November 4, 1993

Contruction of call multigraphs revisited

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 231-6766, -5791 (Fax)
arun@cacs.usl.edu

Extended abstract

1 Introduction

This paper presents an algorithm for constructing call multigraphs for procedural language programs. The call graph construction (CGC) problem is analogous to the problem of performing control flow analysis (CFA) of Scheme [Shi88, Shi91b]. As shown later, this problem is also analogous to the problem of determining types (TD) at specific program points for object-oriented languages [PS91, PR93, Suz81]. Our CGC algorithm may be easily adapted to perform CFA and TD. The algorithm is significant since the CGC, CFA, and TD problems are of fundamental importance for efficient compilation of programs in procedural, functional, and object-oriented languages, respectively.

Pande & Ryder [PR93] have shown that the TD problem is NP-hard in the presence of single-level pointers. We show that the CGC problem is analogous to the TD problem. The CGC problem too is, therefore, intractable. Any polynomial algorithm for it can only give approximate results. The algorithm presented makes the following contributions:

1. It constructs call graphs for a larger class of procedural language programs than previous algorithms [Bur87, CCHK90, HK93, Lak93, Wei80, Spi71, Ryd79].
2. When compared under suitable mapping, the results of our algorithm are more precise than Pande and Ryder's type determination algorithm for C++ [PR93].
3. When compared under suitable mapping, our algorithm is equivalent to Shivers' 1CFA [Shi88, Shi91b]. Our algorithm is however a) more efficient since it does not require CPS conversion and b) gives more precise results for pre-CPS converted programs.

Unlike our previous algorithm for the same problem [Lak93], the new algorithm works for programs with global variables and reference parameters. This increase in scope is brought about by using control flow graph (CFG) representation of programs; the previous algorithm used program dependence graphs (PDG) [OO84]. The improvement comes at additional computational costs. Since in a CFG the data dependence between statements are not available information is propagated through all statements, as against directly between the definition and use statements. Part of this cost may be recovered by using sparse evaluation graphs (SEG) [CCF91] instead of CFGs.

The call graph of a program is affected by program aliases. On the other hand, the set of aliases in a program is influenced by the call graph [Ban79, CBC93, CK89, LR91]. To obtain precise results these two problems should be solved simultaneously. Though we treat global variables and call by references, two of the sources of aliasing, we ignore the aliasing problem. It may be observed that the pointer-induced aliasing algorithms of [CBC93, LR91] and our algorithm have similar "structure." Our algorithm may be combined with one of the aliasing algorithms to simultaneously construct alias sets and call graph. We restrict our focus to the call graph problem and do not delve into the details of how these algorithms may be combined.

The rest of this article is organized as follows. In Section 2 the CGC is formulated and its similarity to the CFA and TD problems is shown. This section defines the constraints under which our algorithm is applicable and also defines a convention to classify the precision of the solutions found by the algorithm. The "essence" of the algorithm is presented in Section 3. Section 4 enumerates the application of our algorithm on two example programs. The complexity of the algorithm is analyzed in Section 5. A more detailed comparison of our work with other related works is presented in Section 6. Finally, Section 7 summarizes the work and reports on its status.

2 Problem formulation

We consider a *while-do* language with provision for call by value-result, call by reference, and call by value parameter passing. The language also allows for procedure-valued variables – variables whose values are references to procedures. A procedure-valued variable may be used just like any other variable i.e. it may be global or local, it may be passed as a parameter using any of the referencing strategies, and it may be assigned to. Furthermore, a procedure-valued variable may also be used to make calls to a procedure referenced by a variable.

We assume that a program's variable has a unique "identity" different from its "name." Different variables with the same names have different identities. Such an identity can easily be generated as a function of the variable's name and the scope in which the it is defined. These identities may be used to determine if a variable is "visible" at a statement, inside, or outside a procedure. We assume the existence of mechanisms to make these determinations. This permits our algorithm to uniformly deal with static scope rules arising from nested procedures, local, global, and static variables.

The **call graph construction problem** may be stated as:

For each call site s in program P which is the set $C(i)$ of procedures that i could be a call to? I.e. if there is possible execution of P such that the procedure q is called from call site s then i must be contained in $C(i)$.

The above statement is modelled after Shivers' statement for the **control flow analysis** problem [Shi88]:

For each call site s in program P, what is the set, $L(s)$, of lambda expressions that s could be a call to? I.e. if there is a possible execution of P such that the lambda expression l is called from call site s, then l must be an element of $L(s)$.

Lambda expressions are essentially functions without names. Nesting of lambda expressions corresponds to nesting of procedure declarations. The two problems become equivalent a) by assigning unique labels to each lambda expression, as done by Shivers, and b) by transforming a function into a procedure with a reference argument for the returned value.

Notice the weakness in the above problem definition. It requires only that if there is a possible execution of P in which q may be called from i then q be in $C(i)$. Thus procedure q may be contained in $C(i)$ even if there is no possible execution in which q may be called from i. The weakness is deliberate since the stronger definition is not computable [PR93]. This is because to obtain a precise solution every possible state of the stack in which a procedure may be called would be needed. Since the number of states may be unbounded for recursive programs and exponential for others, maintaining all the stack states is not pragmatic. As a result all the CGC, CFA, and TD algorithms maintain some approximations of the stack state.

Classifying precision: Shivers [Shi91a] classifies his CFA algorithms as 0CFA, 1CFA, . . ., nCFA, with the numeric prefix giving the depth in the stack upto which the analysis is precise. Instead of a numeric prefix, we use the symbol κ to denote the stack depth. An algorithm that gives precise results for stacks of maximum depth one is said to be $\kappa = 1$ precise. Any state information beyond the κ stack depth is approximated to belong to the same state. A $\kappa = \infty$ precision implies the algorithm is precise for unbounded number of stack depths.

Let us now look at the **type determination problem** (TD) or more precisely the problem of *point specific type determination for single-level pointers* in C++ [PR93]. Assuming that pointers only have a single-level of dereferencing, the problem may be stated as:

Determine the class (types) of objects pointed to by a pointer at a program point.

In C++ a statement of the form `g -> foo(...)` invokes the method `foo` belonging to the class of the object pointed to by `g`. The prime motivation for determining types is that knowing the types of objects pointed to by `g` would help in identifying the set of methods that may be invoked by such a statement. Therein lies the similarity between the TD and CGC problems.

Our CGC algorithm further requires two constraints, referred to as the domain and completeness constraints.

Domain constraint: An assignment statement defining a procedure variable may have on its right hand side either a procedure reference constant or a procedure variable.

That is, an assignment to `gfoo` may take the form `gfoo := &p` or `gfoo := kfoo`, where `&p` evaluates to a reference to the procedure `p` and `kfoo` is a procedure variable. This constraint is not relevant for Shivers CFA algorithms [Shi88, Shi91b] because of the absence of assignment statements in CPS Scheme. Though not explicitly stated in the literature, the TD problem also requires analogous constraints. In C++ a new object instance of type `T` is created by the statement `new T`. Other ways to introduce object references are by the address operator "&". For instance, if variable `f` is declared as an object of type `T` then `&f` gives a reference to an object of type `T`. In both the cases the type of an object instance is statically known directly from declarations.

Completeness constraint: Only procedures contained in a program may be called from a call-site inside the program.

In the absence of this constraint a procedure may call another procedure *external* to the program which may then call some procedure within the program. The TD algorithms [PR93, PS91, Suz81] also implicitly require an analogous constraint, i.e. only objects of types defined in a program may be pointed to. Shivers' algorithms do not require this contraint. He presents a safe approximation for treating procedure references that "escape" to external procedures. His safe approximation can easily be adapted for our algorithm as well.

Realizable paths: In most interprocedural analyses information is propagated back and forth between a call-site and the procedure called from there. A naive analysis may propagate information generated due to a call from one call-site to the return node of another call site. A propagation path in which the call-return nodes are matched is termed as a *realizable path*.

Combining the terminology introduced by Pande and Ryder [PR93] and Shivers [Shi88] we may classify our algorithm as: $\kappa = 1$ precise CGC algorithm for a procedural language with single-level function pointers. It propagates information only over realizable paths. Our algorithm may be customized to create a $\kappa = 1$ precise CFA algorithm for functional languages and $\kappa = 1$ precise TD algorithm for object-oriented languages.

3 A κ=1 precise CGC algorithm

This section discusses the important issues related to the algorithm and how our algorithm addresses them. The actual "code" is not presented due to paucity of space. It would be presented in the full paper.

Program representation Our algorithm requires programs to be represented as a *collection of control flow graphs* (CCFG). A CCFG consists of a set of CFGs, one for each procedure in the program being analysed. A CFG is a directed graph similar to program flow graphs described in [ASU86]. In a CFG every statement of a program is represented by one or more nodes. An assignment or a conditional statement is represented by one node. A procedure call, direct or indirect, is represented by two nodes - a *call* node and a *return* node. Each procedure also has two special nodes - *entry* and *exit*. The edges in this graph represent the sequencing of statements in the code. There are, however, no edges going out of *call* nodes or terminating at *return* nodes.

Our CCFG differs from Landi and Ryder's ICFG (interprocedural CFG) [LR92] in that in a CCFG there are no edges connecting nodes of two CFGs. In an ICFG a calling relationship between a call site and the procedure called is represented by edges between the vertices of the CFGs containing the call site and the procedure. Since our problem is to construct this relationship we cannot assume its existence. The computational complexity of our algorithm may be improved by using sparse evaluation graphs (SEG) [CCF91] as suggested by [CBC93] for their aliasing algorithm. Unlike a CFG, a SEG is constructed specifically for a problem. It has the same set of nodes as a CFG but a different set of edges. The edges are created so as to allow a propagation algorithm for the chosen problem to combine information as early as possible. This reduces the number of intermediate nodes through which information generated at a statement is propagated to statements using it.

Information propagated The call graph construction algorithm is an iterative algorithm. Starting with an initial state, and using some rules, the algorithm propagates information over the CCFG till a fixed point is reached. The information propagated by our algorithm consists of 3-tuples of the form:

`<binding, trigger, site-of-descent>`

The set of elements of this type is called `triple`. We use the field names as access functions to get and set values of the corresponding field. The elements of `binding` and `trigger` fields are 2-tuples represented as

`v = p`

where `v` denotes a program variable and `p` a procedure reference. Informally, the triple:

$<\mathtt{v}_1 = \mathtt{p}_1,\ \mathtt{v}_2 = \mathtt{p}_2,\ \mathtt{s}>$

at a vertex `n` in the procedure $\mathtt{p}_3$ means that: if procedure $\mathtt{p}_3$ is called from call-site `s` with variable $\mathtt{v}_2$ referring to procedure $\mathtt{p}_2$ then there is an execution path from the `entry` of $\mathtt{p}_3$ to node `n` over which variable $\mathtt{v}_1$ may refer to procedure $\mathtt{p}_1$. Notice that variable $\mathtt{v}_2$ is from the static scope of site `s` while variable $\mathtt{v}_1$ is from `n`'s static scope.

There is a 1–1 correspondence between the fields of our triple and the 3-tuples propagated by Choi et. al.'s aliasing algorithm [CBC93]. The correspondence is not in fields' contents but, in an abstract sense, the information they represent. On the other hand the TD algorithm of Pande and Ryder [PR93] propagates 2-tuples containing fields analogous to our `binding` and `trigger` fields. Their 2-tuples do not retain the `site-of-descent` information. The information maintained by Shivers' algorithms [Shi88, Shi91b] correspond to 2-tuples consisting of the `binding` and `site-of-descent` fields. For instance, corresponding to the above triple Shivers' abstract interpreter would maintain a relationship $<\mathtt{b},\ \mathtt{v}_1> \mapsto \mathtt{p}$, stating that the variable $\mathtt{v}_1$ takes the value `p` in "contour" `b`. In the special case of the 1CFA algorithm there is a 1–1 correspondence between "contours" and call sites. The `site-of-descent` `s` may, therefore, be used in place of contour `b`.

With every node of each CFG is associated an attribute named `info`:

• `info:  cfg-node` $\rightarrow$ `set(triple)`

The value of this attribute is computed by the algorithm. At any time, the set of procedures known to be called from an indirect site may be found from the `info` attributes at the corresponding *call* node. If at site `s` an indirect call is made using the value of variable `v`, i.e. site `s` has the statement `(*v)(...)`, then the set of procedures called from site `s` are given by $C(\mathtt{s}) = \{\ \mathtt{p}\ | <\mathtt{v=p}, .., ..> \in \mathtt{info(s)}\ \}$, where $..$ stands for any value.

Outline of the algorithm The algorithm propagates `triples` over CFG nodes. It maintains

• *workset* : a set of `<n, e>` where `n` is a CFG node and `e` a triple.

An element is randomly removed from this set and processed. When processing an element, say `<n, e>`, the triple `e` is included in `info(n)`. If this inclusion changes `info(n)` then, based on the type of the node `n`, a function is invoked with `n` and `e` as its arguments. The processing at this function may add some more entries to the `workset`.

Our CGC algorithm may be divided into four parts:

a. Initialization.
b. Top level `workset` management and invokation of appropriate functions.
c. Establishment of a calling relation.
d. Processing of a triple received at a node.

The `workset` management part has already been outlined. Initialization involves

1. initializing the set of procedures in the call-graph to be ϕ and
2. establishing a call relation from a site **null** to the top-level procedure `main`.

The second item above establishes calling relations defined by direct calls from all procedures transitively called by `main`. It also initializes *workset* as described below.

Establishing a calling relation When it is found that a procedure p is called from a site s the a calling relation is established by performing the following tasks:

1. Establish the relation that procedure p is called from site s.
2. Mark that the procedure p can be reached by a sequence of calls from the top-level procedure main.
3. Initialize the info attribute for all the nodes of the CFG of procedure p.
4. Make triples for each assignment of the type 'v := &q'. Propagate them to the statements successors.
5. Make triples for each procedure reference in the actual parameters of direct calls from p. Propagate them to the entry node of the called procedure.
6. Recursively establish the calling relations known from the direct call sites of p.

Tasks 2 through 6 are performed only when the first call to a procedure is found. In the following discussion the algorithm performing these tasks is called establish-a-call.

Tasks 1, 2, and 3 do not create any workset entries and those created by task 6 may be defined recursively. In task 4, the workset entries for procedure references in an assignment statement are created so as to propagate bindings to its CFG successors. The trigger and site-of-descent fields of these triples are set to **null** to imply that the triple was not generated due to any procedure call.

In task 5 workset entries are created such that triples containing bindings due procedure references in the actual parameter list at a direct call-site are propagated to the entry node of the procedure called. These triples bind the procedure reference to the corresponding formal parameter of the called procedure. The trigger field of these triples record the actual parameter that contained the procedure reference. Their site-of-descent field record the direct call-site whose actual parameter created the triple.

Processing at various nodes In the discussion that follows we use the following symbols:

1. at-node - the node at which processing is being performed,
2. in-val - the triple reaching this node, and
3. v - the variable bound by in-val

Any triple reaching an *entry*, *return*, or *branch* node is propagated to all the CFG successors of the node.

Assignment node: If at-node is an assignment node the processing is relatively simple. If v is not modified at at-node then in-val is propagated to all the CFG successors of the node. Further, if v is used on the right-hand side of the assignment then a new triple representing bindings due to the assignment is created. The trigger and site-of-descent fields of the new triples are copied from the corresponding fields of in-val.

Direct call site: When processing a triple at a direct call-site, distinction is made on whether the variable bound by the triple can be modified by the call or not. A triple binding a variable that cannot be modified by a call can simply be propagated to the corresponding *return* node. Otherwise, one may assume that the binding may be killed by the call. If the variable is used as an actual parameter, assuming call by reference or value-result, the binding is propagated to the corresponding formal parameter. If the variable is visible in the called procedure the original binding is propagated to the procedure. In both cases, however, the trigger field of a propagated triple reflects the fact that the triple was created due to the incoming binding, i.e. trigger(new-triple) = binding(in-val), and that the new triple was created at the call site being processed, i.e. site-of-descent(new-triple) = at-node.

If variable v bound by in-val is used as an actual parameter in the call then the binding field of new-triple binds the corresponding formal parameter of the called procedure. If v is visible to the call then the binding(new-triple) binds v. If both the conditions are possible then different new triples are created to satisfy different conditions.

Indirect call site: The processing at an indirect call site is a generalization of that done at a direct call site. The differences come from the observations that:

1. more than one procedures may be called from an indirect site,
2. a triple binding a variable used for indirection and one binding a "global" variables or a variable used as parameter may appear in arbitrary order, and

3. even though the variable bound by triple `in-val` may be visible in a procedure called from an indirect site, the binding may not be propagated to it.

Owing to the second observation, whenever a triple binding the indirection variable reaches an indirect call site all triples previously received at this site are considered as candidates to be propagated over the newly established call. Similarly, whenever a binding for any variable reaches an indirect call-site it is considered as a candidate to be propagated to all procedures known to be called from this site.

Owing to the third observation, we say that a triple is only a "candidate" for propagation and is not actually propagated. Ignoring this issue would result into a $\kappa = 0$ algorithm. Let t_1 be a triple that is a candidate for propagation. Let t_2 be a triple binding the indirection variable at `at-node`, i.e. `binding(`t_1`) = (v=q)`, for some procedure reference `q`. To ensure $\kappa = 1$ precision, a new triple t_3 is propagated to the procedure `q` if:

1. the `site-of-descent(`t_1`) = site-of-descent(`t_2`)`, i.e. both the triples were triggered due to some propagation from the same call site, or
2. `site-of-descent(`t_1`) = `**`null`** or `site-of-descent(`t_2`) = `**`null`**, i.e. one of the triple is not triggered by any call.

The `trigger` field of the propagated triple reflects the fact it was created due to the bindings in t_1, i.e. `trigger(`t_3`) = binding(`t_1`)`, and that the new triple was created at the call-site being processed, i.e. `site-of-descent(`t_3`) = at-node`. This may be illustrated from the program in Figure 2.

The rules for creating `binding(`t_3`)` may be derived from those for creating `binding(new-triple)` at direct call-sites. Some languages permit a variable to be visible to a collection of procedures but not necessarily all procedures, such as a static global variable in C. It is, therefore, possible that a variable may be visible to one procedure called from an indirect call-site but not to another. A triple binding such a variable has to be treated as if the variable could be both visible as well as not visible in a call. If a variable is neither used as an actual parameter nor is visible to some procedure called from the indirect site, the triple `in-val` may be propagated to the corresponding *return* node.

There is some special work required when `in-val` identifies a new from an indirect call-site. The procedure `establish-a-call` is called to record it. Let `p` be the procedure detected to be called from `at-node`. If this is the first call to `establish-a-call` for `p` it propagates procedure references at assignment statements and direct call-sites in procedures transitively called from `p`. Notice that `establish-a-call` does not process the procedure references in the argument list of indirect call-sites. Therefore, the procedure references in the argument list of `at-node` have not yet been propagated to procedure `p`. This task is performed now. The procedure reference constants in the argument list of the indirect call are propagated to the corresponding formal parameters of the called procedure.

Furthermore, it is quite likely that procedure `p` was previously called from some other site. At that time `establish-a-call` may have detected procedure reference bindings created from within `p`, i.e. not due to a call to `p`. These bindings, identified by a **null** `site-of-descent` field, may have already been propagated to the exit node of `p` and should now be propagated to the return node corresponding to `at-node`. The rules for deriving the `binding` field of triples sent to the return nodes are the same as those used at the exit nodes.

Exit node: Triples reaching an *exit* node should be propagated to the return nodes of the call-sites that may call the procedure, say `p`, containing `at-node`. However, not all triples may be propagated to all such call-sites. A triple may be propagated to the *return* node of a site either if it was generated as a result of a call from that site or if it was not triggered by any call.

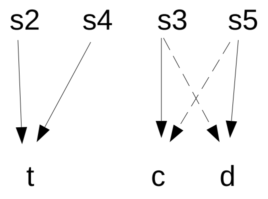
When the triple `in-val` is not triggered by a procedure call, i.e. `site-of-descent(in-val) = `**`null`**, new triples are sent to all the return nodes of call-sites known to call the procedure containing `at-node`. The `site-of-descent` and `trigger` fields of the new triples propagated is also set to **`null`**. This propagates the knowledge that the binding is not due to any procedure call. This means that all triples generated as a result of these triples and reaching the exit node of the calling procedure may also be propagated to all the call-sites. Notice that if an indirect call-site is later known to call procedure `p` then the triple `in-val` will be propagated to it due to the processing at that indirect call-site.

Example program

```
procedure main ()
    local i, x;
    s1:  i := &c;
    s2: t(i, x);
    s3: (*x)();
    s4: t(&d, x);
    s5: (*x)();
end

procedure t($f_{t1}$, $f_{t2}$)
    s6: $f_{t2}$ := $f_{t1}$;
end
```

tuple to process	Info/ Worklist			call relation
	label i:	tuple t_i	-> node n_i	
	1:	<i=c, **null**, **null**>	-> s2	s2 -> t
	2:	<f_{t1}=d, a_{14}=d, s4>	-> entry(t)	s4 -> t
1	3:	<f_{t1}=c, a_{12}=c, s2>	-> entry(t)	
2	4:	<f_{t1}=d, a_{14}=d, s4>	-> s6	
3	5:	<f_{t1}=c, a_{12}=c, s2>	-> s6	
4	6:	<f_{t2}=d, a_{14}=d, s4>	-> exit(t)	
	7:	<f_{t1}=d, a_{14}=d, s4>	-> exit(t)	
5	8:	<f_{t2}=c, a_{12}=c, s2>	-> exit(t)	
	9:	<f_{t1}=c, a_{12}=c, s2>	-> exit(t)	
6	10:	<x=d, **null**, **null**>	-> return(s4)	
7				
8	11:	<x=c, **null**, **null**>	-> return(s2)	
9				
10	12:	<x=d, **null**, **null**>	-> s5	
11	13:	<x=c, **null**, **null**>	-> s3	
12	14:	<x=d, **null**, **null**>	-> return(main)	s5 -> d
13	15:	<x=c, **null**, **null**>	-> s4	s3 -> c
14				
15	16:	<f_{t2}=c, a_{24}=c, s4>	-> entry(t)	
16	17:	<f_{t2}=c, a_{24}=c, s4>	-> s6	
17				



Call graph showing which site calls which procedure. The calls due to dashed edges are not found by our algorithm. They would be found if propagation was over unrealizable paths.

Figure 1 Example demonstrating propagation of information over realizable paths. The table enumerates our CGC algorithm for the given program. See Section 4 on how to interpret the table.

To see what our algorithm does when `in-val` is triggered by a procedure call let us sketch a scenario of how `in-val` may have been created.

A triple e_1 reached the call-site s_1 which calls procedure p. The processing at the call-site propagates a triple e_2 to the entry node of procedure p. The processing of e_2 at this node leads to the propagation of `in-val` to `at-node`, the exit node of p.

Let `new-val` be the new triple created. Our algorithm ensures that `new-val` is sent to the return node of s_1 and that `trigger(new-val) = trigger(`e_1`)` and `site-of-descent(new-val) = site-of-descent(`e_1`)`. To achieve this the triple `e1` and site s_1 need to be known. The site `s1` is known from `site-of-descent(in-val)`. The triple e_1 is found by searching the `info` entries of s_1 for a triple whose `binding` is the same as `trigger(in-val)`.

A new triple created for propagation should bind a variable in the static scope of the calling procedure. Thus if triple `in-val` binds a formal parameter, the new triple should bind the corresponding actual parameter. If `in-val` binds any other variable then the new triple would bind the same variable. But if the variable bound by `in-val` is not visible to the calling procedure and is also not a formal parameter then its binding need not be propagated to any caller.

4 Two examples

Figures 1 and 2 each contain a program, a table, and a graph. The graph in Figure 1 has edges from nodes representing call-sites to nodes representing procedures. It gives the set of procedure called from a call-site. The graph in Figure 2 has edges from nodes representing procedures to nodes representing procedures. It shows the set of procedures called from a procedure. Different graph representations are used because the examples demonstrate different aspects of the algorithm. The first figure demonstrates that the algorithm propagates information over realizable paths. The second figure demonstrates that it is $\kappa = 1$ precise. The graphs diagrammatically represent the final call graphs. Dashed edges in both the graphs represent calling relations that may have been found a) if information was propapagted over non-realizable paths or b) if the algorithm had $\kappa = 0$ precision.

The table in each figure completely enumerates the application of our CGC algorithm for the corresponding program. The wide middle column plays the dual role of maintaining the `workset` and the `info` attribute. This column has three parts: a number followed by a ":", a triple, and an "->" followed by a node identifier. The number gives a label for the triple and node in a cell. Thus, if the label is i the triple in that cell is referred to as t_i and the node as n_i. The node identifier for statements are the same as the labels used in the program. Additionally, the functions `entry` and `exit` give the entry and exit node of a procedure. The function `return` gives the return node corresponding to a call site.

Each row of the table represents processing of a triple. The first column gives the label of the triple to be processed. Let the first column of row r contain the label i. Let k be the last label in the middle column of row r. The `workset` at the instant when the triple i is processed consists of $< n_j, t_j >$ where $i \leq j \leq k$. The entries $1 \leq l \leq i$ in the middle column represent the `info` attributes as follows: $t_l \in$ `info(`n_l`)`. As an example, with respect to Figure 1, let r be the column containing 5 in the first column. Then $i = 5$ and $k = 9$. The `workset` consists of $\{< n_6, t_6 >, < n_7, t_7 >, < n_8, t_8 >, < n_9, t_9 >\}$. Further, `info(s6)=` $\{t_4, t_5\}$, since $n_4 = n_5 =$ `s6`; `info(entry(t))=` $\{t_2, t_3\}$, since $n_2 = n_3 =$ `entry(t)`; and `info(s2)=` $\{t_1\}$.

The rightmost column of each table maintains the call relation established as a result of the processing indicated by that row. The initial call relation, **`null`** `-> main`, is not shown. When a row r is being processed only procedures appearing in call relations above the row r are known to be included in the call graph.

Tracing the propagation of triple t_2 or t_3 in Figure 1 shows how information is propagated over realizable paths. The point to focus is on the creation of these triples and the processing of the triples t_6 and t_8 at the exit node of procedure `t`.

Tracing the propagation of triples in Figure 2 shows how $\kappa = 1$ precision is achieved. The point to focus is on the processing of the triples t_6 and t_8 at indirect site `s3`. One of the triple is propagated to `entry(a)` while the other is propagated to `entry(b)`. A $\kappa = 0$ algorithm would propagate both the triples to both the nodes.

5 Complexity analysis

The complexity of the CGC algorithm depends on several parameters. A simpler asymptotic bound in terms of : $Nodes$ - the CCFG's (and hence program's) size, $PRefs$ - the number of procedure references, and $ISites$ - the number of indirect call-sites may be derived if we assume that:

a. $Vars$, the maximum number of variables visible in any procedure, is bounded by a constant factor of the size of the program, i.e. $\exists c_1$, a constant, such that $Vars \leq c_1 * Nodes$.

b. $Sites$, the total number of call-sites in a program, is bounded by a constant factor of the size of the program, i.e. $\exists c_2$, a constant, such that $Sites \leq c_2 * Nodes$.

Since the processing at individual node is dominated by the processing at indirect call-sites, we derive two sets of complexity measures based on the $ISites/Nodes$ – the proportion of indirect call-sites in the program. The asymptotic complexity of our algorithm is bounded by $O(Nodes^6 * PRefs^2)$ for $ISites/Nodes$ ratio close to 1 and $O(Nodes^5 * PRefs)$ for $ISites/Nodes$ ratio close to 0.

Example program

procedure main ()
  s1: p(&a, &c);
  s2: p(&b, &d);
end

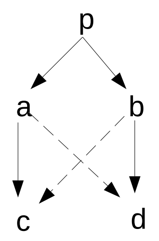
procedure p(f_{p1}, f_{p2})
  s3: (*f_{p1})(f_{p2});
end

procedure a(f_{a1})
  s4: (*f_{a1})();
end

procedure b(f_{b1})
  s5: (*f_{b1})();
end

procedure c() **end**
procedure d() **end**

tuple to process	**Info/Worklist**			**call relation**
	label i	**tuple t_i**	**-> node n_i**	
	1:	$<f_{p1}=a, a_{11}=a, s1>$	-> entry(p)	s1 -> p
	2:	$<f_{p2}=c, a_{21}=c, s1>$	-> entry(p)	
	3:	$<f_{p1}=b, a_{12}=b, s2>$	-> entry(p)	s2 -> p
	4:	$<f_{p2}=d, a_{22}=d, s2>$	-> entry(p)	
1	5:	$<f_{p1}=1, a_{11}=a, s1>$	-> s3	
2	6:	$<f_{p2}=c, a_{21}=c, s1>$	-> s3	
3	7:	$<f_{p1}=b, a_{12}=b, s2>$	-> s3	
4	8:	$<f_{p2}=d, a_{22}=d, s2>$	-> s3	
5	9:	$<f_{p1=a}, a_{22}=d, s2>$	-> exit(p)	s3 -> a
6	10:	$<f_{a1}=c, a_{13}=c, s3>$	-> entry(a)	
7	11:	$<f_{p1}=b, a_{12}=b, s2>$	-> exit(p)	s3 -> b
8	12:	$<f_{b1}=d, a_{13}=d, s3>$	-> entry(b)	
9				
10	13:	$<f_{a1}=c, a_{13}=c, s3>$	-> s4	
11				
12	14:	$<f_{b1}=d, a_{13}=d, s3>$	-> s5	
13	15:	$<f_{b1}=c, a_{13}=c, s3>$	-> exit(a)	s4 -> c
14	16:	$<f_{b1}=d, a_{13}=c, s3>$	-> exit(b)	s5 -> d
15	17:	$<f_{p2}=c, a_{21}=c, s1>$	-> return(s3)	
16	18:	$<f_{p2}=d, a_{22}=d, s2>$	-> return(s3)	
17	19:	$<f_{p2}=c, a_{21}=c, s1>$	-> exit(p)	
18	20:	$<f_{p2}=d, a_{22}=d, s2>$	-> exit(p)	
19				
20				



Call graph showing which procedure calls which procedure. Calls represented by dashed edges are not found by our algorithm. A $\kappa = 0$ algorithm would find those calls.

Figure 2 Example demonstrating that our algorithm achieves $\kappa = 1$ precision. The table enumerates our CGC algorithm for the given program. See Section 4 on how to interpret the table.

6 Comparison with related works

In this section we compare our CGC algorithm with those of others. We also compare our algorithm with the point-specific type determination for object-oriented languages and Shivers' algorithms for control flow analysis of functional languages [Shi88, Shi91b].

The CGC algorithms may be compared on a) the programming language they are applicable for and b) the precision of the call graph they construct. Due to the difference in their domain and results, comparing their computational cost is not meaningful. The algorithms of Burke [Bur87], Callahan et. al. [CCHK90], Hall & Kennedy [HK93], and Ryder [Ryd79] are applicable for Fortran-like languages in which a) only formal parameters can be procedure-valued variables, b) these formal parameters can not be assigned to, and c) procedure references can only be used as actual parameters. All but Ryder's algorithm are applicable for recursive programs. Ryder's and Callahan et. al.'s algorithms are $\kappa = \infty$ precise where as those of Burke and Hall & Kennedy are $\kappa = 0$. Spillman's [Spi71] and Weihl's [Wei80] algorithms are applicable for languages permiting assignment to procedure variables, aliasing, and label-valued variables. Their algorithms are $\kappa = 0$ precise and propagate information over non-realizable paths. Differences between our CGC algorithm and that presented in [Lak93] has been discussed earlier in the Introduction.

There are two key differences between our algorithm and Shivers' 1CFA algorithm [Shi91b]. Shivers' algorithm a) requires converting programs to continuation passing style (CPS) and b) is based on abstract interpretation. A procedure call in CPS never returns, instead it calls a *continuation* function passed as a parameter. Shivers 1CFA algorithm gives $\kappa = 1$ precision over a CPS converted program. Its precision with respect to the program before CPS conversion is, however, $\kappa = 0$ because addition of new procedures and continuations due to CPS conversion increases the number of calls needed to reach one procedure from another. In contrast our algorithm gives $\kappa = 1$ precision for the original program. Shivers has delved into some additional issues that we have ignored. The issues are a) assignment of procedure references to a structure (such as record or array) and b) treatment of incomplete programs. Decisions equivalent to those made by Shivers can easily be incorporated in our CGC algorithm.

The type determination algorithm of Pande & Ryder [PR93] and our algorithm are both worklist based interative algorithms. As mentioned earlier, the Pande-Ryder algorithm propagates 2–tuples with fields equivalent to the `binding` and `trigger` fields of our 3–tuple; it does not have the `site-of-descent` field. As a result the Pande-Ryder algorithm can not determine whether bindings reaching an indirect call-site have descended from the same call site or not. This renders it $\kappa = 0$ precise.

7 Summary and status

An algorithm for constructing call graph for a procedural language is presented. The algorithm is applicable for a larger class of languages than previous algorithms. The CGC problem is analogous to control flow analysis of Scheme and type determination in C++. Our algorithm is adaptable for these problems and improves upon their previous algorithms [PR93, PS91, Shi91b, Suz81].

A prototype of the algorithm has been implemented in Refine [Rea92]. This prototype demonstrates the working of the algorithm on a "toy" procedural language that models scope rules and parameter passing conventions of C. Implementation of the algorithm for C is currently in progress.

8 Bibliography

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

[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 Symposium on Principles of Programming Languages*, pages 29–41. ACM Press, 1979.

[Bur87] M. Burke. An interval-based approach to exhaustive and incremental interprocedural analysis. Technical Report RC 12702, IBM Research Center, Yorktown Heights, NY, September 1987.

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

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic contruction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Programming Languages*, pages 55–66. ACM Press, January 1991.

[CCHK90] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59. ACM Press, January 1989.

[HK93] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, 1993.

[Lak93] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL'93)*, pages 273–284, January 1993.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103. ACM Press, January 1991.

[LR92] William Landi and Barbara G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 235–248, July 1992.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5), May 1984.

[PR93] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. Technical Report TD-93-1, Tata Research Development and Design Center, Pune, India, 1993.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA'91: Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.

[Rea92] Reasoning Systems, Inc. Refine user's guide, 1992.

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

[Shi88] Olin Shivers. Control flow analysis in Scheme. In *SIGPLAN'88 Conference on Programming Languages Design and Implementation*, pages 164–174, 1988.

[Shi91a] Olin Shivers. *Control-flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[Shi91b] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, 1991.

[Spi71] T. C. Spillman. Exposing side-effets in a PL/I optimizing compiler. In *Proceedings IFIPS (Computer Software) Conference*, pages 56–60, 1971.

[Suz81] Norihisa Suzuki. Inferring types in Smalltalk. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 187–199. ACM Press, January 1981.

[Wei80] W. E. 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*, pages 83–94. ACM Press, January 1980.