1**

2 Foundations **2**

2.1 A Simple and Structured Intermediate Representation 3

2.2 Representing Invocation Contexts 5

2.3 Abstract Stack Locations and Points-to Analysis 7

2.3.1 Program-point-specific Points-to Information 8

2.3.2 Mapping Information in the Invocation Graph 8

2.3.3 Resolving Function Pointers 10

3 Developing an Interprocedural Analysis: Generalized Constant Propagation **10**

3.1 GCP in General 10

3.2 Intraprocedural Analysis Functions in GCP 11

3.3 Using Points-to Information to Improve GCP 12

3.4 Using the Invocation Graph to Support Interprocedural GCP 12

3.4.1 Non-recursive Procedure Calls 14

3.4.2 Recursive Procedure Calls 16

3.5 Indirect Procedure Calls in GCP 17

3.6 Summary 17

4 Related Work **18**

4.1 Interprocedural Analysis 18

4.2 Constant Propagation 19

5 Conclusions and Further Work **20**

A GCP Analysis for the `while` Loop **24**

B GCP Analysis for Non-recursive Procedure Calls **26**

C GCP Analysis for Recursive Procedure Calls **27**

Abstract

This paper presents a new and practical approach to context-sensitive interprocedural analysis for C compilers. The method is based on three foundations: a simple and structured intermediate representation for C, a concrete representation for capturing the complete invocation structure of a program (the *invocation graph*), and a points-to analysis that computes the relationships between variables and the appropriate mapping information for the invocation graph. We illustrate the framework by developing a new interprocedural algorithm for generalized constant propagation.

1 Introduction and Motivation

In this paper we present the interprocedural analysis framework that has been designed and implemented for the McCAT optimizing/parallelizing C compiler.[1] This framework has been designed to handle the full complexities of the C language, while at the same time supporting a simple model for implementing complex and sophisticated analyses such as generalized constant propagation and a variety of heap analyses.

When designing our compiler, it became immediately evident that precise interprocedural analysis was critical to obtaining reasonable static approximations that could be used for advanced optimizations and parallelizing transformations. Unlike typical scientific programs written in Fortran, which often have large loop nests that are amenable to optimization and/or parallelization, the C programming methodology tends to promote procedural abstraction, and one often observes that typical C programs have a large number of procedures and procedure calls. Given the large number of, and complex interactions between procedure calls, the imprecision introduced by overly approximating the effect of procedure calls can accumulate and substantially reduce the quality of information collected. Thus, neither traditional global flow analyses that make conservative approximations on encountering function calls, nor interprocedural methods based on summarizing the effects of procedures are powerful enough for our purpose. Rather, we require a more precise *context-sensitive* approach that can accurately measure the effect of a procedure call based on the context of the call.

The major contributions of this paper include:

A general purpose framework: A major contribution of our work is that we provide a unified framework for **general purpose** context-sensitive interprocedural data flow analysis that correctly and precisely estimates the effect of a procedure call for a given calling context. Although techniques have been developed to perform context-sensitive interprocedural analysis [Mye81, HRB88, HRB90, LR92, CBC93], these techniques have typically been geared towards a particular analysis like live variable analysis, program slicing or alias analysis. In contrast, our framework provides a basis for implementing many different interprocedural analyses.

Factoring out the problems due to aliasing, function pointers and recursion: The interprocedural analysis of C programs is considerably complicated by three language features: pointers,

[1] The McGill Compiler/Architecture Testbed is being developed in order to study the interaction between compiler techniques and advanced architectural features [HDE+92].

function pointers, and recursion. We present a framework that uses a preprocessing phase which factors out these problems, thus simplifying the development of further interprocedural analyses. The preprocessing phase: (1) provides a simplified and structured high-level representation of C programs, (2) calculates the points-to relationships among variables, and (3) provides an explicit representation of the complete call structure of the program, the *invocation graph*.

A framework for a complete C compiler: We provide a framework that is not just for a toy language, but has been implemented for the complete C language.

A practical and implemented method: Even though our method, and its correctness, rests on the theoretical foundations laid down in the abstract interpretation world, we have consciously chosen to present the framework in a practical and algorithmic manner that should be accessible to other compiler writers wishing to incorporate our framework in their compilers. In this spirit, we have clearly enumerated the complexities of handling the C language and the resulting design decisions in our approach, and we have made the presentation of the framework as concrete as possible. The implementation is currently operational for the McCAT parallelizing/optimizing C compiler.

A demonstration of the framework: We demonstrate the effectiveness and simplicity of the approach by presenting a new interprocedural generalized constant propagation algorithm that is used to approximate ranges of values associated with variables. In addition to demonstrating how a complex context-sensitive interprocedural analysis is easily implemented within our framework, we also feel that the constant propagation algorithm is an interesting extension to previous work in this area.

The rest of this paper is organized as follows. First we discuss the overall structure and foundations of our approach in Section 2. These foundations provide all of the preprocessing steps that facilitate the development of a wide variety of interprocedural analyses. Based on these foundations we present concrete example of a sophisticated context-sensitive interprocedural analysis for computing generalized constant propagation in Section 3. Finally we compare other interprocedural analyses that are most related to our approach in Section 4 and give our conclusions in Section 5.

2 Foundations

Our approach is based on three foundations: (1) an appropriately designed intermediate language that can handle all of the complexities of C in a simple and structured fashion, (2) an explicit representation of the call-structure of the program, the *invocation graph*, that captures all the necessary invocation patterns even in the presence of function pointers and recursion, and (3) an approach to computing and storing the relevant information about the alias or points-to relationships among variables within procedures and the correspondence between variables in calling and called contexts. The actual organization of our implementation in the McCAT compiler is presented in Figure 1. The foundations are implemented with the preprocessing steps above the dotted line.

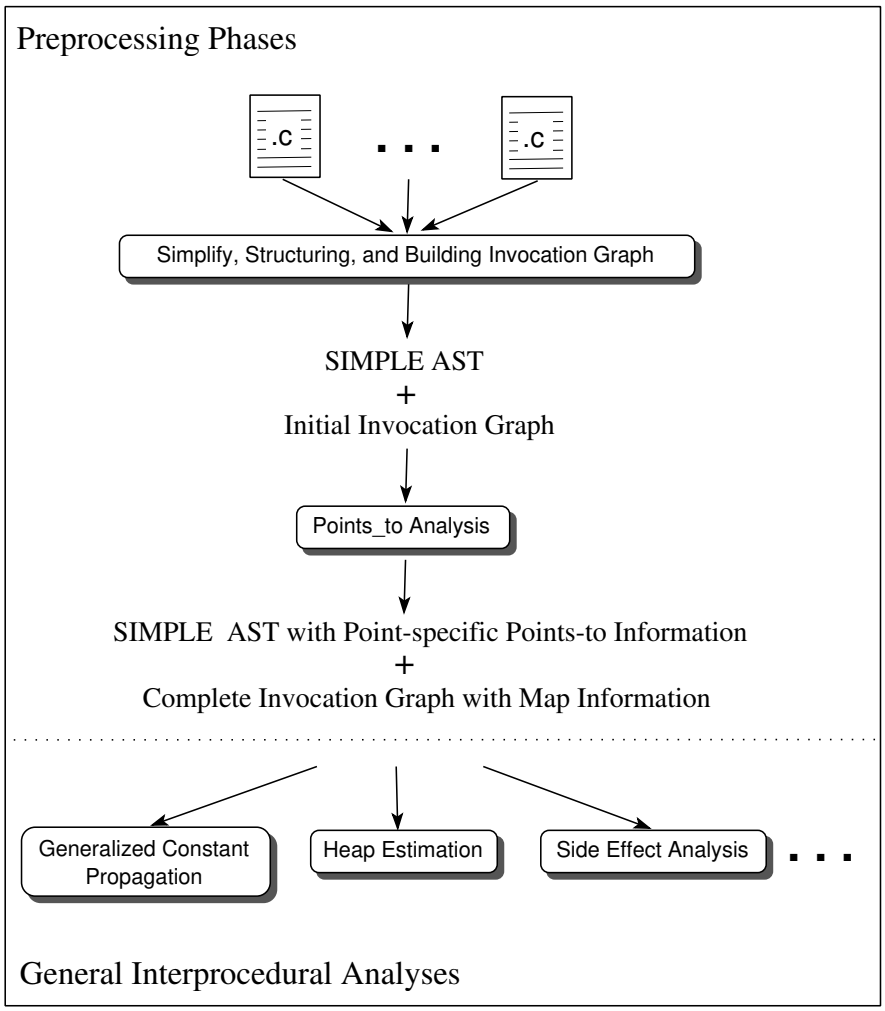


Figure 1: Implementation Overview

2.1 A Simple and Structured Intermediate Representation

One of the foundations of our approach is the design and implementation of a structured intermediate representation called SIMPLE. On one hand, we need to analyze real C programs (not just programs written in a toy subset of C), and therefore our representation faithfully represents these programs. On the other hand, we require as compact and clean a representation as possible, so that new and powerful analyses can be implemented in a straightforward and compositional fashion. As shown in Figure 1, the first phase of our implementation takes a collection of C program files, and produces a list of simplified and structured abstract syntax trees, one tree for each globally declared variable and function.

In designing the SIMPLE intermediate representation there were three major design choices. The first choice was to base our analysis framework on structured (compositional) analyses, and thus the SIMPLE representation is compositional, with control flow regular and explicit. This kind of compositional representation has three advantages: (i) the flow of control is structured and is explicit in the

program representation, (ii) structured analysis techniques and tools supporting such techniques can be used to analyze all the control-flow constructs, and (iii) it is simple to find and transform groups of loop nests. The second design choice was to decide on the level of the representation. It is our belief that most sophisticated analyses need high-level variable references and complete type and type casting information. Thus, in SIMPLE, the identity of array and structure references are retained i.e., the array and structure references are not broken down into a series of lower level statements that perform address calculations [HA92]. The third design choice is to select the basic statements, expressions, and control structures that should be explicitly supported in the intermediate representation. We have designed SIMPLE such that there are 15 basic statements, plus explicit simplified representations of the `while`, `do`, `for`, `if`, `switch`, `break`, `continue` and `return` statements.[2]

Although a complete description of the SIMPLE representation and simplification process is outside of the scope of this paper [Sri92, HDE+92, HS92], we present an example program that we use throughout this paper in Figure 2. On the left is the original C program, and on the right is our simplified version. Typical simplifications include: compiling complex statements into a series of basic statements, simplifying all conditional expressions in `if` and `while` statements to simple expressions with no side-effects, simplifying procedure arguments to either constants or variable names, and moving variable initializations from declarations to statements in the body of the appropriate procedure.

```
void ind_swap(int **ppu, int **ppv)
{ int *ptemp;
  ptemp = *ppu;
  *ppu = *ppv;
  *ppv = ptemp;
}

void incr(int *px)
{ *px = *px + 1;
}


main()
{ int a=1,b=10;
  int *pa,*pb;

  pa = &a; pb = &b;
  incr(pa);

  ind_swap(&pa, &pb);
  incr(pa);
}
```

```
void ind_swap(int **ppu, int **ppv)
{ int  *ptemp;
  ptemp = *ppu;
  *ppu = *ppv;
  *ppv = ptemp;
}

void incr(int *px)
{ int temp_0;
  temp_0 = *px; *px = temp_0 + 1;
}

int main()
{ int a; int b; int *pa; int *pb;
  int **temp_2; int **temp_1;
  a = 1; b = 10;
  pa = &a; pb = &b;
  incr(pa);
  temp_1 = &pa; temp_2 = &pb;
  ind_swap(temp_1, temp_2);
  incr(pa);
}
```

Figure 2: Example Program - *IndirectSwap*

[2]It should be noted that the unrestricted use of `goto` is not compositional and cannot be supported directly. Thus, our McCAT compiler provides a structuring phase that converts programs with unstructured control flow to equivalent programs with structured control flow [Ero93].

2.2 Representing Invocation Contexts

In order to provide precise static estimations for a wide variety of analyses, our framework supports context-sensitive interprocedural analyses. That is, when estimating the effect of a procedure call we wish to estimate the effect of the procedure call within a specific calling context and not just summarize the information for all calling contexts. In general, a calling context depends on the chain of procedure invocations starting with `main` and ending with the particular procedure call under analysis. For example, consider the example program in Figure 3(a). In this case we note that the first call to `f` is in a context where `x` is 1, while the second call to `f` has a context where `x` is 2. Thus, a context-sensitive analysis should be able to determine that in `main`, `x1` gets the value 2, while `x2` gets the value 4.

From this simple example one might assume that the calling context problem can be solved by summarizing the effect of each procedure call site. However, the example program in Figure 3(b) illustrates that a calling context may depend on the complete calling chain from `main`. In this program we have two different contexts in which `f` is called, but only one call site for `f` in the program.

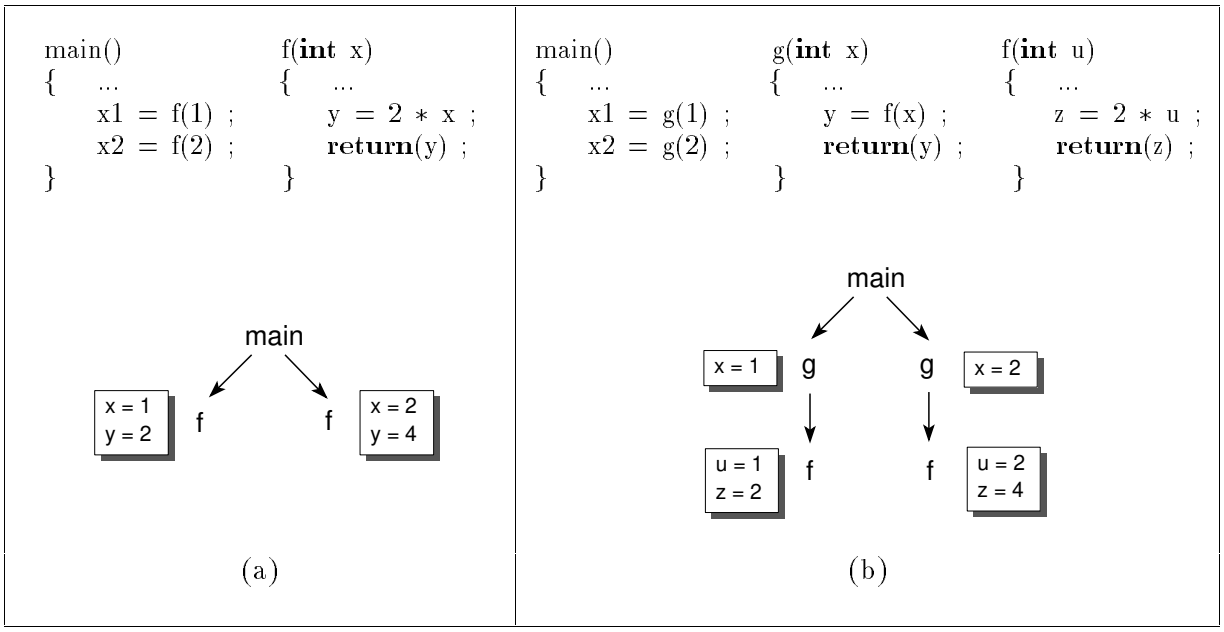


Figure 3: Invocation Contexts

The problem of ensuring that the analysis of a procedure call is specific to a particular calling context has been termed the *calling context problem* by Horwitz et al. [HRB88, HRB90], while Landi and Ryder [LR92] consider this to be the problem of restricting the propagation of information along realizable interprocedural execution paths. One traditional solution to this problem has been to include some context information in the abstraction being calculated. Typical examples of this approach include: memory components [Mye81], procedure strings [Har89, HA92], assumed alias sets [LR92] and source alias sets with the last call-site [CBC93].

Rather than embedding the context in the abstraction being estimated, we have chosen to follow a

different strategy where we **explicitly** represent all invocation paths in an *invocation graph*. As indicated in Figure 1, this invocation graph is built once during the preprocessing phases. Each interprocedural analysis then follows the invocation graph, and uses it as a place to deposit context-sensitive flow information, and as a guide for the computation of fixed-points for recursion. We have given the invocation graphs for the two small examples in Figure 3, decorated with some information about constants.

An important characteristic of the invocation graph is that each procedure invocation chain is represented by a unique path in it, and vice versa. Using the invocation graph we can distinguish not only calls from two different call-sites of a procedure (Figure 3(a)), but we can also distinguish two different invocations of a procedure from the same call-site when reached along different invocation chains (Figure 3(b)).

In the case of programs without recursion, the invocation graph is built by a simple depth-first traversal of the call structure of the program, starting with `main`.[3] In this case the invocation graph corresponds exactly to the invocation structure of the program. For the case of recursion the exact invocation structure of the program is not known statically, and we must approximate all possible unrollings of the recursion. Figure 4 illustrates a program with simple recursion, the set of all possible invocation unrollings for this program, and our invocation graph that is used to approximate all possible unrollings. To build the graph in the case of recursion one terminates the depth-first traversal each time a function name is the same as one of the ancestors on the call chain from `main`. The leaf node (representing the repeated function name) is labeled as an approximate node, and its matching ancestor node is labeled as a recursive node. We indicate the pairings of these nodes with a special back-edge from the approximate node to the recursive node. It should be noted that these back-edges are used only to match the approximate node with its appropriate recursive node, and they are therefore quite different from the other tree edges which correspond to procedure calls. This scheme is completely general. Consider, for example, the invocation graph for a program with mutual recursion as displayed in Figure 5.

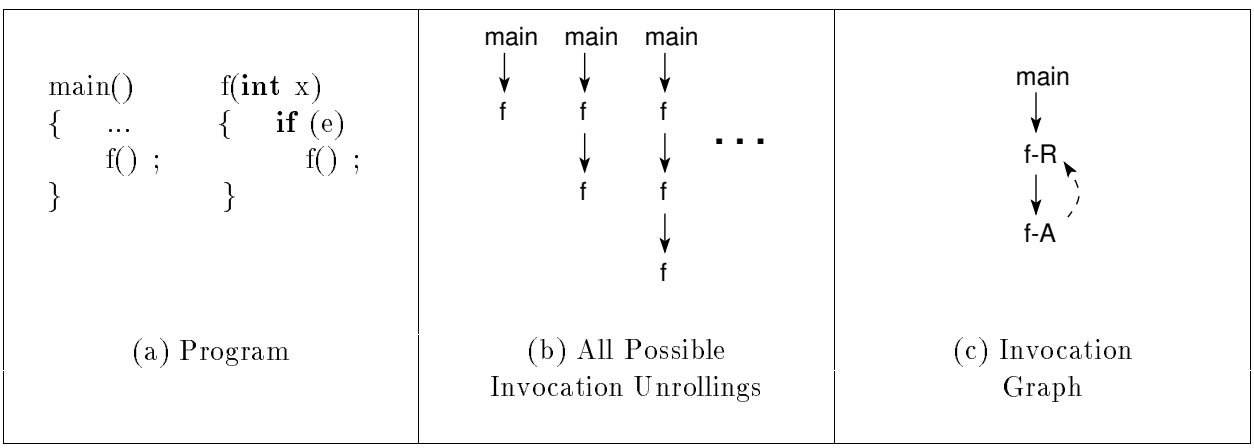


Figure 4: Invocation Graphs for Recursion

Our approach of explicitly building the invocation graph has the following advantages: (1) it cleanly separates the abstraction for any interprocedural analysis from the abstraction required to encode the

[3]The treatment of function pointers is given in Section 2.3.3

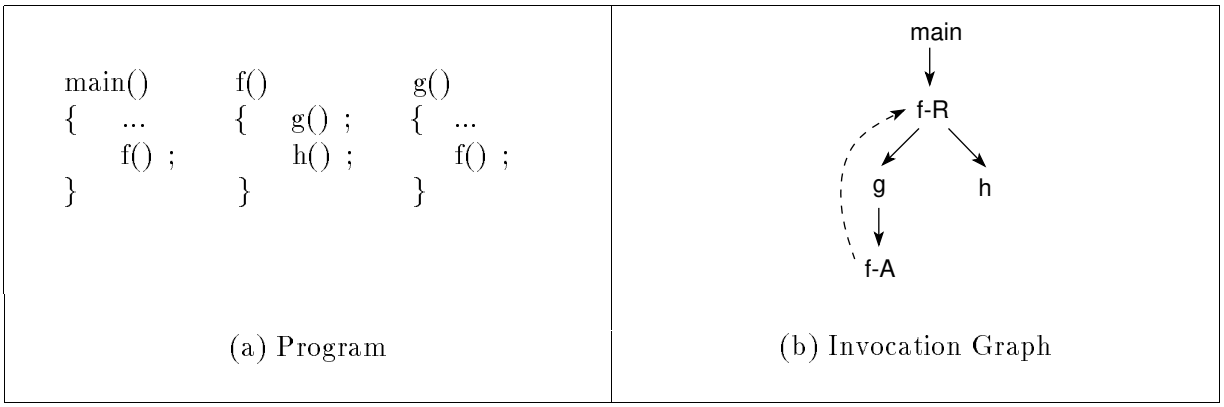


Figure 5: Invocation Graph for Mutual Recursion

calling context, (2) it allows us to deposit information computed from one analysis that can be useful for the next analysis, (3) it provides a place to store (memoize) IN/OUT pairs previously computed to summarize the effect of the function call (so that extra computation can be avoided at analysis time), and (4) it provides a simple framework for implementing simple compositional fixed-point computations for recursion.

2.3 Abstract Stack Locations and Points-to Analysis

When designing and implementing context-sensitive interprocedural analyses for C, there are three difficult problems that arise due to the presence of multi-level pointers and function pointers. In this section we outline the three problems and then show how our approach resolves these problems.

Problem 1: The first problem is that one must deal with indirect variable references of the form `*x`, `**x`, `(*x).next`, and so on. For example, if one were performing constant propagation, then the analysis of statements of the form `*x = 3` requires information about what variables `x` can point to. If `x` definitely points to some variable `y`, then we can safely associate the constant 3 with `y`. However, if `x` possibly points to either `y` or `z`, then we cannot associate the constant 3 with either `y` or `z`. Even worse, if we do not know which scalars are pointed to by `x` we must make a conservative estimate, and conclude that all accessible scalars could either by 3, or their input constant values.

Problem 2: The second problem is that indirect references can refer to variables that are outside of the scope of the function under analysis. Furthermore, these indirect references can point to different variables depending on the invocation context. This is true for our example program in Figure 2. In the first call to the procedure `incr`, `*px` refers to `a`, while in the second call to `incr`, `*px` refers to `b`.

Problem 3: The final major problem is that general pointers can point to not just memory locations, but also to functions. This means that the complete invocation graph cannot be built without resolving which functions are pointed to by function pointer references.

Given that these problems exist for **all** interesting interprocedural analyses, our approach has been to provide a pre-processing analysis called *points-to* analysis that simplifies these problems for the remaining analyses. As indicated in Figure 1, the points-to analysis takes as input the simplified ASTs, and the original invocation graph, and provides as output ASTs decorated with program-point-specific points-to information and the invocation graph decorated with mapping information that gives the correspondence between names in the caller and callee. Points-to analysis also has the effect of resolving any indirect call through a function pointer, to a set of functions that can be possibly called from the given call-site. A complete presentation of the points-to analysis itself is discussed elsewhere [Ema93, EHG93, Ghi92]. In this paper we concentrate on explaining what the points-to analysis computes, and how the results of the points-to analysis facilitate our general interprocedural approach.

2.3.1 Program-point-specific Points-to Information

The solution for **Problem 1** is to compute, for each important program point, the points-to relationships between locations on the stack. These relationships can be expressed as *(a,b,D)* meaning that the stack location for variable `a` *definitely points-to* the stack location for variable `b`, or *(a,b,P)* meaning that `a` *possibly points-to* `b`, at the given program point. Consider, for example, the four program points marked *A*, *B*, *C*, and *D* in Figure 6(a). At each point, the points-to information gives an accurate summary of the relationships among variables (we show a pictorial trace for these four program points in Figure 6(c)). Note particularly that after the call to `ind_swap pa` definitely points-to `b` and `pb` definitely points-to `a`. Thus, the first call `incr(pa)` increments `a`, while the second call increments `b`.

2.3.2 Mapping Information in the Invocation Graph

The solution to **Problem 2** is to compute, for each calling edge in the invocation graph, a mapping between the names of stack locations in the caller and names of stack locations in the called procedure. Within a procedure we perform all static analyses using only names of locals, globals, and symbolic names that represent non-local stack locations pointed to by either the parameters or globals of pointer type. By using these symbolic names we can talk about stack locations in a context-free manner, while the mapping information gives us the context-dependent information.

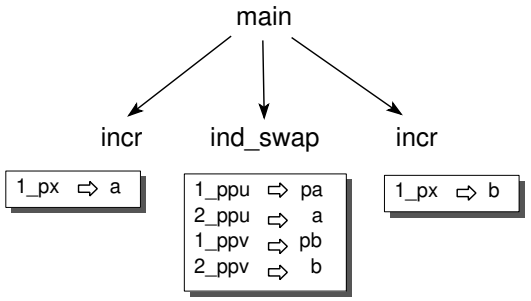
Consider, for example, the stack location names used in the points-to information for the procedure `ind_swap` in Figure 6(a). Within the procedure we refer to symbolic locations like `1_ppu` for the stack location corresponding to `*ppu` and `2_ppu` for the stack location corresponding to `**ppu`.[4]

It is important to note that there are many advantages to using context-free names within procedures. For example, it provides a good way of summarizing the effect of a procedure independent of context, and it leads to a very natural fixed-point analysis for recursion. However, as illustrated in Figure 6(b), we have not lost the context-specific information because it is completely encoded in the invocation graph. Thus, the effect of any procedure call can be computed by mapping information from the calling context to the called context, analyzing the body of the procedure being called using only local, global and symbolic names, and then mapping the information back to the calling context.

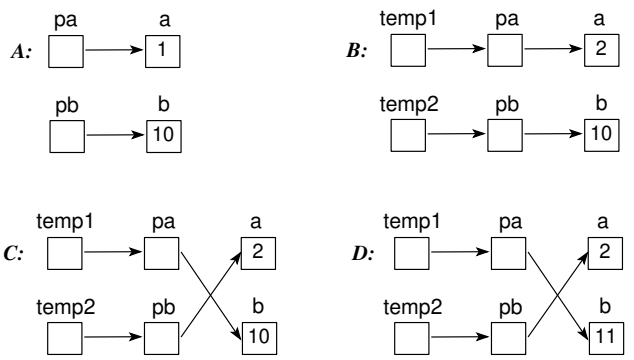
[4]Symbolic names are used only when there are *no* other names that refer to these locations. This ensures that each real stack location corresponds to exactly one abstract stack location.

```
void ind_swap(int **ppu, int **ppv)
{ int  *ptemp;
  /* IN: { (ppu,1_ppu,D), (1_ppu,2_ppu,D),
           (ppv,1_ppv,D), (1_ppv,2_ppv,D) } */
  ptemp = *ppu;
  *ppu = *ppv;
  *ppv = ptemp;
  /* OUT: { (ppu,1_ppu,D), (1_ppu,2_ppv,D),
            (ppv,1_ppv,D), (1_ppv,2_ppu,D) } */
}

void incr(int *px)
{ int temp_0;
  /* IN: { (px,1_px,D) } */
  temp_0 = *px;
  *px = temp_0 + 1;
  /* OUT: { (px,1_px,D) } */
}

int main()
{ int a; int b; int *pa; int *pb;
  int **temp_2; int **temp_1;
  a = 1; b = 10;
  pa = &a; pb = &b;
  /* A: { (pa,a,D), (pb,b,D) } */
  incr(pa);
  temp_1 = &pa; temp_2 = &pb;
  /* B: { (temp_1,pa,D), (pa,a,D),
           (temp_2,pb,D), (pb,b,D) } */
  ind_swap(temp_1,temp_2);
  /* C: { (temp_1,pa,D), (pa,b,D),
           (temp_2,pb,D), (pb,a,D) } */
  incr(pa);
  /* D: { (temp_1,pa,D), (pa,b,D),
           (temp_2,pb,D), (pb,a,D) } */
}
```

(a) Program with Points-to Information



(b) Invocation Graph and Map Information



(c) Memory Trace

Figure 6: The *IndSwap* Example after Points-to Analysis

One can see how different contexts are captured by the mapping information by examining the two calls in the procedure `incr`. Within the procedure `incr` `px` definitely points-to location `1_px` independent of calling context. However, when analyzing the first call `incr(pa)` the value associated with `1_px` will be mapped to `a` in the main program, while analyzing the second call to `incr(pa)`, `1_px` is mapped to `b`.

2.3.3 Resolving Function Pointers

In addition to computing the points-to and mapping information, the points-to analysis must also complete the invocation graph by resolving which functions are invocable from each indirect function pointer call. One might wonder why the completion of the invocation graph must proceed at the same time as the points-to analysis. Consider that the complete invocation graph cannot be built before points-to analysis because the meaning of an indirect call `(*pf)()` is determined by examining the objects that `pf` may point-to. However, points-to analysis itself is a context-sensitive interprocedural analysis that needs the invocation graph. Thus, the two approximations must be calculated at the same time.

As an example, consider the program in Figure 7(a), with an indirect call `fp()` inside the while loop. The (incomplete) invocation graph of the program before points-to analysis is given in Figure 7(b). When points-to analysis encounters the indirect call `fp()`, during the first iteration of the while-loop fixed-point computation, it finds the current points-to set of `fp` to be `(f)`. The invocation graph is updated according to this information, as shown in Figure 7(c) and the function `f` is analyzed in the current calling context. During the second iteration of the fix-point computation, the points-to set of `fp` becomes `(g)`, and the invocation graph again gets updated as shown in Figure 7(d). Future iterations do not modify the points-to set of `fp`, and thus the points-to analysis constructs the complete invocation graph for other interprocedural analyses. Other analyses measure the effect of a function pointer call, by merging the outputs obtained by individually analyzing in the current calling context, all the functions represented by its child nodes in the invocation graph.

3 Developing an Interprocedural Analysis: Generalized Constant Propagation

Given the foundations outlined in the previous section, it is now straightforward to define structured context sensitive analyses for a wide variety of problems. To illustrate this point, we have selected an individual analysis: *Generalized Constant Propagation*. GCP has the advantages of functioning within a non-trivial domain, having a reasonably complex fixed-point calculation, and actually producing interesting and useful information.

3.1 GCP in General

GCP is an analysis that approximates the possible values each variable could assume at each point in the program for any execution of the program. GCP requires a top-down compositional analysis. For the sake of simplicity, we do not attempt to calculate the precise *set* of values for each variable. Instead, we approximate the set with just two values, the minimum and maximum value possible. In other words, our semantic domain is the domain of *ranges*[5]: closed (integer) intervals, partially-ordered by inclusion with both a smallest element ([] – the empty range) and a largest element ([-∞..∞], where by ∞ we mean the largest representable integer value). Each variable at each point in the program is associated with a single range, with the aim that by the end of the GCP calculation, any value a variable might

[5]A word about notation: If $\alpha \leq x \leq \beta$ for some α and some β, then we write "$x : [\alpha..\beta]$" meaning that x must fall within the (inclusive) range α to β.

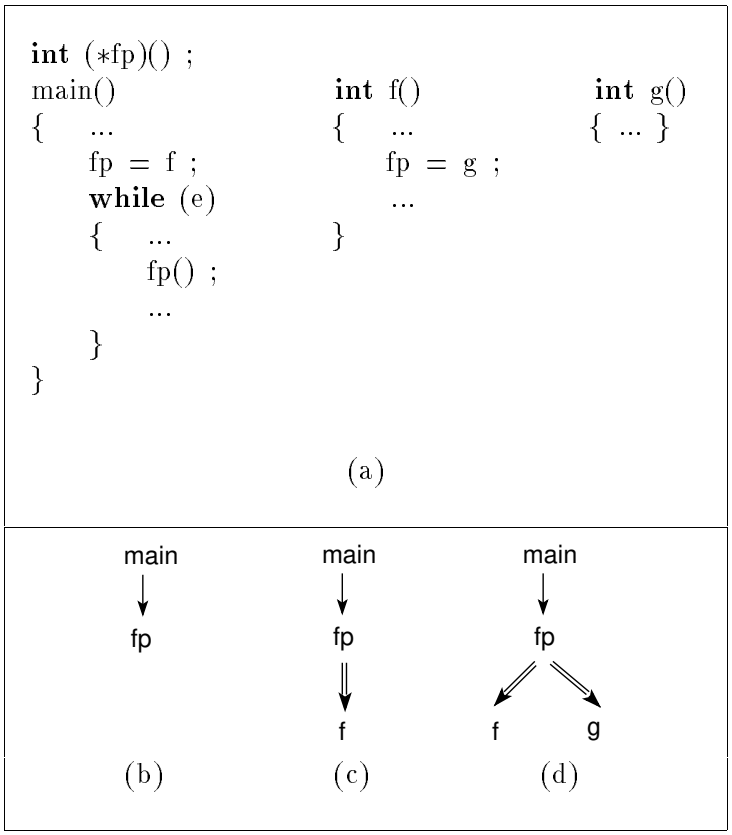


Figure 7: Invocation Graph for Function Pointers

actually assume at a given point during a given execution of the program will be contained within its corresponding range.

Such information has numerous applications. It should be immediately clear that GCP is more powerful than 'standard' constant propagation, and thus should allow for more intelligent detection of dead code. There is also some possibility of using GCP to perform static range-checking. Our particular interest, though, is using GCP to enhance the efficacy of other analyses that depend on values of variables (e.g. various loop transformations, and improving the precision of array dependence testers).

3.2 Intraprocedural Analysis Functions in GCP

GCP is a semantic analysis based on the style of abstract interpretation [CC77]. Thus, every type of statement in the SIMPLE AST (both expression and control) needs a semantic analogue; we need separate methods for determining how GCP information is altered by assignments, the various arithmetic operations, conditionals, loops, and other program constructs. We cannot describe all the semantic functions, but in Figures 8 and 9 we show the semantic operations corresponding to a few different kinds of statements in a compositional form. We hope that this will give the reader some idea of the flavour of the effort. Note that the structured nature of our SIMPLE representation lends itself to concise and

relatively compact analysis rules. Let us consider some of the rules in more detail. A description of the fixed-point analysis for the `while` loop is given in Appendix A.

Addition is quite straightforward; the result of adding two numbers within two ranges is merely the range computed from the sum of the two minimums and the two maximums. There is one complication: in order to ensure the monotonicity of our semantic addition, we cannot allow overflow (i.e. ∞ + anything-greater-than-0 = ∞) or underflow to occur.

Conditionals (*if*-statements) are one of the constructs that actually allow GCP to improve the information being carried through the program. In addition to evaluating the conditional to see if it can be resolved at compile-time [Har77, WZ91], we can also make two refinements to the ranges: (1) the input for the *then*-side of the conditional must be consistent with the conditional, and (2) the input to the *else*-side must be inconsistent with conditional. Our input to the conditional is then split into two (potentially different) inputs for the two sides of the conditional, and then merged at the end of the conditional. Note that the simplicity and ease with which we can express this optimization is a consequence of the structured nature of our approach, and the simplified nature of our intermediate representation.

For example, if our conditional is of the form $a \leq b$ where $a : [5..10], b : [7..9]$, then it must be that as we begin processing the *then*-side $a \leq b$ and $b \geq a$. In terms of ranges, we have $a : [5..9], b : [7..9]$. Similarly, on the *else*-side, it must be that $a > b$ and $b < a$, or $a : [8..10], b : [7..9]$. Once we have processed both sides of the conditional we merge both outputs, so the information is only improved within bodies of the *then* and (and not afterward). Still, this technique can yield useful information.

3.3 Using Points-to Information to Improve GCP

GCP, like many analyses, benefits greatly if points-to information is known. Observe that our assignment rule when the right-hand-side is a dereferenced pointer (in Figure 8) requires locating all variables to which the dereferenced pointer could point, and merging together the corresponding range for each such variable. When we have accurate points-to information for the pointer, this statement will still produce accurate information. However, without points-to, the only safe assumption upon encountering a dereferenced pointer is to assume that any variable could be referenced; the assignment statement mentioned above reduces to a merge of *every* range. Needless to say, this sort of mass merging is devastating to the quality of information gathered.

3.4 Using the Invocation Graph to Support Interprocedural GCP

Without an interprocedural strategy, the evaluation of procedure calls for GCP must make the safe assumption that any variable that could be reached by indirection through the actual parameters of the call, could thereafter be any value, since we cannot know which variables might have been changed by the callee. Note that this also includes all global variables. As illustrated with the rules given in Figure 10, we can easily extend our analysis into a context-sensitive interprocedural GCP analysis by taking full advantage of the invocation graph, the mapping information, and the points-to information.[6] Further, it should be noted that the precision of interprocedural GCP would depend on the way points-to information

[6]For this technical summary we will give the basic idea behind these rules. We also provide illustrative examples of the non-recursive case in Appendix B, and the recursive case in C. A more complete and integrated approach will be given in the full paper.

```
  /* Given a statement S, an input GCP set, and an invocation graph
     node ign, return the output GCP set */
fun process_stmt (S,Input,ign) =
  if basic_stmt(S)
    return(process_basic_stmt(S,Input))
  else
    case S of
      < SEQ(S1,S2) > =>
        return(process_stmt(S2,process_stmt(S1,Input,ign),ign))
      < IF(cond,thenS,elseS) > =>
        return(process_if(cond,thenS,elseS,Input,ign))
      < WHILE(cond,bodyS) > =>
        return(process_while(cond,bodyS,Input,ign))
      ...

fun process_basic_stmt(S,Input) =
  case S of
    < x = y > =>
      /* RangeOf returns the range for the given variable in the given input set */
      return(Input - x:RangeOf(x,Input) + x:RangeOf(y,Input))
    < x = *y > =>
      /* MergeRanges returns the smallest range ≥ every range in a given list.
         Dereference returns a list of variables pointed to by the given variable
         at the given statement. RangesOf is the list-version of RangeOf */
      [a..b] = MergeRanges(RangesOf(Dereference(y,S)),Input)
      return(Input - x:RangeOf(x,Input) + x:[a..b])
      /* Merge pair-wise merges two lists of ranges for the same set of variables.
         DefinitelyPointsTo returns true if the given variables have definite points-to
         relationship at the given program point */
    < *x = y > =>
      derefx = Dereference(x,S)
      if (DefinitelyPointsTo(x,derefx,S))
        return(Input - derefx:RangeOf(derefx,Input) + derefx:RangeOf(y,Input))
      /* If x does not definitely point to a single variable, then the
         strategy is to merge the range from the right-hand-side of the
         statement with the range of all variables x can possibly point to */
      foreach temp in derefx
        Input = Input - temp:Rangeof(temp,Input) +
                        temp:Merge(RangeOf(temp,Input),Rangeof(y ,Input))
      return(Input)
    < x = y + z > =>
      [a..b] = semantic_plus(RangeOf(y,Input),RangeOf( z,Input))
      return(Input - x:RangeOf(x,Input) + x:[a..b])
      ...
```

Figure 8: Compositional Intraprocedural Rules for GCP (continued on next figure)

```
fun semantic_plus([a..b],[c..d]) =
  if (a+c < −∞) e = −∞ else e = a+c
  if (b+d > ∞) f = ∞ else f = b+d
  return([e..f])

  /* Given a simple conditional cond, the statements in the then and
     else part, GCP input and an invocation graph node ign, return
     the gcp output set. */
fun process_if(cond,thenS,elseS,Input,ign) =
  /* ConstrainConditional splits the input set into two sets; one consistent
     with the conditional (to serve as input to the then statement), and one
     inconsistent (for the else statement). If either set is empty, then
     the input is such that the conditional can only have one possible outcome */
  (consistentInput,inconsistentInput) = ConstrainConditional(cond,Input)
  /* Do not process the then statement if we've determined that it
     cannot be executed! */
  if (consistentInput != {})
    thenOutput = process_stmt(thenS,consistentInput,ign)
  else
    thenOutput = {}
  if (inconsistentInput != {})
    elseOutput = process_stmt(elseS,inconsistentInput,ign)
  else
    elseOutput = {}
  return(Merge(thenOutput,elseOutput))
```

Figure 9: Compositional Intraprocedural Rules for GCP

is stored in the SIMPLE AST. The most accurate method is to store the points-to information for each invocation of a function, by tagging the points-to information with the corresponding invocation graph node. A more concise, but potentially less accurate method is to merge the points-to information for all calling contexts of the given function.

3.4.1 Non-recursive Procedure Calls

The first case in Figure 10 corresponds to a call to an ordinary invocation graph node (i.e. one that is not involved in a fixed-point computation). The general idea is that the information coming from the calling invocation graph node is mapped using the information stored on the appropriate calling arc, the body of the procedure is evaluated, and the output is unmapped and returned. The map and unmap functions for GCP are given in Figure 11. Note that we have also included an optimization for this scheme whereby the last input/output pair computed for this call is stored in the invocation graph node, and the output is only recomputed when the input changes. This is an obvious use of the invocation graph, and it avoids many redundant computations for calls that are inside of loops or recursion.

```
fun process_call(Input,actualList,formalList,ign,funcBody,mapInfo) =
  case ign of
    < Ordinary > =>
      funcInput  = gcp_map(Input,formalList,actualList,mapInfo)
      if (funcInput == ign.storedInput)  /* already computed */
        return(gcp_unmap(Input,ign.storedOutput,mapInfo))
      else /* compute output,  store input and output */
        funcOutput = process_stmt(funcBody,funcInput,ign)
        ign.storedInput = funcInput   ign.storedOutput = funcOutput
        return(gcp_unmap(Input,funcOutput,mapInfo))

    < Approximate > =>
      funcInput  = gcp_map(Input,formalList,actualList,mapInfo)
      recIgn = ign.recEdge  /* get partner recursive node in inv. graph */
      /* if this input is contained in  stored input, use stored ouput */
      if isSubsetOf(funcInput,recIgn.storedInput)
        return(gcp_unmap(Input,recIgn.storedOutput,mapInfo))
      else  /* put this input in the pending list, and return Bottom */
        addToPendingList(funcInput,recIgn.pendingList)
        return Bottom

    < Recursive > =>
      funcInput  = gcp_map(Input,formalList,actualList,mapInfo)
      if (funcInput == ign.storedInput)  /* already computed */
        return(gcp_unmap(Input,ign.storedOutput,mapInfo))
      else
        ign.storedInput  = funcInput  /* initial input estimate */
        ign.storedOutput = Bottom     /* initial output estimate */
        ign.pendingList  = {}         /* no unresolved inputs pending */
        done = false
        do
          /* process the body */
          funcOutput = process_stmt(funcBody,storedInput,ign)
          /* if there are unresolved inputs, merge inputs and restart */
          if (ign.pendingList != {})
            ign.storedInput = Merge(ign.storedInput,pendingListInputs)
            ign.pendingList = {}   ign.storedOutput = Bottom
          /* check to see if the new output is included in old output */
          else if isSubsetOf(funcOutput,ign.storedOutput)
              done = true;
          else /* merge outputs and try again */
            ign.storedOutput = Merge(ign.storedOutput,funcOutput)
        while (not done)
        /* reset stored input to initial input for future memoization */
        ign.storedInput = funcInput
        /* return the fixed-point after unmapping */
        return(gcp_unmap(Input,ign.storedOutput,mapInfo))
```

Figure 10: Compositional Interprocedural Rules for GCP

```
fun gcp_map(Input,formalList,actualList,mapInfo) =
  funcInput = {}
  foreach formalI in (formalList) /* formals inherit the range from actuals */
    funcInput = funcInput + formalI:RangeOf(actualI,Input)
  foreach globalI in (globalVarList) /* the range of globals remains the same */
    funcInput = funcInput + globalI:RangeOf(globalI,Input)
  foreach x in (SymbolicVars(mapInfo))
    mappedVars = getMappedVars(x,mapInfo)
    funcInput = funcInput + x:MergeRanges(RangesOf(mappedVars),Input)
                /* a symbolic variable receives the merged range of the
                   variables it represents */
  return(funcInput)

fun gcp_unmap(Input,funcOutput,mapInfo) =
  Output = Input /* initialize the Output of the call to its Input */
  foreach globalI in (globalVarList)
    Output = (Output - globalI:RangeOf(globalI,Input) +
              globalI:RangeOf(globalI,funcOutput))
              /* each global gets the new range from the called function */
  foreach x in (SymbolicVars(mapInfo))
    mappedVars = getMappedVars(x,mapInfo)
    foreach var in (mappedVars)
      Output = (Output - var:RangeOf(var,Input) + var:RangeOf(x,funcOutput))
               /* each variable represented by a symbolic variable in
                  the called function, gets the range of the symbolic
                  variable */
  return (Output)
```

Figure 11: The Map/Unmap Functions for GCP

3.4.2 Recursive Procedure Calls

The second and third cases in Figure 10 work together to implement a safe and accurate fixed-point computation for recursion. As we have explained previously, all possible unrollings for call-chains involving recursion are approximated by introducing matched pairs of recursive and approximate nodes in the invocation graph. Each recursive node marks a place where a fixed-point computation must be performed, while each approximate node marks a place where the current stored approximation for the function should be used (instead of evaluating the call, the stored output is used directly).

At each recursive node we store an input, an output, and a list of pending inputs. The input and output pair can be thought of as approximating the effect of the call associated with the recursive function (let us call it `f`). The fixed-point computation generalizes the stored input until it finds an input that summarizes all invocations of `f` in any unrolled call tree starting at the recursive node for `f`. Similarly, the output is generalized to find a summary for the output for any unrolling of the call tree starting in the recursive node for `f`. The generalizations of the input and output may alternate, with a new generalization of the output causing the input to change.

Let us consider the rule for the approximate node in Figure 10. In this case, the current input is compared to the the stored input of the matching recursive node. If the current input is contained in the stored input, then we use the stored output as the result. Otherwise, the result is not yet known for this input, so the input is put on the pending list, and BOTTOM is returned as the result. Note that an approximate node never evaluates the body of a function, it either uses the stored result, or returns BOTTOM.

Now consider the recursive rule. In this case we have an iteration that only terminates when the input is sufficiently generalized (the pending list of inputs is empty) and the output is sufficiently generalized (the result of evaluating the call doesn't add any new information to the stored output).

3.5 Indirect Procedure Calls in GCP

Indirect procedure calls through function pointers are easily incorporated in the analysis. The points-to analysis has resolved all the functions possibly pointed to by the function pointer, and so one just analyzes each possibility, merging the outputs. The exact rule is given in Figure 12.

```
fun process_call_indirect(Input,actualList,ign) =
  ignSet = childNodesOf(ign) /* set of functions invocable */
  funcOutput = {}
  foreach igNode in (ignSet)
    igNodeOutput = process_call(Input,actualList,igNode.formalList,igNode,
                                igNode.funcBody,igNode.mapInfo)
                   /* get the output for each invocable function */
    funcOutput = Merge(funcOutput,igNodeOutput) /* merge the outputs */
  return(funcOutput)
```

Figure 12: Compositional Rule for an Indirect Procedure Call

3.6 Summary

From the GCP example, it should be clear that it is straightforward to define a new and sophisticated interprocedural analysis based on our framework. We can summarize the process as follows:

1. Define an analysis function for each simple statement and expression that affects the approximation under analysis. For those statements that involve pointer indirections take full advantage of the points-to information stored in the AST.

2. Define a rule for each structured statement, possibly taking advantage of the conditionals and incorporating the appropriate stepping functions (described in Appendix A).

3. Define a map and unmap function for the approximation under analysis. These functions are simple to define given the simplified procedure calls in our intermediate representation, and the mapping information stored in the invocation graph.

4. Define the rules to support both non-recursive and recursive procedure calls. The basic structure of these rules will be exactly the same as for GCP.

4 Related Work

4.1 Interprocedural Analysis

Most of the work done on interprocedural data flow analysis has focused on computing summary data flow information for procedure calls [Spi71, All74, Bar77, Ban79, Ros79, CK84, CK88, Bur90]. [7] The presence of aliases induced by reference formals, poses the major complication for these analyses. More recently, algorithms for determining modification side effects of procedure calls, for languages with aliasing induced by general pointers, have been proposed [CBC93, LRZ93].

Myers [Mye81] was the first to formulate context-sensitive interprocedural data flow solutions for problems like available expression and live variable analysis. He partially resolved the calling context problem by associating a *memory component* (essentially the last call-site) with the flow information, to propagate information only along execution paths. However, he proved that finding precise solutions to these problems is NP-complete in the presence of aliasing.

Jones and Muchnick [JM82] describe a general framework for interprocedural analysis. They use a token set to capture the status of the call stack of the program. Based on the choice of the token set, they can represent a whole range of interprocedural analyses: from collecting summary information to exact computation description. Although flexible, their method is mostly of theoretical interest. In contrast, we have given a more concrete and implemented approach.

Horwitz et al. [HRB88, HRB90] developed an algorithm for interprocedural static program slicing. They use a variation of attribute grammars called linkage grammars along with their system dependence graph program representation to effectively side-step the calling context problem. Their technique is particularly geared towards slicing, and their formulation of the problem is not exactly like a data flow problem.

Harrison and Ammarguellat [Har89, HA92] use procedure strings for the interprocedural analysis of Scheme and C programs. The procedure string records all of the procedure calls and returns made from the beginning of the program, and thus helps in correct pairing of calls and returns during interprocedural analysis. Their goals are very similar to ours, however, their analysis works on lower-level intermediate representation. A more precise comparison between the two approaches will require substantial experiments to compare the complexity and results.

Landi and Ryder [LR92] propose a *conditional* approach for context-sensitive interprocedural alias analysis in the presence of general pointers. They associate an *assumed-alias* pair with every alias pair, where the validity of the alias pair at a program point is conditional on the validity of the assumed-alias pair at the entry node of the procedure containing the program point. They recover the calling context by determining the call-sites which can propagate the assumed-alias pair to the entry node of the procedure under analysis. Their scheme is precise for single level pointers. However, in the presence

[7] As our topic is very general, it encompasses a considerable body of literature. Thus, we have endeavored to provide those references most relevant to our approach. However, if we have missed something that the reviewers find pertinent, we will certainly include it in our final paper.

of multi-level pointers, it can merge information arriving from different call-sites and also propagate information to extraneous call-sites. Further, their *conditional* approach is not easily extensible to other analyses. It requires designing a completely new abstraction for each context-sensitive interprocedural analysis [PRL91].

Choi et al. [CBC93] also give a context-sensitive algorithm for interprocedural pointer-induced aliasing. They associate a source alias set $\texttt{SAS}_C$ and a call-site `C` with each alias pair, where the alias pairs in $\texttt{SAS}_C$ induce the given alias pair, via the call-site `C`. They use the call-site information to recover context, and to avoid merging alias information arriving from different call-sites. The role of source alias set abstraction is not clear from their paper. Essentially, they summarize context by using call stack information of depth one, and hence cannot properly handle information propagated along call-chains of size greater than one. Further, they cannot distinguish between information propagated by two different invocations of a procedure from the same call-site.

Compared to the previous two approaches, we use the invocation graph to avoid propagating information to inappropriate call-sites. For the same reason, we never merge information arriving from different call-sites or from different invocations of a procedure from the same call-site, to derive new (spurious) information. Further, none of the above techniques can handle function pointers (alternatively termed as procedure variables). While we can easily accommodate them in our analysis framework, during the points-to analysis phase [Ghi92, EHG93]. Another important point is that our analysis framework is more general purpose than most of the related work. As demonstrated by the development of GCP, we have a framework that allows one to easily build many different context-sensitive interprocedural analyses.

The problem of constructing the call multigraph of a program in the presence of procedure variables has been previously looked at [Spi71, Ryd79, Wei80, CCHK90]. Lakhotia [Lak93] has proposed the most recent solution to this problem. He formulates it as a problem of propagating sets of procedure-values, which is similar to constant propagation, except that the domain is the powerset of all procedure constants. However, his technique cannot handle C features like multi-level function pointers, arrays of function pointers, and the type-casting of function pointers into pointers and vice versa, which we can easily handle within our points-to analysis framework [Ghi92].

4.2 Constant Propagation

Constant propagation is a popular analysis that has appeared in numerous forms over the years [CCKT86, SK91, BDK91, WZ91, GT93], both as an intraprocedural and as an interprocedural analysis. Generalized constant propagation, however, has not enjoyed as much attention; Harrison's [Har77] article on the *range propagation* problem is one of the few papers to address this type of analysis, albeit in a more 'traditional' setting based on precomputation of control-flow diagrams and *def* and *use* sets. GCP information also bears some resemblance to the output of analyses that aim to determine program invariants (applied in hopes of producing provably correct programs) [DM81]. This is a large body of literature with very different goals, but GCP can in some sense be considered a spawn of such efforts. Our concept of 'stepping' (described in Appendix A) also has relatives; the *widening/narrowing* tactic of Cousot and Cousot [CC92], though directed at infinite domains rather than very 'tall' finite ones, has much the same effect: making semantic analyses tractable in the presence of uncooperative domains.

5 Conclusions and Further Work

In this paper we have presented a practical, general-purpose, context-sensitive interprocedural framework and we have demonstrated how this approach has been implemented for an optimizing/parallelizing compiler.

Our approach rests on providing a concise structured intermediate representation that can be used to represent C programs, and a pre-processing analysis that computes point-specific points-to information, and an invocation graph decorated with mapping information that explicitly encodes all calling contexts. By providing these foundations, we resolve many of the difficulties in analyzing C, and provide a base upon which many interprocedural analyses can be built. We have demonstrated this by providing a new interprocedural analysis for generalized constant propagation.

We have implemented the method, and empirically it appears to have reasonable complexity for a variety of complex analyses and benchmark programs. The worst-case complexity of the analysis depends on the height of the partial ordering used in a particular analysis, the stepping heuristic used for high partial orderings, the maximum loop nesting level, the invocation structure of the program and the nesting of recursion. To date, we have observed that real analyses and real programs have characteristics that make our approach realistic. In the future we plan to make an empirical study to make this observation more concrete.

References

[ACM77] ACM SIGACT and SIGPLAN. *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, January 17–19, 1977.

[ACM88] ACM SIGPLAN. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 22–24, 1988. Also in *SIGPLAN Notices*, 23(7), July 1988.

[All74] Frances E. Allen. Interprocedural data flow analysis. *Proceedings IFIP Congress 74*, pages 398–402, 1974.

[Ban79] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, January 1979.

[Bar77] Jeffrey M. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* [ACM77], pages 119–131.

[BDK91] N. Tawbi B. Dehbonei, C. Laurent and R.S. Kulkarni. Pmac: An environment for parallel programming. In *The Fifth International Parallel Processing Symposium*, pages 638–641. IEEE, May 1991.

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

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

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages* [ACM77], pages 238–252.

[CC92] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. Technical Report LIX/RR/92/09, Ecole Polytechnique Laboratoire d'Informatique, 91128 Palaiseau Cedex, France, Juin 1992.

[CCHK90] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, April 1990.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, Palo Alto, California, June 25–27, 1986. ACM SIGPLAN. Also in *SIGPLAN Notices*, 21(7), July 1986.

[CK84] Keith Cooper and Ken Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 247–258, Montréal, Québec, June 17–22, 1984. ACM SIGPLAN. Also in *SIGPLAN Notices*, 19(6), June 1984.

[CK88] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* [ACM88], pages 57–66. Also in *SIGPLAN Notices*, 23(7), July 1988.

[DM81] Nachum Dershowitz and Zohar Manna. Inference rules for program annotation. *IEEE Transactions on Software Engineering*, SE-7(2):207–222, March 1981.

[EHG93] M. Emami, L. Hendren, and R. Ghiya. Context-sensitive interprocedural points-to analysis in the presence of function pointers. ACAPS Technical Memo 54, School of Computer Science, McGill University, Montréal, Québec, In Preparation 1993.

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, August 1993.

[Ero93] A. Erosa. A program structurer for the McCAT C compiler. Master's thesis, McGill University, expected December 1993.

[Ghi92] Rakesh Ghiya. Interprocedural analysis in the presence of function pointers. Technical report, McGill University, December 1992. ACAPS Technical Memo 62.

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

[HA92] W. Ludwell Harrison III and Zahira Ammarguellat. A program's eye view of Miprac. In *Conference Record of Fifth Workshop on Languages and Compilers for Parallel Computing, YALEU/DCS/RR-915*, pages 339–353. Yale University, August 1992. To appear as LNCS.

[Har77] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, May 1977.

[Har89] Luddy Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. Technical Report 860, February 1989.

[HDE+92] Laurie J. Hendren, Chris Donawa, Maryam Emami, Guang R. Gao, Justiani, and Bhama Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Conference Record of Fifth Workshop on Languages and Compilers for Parallel Computing, YALEU/DCS/RR-915*, pages 261–275. Yale University, August 1992. To appear as LNCS. Also available as ACAPS Technical Memo 46, McGill University.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* [ACM88], pages 35–46. Also in *SIGPLAN Notices*, 23(7), July 1988.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HS92] L. Hendren and B. Sridharan. The SIMPLE AST - McCAT compiler. Technical report, McGill University, October 1992. ACAPS Design Note 36.

[JM82] N. D. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Lak93] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 273–284, 1993.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 SIGPLAN Symposium on Programming Language Design and Implementation*, 1992.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the 1993 SIGPLAN Symposium on Programmi ng Language Design and Implementation*, 1993.

[Mye81] Eugene W. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, January 1981.

[PRL91] H. D. Pande, B. G. Ryder, and W. Landi. Interprocedural def-use associations for C programs. In *Proceedings of the ACM SIGSOFT Conference on Testing, Analysis and Validation*, pages 139–153, October 1991.

[Ros79] Barry K. Rosen. Data flow analysis for procedural languages. *JACM*, 26(2):322–344, April 1979.

[Ryd79] Barbara Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, May 1979.

[SK91] B. Steffen and J. Knoop. Finite constants: Characterizations of a new decidable set of constants. *Theoretical Computer Science*, 80(2):303–318, March 1991.

[Spi71] T. C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Proceedings of the 1971 IFIPS Congress*. North Holland Publishing Co., Amsterdam, 1971.

[Sri92] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.

[Wei80] William Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, January 1980.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

A GCP Analysis for the `while` Loop

Since the GCP information must be valid for any run of the program, we need to ensure that the GCP ranges within a loop are valid for any number of iterations, even in the absence of explicit loop bounds. We need, in other words, to compute the fixed point of the loop with respect to GCP information. Of course we have to be certain that such a fixed point exists in all cases; this can be done by ensuring that all the semantic operations are monotonic in the semantic domain of ranges.

The process of calculating this fixed point is described both diagrammatically and functionally in Figure 13. The original input is first merged with the previous loop-body output. The result is passed through the conditional of the loop, splitting the input into two; the set which is consistent with the condition imposed by the loop conditional runs through the loop body, and the set which is not consistent with the conditional is merged with the final output of the loop. This repeats until the fixed point is reached.

```
fun process_while(cond,bodyS,Input,ign) =
  iterations = 0
  Output = oldBodyInput = {}
  do
    Input = Merge(Input,bodyOutput)
    (consistentInput,inconsistentInput) = ConstrainConditional(cond,Input)
    converged = (consistentInput == oldBodyInput)
    if (not converged and iterations > maxiterations)
      /* StepUp artificially moves a non-converging range up in the
         semantic domain */
      StepUp(consistentInput,oldBodyInput)
      iterations = 0
    bodyOutput = process_stmt(bodyS,consistentInput,ign)
    Output = Merge(Output,inconsistentInput)
    iterations = iterations + 1
  while (not converged)
  return(Output)
```

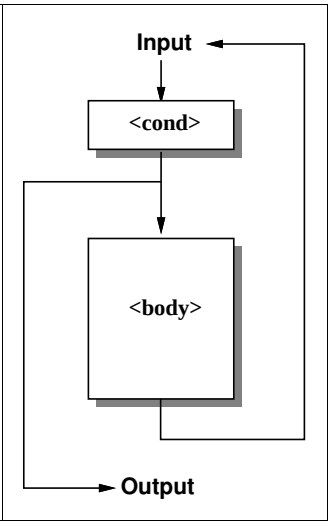


Figure 13: Calculating the Fixed Point of a While Loop

As mentioned, each of our semantic functions is monotonic in the domain of ranges, so we will reach a fixed point eventually. Unfortunately, the domain of ranges can be quite 'tall;' there are monotonic chains of *very* long length: e.g. $[0..0] \leq [0..1] \leq [-1..1] \leq \ldots \leq [-\infty..\infty]$ where ∞ is typically 2^{31}. In the worst case then, our monotonicity requirement only ensures convergence after 2^{32} steps per variable, which is clearly unacceptable.

We can, however, speed up this process at a cost of reduced quality of information. We can, for instance, artificially 'step up' (raise in the semantic domain) the ranges for variables that refuse to converge after some fixed number of iterations. If each range can be stepped only so many times before reaching $[-\infty..\infty]$, then the monotonicity of our semantic operations guarantees convergence in much less time. By using some heuristics to guide the choice of which variable to step (e.g. choosing the loop index first), we can achieve a reasonable compromise between efficiency and accuracy. [8] Note that this

[8] Even with this optimization, in the worst case our approach still has a running time bounded exponentially in the depth

same idea can be applied to the fixed-point calculation for recursion in the interprocedural case.

The following example illustrates a typical fixed-point calculation on a while loop.

```
main() {
  int i = 1;
  int n = 20;
  int sum = 0;

  /* GCP: {i:[1..1],n:[20..20],sum:[0..0]} */

  /* Input to Loop: Point I */
  while (i <= n) {
        /* Split I over the conditional: Point S */
        sum = sum + i;
        i++;
        /* Out from Loop Body: Point B */
  }
  /* Out from Loop: Point O */
```

It takes 5 iterations to reach a fixed point, and the steps in each iteration are:

1. Merge the info from point B with point I.
2. Split I over the conditional, forming two sets of ranges:
   One consists of the ranges that result if the conditional is True, and forms the input set to the body of the loop (point S).
   The other consists of the ranges that result if the conditional is False, and is merged with the information at point O to form the output of the entire loop.
3. The loop body is processed, producing a new rangeset B.
4. Go back to step 1 and repeat until convergence.

And we can summarize the information at each point as it develops over the 5 iterations in the tables below:

Point	Iteration 1	Iteration 2
I	{i:[1..1],n:[20..20],sum:[0..0]}	{i:[1..2],n:[20..20],sum:[0..1]}
S	{i:[1..1],n:[20..20],sum:[0..0]}	{i:[1..2],n:[20..20],sum:[0..1]}
B	{i:[2..2],n:[20..20],sum:[1..1]}	{i:[2..3],n:[20..20],sum:[1..3]}
O	{}	{}

After two iterations we notice that neither `i` nor `sum` have converged. We therefore decide to step one of them up; heuristically the variable(s) used in the loop conditional is a good one to try first, so we try stepping `i` up to [1..20].

of loop nesting. Such worst case behaviour, though, happens rarely in practice; a perception we find vindicated by early experimental observations.

Point	Iteration 3	Iteration 4
I	{i:[1..20],n:[20..20],sum:[0..3]}	{i:[1..21],n:[20..20],sum:[0..23]}
S	{i:[1..20],n:[20..20],sum:[0..3]}	{i:[1..20],n:[20..20],sum:[0..23]}
B	{i:[2..21],n:[20..20],sum:[1..23]}	{i:[2..21],n:[20..20],sum:[1..43]}
O	{}	{i:[21..21],n:[20..20],sum:[0..23]}

Again it is not converging rapidly enough, so we try stepping the only remaining non-converged variable; we set `sum` to $[0..\infty]$.

Point	Iteration 5
I	{i:[1..21],n:[20..20],sum:[0..∞]}
S	{i:[1..20],n:[20..20],sum:[0..∞]}
B	{i:[2..21],n:[20..20],sum:[1..∞]}
O	{i:[21..21],n:[20..20],sum:[0..∞]}

And we finally reach a fixed point.

B GCP Analysis for Non-recursive Procedure Calls

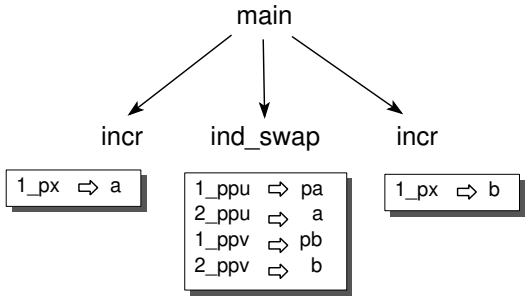
We demonstrate the GCP analysis for non-recursive procedure calls, using the simplified version of the indirect swap program given in Figure 14(a). Its invocation graph decorated with the mapping information is shown in Figure 14(b). On encountering the first call to `incr` at program point A, the GCP range information is (a:[1..1], b:[10..10]).

By inspecting the corresponding invocation graph node, we find this call to be an ordinary call, with the local variable `a` mapped to the symbolic name `1_px` in the procedure `incr`. By following the GCP map rules given in Figure 11, we obtain the input GCP information for the procedure `incr` to be (1_px:[1..1]). After performing the analysis of the body of `incr`, we get the output information: (1_px:[2..2], temp_0:[1..1]). Next, following the GCP unmap rules given in Figure 11, we obtain the range for variable `a` by copying over the range of symbolic name `1_px` which represents it in procedure `incr`. Thus we obtain the following GCP information at program point B: (a:[2..2],b:[10..10]). The call to indirect swap does not affect the GCP information. The second call to `incr` is processed with the input information as (1_px:[10..10]) as for this call `b` is mapped to `1_px`. The final GCP information at important program points is given below. Note that the information at program point E is obtained by merging the ranges calculated during the two visits to procedure `incr`.

Point	GCP Information
A	{a:[1..1],b:[10..10]}
B	{a:[2..2],b:[10..10]}
C	{a:[2..2],b:[10..10]}
D	{a:[2..2],b:[11..11]}
E	{temp_0:[1..10],1_px:[2..11]}

```
void ind_swap(int **ppu, int **ppv)
{ int  *ptemp;
  ptemp = *ppu;
  *ppu = *ppv;
  *ppv = ptemp;
}

void incr(int *px)
{ int temp_0;
  temp_0 = *px;
  *px = temp_0 + 1;
  /* Point E */
}

int main()
{ int a; int b; int *pa; int *pb;
  int **temp_2; int **temp_1;
  a = 1; b = 10;
  pa = &a; pb = &b;
  /* Point A */
  incr(pa);
  /* Point B */
  temp_1 = &pa; temp_2 = &pb;
  ind_swap(temp_1,temp_2);
  /* Point C */
  incr(pa);
  /* Point D */
}
```

(a) Example Program



(b) Invocation Graph and Map Information

Figure 14: Non-recursive Example

C GCP Analysis for Recursive Procedure Calls

We demonstrate the computation of a typical recursion fixed point for the simplified version of the example program in Figure 15 in which we have marked the specific program points of interest. This program is the recursive counterpart of the example program in Appendix A. Figure 16 shows the invocation graph of the program, decorated with appropriate mapping information as computed by the points-to analysis.

The GCP analysis of the example program begins with the body of `main` and the root of the invocation graph, and proceeds through several basic statements until the the call to procedure `recsum` is encountered at program point A. At this point, the GCP approximation is: (n:[20..20], sum:[0..0]). The effect of the call to `recsum` is approximated by traversing the appropriate invocation graph edge, in this case traversing the invocation graph node labeled `recsum-R`, applying the gcp-map to get the original input approximation for `recsum`, analyzing the effect of the body of `recsum` via the appropriate fixed-point computation, and then returning the gcp-unmapped output back to the calling site in `main`.

More specifically, the computation proceeds as follows. Following the call edge from `main` to `recsum-R`, the local variables `n` and `sum`, are mapped to symbolic names `1_m` and `1_tot`. Following the GCP map rules, the input for procedure `recsum` is (i:[1..1], 1_m:[20..20], 1_tot:[0..0]).

```
void recsum(int i, int *m, int *tot)
/* Sum  1 to *m,  return in *tot */
{ if (i <= *m)
    { *tot = *tot + i;
      recsum(i+1, m, tot);
    }
}

main()
{ int n = 20;
  int sum = 0;
  recsum(1,&n,&sum);
}
```

```
void recsum(int i,int *m,int *tot)
{ int temp_0; int temp_1; int temp_2;
  temp_0 = *m;
  /* Point I */
  if (i <= temp_0)
    { temp_1 = *tot;
      *tot = temp_1 + i;
      temp_2 = i + 1;
      /* Point C */
      recsum(temp_2,m,tot);
      /* Point D */
    }
  /* Point E */
}

int main()
{ int n; int sum;
  int *temp_4; int *temp_3;
  n = 20; sum = 0;
  temp_3 = &n; temp_4 = &sum;
  /* Point A */
  recsum(1,temp_3,temp_4);
  /* Point B */
}
```

Figure 15: Example Recursive Program - *RecSum*

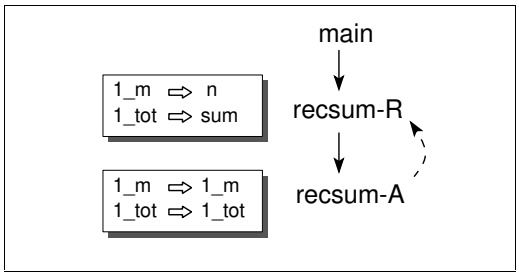


Figure 16: Invocation Graph for *RecSum*

As explained in section 3.4.2, for a recursive procedure call, the fixed-point analysis proceeds by collecting the most general input to the procedure, taking into account all invocations of the procedure due to all possible unrollings of the recursion. The same applies to the output. We refer to the generalized

input and output as stored input and stored output. Each new iteration during the fix-point computation, begins with the current generalized input (stored input). The stored output reflects the output GCP information for the stored input. An output of Bottom indicates that the output is not yet known for the current stored input. The actual fixed-point analysis for the example program proceeds in the following steps:

1. Initialize the stored input to the initial input, the stored output to Bottom, and the pending inputs list to empty.
2. Analyze the body of `recsum`, with stored input as the input.
3. At program point I, split the GCP information into two sets: one consistent with the conditional and another inconsistent with the conditional.
4. Analyze the *ifBody* with the consistent set.
5. On encountering the approximate call to `recsum` at program point C, the calling edge from `recsum-R` to `recsum-A` is traversed, the current GCP information is mapped to obtain the new input for this function invocation. Since this invocation corresponds to an approximate node in the invocation graph, the body of `recsum` is not analyzed. Rather, one of the following actions is taken. If the mapped input is a subset of the stored input, then use the stored output and unmap the stored output to obtain the GCP approximation at program point D. Otherwise, pass Bottom as the approximation at D, and put the mapped input in the pending list in the `recsum-R` node, indicating that an output is not yet available for this input to `recsum`.
6. Merge the inconsistent set at I with the information at D, to obtain the output GCP information for this iteration.
7. If the pending list is not empty, merge the pending inputs with the stored input, discard the output of this iteration, reset stored output to Bottom and go back to step 2. Otherwise, if the pending list is empty, but the output obtained in step 6 is not a subset of the stored output, merge it with the stored output and go back to step 2.
8. Unmap the stored output, to provide the GCP information at program point B.

Now, we summarize the iterations required to compute the recursion fixed-point computation.

The process begins with both the pending list and the StoredOutput initialized to {}. The initial input is first split over the conditional into a set consistent with the conditional, and a set inconsistent with the conditional. Note that in this first couple of iterations `i` has a value small enough that there does not exist an inconsistent set i.e. the inconsistent set will just be {}.

Point	Iteration 1
StoredInput	{i:[1..1],1_m:[20..20],1_tot:[0..0]}
StoredOutput	{}
Consistent (I)	{i:[1..1],1_m:[20..20],1_tot:[0..0],temp_0:[20..20]}
Inconsistent (I)	{}
C	{i:[1..1],1_m:[20..20],1_tot:[1..1],temp_0:[20..20],temp_1:[0..0],temp_2:[2..2]}
D	{}
Output (E)	{}
pending	{i:[2..2],1_m:[20..20],1_tot:[1..1]}

Since the ranges in the StoredInput are not a superset of the corresponding ranges at point C (after mapping), the appropriate ranges are added to the pending list and output of {} is returned. This output is merged with the inconsistent set to form the output of the function.

At the end of the first iteration, the pending list is non-empty, so the second iteration begins with a new StoredInput that is the merge of the initial StoredInput and the pending list. The pending list is reinitialized to {}, and the second iteration is performed.

Point	Iteration 2
StoredInput	{i:[1..2],1_m:[20..20],1_tot:[0..1]}
StoredOutput	{}
Consistent (I)	{i:[1..2],1_m:[20..20],1_tot:[0..1],temp_0:[20..20]}
Inconsistent (I)	{}
C	{i:[1..2],1_m:[20..20],1_tot:[1..3],temp_0:[20..20],temp_1:[0..1],temp_2:[2..3]}
D	{}
Output (E)	{}
pending	{i:[2..3],1_m:[20..20],1_tot:[1..3]}

After the second iteration, there are two ranges that have so far refused to converge (`i` and `1_tot`), so one range is stepped up. If the fixed-point analysis was aware of the conditional closest to the recursive call, then a good heuristic choice is to step up `i` to [1..20]; the convergence would then be very similar to the *while*-loop example. In the absence of this information, however, one can still can make a slightly-worse choice by stepping `i` up to [1..∞].

Point	Iteration 3
StoredInput	{i:[1..∞],1_m:[20..20],1_tot:[0..3]}
StoredOutput	{}
Consistent (I)	{i:[1..20],1_m:[20..20],1_tot:[0..3],temp_0:[20..20]}
Inconsistent (I)	{i:[21..∞],1_m:[20..20],1_tot:[0..3],temp_0:[20..20]}
C	{i:[1..20],1_m:[20..20],1_tot:[1..23],temp_0:[20..20],temp_1:[0..3],temp_2:[2..21]}
D	{}
Output (E)	{i:[21..∞],1_m:[20..20],1_tot:[0..3],temp_0:[20..20]}
pending	{i:[2..21],1_m:[20..20],1_tot:[1..23]}

After the third iteration, the recursive call still returns {}. Nevertheless, there is a non-trivial output (E), and this is because the final output is formed from the merge of the inconsistent set (which is non-trivial now that we have stepped up `i`) and the output of the *then*.

Again, note that `1_tot` is not converging fast enough, and so it is stepped up to [0..∞].

Point	Iteration 4
StoredInput	{i:[1..∞],1_m:[20..20],1_tot:[0..∞]}
StoredOutput	{}
Consistent (I)	{i:[1..20],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20]}
Inconsistent (I)	{i:[21..∞],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20]}
C	{i:[1..20],1_m:[20..20],1_tot:[1..∞],temp_0:[20..20],temp_1:[0..∞],temp_2:[2..21]}
D	{}
Output (E)	{i:[21..∞],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20]}
pending	{}

This fourth iteration looks very similar to the third, but a number of important changes have occurred. At point C, note that the mapped input from the approximate call is now a subset of the StoredInput. Thus, the StoredOutput is used to approximate the result of the call — in this case as the StoredOutput is {}, so the output of the approximate call is still {}.

Because of this change, however, nothing is added to pending list (no new inputs have been found), and the output is approximated by merging the latest output with the StoredOutput.

Point	Iteration 5
StoredInput	{i:[1..∞],1_m:[20..20],1_tot:[0..∞]}
StoredOutput	{i:[21..∞],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20]}
Consistent (I)	{i:[1..20],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20]}
Inconsistent (I)	{i:[21..∞],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20]}
C	{i:[1..20],1_m:[20..20],1_tot:[1..∞],temp_0:[20..20],temp_1:[0..∞],temp_2:[2..21]}
D	{i:[1..20],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20],temp_1:[0..∞],temp_2:[2..21]}
Output (E)	{i:[1..∞],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20],temp_1:[0..∞],temp_2:[2..21]}
pending	{}

The fifth iteration is the penultimate iteration. Again, the input from the approximate call is a subset of the StoredInput, so the StoredOutput can be used. This StoredOutput is now non-trivial, and so it is used as the effect of the approximate call, and unmap is applied to the StoredOutput.

For the final iteration, the output of the procedure from the fourth iteration is merged in with the StoredOutput. Once again the input from the approximate call is a subset of the StoredInput, and so the unmapped StoredOutput is used. This time, however, the final output of the procedure will not change, indicating that we have reached a fixed point.

Point	Iteration 6
StoredInput	{i:[1..∞],1_m:[20..20],1_tot:[0..∞]}
StoredOutput	{i:[1..∞],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20],temp_1:[0..∞],temp_2:[2..21]}
Consistent (I)	{i:[1..20],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20]}
Inconsistent (I)	{i:[21..∞],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20]}
C	{i:[1..20],1_m:[20..20],1_tot:[1..∞],temp_0:[20..20],temp_1:[0..∞],temp_2:[2..21]}
D	{i:[1..20],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20],temp_1:[0..∞],temp_2:[2..21]}
Output (E)	{i:[1..∞],1_m:[20..20],1_tot:[0..∞],temp_0:[20..20],temp_1:[0..∞],temp_2:[2..21]}
pending	{}

Finally, the result is returned to `main` by unmapping the StoredOutput to obtain the GCP information at program point B: (n:[20..20], sum:[0..∞]).