McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

ALPHA: A Family of Structured Intermediate Representations for a Parallelizing C Compiler

Laurie J. Hendren
Guang R. Gao
Vugranam Sreedhar

ACAPS Technical Memo 49
November 17, 1992

hendren@cs.mcgill.ca, gao@cs.mcgill.ca, sreedhar@cs.mcgill.ca

Abstract

The design of intermediate languages plays a crucial role in the development of optimizing and parallelizing compilers. In this paper, we present the design of *ALPHA*, a Hybrid Executable Intermediate Representation. ALPHA is a hybrid in the sense that it combines a structured abstract syntax tree (AST) representation with a structured graph representation.

The AST representation has been designed to support general, pervasive and accurate dependence analysis for imperative languages with pointer data structures. In particular, our AST representation supports a location-based *alias analysis* that is implemented using abstract interpretation techniques. Aliasing information collected by the structured alias analysis is used to build the other part of the hybrid: a structured graph representation that encodes dependence information. As well as exposing parallelism, this graph representation can be used for sparse flow analyses and transformations.

Our graph representation is based on argument-fetching style dataflow graphs [DG88]. The executable nature of these graphs facilitates systematic algorithm development and verification of correctness. Furthermore, unlike most dataflow-model intermediate representations, which are based on *token-pushing* semantics, our graph representation supports an imperative (multiple updatable) storage model while keeping a well-defined operational semantics. This allows us to fully support imperative programs even in the presence of pointers.

We are currently implementing ALPHA as a part of the McCAT (McGill Compiler/Architecture Testbed) compiler. In this paper we present the methods used to build ALPHA graphs, and we illustrate the use of ALPHA for two case studies: constant propagation and induction variables detection.

Contents

List of Figures

List of Tables

1 Introduction and Motivation

With the recent advances in high performance architectures, the need for sophisticated compile-time analysis is increasing. This is particularly evident in compilation for architectures supporting some level of fine-grain or coarse-grain parallelism. It is becoming increasingly clear that the choice of intermediate representation greatly impacts the development of analysis methods. For example, the recent development of representations like the SSA form [CFR+89], and *sparse data flow evaluation graphs* [CCF91], has led to the development of efficient analyses that use these forms [RWZ88, CLZ86, AWZ88, WZ91].

The main motivation in our work was to investigate and develop a family of intermediate representations that would allow us to support a wide variety of analyses in a straightforward and efficient manner. We feel that a family of representations is required because different analyses require different representations. On one hand, one requires a representation that lends itself to the development and application of accurate alias and dependence analysis.[1] On the other hand, one would like dependence-based representations that: (1) support a wide-variety of efficient (sparse) analyses including those that have been successfully developed for the SSA form and other sparse representations, and (2) are directly executable and have a well-defined concurrent semantics. The latter feature is important for several reasons. First, a directly executable intermediate representation allows one to verify the correctness of various sophisticated analyses and transformations. Second, it is essential to support a concurrent execution semantics for the development of optimization techniques such as some software pipelining approaches [GWN91b, NG93].

An important facet of our work is that we wanted intermediate representations that can handle imperative programs including programs that use pointer data structures. Thus, we require both an intermediate representation that supports alias analysis in the presence of pointers, and a dependence-based intermediate representation that can represent dependences due to pointer aliasing in a natural fashion.

The development of our representations has been motivated by our experiences in building the McCAT compiler, a compiler targeted at high-performance architectures that support fine-grain parallelism and/or multi-threading [HDE+92]. Figure 1 shows both the overall structure of the compiler, and the new additions proposed in this paper. The high-level structure, as shown at the top of the figure, makes use of three structured intermediate representations FIRST, SIMPLE, and LAST [HDE+92]. Each representation is a structured AST that fulfills a specific role in the compiler. The FIRST representation provides a clean separation between the front-end and the back-end of the compiler, the SIMPLE representation provides a structured and simplified program representation that is convenient for alias and dependence analysis, and the LAST representation exposes low-level details that are appropriate for register allocation and instruction scheduling [HGAM92].

In this paper we focus on the ALPHA family: the SIMPLE, α-TREE and σ-α-GRAPH representations. As illustrated in the lower part of Figure 1, we start with the SIMPLE representation, and through various analyses and transformations we create the other two members of our family. The first phase of the process creates the α-TREE representation, while the second phase creates the structured σ-α-GRAPH representation.

The main contributions of this paper are:

- We discuss the foundations upon which we build: (1) the SIMPLE intermediate representation, and (2) the interprocedural *stack matrix analysis* that is used to compute accurate dependence

[1] In fact, it is these analyses that allow one to build the dependence-based representations.

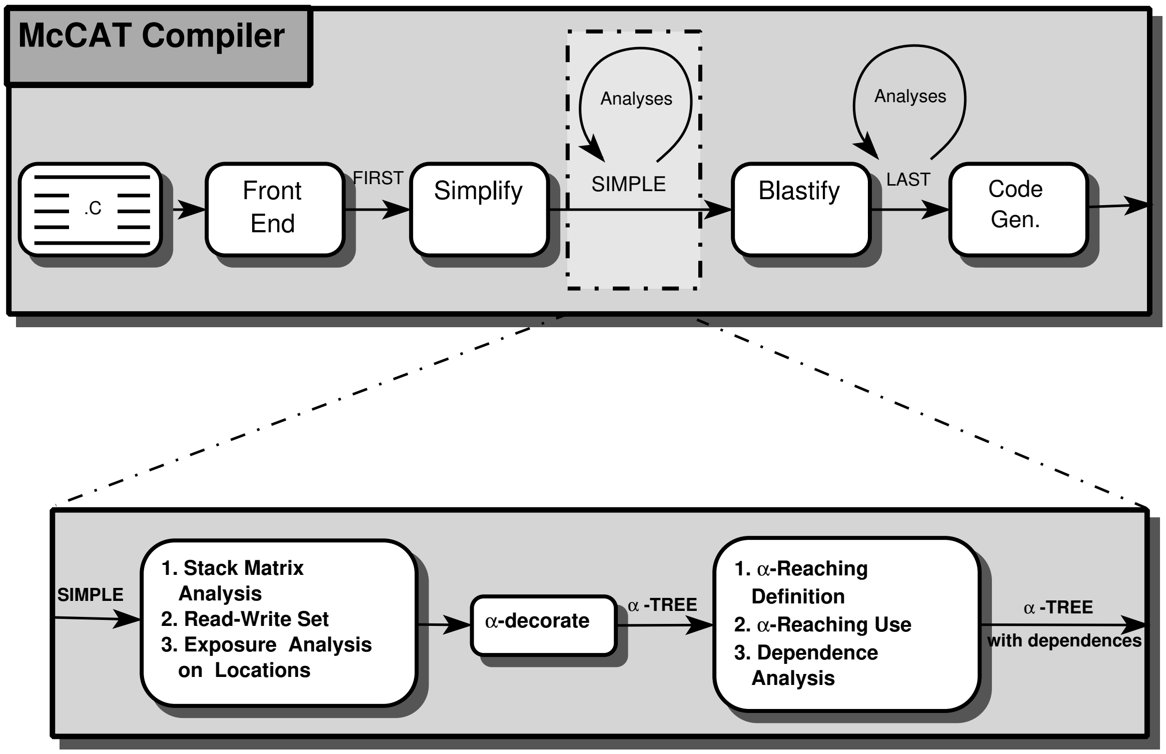


Figure 1: The McCAT Compiler

information even in the presence of stack-based pointers. The SIMPLE representation, and the stack matrix analysis makes use of the concept that every important location should have a name. This feature enables us to build the other representations in a straightforward manner.

- We introduce a new type of structured intermediate representation, α-TREEs and we demonstrate how this representation can be efficiently constructed from SIMPLE. The α-TREE representation differs from SIMPLE in that it includes α-statements that capture dependences entering and exiting structured control regions corresponding to conditionals and loops. This intermediate form is efficiently and correctly constructed even in the presence of pointers, `break` and `continue` statements in loops, and recursion. Furthermore, we show how the α-TREE representation lends itself to the development of efficient analyses. In particular, we show how reaching definitions can be efficiently implemented in both space and time. We also discuss other applications of α-TREEs for sparse analyses.

- We present a novel structured σ-α-GRAPH representation and illustrate how it can be directly obtained from the α-TREE representation. The graph representation is a major extension of argument-fetching style dataflow program graphs [DG88], and it features a parallel execution semantics. This representation provides ample opportunity for advanced loop optimization as well as code generation for multithreaded architectures.

The remainder of this paper is structured as follows. The next section, Section 2, provides a short introduction to both the SIMPLE intermediate form, and the alias analysis method we use as the foundation for further analysis and transformation. In Section 3 we discuss the two-phase process that is used to create the α-TREE and σ-α-GRAPH representations. In Section 4 we provide an overview the potential uses of the α-TREE and σ-α-GRAPH representations. Finally, we discuss how our work relates to other intermediate representations in Section 5, and give our conclusions in Section 6.

2 SIMPLE and Stack Matrix Analysis

As we stated in the introduction, one of our major goals was to develop an intermediate representation that can deal naturally with imperative programs and pointers. In order to achieve this goal we have designed both an appropriate structured intermediate form - SIMPLE, and an interprocedural alias analysis method called *stack matrix analysis*[HDE+92, Ema92]. Both SIMPLE and *stack matrix analysis* form the foundations for the development of our new α-TREE and σ-α-GRAPH representations.

2.1 SIMPLE

The SIMPLE intermediate representation has been designed specifically to provide a good basis for interprocedural alias analysis. The key features of SIMPLE are:

- The simplified program is composed of simple assignment statements, and structured compositional statments (`while`, `do-while`, `for`, `if-else`, and `switch-case`). The simplified program has structured control flow except for the explicit use of `break` and `continue` in loops, and `return` to exit from procedures. All of the compositional statements are simplified to conform to simple rules. For example, switch statments are transformed so that each alternative branch ends with a `break`, `continue`, or `return`. Furthermore, any expression in the loops or conditional constructs are transformed so that all side-effects occur in their bodies, and thus expressions are guaranteed to be side-effect free.
- Variable references are simplified to one of: a scalar reference, a structure reference, an array reference, or a one-level pointer reference. In addition, all type information is available and all type-casting is made explicit.

Two examples of simplified programs are given in Figure 2(b) and Figure 3(b). Further details about SIMPLE may be found in [HDE+92, Sri92]

2.2 Stack Matrix Analysis and Abstract Stack Locations

The fundamental point of our approach is that rather than collecting alias information in the form of alias pairs [Bar78, Ban79, Coo85, CK89, LR92, Lan92], we have chosen to directly abstract relationships between locations. More specifically, we use an abstraction which associates a name with every important stack location, and we use abstract interpretation techniques to approximate the *relationships* between these abstract locations.[2]

The goal of our analysis is to: (1) compute all the relationships between the important abstract stack locations, and (2) use the information about relationships to compute read and write sets for each

[2] In this paper we concentrate on stack-allocated pointers, dynamically-allocated pointers will be handled using a variation of *path matrix analysis*[Hen90, HN90, HHN92].

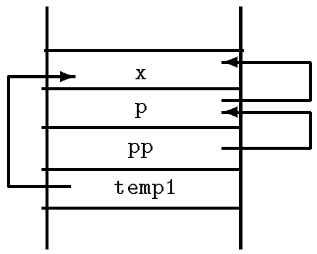
statement. Let us begin by illustrating the *stack-matrix* problem with a very simple example. Consider the program in Figure 2(a), and its SIMPLE form in Figure 2(b). Note that the statement `**p = 2` has been converted into two statements `temp_1 = *p; *temp_1 = 2`. By simplifying each reference to one level of indirection, we provide a name for each location on the stack.

```
{ int x, *p, **pp;

  x = 3;
  p = &x;
  pp = &p;
  **pp = 2;
}
```

(a) Example program

```
{ int x, *p, **pp;
  int *temp_1;

  x = 3;          /* 1 */
  p = &x;         /* 2 */
  pp = &p;        /* 3 */
  temp_1 = *pp;   /* 4 */
  *temp_1 = 2;    /* 5 */
}
```

(b) SIMPLE representation



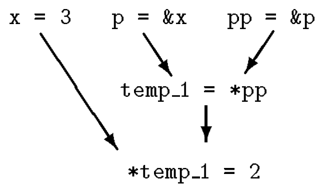
(c) Abstract Stack

	x	p	pp	temp1
x				
p	*D*			
pp		*D*		
temp1	*D*			

(d) Stack Matrix

S	Write(S)	Read(S)
`x = 3`	{x}	ϕ
`p = &x`	{p}	ϕ
`pp = &p`	{pp}	ϕ
`temp_1 = *pp`	{temp_1}	{pp, p}
`*temp_1 = 2`	{x}	{temp_1}

(e) Read/Write Sets



(f) Dependencies

Figure 2: A simple example of SIMPLE and *stack matrix analysis*

Given the simplified representation, we perform the first step of the analysis by computing for each program point the "points-to" relationships between important locations on the stack. Figure 2(c) gives a pictorial view of the relationships between the abstract locations at program point 4 for the simplified program given in Figure 2(b). In this example, we see that "`pp` points to `p`", "`p` points to `x`", and "`temp_1` points to `x`". Figure 2(d) gives the concrete *stack matrix* representation actually used in the analysis. In general, each entry in a stack matrix is one of: (1) empty indicating there is guaranteed to be no relationship between the abstract stack locations, (2) D indicating there is a *definite* relationship, or (3) P indicating that there is a *possible* relationship. As there is no approximation due to control flow in our example, we find that the stack matrix in Figure 2(d) contains only empty and D entries.

Given the stack matrix information at each program point, it becomes quite straightforward to define exactly which stack locations are written and read by each statement. Figure 2(e) summarizes the abstract stack locations read and written for our simple program, and Figure 2(f) gives the dependencies that are imposed by the reads and writes for each statement. Note that in the statement `temp_1 = *pp`, both the abstract location `pp` and the abstract location `p` are read.

It should be noted that our small example is intended as an introduction and it demonstrates only a small part of the stack matrix analysis methods. In fact, the complete interprocedural analysis properly handles both aliasing due to procedure calls with pointer arguments and recursion [Ema92]. The principal idea for procedures is that the correct number of abstract location names must be generated for each parameter that has a pointer type, and that the relationships between these generated names must safely approximate the actual relationships that hold at a call site. Thus, the generated abstract names are used to safely approximate all non-visible locations that may be affected by executing the procedure.[3] The complete analysis has been implemented for both scalars and structured data, and it is operational at the SIMPLE level in the McCAT C compiler.

For the purposes of this paper, the important point is that the location-based stack matrix analysis provides exactly the information required to define accurate read and write sets for each statement. By ensuring that every important location is safely represented in the stack matrices, we can use the read and write sets as the basis for further analysis. Table 1 summarizes the read and write sets for each basic statement type in SIMPLE. Given a statment s, and a stack matrix SM, the table shows how to compute: (1) $W_p(s, SM)$ - the set of abstract locations possibly written by s, (2) $W_d(s, SM)$ - the set of abstract locations definitely written by s, and (3) $R(s, SM)$ - the set of abstract locations read by s. Note that in this table $P_d(x, SM)$ denotes the set of abstract locations definitely pointed to by abstract location x (all abstract locations y such that $SM[x, y] = D$), $P_p(x, SM)$ denotes the set of abstract locations possibly pointed to by abstract location x (all abstract locations y such that $SM[x, y] = P$), and $P(x, SM)$ denotes $P_d(x, SM) \cup P_p(x, SM)$. Furthermore, we can define $W(s, SM)$ to be $W_p(s, SM) \cup W_d(s, SM)$.

s	$W_p(s, SM)$	$W_d(s, SM)$	$R(s, SM)$
`x = y`	ϕ	{`x`}	{`y`}
`x =` *unop* `y`	ϕ	{`x`}	{`y`}
`x = y` *binop* `z`	ϕ	{`x`}	{`y`,`z`}
`*x = y`	$P_p(x, SM)$	$P_d(x, SM)$	{`x`,`y`}
`*x =` *unop* `y`	$P_p(x, SM)$	$P_d(x, SM)$	{`x`,`y`}
`*x = y` *binop* `z`	$P_p(x, SM)$	$P_d(x, SM)$	{`x`,`y`,`z`}
`x = &y`	ϕ	{`x`}	ϕ
`x = *y`	ϕ	{`x`}	{`y`} $\cup P(y, SM)$

Table 1: Read and Write Sets for statement s relative to stack matrix SM

3 Constructing the α-TREE and σ-α-GRAPH Representations

In the previous section we introduced the stack matrix analysis that is performed on the SIMPLE intermediate representation, and we demonstrated how the stack matrix information can be used to accurately calculate read and write sets for the basic statements in SIMPLE. In this section we discuss the various analyses and transformations that are required to transform SIMPLE trees to our new α-TREE and σ-α-GRAPH representations.

3.1 An example program

Figure 3(a) provides a C program that demonstrates conditionals, loops, breaks and pointers. Figure 3(b) shows the resulting program after simplification and stack matrix analysis. Note that we have indicated the results of the stack-matrix analysis at three important program points. At program point `A`, the stack matrix analysis indicates that abstract location `q` definitely points to `x`, and `p` definitely points to `y`. Similarly, at program point `B`, just before the `break` from the loop, the analysis reveals that both `q` and `p` definitely point to `y`. Just after the loop, at program point `C`, the analysis shows that `b` definitely points to `y`, while `q` points to either `x` or `y`.

[3] This appears to be a generalization of the `nonvisible` name used in [Lan92, LR92].

```
main()
{ int u, v, x, y;
  int *p,*q ;

  x = read_int() ;
  y = read_int() ;
  p = &y ;
  q = &x ;
  u = *p + *q ;

  while (u < 100)
  { v = read_int() ;
      if (u == v)
        { q = p ;
          p = &y ;
          break ;
        }
      u++ ;
      (*p)++ ;
  }

  if (v == 10)
    *q = *q + 1 ;
  else
    *p = *p + 1 ;

  return(*p + *q) ;
}
```

(a) original program

```
int main()
{ int u,v x,y;
  int  * p;
  int  * q;
  extern int read_int();

  int temp_0, temp_1, temp_2, temp_3;
  int temp_4, temp_5, temp_6, temp_7;

  x = read_int();
  y = read_int();
  p = (& y);
  q = (& x);
  temp_0 = (* p);
  temp_1 = (* q);
  u = (temp_0 + temp_1);

  /*-- POINT A:  -------*/
  /*-- (p->y), (q->x) --*/
  while ( (u <= 99) )
    { v = read_int();
      if ((u == v))
        { q = p;
          p = (& y);
          /*-- POINT B: -------*/
          /*-- (q->y), (p->y) -*/
          break;
        }
      u = (u + 1);
      temp_2 = (* p);
      (* p) = (temp_2 + 1);
    }
 /*-- POINT C: ----------------*/
 /*-- (p->y), (q->x)?, (q->y)?-*/

  if ((v == 10))
    { temp_3 = (* q);
      (* q) = (temp_3 + 1);
    }
  else
    { temp_4 = (* p);
      (* p) = (temp_4 + 1);
    }

  temp_6 = (* p);
  temp_7 = (* q);
  temp_5 = (temp_6 + temp_7);
  return (temp_5);
}
```

(b) after simplification and stack-matrix analysis

```
int main()
{ int u,v x,y;
  int  * q;
  extern int read_int();

  int temp_3, temp_5, temp_7;;

  x = read_int();
  y = read_int();
  q = (& x);
  u = (y + x);

  /*-- POINT A:  -------*/
  /*-- (p->y), (q->x) --*/
  while ( (u <= 99) )
    { v = read_int();
      if ((u == v))
        { q = (& y);
      /*-- POINT B: --------*/
      /*-- (q->y), (p->y) --*/
          break;
        }
      u = (u + 1);
      y = (y + 1);
    }
/*- POINT C: --------------*/
/*- (p->y),(q->x)?,(q->y)?-*/

  if ((v == 10))
    { temp_3 = (* q);
      (* q) = (temp_3 + 1);
    }
  else
    y = (y + 1);

  temp_7 = (* q);
  temp_5 = (y + temp_7);
  return (temp_5);
}
```

(c) after pointer-elimination, copy propagation, and dead-code removal

Figure 3: An example program with simplification and pointer-elimination

Figure 3(c) shows how the results of stack matrix analysis can be used to replace pointer accesses with the appropriate scalar. Whenever a pointer variable *definitely* points to an abstract stack location corresponding to a variable, we can replace the pointer reference with the appropriate scalar. For example, the statement just before program point `A` we can replace `*p` with `y` and `*q` with `x`. Figure 3(c) gives the program after pointer-replacement has been applied to the entire program. In order to simplify further discussion, we have also applied copy-propagation and dead-code removal to give a smaller and cleaner example.

3.2 Constructing α-TREEs

The SIMPLE intermediate representation provides a natural way of representing nested regions of control structures. In SIMPLE the following basic constructs introduce a new region: `if-else`, `switch-case`, `for`, `while`, and `do-while`. In most cases control flows into the entry of the control region and out of the exit in the control region. In the case of `break` and `continue` statements, control goes to the exit or entry of the closest enclosing loop.

The central idea behind the α-TREE representation is that we wish to provide interception points for all dependences (flow, output, and anti) for locations that are read and/or written in the control region. This is accomplished by placing α-lists at the entry and exit of control regions. An α-list is simply a list of α-statements where each α-statement is a pseudo-assignment (identity assignment) of the form `l=`α`(l)`, and `l` is the name of an abstract location. We call the process of transforming a SIMPLE AST to an α-TREE , *α-decorating.* Figure 4 shows where the α-statements are placed relative to the original components for the `if-else` and `while` statements. Note that in both cases control must enter the region through the *inlist* ($l_{in_1}=\alpha(l_{in_1})$; $\ldots$; $l_{in_m}=\alpha(l_{in_m})$;) and exit through the *outlist* ($l_{out_1}=\alpha(l_{out_1})$; $\ldots$; $l_{out_n}=\alpha(l_{out_n})$;). In the case of the while loop at region k, any `break` statement in the body of the while loop would correspond to "`goto exit_k`" and any `continue` statement would correspond to "`goto entry_k`".

entry_k:
    $l_{in_1} = \alpha(l_{in_1});$
    ...
    $l_{in_m} = \alpha(l_{in_m});$
    if not < e >goto else_k;
    < s_then >
    goto exit_k;
else_k:
    < s_else >
exit_k:
    $l_{out_1} = \alpha(l_{out_1});$
    ...
    $l_{out_n} = \alpha(l_{out_n});$

IF(inlist,e,s_then,s_else,outlist)

entry_k:
    $l_{in_1} = \alpha(l_{in_1});$
    ...
    $l_{in_m} = \alpha(l_{in_m});$
    if not < e >goto exit_k;
    < s_body >
    goto entry_k;
exit_k:
    $l_{out_1} = \alpha(l_{out_1});$
    ...
    $l_{out_n} = \alpha(l_{out_n});$

WHILE(inlist,e,s_body,outlist)

Figure 4: Alpha-decorated conditional and loop

By inserting the α-statements, we provide pseudo-assignments that force a *definite* use and a *definite* definition for each location.[4] Thus, just as the ϕ-functions in the SSA form provide a place for collecting dependences at the exit of control regions, the α-lists provide the appropriate places at both the entry

[4] Note that we consider abstract location names and not variable names. This distinction is very important since it allows us to handle pointer variables very naturally.

and exit of control regions. It should be noted that these pseudo-assignments do not correspond to any instructions that must be generated, they are only a tool for further analysis. Furthermore, unlike the SSA form, there is no need to insert extra names for locations. We only introduce new names when we explicitly wish to break output dependences.[5] Further comparisons with the SSA form are given in Section 5.

Given our SIMPLE tree representation, and the stack matrix analysis, the α-decorating procedure is straightforward and inexpensive to compute. Figure 5 summarizes the rules used to determine which locations should be captured by α-statements. The left column illustrates the rules for capturing an abstract location `x` on entry to a region, while the right column illustrates the rules for capturing `x` on exit of a region. The columns illustrate capture dues to *flow*, *anti*, and *output* dependences. In the figure, the square boxes correspond to conditional or loop statements which are denoted by S in the associated formula. Note that these formulae make use of *side-effect analysis* ($read(S)$ and $write(S)$), and *exposure analysis* ($will\_be\_read(S)$, $will\_be\_written(S)$, $was\_read(S)$, and $was\_written(S)$).

We will demonstrate the simplicity and inexpensive nature of the capturing process by examining the capture of flow dependences exiting out of control regions (the upper right entry in Figure 5). In this case we need to know the set of abstract locations written by the region corresponding to statement S. As indicated in the figure, a location `x` should be captured if it is in the set $write(S) \cap will\_be\_read(S)$. We can compute the *write* set of any statement S by simply taking the union of the *write* sets of the components of S. For example, $write(IF(e, s\_then, s\_else)) = write(s\_then) \cup write(s\_else)$. The read and write sets can clearly be collected by one pass over the SIMPLE tree. The $will\_be\_read(S)$ set is the set of abstract locations that may be read (before redefinition) between the exit of S and the exit of the procedure. This corresponds to the normal notion of live variable analysis and can also be computed in $O(n)$ time (this algorithm is a variation of the structured flow analysis algorithm found in [ASU86], and is given in full in Appendix A).

All the other sets required for capturing can also be computed in a similar manner. For these one calculates $will\_be\_written(S)$ - locations after the exit from S, $was\_read(S)$ - locations read before entry into S, and $was\_written(S)$ - locations written before entry into S.

We use the capturing process to transform a SIMPLE tree to an α-TREE . For each conditional and loop statement we store locations captured on entry in a list of α-statements, *inlist*, and locations captured on exit in *outlist*. In each case we use the rules as given in Figure 5 to determine which locations should be captured. The only exception is that a conditional that contains a break must inherit its *inlist* and *outlist* from the *inlist* of the closest enclosing loop. Figure 6 gives the α-TREE version of our example program. In the case of the `while` loop in this example, we see that the locations read are $\{q, x, y\}$, and the locations written are $\{u, v, q, y\}$. Using these read and write sets, plus the exposure analysis, we find that locations $\{u, y, q\}$ must be captured on entry by the inlist, and $\{y, q\}$ must be captured on exit by the outlist. Note that since `v` is only used inside the loop, it does not occur in either the inlist or outlist, and since `u` is not used after the loop, it does not appear in the outlist.

3.3 Computing α-Reaching Definitions

Once the α-TREE representation has been created, we have the complete infra-structure required to support efficient analyses (including dependence analysis). We illustrate this by presenting the solution for α-reaching-definitions. This problem is also useful as the first step in building the σ-α-GRAPHs .

Figure 7 gives the complete structured analysis for α-reaching-definitions. The major difference between this analysis and the ordinary reaching-definitions analysis is that we make use of the fact that all definitions entering control regions must have been captured by an α-statement. This means that we propagate only the definitions generated by the α-statements of the inlist inside each control region. For example, consider the rule for IF(inlist,e,s_then,s_else,outlist) in Figure 7. The first action in this rule

[5] This is done using an extension of the renaming algorithm found in [ZC90].

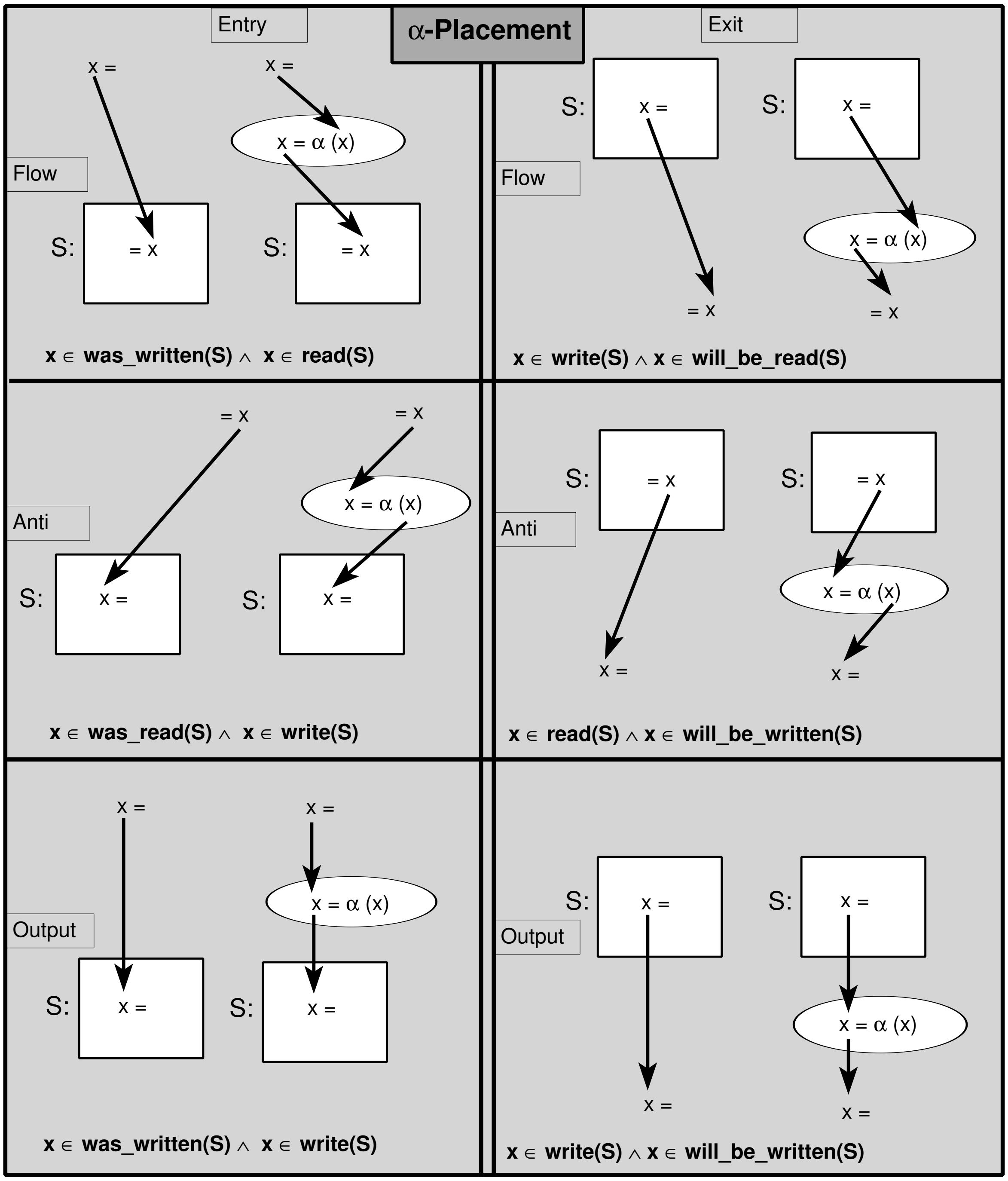


Figure 5: Capturing dependences with α-nodes

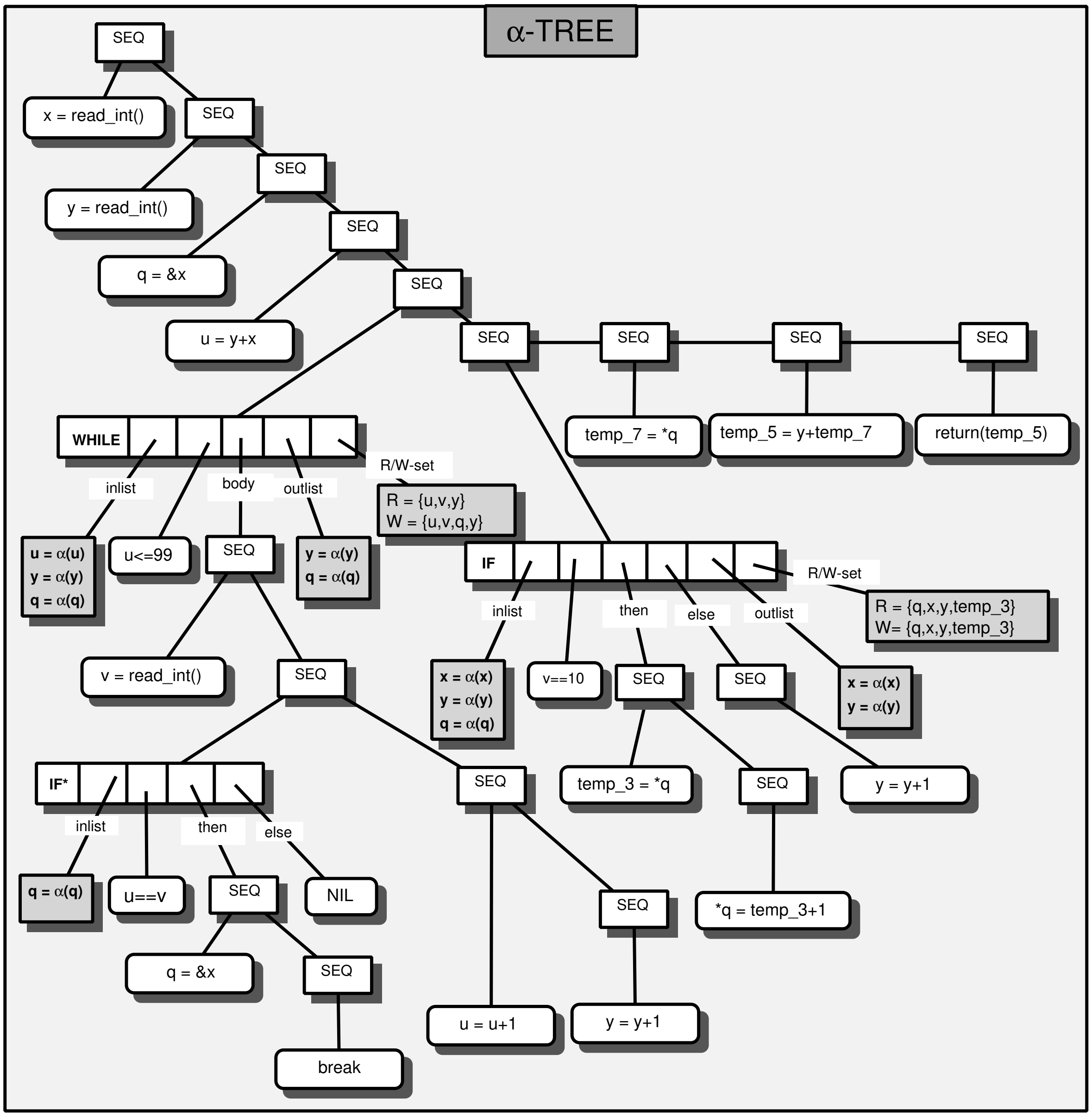


Figure 6: The α-TREE with R/W sets and α-lists

specifies that we should store in s.inlist_rdefs the set of all reaching definitions for locations that will be captured by α-statements in the inlist. These reaching definitions are used at a later stage to insert flow and anti dependences between definitions outside of the region and the appropriate α-statement capturing the dependence. The second statement, in_defs := defs(inlist);, specifies that we should create a set of α-reaching-def pairs, one pair for each α-statement in the inlist. A statement of the form *s_i: l=*α*(l)* corresponds to the pair < *s_i, l* >. The third statement uses in_defs as input and recursively applies the α-ReachDef function to the components s_then and s_else, and stores the merge ($\bowtie$) of the two output sets in s.outlist_rdefs. This set will be used to insert flow and output dependences between statements with the IF statement and the α-statements in the outlist. Finally, if the result of analyzing the components is not $\perp$ ($\perp$ indicates that a path was interrupted by a `break` or `continue` statement), then we return the reaching definitions generated by the α-statements of the outlist, along with any definitions that were not redefined (were skipped) by this IF statement. Note that any definitions generated inside the IF statement are captured by an α-statement in the outlist. The other rules in Figure 7 behave similarly.

The important point to note is that the α-reaching_def analysis is efficient in time because it visits each node in the α-TREE only once (there is no need for iteration), and it is efficient in space because we propagate only the information needed for the inlist inside a region. In fact, we have developed a definition number system that allows us to propagate short bit-vectors inside each region. We expand upon this idea in the full paper. Another point to note is that the α-lists provide a natural barrier between regions, and provide a simple way to chop up a dataflow problem into independent pieces. Thus, this supports parallel flow analysis.

3.4 Constructing σ-α-GRAPHs

Given the α-reaching_defs and the results of a similar analysis for α-reaching_uses, we can completely construct flow, anti, and output dependences. We then use these dependences to construct our final intermediate representation, σ-α-GRAPHs . Figure 8 gives the resulting σ-α-GRAPH for our example program.

The σ-α-GRAPH representation, a major extension of the argument-fetching style dataflow graphs [DG88], supports a parallel execution semantics. Nodes in a σ-α-GRAPH represent operations, and edges represent precedence relations between nodes. That is, a node is "enabled" for execution as long as all precedence relations are satisfied — there is no a priori fixed sequential total order imposed on the execution of the nodes. It also retains another feature of the argument-fetching dataflow model, in that *signals*, not data "tokens" are sent along the output arcs to indicate the completion of execution a node.

Note that the example graph is composed from three types of nodes: simple-nodes, σ-nodes, and α-nodes (refer to the legend in Figure 8). Simple nodes are self explanatory. The σ-nodes are introduced to implement the "switch" functions required in conditionals and loops. A σ-node, first waits for a signal from all of its input arcs, and then executes. When executed, it first evaluates its boolean predicate, and then according to the boolean result (true or false) it send signals to either the successor nodes in its T list or its F list (as indicated in the figure). The α-nodes, placed at the output of conditionals and the input and output of loops, play the function of "merging" its incoming signals: i.e. when an incoming signal arrives along one of its arcs, it will forward the signal to all of its output arcs. Following this explanation, the reader is encouraged to hand simulate the execution of the example graph.

It should be noted that σ-α-GRAPHs are different from any traditional dataflow graphs in several major aspects. First, the σ-α-GRAPH model has, as a part of the operational semantics, an imperative store which is updatable. As a result, the arcs can represent flow dependence as well as other storage-based dependences (for example, anti and output dependences). The argument-fetching style dataflow model facilitates this extension. Secondly, as an intermediate form for compiler optimization, σ-α-GRAPHs have retained the sequential order between the iterations of the original loops — the sig-

```
/* Given s - a statement,   in - the α-reaching_def set at the input of s,   and loop - the
   closest enclosing loop, compute the α-reaching_def for the components of s, and return the
   α-reaching_defs at the output of s. Each α-reaching_def is a pair < stmt#,l >where stmt# is a
   statement id, and l is a location being defined by that statement. As a side-effect,
   definitions reaching the input of statement s will be stored in s.in_rdefs, and definitions reaching
   break statements will be stored in s.break_rdefs where s in the closest enclosing loop for the break.
   For the IF and WHILE constructs, assume that the α-lists have already been inserted, and store
   the definitions reaching the input of the inlist (s.inlist_rdefs) and outlist (s.outlist_rdefs).
   Note: ⊥⋈⊥→⊥, ⊥⋈s→s, s⋈⊥→s, and s1⋈s2→(s1 ∪ s2) */

α-ReachDef(s,in,loop) =
{ case s of
    ASSIGN (lhs,rhs) ⇒
      { s.in_rdefs := in;
        kill_defs := { the set of all defs < s#,l >∈ in, such that l ∈ W_d(s,s.SM) };
        gen_defs := { the set of all pairs < s.stmtid,l >such that l ∈ W(s,s.SM) };
        return(gen_defs ∪ (in - kill_defs));
      }
    SEQ (s1,s2) ⇒
      { s1.in_rdefs := in;
        s2.in_rdefs := α-ReachDef(s1,in,loop);
        if (s2.in_rdefs = ⊥ ) then return(⊥ ) else return(α-ReachDef(s2,s2.in_rdef,loop));
      }
    IF (inlist,e,s_then,s_else,outlist) ⇒
      { s.inlist_rdefs := { the set of all defs < s#,l >∈ in, such that the statement l=α(l) ∈ inlist };
        in_defs := defs(inlist);
        s.outlist_rdefs := α-ReachDef(s_then,in_defs,loop) ⋈ α-ReachDef(s_else,in_defs,loop);
        if (s.outlist_rdefs = ⊥ ) then
          return(⊥ );
        else
        { out_defs := defs(outlist);
          skip_defs := { the set of all defs < s#,l >∈ in, such that the statement l=α(l) ∉ outlist };
          return(out_defs ∪ skip_defs);
        }
      }
    WHILE (inlist,e,s_body,outlist) ⇒
      { s.inlist_rdefs := { the set of all defs < s#,l >∈ in, such that the statement l=α(l) ∈ inlist };
        in_defs := defs(inlist);
        s.inlist_rdefs := α-ReachDef(s_body,in_defs,s) ⋈ s.inlist_rdefs;
        s.outlist_rdefs := in_defs ∪ s.break_rdefs;
        out_defs := defs(outlist);
        skip_defs := { the set of all defs < s#,l >∈ in, such that the statement l=α(l) ∉ outlist };
        return(out_defs ∪ skip_defs);
      }
    CONTINUE ⇒
      { loop.inlist_rdefs := loop.inlist_rdefs ∪ in;   return(⊥ ); }
    BREAK ⇒
      { loop.break_rdefs := loop.break_rdefs ∪ in;   return(⊥ ); }
    NULLSTMT ⇒  return(in);
}
```

Figure 7: Computing the *alpha-reaching_def* sets for each statement

nificance of this decision is elaborated in the following sections. Thirdly, the $\alpha-$nodes are "virtual" merge nodes, that is they need not be executed. In fact, as an acute reader may observe, they can be subsumed into their successor nodes for efficient execution. In Figure 8), we placed the α-statements in the nodes for the purpose of simple comparisons between the α-TREE and σ-α-GRAPH examples.

Finally, it should be noted that σ-α-GRAPHs are structural and compositional, and can be constructed via straightforward structured translation rules from the corresponding α-TREE . We leave the description of these rules to [HGS92]. These graphs are also *well-formed* in the sense that, although it has a parallel operational model, the execution of the graphs will never deadlock.[6]

4 Applications

In this section we briefly discuss some of the most important applications that can be efficiently implemented using our representations. We will discuss these in more detail in the full paper.

4.1 Applications of the SIMPLE Representation

The SIMPLE representation lends itself to the development of structured (compositional) analyses. In this paper we have introduced the more complex *stack matrix analysis*, and we have presented a very simple analysis for computing the *will_be_written* set. In our experience with the McCAT compiler, we have found SIMPLE to be a good setting for the development of a variety of analyses that are based on abstract interpretation techniques [Sri92, Ema92]. In addition, we have used it for the development of an analyzer generator, McTAG [Sri92].

4.2 Applications of the α-TREE Representation

As illustrated by the α-ReachDef algorithm presented in Figure 7, we can make use of the α-lists in the α-TREE representation to support structured analyses that are efficient in both time and space. The reaching definition and reaching use information can be used to efficiently compute dependence information in the α-TREEs , and thus provide the framework for implementing efficient algorithms that make use of the sparse dependence structure [HGS92]. For example, we have studied how to adapt the techniques sparse constant propagation [WZ91], and induction variable detection [Wol92].

We also expect that we can improve some analyses that have previously been implemented on SIMPLE by exploiting both the sparse dependence information, and the region separation provided by the α-lists. Furthermore, based on our experience of implementing loop transformations such as unimodular transformation [SV92, SG92], and collective loop optimization [GOST92, Ols92], we believe that α-TREE representation provides a convenient and efficient platform for such loop optimizations. Although these transformations could also be done using the σ-α-GRAPH representation, we see no advantages to using this more complex representation.

4.3 Applications of the σ-α-GRAPH Representation

The σ-α-GRAPH representation also has important applications. For example, with its parallel execution model, it is a powerful base to perform such loop scheduling. In fact, the efficient exploitation of fine-grain parallelism in loops has been a major goal in the design of the McCAT compiler. Fine-grain loop scheduling such as [Lam88] has been proposed for high-performance pipelined processor

[6] This should not be a surprise, since we derive the graphs from a sequential program. However, this is a very desirable property for any parallel operational model. For our σ-α-GRAPHs , it can be proved via *structural induction*. However, a formal exposition is beyond the scope of this paper.

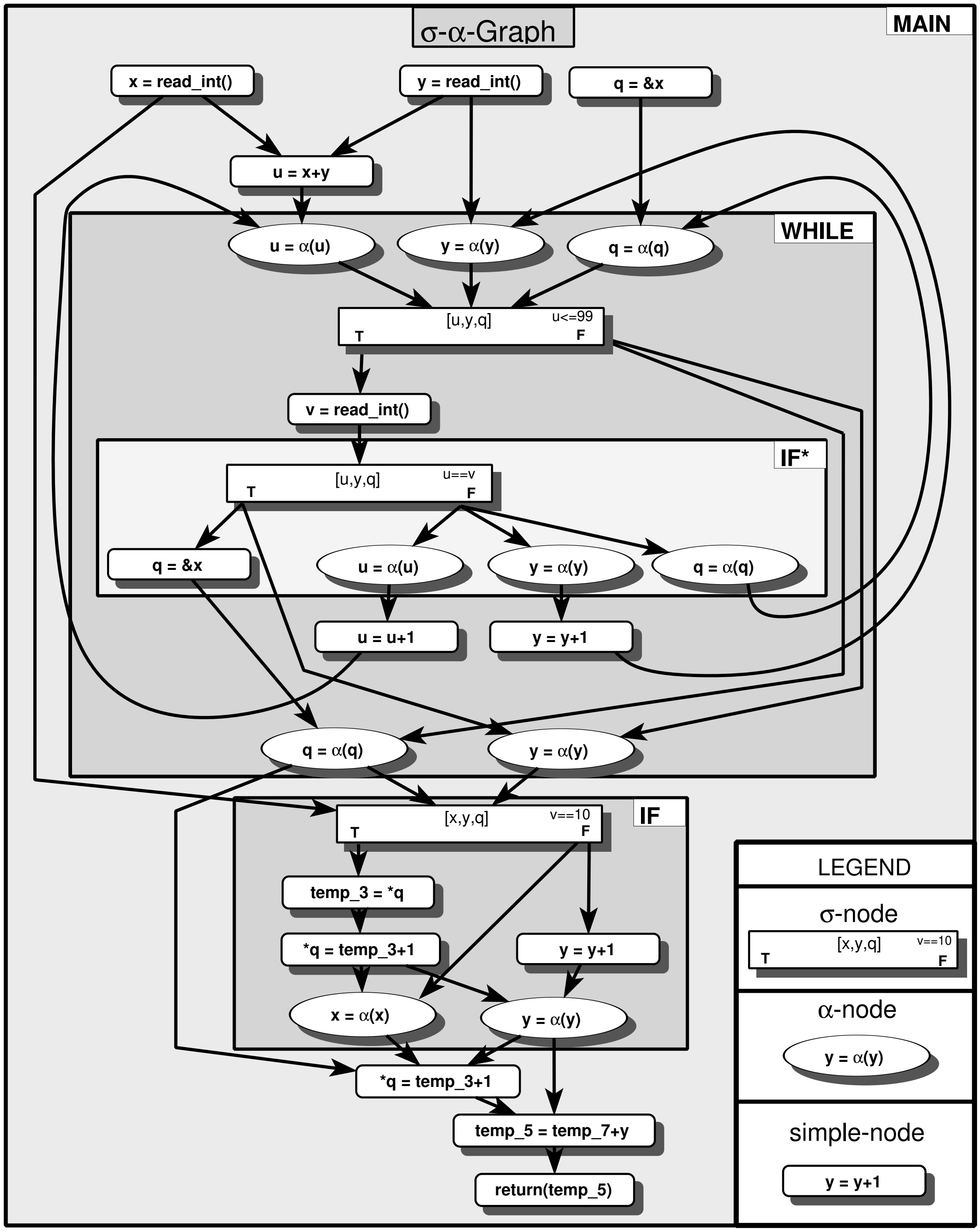


Figure 8: The α-GRAPH: an executable representation

architectures, as well as super-scalar and VLIW architectures. One approach for software pipelining [AN88, GWN91b], is to let a compiler to emulate the parallel execution of the loop under a certain operational model, until a periodic pattern is "discovered" which often consists of instructions from a number of consecutive iterations. For example, in [GWN91b], we have proposed an efficient method where the loop is modeled by *static dataflow graph*, which is directly executed (at compile-time) under the "earliest firing rule" to determine a software pipelined schedule under resource constraints.

We also expect that σ-α-GRAPHs naturally form a basis for the implementation of a unified framework of concurrent loop scheduling and register allocation [NG92, NG93]. A parallel operational model of loops (as its intermediate form) is essential in such framework. Finally, the σ-α-GRAPH representation will be used for generating threads for multithreaded architecture models, and also for generating parallel tasks for multiprocessor execution.

5 Related Work

Our family of intermediate representations has been designed to benefit from a variety of techniques that apply to tree-based and/or graph-based representations.

Tree-based representations have traditionally been used to represent a program as a high-level abstract syntax tree that is used for semantic analysis, for source-to-source transformations, or as a source for lower-level intermediate code generation. Trees have also been used for program development tools [Ros85], and in some dependence analyzer tools (parallelizing compilers) for performing loop transformations and other source-to-source transformations [AK82, Wol91]. With the advent of structured programming languages, automatic structuring techniques, and abstract interpretation analysis, it has become clear that many compile-time analyses naturally use a tree-like representation. Two compilers that follow this route include the MUG2 compiler [Wil81], and the recent Miprac parallelizing compiler [HIA92] that uses a low-level tree representation as the basis for a wide variety of interprocedural abstract interpretation analyses. Our SIMPLE and α-TREE representations allow us to fully exploit similar techniques. We believe that SIMPLE is particularly suited to the development of accurate alias analysis

In many compiler projects, the intermediate representation is based on some kind of graph. These include: *control-flow graphs* [ASU86], *data dependence graphs* [KKP+80], *program dependence graphs* (PDG)[FOW87] and *program dependence webs* [BMO90], the *static single assignment form* [CFR+89] and *sparse evaluation graphs* [CCF91], *dataflow program graphs* [Ack84, Tra86] and *dependence flow graphs* [PBJ+91]. Our α-TREE and σ-α-GRAPH representations have been designed to incorporate many of the interesting features of these other approaches.

The development of the α-TREE representation was influenced by the work on sparse representations based on single assignment principle [CFR+89, CCF91]. However, unlike the SSA form, we do not introduce unnecessary new names. We introduce names solely for the purpose of naming locations and removing (spurious) dependences. Furthermore, unlike the SSA form where one must process the ϕ-functions after analysis, one may generate code directly from the α-TREE representation by simply ignoring the α-statements. As we have demonstrated, the α-TREE representation provides an excellent framework to perform dataflow analyses and other optimizations efficiently at the desirable level of sparsity. We get sparseness and compactness both in terms of the evaluation process and as well as the size (i.e. bit-vector length) of the information that is being propagated to perform the analysis. Another major feature of our approach is that it has been designed, from the start, to handle programs with pointer data structures, and to represent dependence information both accurately and efficiently in the presence of pointer aliasing in such programs.

Another interesting representation that is relevant to our σ-α-GRAPH representation is the dependence flow graph (DFG) approach [PBJ+91]. Like σ-α-GRAPHs , the DFG representation is designed to have a parallel executable model. One feature of the σ-α-GRAPH representation is that it is based on a generalization of the argument-fetching dataflow graph model which is among the first to advocate an operational semantics that maintains the advantage of dataflow model without using a token-pushing style execution. Furthermore, σ-α-GRAPH support a general updatable storage model in the sense that the arcs represent precedence relations due to any storage related dependences, not restricting to just flow dependence. Another important difference between σ-α-GRAPHs and DFGs is the representation of loops. The semantics of a loop, as modeled in DFGs [PBJ+91], is essentially based on the loop unravelling semantics of dynamic dataflow model [AG82]. Although it allows maximum fine-grain parallelism to be fully represented (and can be exploited if run on an idealized dynamic dataflow machine), there is an apparent gap between such parallel execution model and the conventional machine architectures. Perhaps such gap can be filled by novel compiling techniques yet to be developed. In designing σ-α-GRAPHs , however, we have decided to retain the sequential order between iterations in the original loop, while providing explicit loop-carried dependence in the representation for the purposes of loop transformation. Futhermore, while dataflow scheduling semantics is directly useful for compile-time optimization such as software pipelining, it is not clear that **dynamic** dataflow semantics is the best choice. For instance, in [GWN91a], our software pipelining technique is based on dataflow graphs with **static** dataflow semantics. On the other hand, our work on simultaneous loop scheduling and register allocation is based on dataflow graphs with FIFO dataflow semantics [NG93] – somewhere in between pure dynamic and pure static dataflow execution models. Our position is that σ-α-GRAPHs provide a template representation for loops and provides the compilers with ample opportunity and flexibility to adapt suitable loop execution models for different optimizations. Furthermore, it also can serve as a good base for generating code for multithreaded architectures [ACC+90, NPA92].

6 Conclusions and Future Work

In this paper we have presented ALPHA, a family of structured intermediate representations that are used in our McCAT C compiler. The fundamental point of the paper is that we require a family of representations because each representation facilitates different sorts of analyses. The three members of the ALPHA family are the SIMPLE, α-TREE , and σ-α-GRAPH representations.

The SIMPLE representation and the associated stack matrix analysis are used as the foundations upon which we build the other two representations. These foundations are important because they allow us to provide intermediate representations that naturally handle dependences involving pointers.

We have given the procedure for producing the α-TREE representation from SIMPLE, and we have shown that this procedure is efficient. Furthermore, we demonstrated that by introducing dependence capturing α-statements, we provide an intermediate representation that supports analyses that are efficient in space and time.

The third member of the ALPHA family, the σ-α-GRAPH representation, is executable and has a parallel operational semantics. We argued that this representation facilitates the development of loop transformations like software pipelining, as well as multithreaded code generation.

The SIMPLE representation and the associated stack matrix analysis has been completely implemented in the McCAT compiler. We have designed and are currently in the process of completing the implementation of the α-TREE and σ-α-GRAPH representations. We are continuing our work on developing applications for each member of the ALPHA family.

References

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International*

Conference on Supercomputing, pages 1–6, Amsterdam, The Netherlands, June 11–15, 1990. ACM. *Computer Architecture News,* 18(3), September 1990.

[Ack84] W. B. Ackerman. *Efficient Implementation of Applicative Languages.* PhD thesis, Massachusetts Institute of Technology, April 1984. Also published as Technical Report MIT/LCS/TR-323, MIT Laboratory for Computer Science.

[AG82] Arvind and K. P. Gostelow. The U-Interpreter. *Computer*, 15(2):42–49, February 1982.

[AK82] J. R. Allen and K. Kennedy. PFC: A program to convert FORTRAN to parallel form. Technical Report MASC TR82-6, Rice University, Houston, TX, March 1982.

[AN88] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 22–24, 1988. ACM SIGPLAN. *SIGPLAN Notices,* 23(7), July 1988.

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

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

[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.

[Bar78] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, New York, June 20–22, 1990. ACM SIGPLAN. *SIGPLAN Notices,* 25(6), June 1990.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991. ACM SIGACT and SIGPLAN.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single-assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[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, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[CLZ86] Ron Cytron, Andy Lowry, and Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, Florida, January 13–15, 1986. ACM SIGACT and SIGPLAN.

[Coo85] Keith Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT and SIGPLAN.

[DG88] Jack B. Dennis and Guang R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of Supercomputing '88*, pages 368–373, Orlando, Florida, November 14–18, 1988. IEEE Computer Society and ACM SIGARCH.

[Ema92] M. Emami. An alias analysis for stack-allocated data structures. Master's thesis, McGill University, expected December 1992.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GOST92] G. R. Gao, R. Olsen, V. Sarkar, and R Thekkath. Collective loop fusion for array contraction. In *5th Workshop on Languages and Compilers for Parallel Computing.* August 1992. Also to appear in LNCS.

[GWN91a] Guang R. Gao, Yue-Bong Wong, and Qi Ning. A Petri-Net model for fine-grain loop scheduling. ACAPS Technical Memo 18, School of Computer Science, McGill University, Montréal, Québec, January 1991.

[GWN91b] Guang R. Gao, Yue-Bong Wong, and Qi Ning. A timed Petri-Net model for fine-grain loop scheduling. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 204–218, Toronto, Ontario, June 26–28, 1991. ACM SIGPLAN. *SIGPLAN Notices,* 26(6), June 1991.

[HDE+92] L. J. Hendren, C. Donawa, M. Emami, G. Gao, J. Justiani, and B Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *5th Workshop on Languages and Compilers for Parallel Computing.* August 1992. Also to appear in LNCS.

[Hen90] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, January 1990.

[HGAM92] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In U. Kastens and P. Pfahler, editors, *Proceedings of the International Conference on Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 176–191. Springer-Verlag, October 1992.

[HGS92] L. J. Hendren, G. R. Gao, and V. C. Sreedhar. ALPHA: A dependence-based intermediate representation for an optimizing/parallelizing C compiler. ACAPS Technical Memo 49, School of Computer Science, McGill University, Montréal, Québec, November 1992.

[HHN92] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, San Francisco, California, June 17–19, 1992. ACM SIGPLAN. *SIGPLAN Notices,* 27(7), July 1992.

[HIA92] W. L. Harrison III and Z Ammarguellat. A program's eye view of miprac. In *5th Workshop on Languages and Compilers for Parallel Computing.* August 1992. Also to appear in LNCS.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[KKP+80] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Analysis and transformation of programs for parallel computation. In *Proceedings of the Fourth International Computer Software and Application Conference*, October 1980.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22–24, 1988. ACM SIGPLAN. *SIGPLAN Notices,* 23(7), July 1988.

[Lan92] William A. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers University, 1992.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Symposium on Programming Language Design and Implementation*, pages 235–248, 1992.

[NG92] Q. Ning and G. Gao. Minimizing loop storage allocation for an argument-fetching dataflow architecture model. In D. Etiemble and J.-C. Syre, editors, *Proceedings of PARLE '92 – Parallel Architectures and Languages Europe*, pages 585–600, Paris, France, June 15–18, 1992. Springer-Verlag, Lecture Notes in Computer Science 605.

[NG93] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, January 10–13, 1993. ACM SIGACT and SIGPLAN.

[NPA92] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[Ols92] R. Olsen. Collective analysis and transformation of loop clusters. Master's thesis, McGill University, May 1992.

[PBJ+91] K. Pingal, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graph: An algebriac approach to program dependencies. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991. ACM SIGACT and SIGPLAN.

[Ros85] David S. Rosenblum. A Methodology for the Design of Ada Transformation Tools in a DIANA Environment. *IEEE Software*, pages 24–33, 1985.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

[SG92] N. Shiri and G. R Gao. Unimodular transformations of monolithic array constructors. ACAPS Technical Memo 43, School of Computer Science, McGill University, Montréal, Québec, June 1992.

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

[SV92] Nematollaah Shiri-Varnaamkhaasti. A design and implementation of unimodular transformations of loops. Master's thesis, McGill University, Montréal, Québec, May 1992.

[Tra86] Kenneth R. Traub. A compiler for the MIT tagged-token dataflow architecture. Master's thesis, Massachusetts Institute of Technology, August 1986. Also published as Technical Report MIT/LCS/TR-370, MIT Laboratory for Computer Science.

[Wil81] R. Wilhelm. Global flow analysis and optimization in the MUG2 compiler generating system. *Program Flow Analysis: Theory and Applications*, 1981.

[Wol91] M. Wolfe. The Tiny loop restructing research tool. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, St. Charles, Illinois, August 12–16, 1991.

[Wol92] M. Wolfe. Beyond induction variables. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, California, June 17–19, 1992. ACM SIGPLAN. *SIGPLAN Notices,* 27(7), July 1992.

[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.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.

A Structured Flow Analysis for the *will_be_read* Sets

The purpose of this appendix is to give the algorithms for computing the *will_be_read* sets for each statement in a SIMPLE tree. These can be computed in two passes over the tree. In the first pass, the algorithm presented in Figure 9 is used to collect the gen and kill sets for each statement in the tree. The proceeds from the bottom leaf nodes upwards through the compositional nodes.

Given that the gen and kill sets have been calculated and stored in the fields s.gen_wbr and s.kill_wbr respectively, the algorithm presented in Figure 10 can be used to compute the *will_be_read* sets for each statement in a top down fashion. Note that no fixed-point iteration is required, and thus these sets can be computed by one pass down through the tree.

```
{ Given a statment s, compute the gen_wbr and kill_wbr bottom-up.
  Store these sets in s.gen_wbr and s.kill_wbr. }

GenKillWbr(s) =
{ case s of
    ASSIGN (lhs,rhs) ⇒
      { s.gen_wbr := R(s,s.SM);   s.kill_wbr := W_d(s,s.SM); }
    SEQ (s1,s2) ⇒
      { GenKillWbr(s1);   GenKillWbr(s2);
        s.gen_wbr := s1.gen_wbr ∪ (s2.gen_wbr - s1.kill_wbr);
        s.kill_wbr := s1.kill_wbr ∪ (s2.kill_wbr - s1.gen_wbr);
      }
    IF (e,s_then,s_else) ⇒
      { GenKillWbr(s_then);   GenKillWbr(s_else);
        s.gen_wbr := s_then.gen_wbr ∪ s_else.gen_wbr ∪ R(e,e.SM);
        s.kill_wbr := s_then.kill_wbr ∩ s_else.kill_wbr;
      }
    WHILE (e,s_body) ⇒
      { GenKillWbr(s_body);
        s.gen_wbr := s_body.gen_wbr ∪ R(e.e.SM);   s.kill_wbr := φ;
      }
    CONTINUE ⇒  { s.gen_wbr := φ;   s.kill_wbr := φ; }
    BREAK ⇒  { s.gen_wbr := φ;   s.kill_wbr := φ; }
    NULLSTMT ⇒  { s.gen_wbr := φ;   s.kill_wbr := φ; }
}
```

Figure 9: Computing the gen and kill sets for the *will_be_read* problem

```
{ Given a statement s, the will_be_written set at the output of s (out), and
  the closest enclosing loop (loop), return the will_be_written set at the
  input of s. As a side-effect, store the will_be_written sets for the exit
  of each IF, and the break-exit and normal-exit of each WHILE.
  Note that the gen_wbr sets for each node should have been already calculated. }

WBR(s,out,loop) =
{ case s of
    ASSIGN (lhs,rhs) ⇒
      return(s.gen_wbr ∪ (out - s.kill_wbr));
    SEQ (s1,s2) ⇒
      { in_2 := WBR(s2,out,loop);
        return(WBR(s1,in_2,loop));
      }
    IF (e,s_then,s_else) ⇒
      { s.exit_wbr := out;
        in_then := WBR(s_then,out,loop);   in_else := WBR(s_else,out,loop);
        return(in_then ∪ in_else ∪ R(e,e.SM));
      }
    WHILE (e,s_body) ⇒
      { s.break_wbr := out;   s.exit_wbr := out ∪ s.gen_wbr;
        in_body := WBR(s_body,s.exit_wbr,s);
        return(in_body ∪ R(e,e.SM));
      }
    CONTINUE ⇒  return(loop.exit_wbr);
    BREAK ⇒  return(loop.break_wbr);
    NULLSTMT ⇒  return(out);
}
```

Figure 10: Computing the will_be_read sets