Sharlit—A tool for building optimizers

Steven W.K. Tjiang
John L. Hennessy
Computer Systems Laboratory
Stanford University

A complex and time-consuming function of a modern compiler is global optimization. Unlike other functions of a compiler such as parsing and code generation which examine only one statement or one basic block at a time, optimizers are much larger in scope, examining and changing large portions of a program all at once. The larger scope means optimizers must derive and use complex relationships among widely separated parts of a program. To get the best code, optimizers must perform many program transformations. Each of these transformations makes its own particular demands on the internal representation of programs; each can interact with and depend on the others in different ways. This makes optimizers large and complex.

Despite their complexity, few tools exist to help in building optimizers. This is in stark contrast with other components of the compiler where years of experience have culminated in tools that reduce the effort required to build these components. For example, parser generators are used to build front-ends, and peephole optimizers and tree matchers are used to build code generators.

This paper presents Sharlit, a tool to support the construction of modular and extensible global optimizers. We will show how Sharlit helps in constructing data-flow analyzers and the transformations that use data-flow analysis information: both are major components of any optimizer.

Sharlit is implemented in C++ and uses C++ in the same way that YACC uses C. Thus we assume the reader has some familiarity with C++ [9].

1 Sharlit and Optimizers

Traditionally optimizers are organized as a fixed sequence of phases. Before optimization, a front-end translates a source program into some intermediate form. The optimizer reads the program one procedure at a time, and converts the procedure into an internal representation (IR), typically a control-flow graph. Each optimizer phase analyzes and transforms the flow graph. When all the phase are finished, the optimizer writes out the IR as an improved program.

To use Sharlit, a compiler writer programs each optimization phase as a Sharlit specification. From a specification, Sharlit automatically generates an optimization phase that consists of two steps: a data-flow analysis (DFA) of the flow graph collects data-flow information, and a flow graph traversal uses the collected information to perform optimizations. In the literature and in practice this two-step organization is often used—It is used by Aho, Sethi and Ullman[1] in their descriptions of optimizations, and by Chow[4] in his implemention of optimizations.

Because data-flow analysis consumes a lot of time and space, compiler writers are concerned with implementing the analyzers efficiently. This concern has led to two deficiencies in existing optimizers. First, compiler writers lower time and space requirements by limiting the kinds of DFA solved. Unfortunately, such limits restrict the kinds of code optimizations that can be done, a restriction contrary to our goal of building an extensible optimizer. Second, optimizers are designed using a control-flow graph with two levels: a global level consisting of a flow graph whose nodes are basic blocks, and a local level consisting of the instructions that form the basic block. A two-level graph reduces the number of nodes and the size of the data-flow information required during DFA. Unfortunately, the two-level graph obscures data-flow analysis and code transformation in several ways.

The major drawback of the two-level approach is that it complicates the programming of DFAs. Each analyzer must have both a local analysis phase and a propagation phase. Local analysis computes the data-flow effect of a basic block by composing the data-flow effects of its instructions. The propagator uses the global DFA solution at each basic block to compute the local solution at each of its instructions. These two phases are custom built by the compiler writer.

A second drawback is that compiler writers must understand the DFA and all the data structures at two levels before they can build prototype solvers for the DFAS. Given that any DFA can be understood and solved knowing only the data-flow effect of each instruction, the two-level structure is a complicating implementation detail that hinders experimentation of different optimizations.

A third drawback is that the two-level structure encourages non-modularity. To minimize the number of CFG traversals, compiler writers often combine the local analysis of several DFA problems in one traversal, and the corresponding propagators in another traversal. Hence, the compiler writer must shoulder the burden of orchestrating these phases and their side-effects so they will perform as desired.

Instead of limiting the analyses or using basic blocks, Sharlit lets compiler writers program global analyses and optimizations in a modular fashion with the following abstractions:

1. Flow graphs: specified by the compiler writer. The nodes of the graph can be basic blocks or individual IR instructions. We advocate the latter.

2. Flow values: values that flow through the graph. The solution of the DFA is one flow value for each flow-graph node. For traditional DFA problems, the flow values are bit vectors that represent, for example, sets of expressions.

3. Flow functions: functions that represent the effect of flow graph nodes and paths on the flow values; these are specified as input to Sharlit.

4. Action routines: routines that use the DFA solution to perform program optimizations.

5. Path simplification rules: rules that show how to combine flow functions into other flow functions. There are four kinds of rules: **create**, **absorb**, **plus**, and **star.**

With these abstractions, a compiler writer can implement a wide range of program analysis that form the core of optimizers, from the traditional bit-vector based DFAs to ones using sophisticated symbolic analysis. A compiler writers can pose

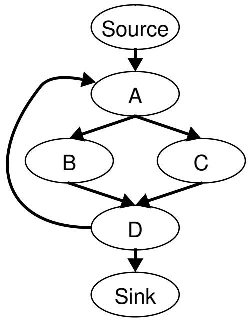


Figure 1. A small graph.

the DFA problems as if each node of the CFG is an instruction. The responsibility for generating space- and time-efficient solvers for the DFAs rests with Sharlit. The generated solvers are efficient because they make the graph smaller with a technique called *path simplification*.

1.1 Path simplification

Path simplification is based on a version of Tarjan's algorithm to solve the single-source path problem[11]. Tarjan's algorithm finds a *path expression* representing all paths from the source to every node in a graph. Path expressions are regular expressions built from node labels and the operators: ·, +, and *. For example, in Figure 1, the path expression from the exit of the source node to the exit of node D is $A((B+C)DA)^*(B+C)D$. Sharlit uses path expressions in data-flow analysis by interpreting the node labels as flow functions, path concatenation(·) as function composition, path addition(+) as taking the *meet* of functions, and Kleene-star (*) of a path p as $p + p \cdot p + p \cdot p \cdot p + \ldots$. The compiler writer tells Sharlit how to perform these operations with path simplification rules.

Path simplification is versatile: the rules can describe a wide range of DFA solvers, from simple iterative ones, to solvers using local analysis, to sophisticated solvers that use interval analysis. When confronted with irreducible graphs or incomplete sets of rules, path simplification degrades gracefully—it is still able to simplify significant portions of the graph.

2 Writing DFAs with Sharlit

This section provides an overview of Sharlit and path simplification, and shows how one can use Sharlit to write DFAs. We demonstrate Sharlit with three examples, all solving the *available expression problem* (AVEP)[1]. An expression is available at a node n if every path from the source to n has evaluated the expression and the value of the expression remains the same if reevaluated at n. Determining available expressions is important for code motion [4].

These three examples show Sharlit's ability to apply a variety of solution techniques and solve versions of AVEP using progressively more sophisticated and potentially faster solution techniques.

To underscore the extensibility of Sharlit-generated analyzers, we build each example from the previous one by adding new flow functions and new path simplification rules. The first example is a prototype solver that uses iteration. In this example, each IR instruction is a node in the flow graph, making the prototype inefficient. In the second example, we extend the first example with path simplfication rules, resulting in a DFA solver which performs local analysis, and reducing the work needed during iteration. In the third example, we enhance the solver with more rules—**plus** and **star** rules—to create a solver that uses interval analysis.

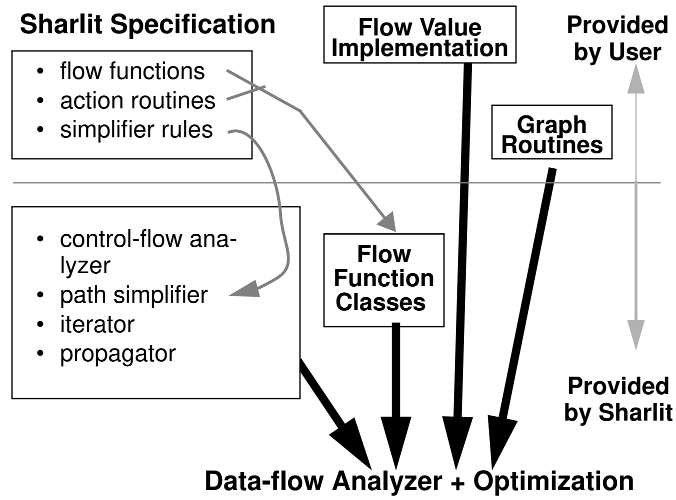


Figure 2. Structure of a Sharlit-based optimizer phase.

2.1 Preliminaries

Figure 2 shows how an internal phase of an optimizer is created using Sharlit. Each automatically generated data-flow analyzer consists of four major components: control-flow analyzer, path simplifier, iterator, and propagator. The control-flow analyzer discovers the loop structure of the control-flow graph and summarizes the structure (control-flow analysis is done once for several DFAs) [8, 10]. The path simplifier, generated from the rules in the description, uses the control-flow information to eliminate nodes from the flow graph, creating a reduced graph that is easier to solve. The iterator makes one or more phases over the reduced graph, calling flow functions until a solution (consisting of flow values) to the data-flow problem is found. The propagator computes the solution at nodes eliminated by the path simplifier. In the process, the propagator calls user-supplied action routines for each node, routines that use the solution to perform program optimizations.

All the components are compiled and linked (black arrows) into an internal phase of the optimizer. The flow value is implemented as a C++ class by the compiler writer. The compiler writer can implement the flow graph in any way, but must provide graph routines that let the generated data-flow analyzer *view* the flow graph as a doubly-linked directed graph with edges stored as adjacency vectors.

In our examples, the IR is quad based and each IR instruction is implemented using the following structure:

```
class IR_instruction {
    ...
public:
    int kind;
    IR_variable dst,src1,src2;
    Expression expr_no;
    ...
};
```

Only those fields relevant to our analysis are shown.

The IR has several types of instructions, distinguishable by the value of the field `kind.` Since expressions are built recursively from arithmetic operations and variables, we are interested in the following values of `kind`:

- `IR_op`: performs an arithmetic operation on `src1` and `src2`, storing the result into the variable `dst`.
- `IR_ldc`: loads a constant into the variable `dst`.
- `IR_kill:` modifies the variable `dst`.
- `IR_initial:` is the instruction type used to mark the source node of the graph.
- `IR_others:` doesn't affect the contents of any of the variables.

We assume that value numbering [1] has been performed—possibly in a previous phase—on the instructions. For each IR instruction, value numbering has set the field `expr_no` to identify the symbolic expression that is being computed ty the instruction. For each variable, value numbering has computed the set of expressions *killed* by a write to the variable (See `kill_sets` below).

2.2 Example 1: iterative analysis

Figure 3 shows a Sharlit specification for an iterative solver for AVEP. It is the simplest kind of specification possible, and contains the minimal ingredients necessary for a working DFA solver.

The braces in the specification delimit C++ code, which Sharlit inserts into a C++ code framework to build a solver, just as YACC [ ] inserts user-supplied C code into a C code framework to build a parser. Within the C++ parts, variables that begin with an underscore are special. Playing the same role as the $ variables of YACC, they refer to variables with which the Sharlit generated parts of the solver communicate with the user-supplied code.

The major parts of the specifications are described below.

1. Data specific to the DFA problem

In Figure 3(A), the keyword **solver** begins a declaration for variables that are specific to DFA problem. These variables will be used by user-supplied code in other parts of the Sharlit specification. For example, the flow functions refer to the kill information with the expression `_P->kill_sets`.

2. Flow graph nodes

In Figure 3(A), the **node_base** line tells Sharlit that each node in the flow graph is an object of type `IR_instruction`. Sharlit doesn't need to know the internals of the type to generate an analyzer but it does need the name of the type to declare variables.

3. Flow values:

The other keywords in Figure 3(A)—**value_base**, **new_value**, **copy_value**, and **meet**—tell Sharlit about the type of flow values, and the operations on flow values, operations called in the process of solving the DFA problem. The **value_base** line indicate to Sharlit the name of the type with with flow values should be declared. The other keywords define operations. To allocate a new flow value, the DFA solver calls the C++ code following **new_value**. To copy a flow value (`*_SRC`) into another (`*_DST`), the solver calls the code following **copy_value**. To take the meet of a set of flow values—pointed to by an array of pointers `_SRCS`—the solver calls the code following **meet**.

The **copy_value** operation has another important duty—it tells the solver whether the source flow value (`*_SRC`) was equal to the destination flow value (`*_DST`) before copying. Returning an indication of whether the destination changed with the copy is used as a termination test within the solver.

```
Avail: =
solver: {
    Expr_set *kill_sets[];
};
node_base = IR_instruction;
value_base = Expr_set;
new_value: {
    Expr_set *v=new Expr_set;
    v->mk_universal();
    return v;
};
copy_value: {
    if(*_SRC == *_DST)
        return 0;
    *_DST = *_SRC;
    return 1;
};
meet: {
    int i;
    _DST->mk_universal();
    for(i=0;i<_NSRCS;i++)
        *_DST &= *_SRCS[i];
};
```

(A)

```
AV_expression: flow {
    *_V += _N->expr_no;
    *_V -=_P->kill_sets[_N->dst];
};
AV_initial: flow{
    _V->mk_empty();
};
AV_kill:flow {
    *_V-=_P->kill_sets[_N->dst];
};
AV_identity: flow {};
```

(B)

```
flow_map: {
    switch(*_N->kind){
    case IR_op:
    case IR_ldc:
        return new AV_expression;
    case IR_kill:
        return new AV_kill;
    case IR_initial:
        return new AV_initial;
    case IR_others:
        return AV_identity;
    }
};
end;
```

(C)

Figure 3. Iterative solver for available expression.

```
iterate(L)
{
    Let L = [u_1, u_2, ..., u_n] be a df-order of the CFG.
    Let O(u) be the output flow variable associated with u
    _V ← new_value()
    c ← true
    (flow(u_1))(_V)
    copy_value(O(u_1), _V)
    while c is true {
        c ← false
        for i=2 to s {
            if ancestor(u_i) ≠ u_i {
                copy_value(_V, O(ancestor(u_i)))
                (new_flow(u_i))(_V)
            } else if has_meet(u_i) {
                Let {w_1, w_2, ..., w_a} be predecessors of u_i
                meet(_V, [O(w_1), O(w_2), ..., O(w_a)])
                (flow(u_i))(_V)
            } else
                (flow(u_i))(_V)
            if O(u_i) exists {
                c ← c| copy_value(O(u_i), _V)
            }
        }
    }
    solved(u_i) ← true  for i = 1, 2, ..., s
}
```

Figure 4. The iterator—solution by iteration.

4. *Flow functions:*

   Figure 3(B) shows the flow functions. In each of the flow functions, the incoming flow value is pointed to by the variable `_V`. The code following the keyword **flow** computes an output flow value, leaving a pointer to it in `_V`. In available expressions, and many other DFAs, the code can modify the incoming value directly into the outgoing value. For large, complex values, modification-in-place avoids unnecessary copies.

   There is a flow function corresponding to each instruction type discussed above.

   - `AV_expression` corresponds to `IR_ldc` and `IR_op` instructions. This function inserts `expr_no`, the expression number being generated by the current node `_N`, into the incoming flow value, then removes the killed expressions, those whose value may change when the variable `_N->dst` is modified.
   - `AV_kill` corresponds to the instruction `IR_kill`. This function removes all the killed expressions from the flow value.
   - `AV_initial` corresponds to the initial node `IR_initial`. This function initializes its output value to the empty set.
   - `AV_identity` corresponds to the `IR_others` instruction. This function uses its incoming flow value as its outgoing value.

5. *Flow map:*

   We must provide the above mapping between instructions and flow functions. Part C is a flow map function called from control-flow analysis to establish the correspondence between flow functions and instructions.

Figure 4 shows the iteration procedure that Sharlit uses to solve DFA problems. The procedure takes as its argument, a flow graph represented as a list of nodes ordered in the depth-first order [6] computed during control-flow analysis. The ordering is used because it is easy to compute and it has fast convergence for bit-vector problems [6].

The procedure **iterate** refers to the following information about each node u:

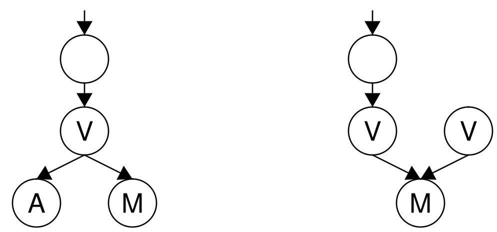
- flow(u) is the flow function of the node u, obtained by calling the **flow_map** code provided by the user.
- ancestor(u) and new_flow(u) are computed during path simplification. Since this example has no rules, the condition ancestor$(u_i) \neq u_i$ is always false. Thus, for now, we can ignore the first if statement in the loop.
- $O(u)$ is the flow variable that holds the output flow value of node u. To save space, not every node will have a variable (see Section 2.3).
- has_meet(u) indicates whether a meet operation is necessary to compute the input flow value of u.

In the *one-instruction-one-node* flow graph, if Sharlit kept a copy of data-flow information—a flow variable—at every node, the data-flow analyzers would use a prohibitive amount of storage. Thus Sharlit needs to avoid keeping flow variabes at every node—the last two items above help to do just that.

2.3 Eliminating flow variables I

In a Sharlit specification, a compiler writer cannot write code that references flow variables directly. In particular, flow functions do not reference flow variables. Instead they modify the variable `_V`. At the beginning of a flow function, `_V` contains the input flow value of the node u with which the flow function is associated. The flow function modifies `_V` directly to arrive at the output flow value of u.

Denying direct access to flow variables frees Sharlit to decide which flow variables to eliminate. Sharlit allocates flow variables at three kinds of nodes. We show two of the three below. The third kind will be discussed later in Section 2.7.



Nodes labelled V need flow variables and nodes labelled M need a meet operation or a copy operation to compute their input flow values. In the graph on the left, the node V needs an output flow variable because one of its successors, the node M, does not follow it in the depth-first order. Although `_V` as computed by V can be used directly as input to node A, V's flow value must be stored until node M is referenced during iteration. In the graph on the right, the nodes labelled V need variables because the meet operation that computes the input flow value for node M will refer to the output flow values of the two V's.

In general, a node u requires a flow variable $O(u)$ when u has more than one successor, or when u is the predecessor of a node that requires a meet operation to compute its input flow value.

Similarly, has_meet(w) is true for a node w—indicating that w requires a meet operation to compute its input flow value—when w is the successor of a node which has more than one successor, or when w has more than one predecessor. In the left graph above, M needs a meet operation to copy the flow variable of its predecessor. In the right graph, M needs a meet operation to combine the flow variables of its predecessors.

Sharlit's approach to maintaining flow variables differs from the traditional scheme that keeps a flow variable at the input and output of each node or basic block. Instead, Sharlit keeps an output flow variable at the end of a linear sequence of instructions. Consider two nodes u and w such that u's unique successor is w and w's unique predecessor is u. After calling the flow function for u, _V contains the output flow value of u, which is equal to the input flow value of w. Hence, _V can be passed directly to the flow function of w as its input. Since no other nodes need the value of $O(u)$, it can be eliminated.

2.4 Example 2: local analysis

Even though the Sharlit example in Figure 3 saves space by eliminating many flow variables, it must still visit every IR instruction on every iteration, using a lot of time. Our second version of AVEP shows how we can reduce the number of nodes visited by adding path simplification rules to Figure 3.

Many optimizers reduce the number of nodes by using a flow graph where the nodes are basic blocks. By adding path simplification rules to a Sharlit specification, a compiler writer can get the same reduction afforded by basic blocks, but without the need for ad hoc local analysis and propagation phases. In the following example, we show how rules can shrink the flow graph so that it contains only those nodes that form the boundaries of *extended basic block.*

To add local analysis to Figure 3, we add the code in Figure 5. Part A declares a path function AV_bb that represents the data-flow effect of a sequence of instructions. A path function is a special kind of flow function that represents the data-flow effect of a flow-graph path. In the flow function AV_bb, two local variables, defs and kills, are used in computing the effect of a path:

$$\text{Output} = (\text{Input} - \text{kills}) \cup \text{defs}\,.$$

The simplification rules compute defs and kills as a path is being built.

Two kinds of rules—**create** and **absorb** rules—follow the definition of AV_bb; Section 2.6 will show more kinds. Create rules generate identity flow functions for the empty basic block or empty path. Absorb rules grow this empty path by *absorbing* the flow function that immediate follows the path. Figure 5 shows four **absorb** rules, one for each of the flow functions given in Figure 3, and one for the path function AV_bb.

We summarize what the path simplifier (Section 2.7) does for extended basic blocks in Figure 6. By applying a series of **create** and **absorb** rules, the simplifier computes instances of the flow function AV_bb that relates the flow value leaving node A to the value leaving the leaves. Thus the nodes B and C are no longer necessary and are eliminated from consideration by the iterator.

For example, say the flow functions at node C and F are

```
....
AV_bb: local {
    Expr_set defs;
    Expr_set kills;
}
path {
    _V -= kills;
    _V += defs;
};
....
simplifier
(:AV_bb) create = { };
(b:AV_bb) absorb (l:AV_identity) = {};
(b:AV_bb) absorb (l:AV_expression,N) = {
    Expr_set*kill_set=&_P->kills_set[N->dst];
    b->kills += *kill_set;
    b->defs.add(N->expr_no);
    b->defs -= *kill_set;
};
(b:AV_bb) absorb (l:AV_kill,N) = {
    Expr_set*kill_set=&_P->kills_set[N->dst];
    b->kills += *kill_set;
    b->defs -= *kill_set;
};
(b:AV_bb) absorb (l:AV_bb) = {
    b->defs -= l->kills;
    b->defs += l->defs;
    b->kills += l->kills;
};
```

A

Figure 5.The sharlit specification for available expressions augmented with local analysis.

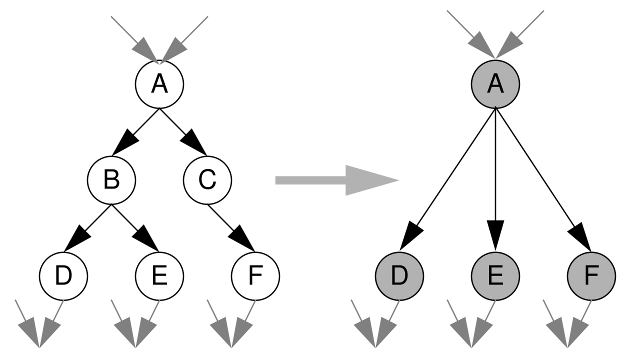


Figure 6. Simplifying extended basic blocks.

propagate(L)
{
  Let $L = [u_1, u_2, \ldots, u_n]$ be a df-order of the CFG.
  _V $\leftarrow$ new_value( )
  for $i = 2$ to s do {
    if has_meet(u_i) {
      Let $\{v_1, v_2, \ldots, v_a\}$ be predecessors of u_i
      meet(_V, $[O(v_1), O(v_2), \ldots, O(v_a)]$)
    }
    (in(u_i))(_V)
    if solved(u_i)
      copy_value(_V, $O(u_i)$)
    else
      (flow(u_i))(_V)
    (out(u_i))(_V)
    if $O(u_i)$ exists and not solved(u_i)
      copy_value($O(u_i)$, _V)
  }
}

Figure 7. Propagation—computing solutions on the whole flow graph using the solutions from the reduced graph.

both of type `AV_expression`. To find the path function from the exit of A to the exit of F, the path simplifier uses the create rule to create a new path function. That path function *absorbs* the `AV_expression` function of C, using the rule

`(b:AV_bb)` **absorb** `(l:AV_expression,N)`

When the code part of this rule executes, the variable `b` points to the path function `AV_bb`; the variable `l` to `AV_expression`; and the variable `N` to the node C. The code updates the path function so that it represents the data-flow effect of the original path function composed with the flow function of C. Another application of the above rule will compose the above path with the flow function of node F.

2.5 Propagation

Path simplification builds a smaller graph that consists only the nodes at the boundaries of extended basic blocks, such as nodes A, D, E and F in Figure 6. It sets the *ancestor* of the leaves to the node that heads their corresponding block. For nodes D, E and F, that ancestor is A. It computes for each leaf u a new flow function new_flow(u) to represent the data-flow effect from ancestor(u) to u. The reduced graph, represented as a list, is given to **iterate**. The test for ancestor in **iterate** now has a purpose—to call new_flow(u) instead of flow(u).

Once iteration has found a solution to the reduced graph, the routine **propagate** in Figure 7 is called to compute the flow values of nodes in the larger flow graph.

```
name: local {
    local variables
}
flow {
    ....
}
in {
    code that uses flow value at entry of node
}
out {
    code that uses flow value at exit of node
};
```

Figure 8. Flow function declaration with in and out functions.

As **propagate** computes the flow values, it calls in(u) and out(u), action routines that can use the flow values at the input and output of u to perform optimizations. The compiler writer specifies these action routines together with a flow function, as shown in Figure 8.

2.6 Example 3: interval analysis

The advantage of iteration, with or without local analysis, is that it works on all graphs, even irreducible graphs. But for reducible graphs, we can avoid iteration, potentially speeding up DFA and allowing us to solve those DFAs which may not converge with iteration.

The previous example (Section 2.4) uses path simplification to perform local analysis, building a reduced graph with fewer nodes. We can use path simplification to perform interval analysis, building a reduced graph in which there are no loop back-edges. For an reducible flow graph, Sharlit can remove all back-edges, and the DFA problem can be solved by making one pass of the reduced graph, followed by a propagation step.

To remove a back-edge, path simplification summarizes the data-flow effect of a loop and *replaces* the loop with a tree consisting of some of the nodes of the loop. The flow functions of the nodes are replaced. This replacement occurs with the innermost loop, proceeds outwards, and ends with the entire graph.

For example, in graph of Figure 9(1), a tree replaces the innermost loop, whose header is the node D. The result in Figure 9(2) has the single tree node highlighted. The node E is eliminated[1], just as the local analysis of Section 2.4 would. The flow function of F is replaced by a new flow function f which is equal to EF. The graph Figure 9(3) shows the resulting graph after processing the loop headed by A. Here, the new flow functions are d=D(fD)* and g=(BC+df)G. Replacing A's loop eliminated the back-edge entering into D—its data-flow effect summarized into the flow function f. Finally, in Figure 9(4), the outermost interval headed by Source is replaced and that removes the back-edge entering A.

In the example, all the back-edges were removed and the

1. The nodes are actually only marked and eliminated in a later step, but we show eliminate them eliminated here to simplify the explanation.

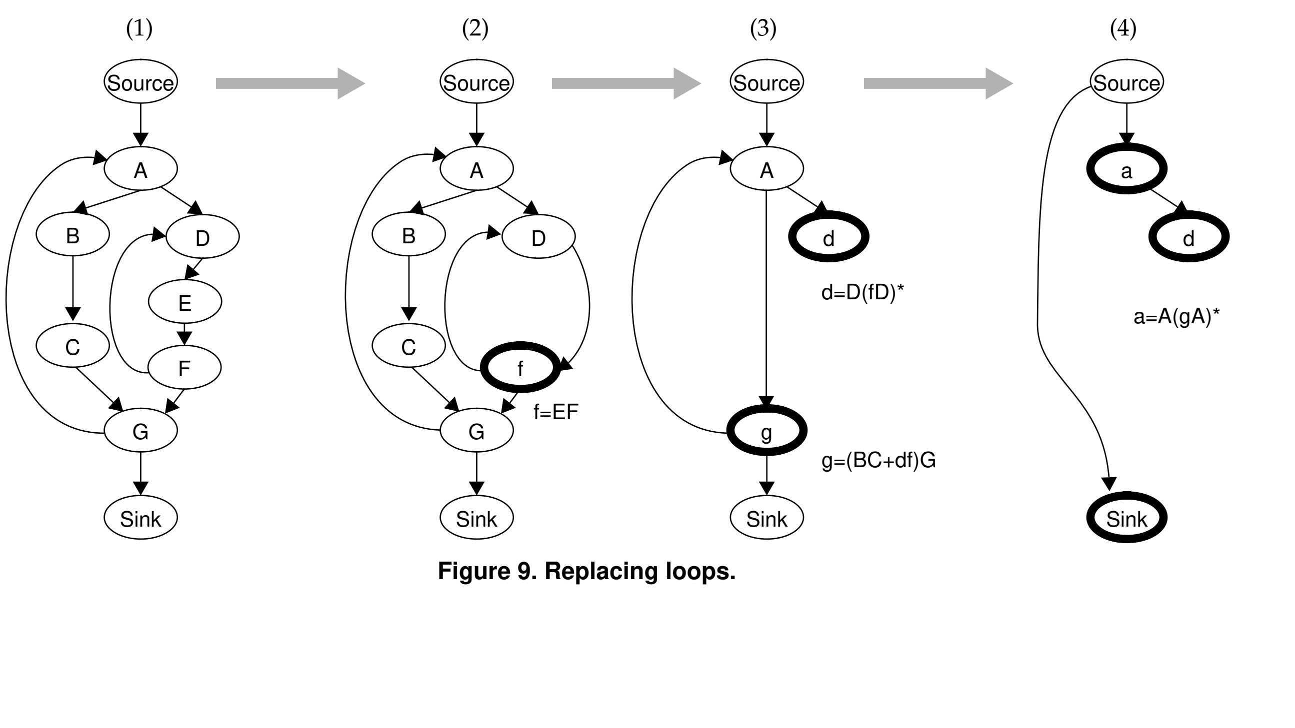


Figure 9. Replacing loops.

```
(b1: AV_bb) plus (b2: AV_bb) = {
    Expression_set sum1, sum2;
    sum1.universal(_P->etable);
    sum2.universal(_P->etable);
    sum1 -= b2->defs;
    sum1 *= b2->kills;
    sum1 += b1->kills;
    sum2 -= b1->defs;
    sum2 *= b1->kills;
    sum2 += b2->kills;
    b1->kills = sum1;
    b1->kills *= sum2;
    b1->defs *= b2->defs;
};
(b:AV_bb) star = {
    b->kills -= b->defs;
    b->defs.clear();
};
```

Figure 10. Enhancing solver to use interval analysis.

cfa(G) /* Control-flow analysis */
{
    Let G be the control-flow graph.
    Perform a depth-first traversal to discover the loop structure of G.
    The output are two lists: L which is a df-ordering of the G, and a list of lists $[L_1, \ldots, L_K]$. Each list is a df-ordering of the nodes within a loop body. The lists are mutually exclusive—all pairwise intersections are empty. The lists satisfies the condition
    $\mathrm{depth}(L_1) \geq \mathrm{depth}(L_2) \geq \ldots \geq \mathrm{depth}(L_K)$.
    Also computes for each node u the loop_head(u), the node which heads the loop in which u belongs.
}
simplify(L, $[L_1, \ldots, L_K]$)
{
    Let L and $[L_1, \ldots, L_K]$ be the result of control-flow analysis.
    for i=1 to K
        simplify_loop(L_i)
    return **reduce**(L)
}
reduce(L)
{
    Let $L = [u_1, u_2, \ldots, u_s]$ be a reverse post-ordering of the flow graph
    Let M be an empty list of nodes.
    for $i = s$ to 2 {
        if u_i is a loop header { Prepend u_i to M. }
        else if keep(u_i) { Prepend u_i to M }
        else continue
        if ancestor$(u_i) \neq u_i$ {
            keep(ancestor(u_i)) $\leftarrow$ true
        } else {
            keep(v) $\leftarrow$ true for all predecessors v of u_i.
        }
    }
    Prepend u_1 to M
    return M
}

Figure 11. Pseudo-code for Sharlit's simplifier.

simplify_loop(L)
{
    let $L = [u_1, u_2, \ldots, u_l]$ a df-order of a loop.
    for i=1 to l {
        Let f = flow(u)
    /* Classify predecessors */
        Let $V = \{v_1, v_2, \ldots, v_a\}$ be predecessors such that u_i = loop_head(v_j) and $v_j \to u_i$ is a back-edge for $j = 1, 2, \ldots, a$.
        Let $\{w_1, w_2, \ldots, w_b\}$ be the other predecessors of u_i
    /* Compute the path expression through back-edges */
        if $V \neq \varnothing$ {
            $q_j \leftarrow$ eval(v_j, u_i) for $j = 1, 2, \ldots, a$
            $q \leftarrow ((q_1 + q_2 + \ldots + q_a) f)$ /* form path expression */
        } else
            $q \leftarrow f_{Id}$ /* the identity flow function */
    /* Compute the path expression from a dominator to u*/
        Call eval(w_j) for $j = 1, 2, \ldots, b$
    /* Find common ancestor */
        if ancestor(w_1) = ancestor(w_2) = … = ancestor(w_b) {
            Let a = ancestor(w_j)
            Let p_j = new_flow(w_j) for $j = 1, 2, \ldots, b$
            $p \leftarrow (p_1 + p_2 + \ldots + p_b) f$ /* form path expression */
        } else if we can't find a then
            $p \leftarrow f_{\perp}$
    /* Compute new_flow */
        $p \leftarrow pq^*$
        if $p \neq f_{\perp}$ {
            ancestor$(u_i) \leftarrow a$
            new_flow$(u_i) \leftarrow p$
        } else {
            keep(u_i) = true
            keep(v_j) = true for $j = 1, 2, \ldots, a$
        }
    }
}
eval(u)
{
    if ancestor$(u) \neq u$ {
        path_compress(u)
        return new_flow(u)
    }else
        return f_{Id}
}
eval(u,h)
{
    if ancestor$(u) \neq u$
        path_compress(u)
    if $u = h$ { return f_{Id} }
    if ancestor$(u) = h$ { return new_flow(u) }
    return $f_{\perp}$
}
path_compress(u)
{
    if ancestor(ancestor(u)) $\neq u$ {
        path_compress(ancestor(u))
        Let f = new_flow(ancestor(u)) $\cdot$ flow(u)
        if $f \neq f_{\perp}$ {
            new_flow$(u) \leftarrow f$
            ancestor$(u) \leftarrow$ ancestor(ancestor(u))
        }
    }
}

Figure 12. Pseudo-code for Sharlit's simplifier (continued).

DFA problem can be solved by making one iteration over the much smaller, acyclic graph of Figure 9(4), followed by a propagation step that computes the solution at all the other nodes from the solution at A and D.

To add interval analysis to Figure 5, we add two types of rules—**plus** rules and a **star** rule—shown in Figure 10. These rules tell Sharlit how to take the meet of two alternative paths that share the same end-points; and how to take the Kleene-star of a path. The last rule is necessary for removing back-edges.

2.7 Eliminating flow variables II

After path simplification in the examples of Section 2.4 and Section 2.6, Sharlit needs to store variables at the nodes that are ancestors to others. This is necessary because the procedure **iterate**—when it visits a node u with an ancestor—uses the output flow value of the ancestor of u. In general **iterate** may not visit u immediately after $\text{ancestor}(u)$, and $\text{ancestor}(u)$ may be ancestors to other nodes. Therefore a flow variable is needed at ancestors to hold its output flow value.

3 The Path Simplifier

Sharlit's simplifier is shown in Figure 11 and Figure 12, and is based on Tarjan's algorithm for the single-source path problem. The algorithm computes for each node a path expression representing all paths from the source to that node. Sharlit's path simplifier uses this path expression computation to guide the composing, meeting and taking the Kleene-star of flow functions. In this section, we discuss the four key ideas behind the operation of the simplifier. We explain the simplifier in the context of computing path expressions; Section 3.1 will show how that computation corresponds to calls of the rules provided in a Sharlit specification.

First, changing the graph directly to replace loops is expensive. Instead a tree edge (a, u) is represented by setting $\text{ancestor}(u)$ to a. In Figure 9(2) for example, $\text{ancestor}(\mathsf{E})$ and $\text{ancestor}(\mathsf{F})$ are both D. Also, the new flow functions computed when replacing the loop are stored in $\text{new\_flow}(u)$. We can view the tree edge (a, u) replacing all the flow graph edges of u.

Second, when forming path expression the simplifier can fail because it is unable to find a sequence of simplification rules that can simplify the path expression into a single flow function. When that occurs, the result of the path expression formation is $f_{\perp}$. When that occurs for a node u, its ancestor remains u, and u becomes eligible to become ancestors of other nodes (See Section 3.2).

Third, the simplifier builds the ancestor tree by visiting each loop in depth-first order, and visits loops from the innermost to the outermost. This ensures that when it comes time to compute the path expression for a node u, the simplifier has already computed the path expressions for the predecessors of u within the loop. For example, in Figure 9, the visit order is [E, F] for the loop headed by D, [B,C,D,G] for the loop headed by A, and [A,Sink] for the outermost interval headed by Source.

Fourth, tree nodes—those nodes u whose ancestor is not u—have their path expressions computed on demand. This computation occurs when the procedure **eval** is called from **simplify_loop**. In Figure 9(2) for example, the path expression between A and f is computed during calls to eval when the simplifier is visiting G.

We can see which path expressions are computed in the main body of **simplify_loop** and **eval**. A chronology of the order in which path expressions are computed follows:

- path expression E for the path from D to E.
- EF for path from D to F.
- B for path from A to B.
- BC for path from A to C.
- When visiting D, the simplifier calls eval(F, D) which returns f=EF. Hence, the path expression computed for D is d=D(fD)*.
- When visiting G, the simplifier calls eval(F) which yields df, the path expression from A to F. Hence the path expression for G is g=(BC+df)G.
- A(gA)* for the path from Source to A, with a call of eval(G, A).
- Finally A(gA)*gSink for the node Sink, with a call of eval(G).

3.1 Path simplification rules

Path simplification builds path expressions and in the process computes flow functions using composition, join, and Kleene-star of other flow functions. These operations are directed by the compiler writer with path simplification rules.

An obvious way to tell the analyzer how to combine flow functions is to use the *times table* approach. For example, when the analyzer needs to compose two flow functions, it would index a special table with the two flow functions to find what actions are necessary to do the composition. Although tables are simple, specifying them directly is tedious: A complete specification of the composition table for M flow functions would require M^2 entries. Instead Sharlit generate a complete table using only $M + 1$ rules and a new kind of flow function: a path function.

During iteration (the solution of the DFA problem), a path function behaves just like any other flow function, except that it represents the data-flow effect of a path computed by the path simplifier. During path simplification, path functions are treated differently for efficiency reasons—they are modified directly or reused.

Below we show the five kinds of rules and describe how they work conceptually. Each rule has a code part that when executed computes a new path function by modifying the path function specified in the left-hand side of the rule.

1. *p* create: Create a new path *p* whose data-flow effect is the identity. This rule is used in the situations depicted in Figure 13. The analyzer inserts identify flow functions with the intention of extending that path (see below). This insertion does not change the data-flow problem—the solution is the same as the solution without the insertion of *p*.

2. *p* create *f*: Create a new path *p* whose data-flow effect is the same as the flow function *f* (see Figure 14). Again this replacement will not change the solution of the data-flow problem. This replacement is used by the analyzer to create paths in the graph that begins with nodes with flow function *f*.

3. p absorbs f: Extend a path p so that it has the same data-flow effect as the old p composed with *f*, that is modify p so that it represents the data-flow effect $p \cdot f$ (Figure 15).

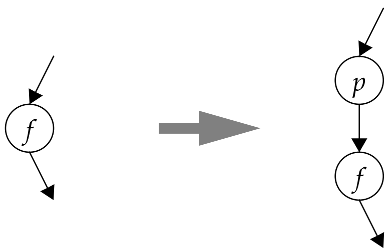


Figure 13. Applying the "*p* create" rule.

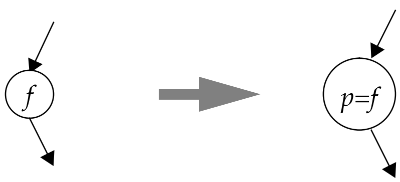


Figure 14. Applying the "*p* create *f*" rule

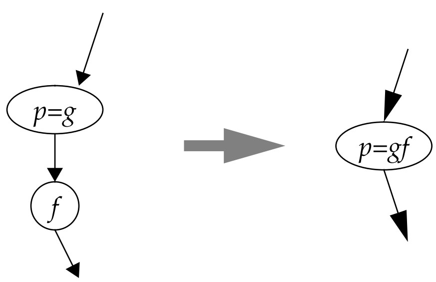


Figure 15. Applying the "*p* absorb *f*" rule

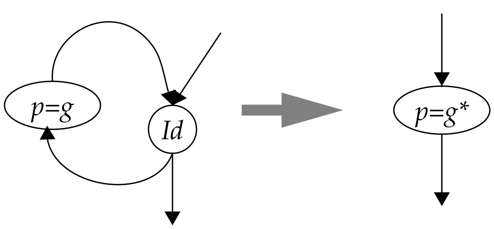


Figure 16. Apply the "*p* star" rule
The node labelled *Id* is an identity node. They are not actually present in the graph but are present for expository reasons only.

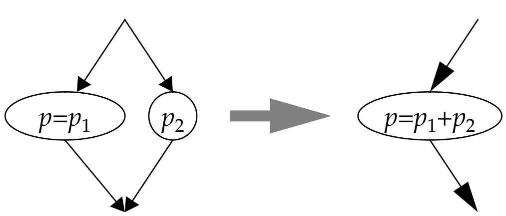


Figure 17. Applying the "p_1 plus p_2" rule
This rule only applies when the two paths share common endpoints. This condition is enforce in the path simplifier because the ancestors of the nodes must be the same

4. p star: Replace the path p with a path that represents the data-flow effect of p^* . This rule, together with other rules, is used to remove back-edges (See Figure 16).

5. p_1 plus p_2: Replace the path p_1 with a path that represents the data-flow effect of the path $p_1 + p_2$ (Figure 17).

Relating this back to Figure 11, we can see that the path computation steps are application of these rules. So computing a path expression, such as $q \leftarrow ((q_1 + q_2 + \ldots + q_a)f)$ means applying a sequence of path simplification rules.

3.2 Incomplete path simplification

Sharlit's path simplification differs from Tarjan's algorithm in how it handles irreducibilities and incomplete rule sets. For both situations, path simplification can still reduce significant parts of the graph, although it may not be able to remove all back-edges. Tarjan's algorithm works only on reducible flow graphs.

We can characterize **simplify** by saying that it tries to find a flow function from a dominator of u whose path to u can be simplified into one flow function. Simplification can fail to find such a flow function when the simplification rules are incomplete—they cannot simplify some combination of flow functions—or when the graph is irreducible. When there are no meet rules but the rules are otherwise complete, this dominator will be the entry node of the extended basic block. For irreducible graphs, **simplify** fails because a common ancestor cannot be found (see Figure 11). Even then, the simplifier can still succeed on the reducible parts of the graph.

The simplifier degrades gracefully, reverting to local analysis where it can, or even to just partial local analysis (only parts of extended basic blocks are simplified) when the rules are extremely incomplete. This gradual degradation makes it easy to test and develop DFA incrementally.

3.3 Using rules in path computation

The procedures **simplify**, **eval**, and **path_compress** build path expressions abstractly using operators ·, +, and *. We show how Sharlit turns this process of computing path expressions into invocations of rules. We define three functions corresponding to the path operators:

1. compose(f, g):
   a) If $f = f_\perp$ or $g = f_\perp$ then return $f_\perp$ to indicate failure.
   b) If $f = f_{\mathrm{Id}}$ then create a new path function representing g, that is, with the same data-flow effect as g. This creation can occur by either applying the rules—p **create** g; or the sequence—p **create** followed by p **absorb** g. If exactly one of these possibilities are possible by the rules given in the specification, then return p; otherwise return
   c) If $f = p$ for some path function p. Then apply p absorb g if possible; otherwise go to the next case.
   d) Otherwise, return compose(compose(f_{Id}, f), g) but only if the rule sequence to do it exists.

2. plus(p_1, p_2, ..., p_a):
   a) If $a = 1$ return p_1.
   b) Make a copy of the first path by doing $p \leftarrow \mathrm{compose}(f_{\mathrm{Id}}, p_1)$, then apply p **plus** p_2, ..., p plus p_a. Return p if successful, otherwise return f_{Id}.

3. star(p):

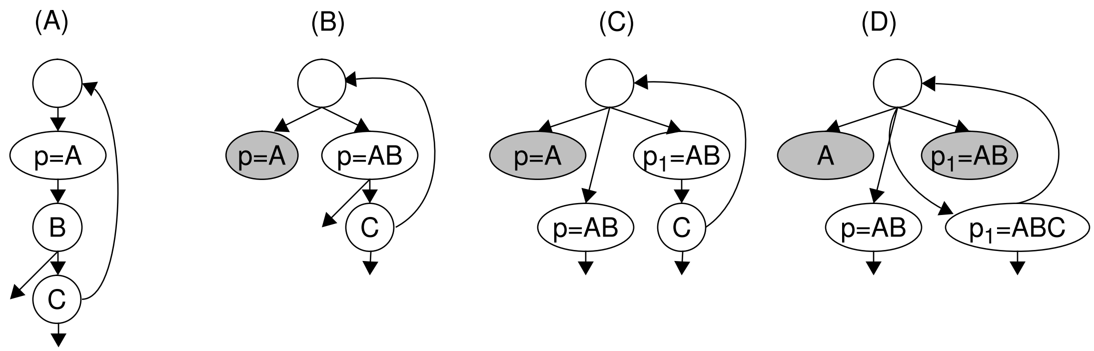


Figure 18. Reusing flow functions in sequences of node

a) Invoke the rule p **star**, if the rule is present in the specification, otherwise return $f_\perp$

With these functions, we can interpret the path computation $(p_1 + p_2 + \ldots + p_a)f$ as compose(join($p_1, p_2, \ldots, p_a$), f), and pq^* as compose(p, star(q)).

The above functions are generated as table lookups. For example, compose would index into a composition table to determine what action to take. Because rule sets are small, Sharlit generates these tables by exhaustive search of the rule sequences.

3.4 Absorb rules save space

Path functions computed by the analyzers can potentially take a lot of storage. For example, the path function `AV_bb` in Figure 5 has two sets of expressions.Thus, just as we must be careful about which flow variables to retain, we must be careful about which path functions to retain. The traditional way of dealing with this problem in traditional optimizers is to group instructions into basic block. Therefore only one path function is required for each basic block.

In Sharlit, the primary mechanism to save space is absorb rules coupled with reference counting of flow functions. To see how absorption saves on space, see Figure 18. Assume that we have the rules: p create, p absorb p, p absorb A, p absorb B, and p absorb C. In Figure 18(A) we apply the rules p create, followed by p absorb A to compute the path function corresponding to the path expression A. In (B), we apply p absorb B giving the path expression AB. We can reuse p because we know that the node A won't need a path function because it will be eliminated by **reduce** (Figure 11). However, in Figure 18(B) this cannot be done because node B's path expression p(now corresponding to path expression AB) cannot be reused because B's other successor needs p to compute their path functions. Hence, p must be copied, as shown in Figure 18(C). Absorption continues in Figure 18(D) and reuses the copy of AB.

To decide when to reuse a path function, we keep a reference count—a count of the successors—for each path function. When composing a path function p with its successor, and p's count is one, we can use p directly in an absorb rule; if p's count is not one, then p must be copied, its count decremented, and the copy used directly in absorption. When we add a successor to a path function (as when path simplification connect nodes to their ancestors), p's count must be incremented.

Not all path functions are reference counted. Path functions associated with loop headers are treated as though they have infinite reference counts to ensure that they are never eliminated because ancestors are never eliminated by **reduce**. Identity flow functions are treated similarly.

4 Sparse Data-flow Evaluation Graphs

Besides being able to do local analysis and interval analysis, path simplification can also build sparse data-flow evaluation graph (SDFEG) [4] for reducible flow graphs to solve sparse data-flow problems efficiently. In the context of reaching definitions problem, a SDFEG corresponds to a SSA graph for one variable. But unlike SSA, SDFEGs can be applied to other data-flow problems.

To compute an SDFEG, it is necessary to find paths which consists solely of identity flow functions, and to prune away all those flow-graph nodes which are connected to nodes in the SDFEG nodes through identity paths. Sharlit can identify such paths with the path simplification rules in Figure 19. In that specification the keywords **identity** tells Sharlit the flow function or path function is the identity flow function. Figure 20 compares an SDFEG computed using Sharlit and that of Choi's.

Sharlit's SDFEG differs in that we use nodes to represent some of the edges in the Choi's SDFEG—We indicate those nodes with dashed circles. These nodes are required because the only way Sharlit can modify a node u's edges in the graph is to replace all of u's edges with one edge from u's ancestor to u. Therefore, at the node F, the only way to connect F to its SDFEG predecessor is indirectly, as ancestors of C and E.

The key point that makes SDFEG efficient is it can eliminate so many more flow variables than our current scheme. The current version of Sharlit's simplifier is capable of finding the paths composed of identity flow function. However, the current version of **propagate** and **reduce** needs to be modified in order for us to eliminate the flow variables at the ends of these identity paths. We continue to work on integrating SDFEG into Sharlit.

5 Conclusion

We have found Sharlit to be a useful tool. Its facilities are ideal for describing DFAs, from simple iterative ones, to ones that use symbolic analysis (recurrence detection) to ones that use interval analysis. Because of its extensibility, it is ideal for prototyping optimizations and to experiment with different internal representations.

We have used Sharlit to build an optimizer as part of the Stanford SUIF compiler project [12]. Our results (Figure 21) show that this optimizer can optimize large, real programs—such as the PERFECT club [3]—as effectively as commercial

```
...
Defs: flow { ... }
Id_path: path Identity;
...
simplifier
Id_path create = {};
Id_path absorb Identity = {};
Id_path plus Id_path = {};
Id_path star = {}
end
```

Figure 19. Rules for doing SDFEG.

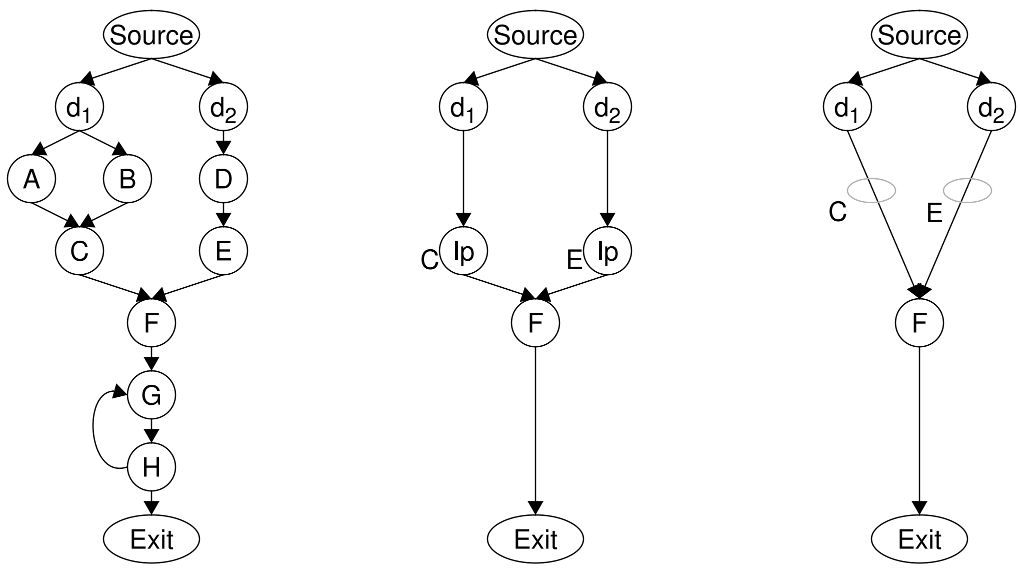


Figure 20. Example of Sharlit's version SDFEG. Ip represents an Identity path.

	SUIF -O	MIPS -O2	SUIF/MIPS
LG	97	106	0.82
LW	1437	1405	1.02
NA	421	377	1.12
SD	88	105	0.84
SR	1270	1253	1.01
TF	243	232	1.05
TI	259	244	1.06

Figure 21. PERFECT club execution results in seconds from runs on a Decstation 5000 with version 1.31 of the Mips compiler.

compilers. Our optimizer uses Sharlit to implement all its internal phases:

1. Value numbering.

2. Reaching definitions.

3. Elimination of partial redundancy[7]: Global common subexpression elimination, strength reduction, backward code motion out of loops.

4. Global register allocation.

5. Constant propagation and copy propagation.

6. Liveness analysis and Dead code elimination.

7. Recurrence detection[2].

Our experiences suggest that Sharlit is a good and practical. Constructing the optimizer using Sharlit has required much less effort and has resulted in a compiler that is simpler, easier to extend, and more versatile.

6 Acknowledgments

This research has been supported by DARPA under contract N00039-91-C-0138. We would like to thank Rob French, Monica Lam, Bob Wilson, and Michael Wolf for their helpful comments.

7 References

1. A.V. Aho, R. Sethi, J.D. Ullman, *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Z. Ammarguellat and W. L. Harrison III, Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*
3. M. Berry et al, The PERFECT Club Benchmarks: effective performance evaluation of supercomputers. Technical Report UIUCSRD Rep. No. 827, University of Illinois Urbana-Champaign, 1989.
4. J. Choi, R. Cytron, and J. Ferrante, Automatic construction of sparse data flow evaluation graphs, In *POPL 1991*.
5. F. Chow, *A Portable Machine-Independent Global Optimizer,* PhD thesis, Stanford University, 1983
6. M.S. Hecht and J.D. Ullman. A simple algorithm for global data-flow analysis programs, In *SIAM J. Computing*. (4):519-532.
7. E. Morel and C. Renvoise, Global optimization by suppression of partial redundancies, In *Communications of the ACM*. (22):96-103
8. M. Sharir. "Structural analysis: a new approach to flow analysis in optimizing compilers". *Computer Languages 5* (1980), 141-153.
9. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, Reading, Mass.
10. R. E. Tarjan. "Testing flow graph reducibility". *J. of Comput. and Syst. Sci.* (9 1974), 355-365.
11. R.E. Tarjan, Fast algorithms for solving Path problems. *Journal of the ACM*, 3(28):591-642, July 1981.
12 S. Tjiang, M. E. Wolf, M. S. Lam, K. L. Pieper, and J. L. Hennessy, Integrating scalar optimization and parallelization, In *4th Workshop on Languages and Compilers for Parallel Computing*, Aug 1991.
13 J. D. Ullman. "Fast algorithms for the elimination of common subexpressions". *Acta Inf. 2*, 3 (July 1973), 191-213.