Verified Validation of Program Slicing

Sandrine Blazy
IRISA - University of Rennes 1
sandrine.blazy@irisa.fr

Andre Maroneze
IRISA - University of Rennes 1
aoliveir@irisa.fr

David Pichardie
IRISA - ENS Rennes
david.pichardie@ens-rennes.fr

Abstract

Program slicing is a well-known program transformation which simplifies a program with respect to a given criterion while preserving its semantics. Since the seminal paper published by Weiser in 1981, program slicing is still widely used in various application domains. State of the art program slicers operate over program dependence graphs (PDG), a sophisticated data structure combining data and control dependences.

In this paper, we follow the *a posteriori* validation approach to formally verify (in Coq) a general program slicer. Our validator for program slicing is efficient and validates the results of a run of an unverified program slicer. Program slicing is interesting for *a posteriori* validation because the correctness proof of program slicing requires to compute new supplementary information from the PDG, thus decoupling the slicing algorithm from its proof.

Our semantics-preserving program slicer is integrated into the CompCert formally verified compiler. It operates over an intermediate language of the compiler having the same expressiveness as C. Our experiments show that our formally verified validator scales on large realistic programs.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Validation

Keywords program slicing, verified compiler, verified static analysis

1. Introduction

Program slicing is a technique to simplify a program with respect to a criterion, that is, either a variable at a given program point or a program point. This second criterion is more general than the first one, so we choose program points as slicing criteria in this paper. Slicing a program with respect to a program point consist in slicing it with respect to all the variables that are used at this program point. It consists in the removal of statements that have no influence on the criterion, and are said to have been sliced away. The result of this process, which is called a sliced program, is another program, a simplified version of the original one which behaves in the same way, with respect to the slicing criterion.

Since the seminal paper published by Weiser [29] in 1981, a great deal of papers on different kinds of program slicing, operating over various kinds of programming languages were published, including several surveys (e.g. [25] and [30]). The main application domains of program slicing are debugging, program understanding and software maintenance. Recent application domains are still emerging. For instance, program slicing is used by automatic program proof to simplify programs and specifications so that they can be easier to prove with deductive verification tools [3].

Program slicing is also often used in conjunction with other analyses, as a preprocessing step, thus improving both precision and efficiency of the analyses. For instance, reference tools for estimating the worst case execution time (WCET) of programs perform a preliminary program slicing step when estimating loop bounds of programs; improvements due to program slicing are crystal clear on programs with nested loops. Indeed, the program slicing step reduces the number of program states computed by the WCET analysis and improves the precision of the WCET estimation without requiring more complex analyses.

State of the art program slicers operate mainly over program dependence graphs (PDG), a sophisticated data structure representing explicitly data and control dependences [13]. Constructing a slice consists mainly in traversing a PDG (in order to compute paths) and performing a post-dominance analysis. Program slicers that construct executable slices in the presence of jump statements implement delicate and error-prone graph algorithms. A few paper-and-pencil proofs related to the correctness of program slicing exist in the literature [24, 23, 2]. The key ingredients to these proofs are presented in [23, 2], where the authors define the notions of relevant variables and specific vertices called next observable vertices that are used only in the proof.

To the best of our knowledge, the only mechanized proof of program slicing is detailed in [28]. It uses the Isabelle/HOL proof assistant to formalize in a non-constructive way a PDG and its use to produce program slices. We follow a different approach, called *a posteriori* validation [19, 26] to formally verify a program slicer operating over a PDG. Instead of formalizing a PDG, we write in OCaml an unverified program slicer and we validate the results of each of its runs. Our validator is written in Coq; it is efficient and proved correct. We chose this pragmatic approach to formal verification because our program slicer relies on sophisticated data structures and delicate algorithms.

Program slicing is also interesting for *a posteriori* validation because the correctness proof of program slicing requires to compute new supplementary information from the PDG (relying on relevant variables and next observable vertices), thus decoupling the slicing algorithm from its proof.

Our semantics-preserving program slicer is integrated into the CompCert formally verified compiler [17]. It operates over an intermediate language of the compiler having the same expressiveness as C. Our work has been currently used for formally verifying a loop bound estimation analysis [7]. Some details about this analysis and comparisons with the current work are given in the related work section.

CPP '15, January 12–14, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676724.2693169

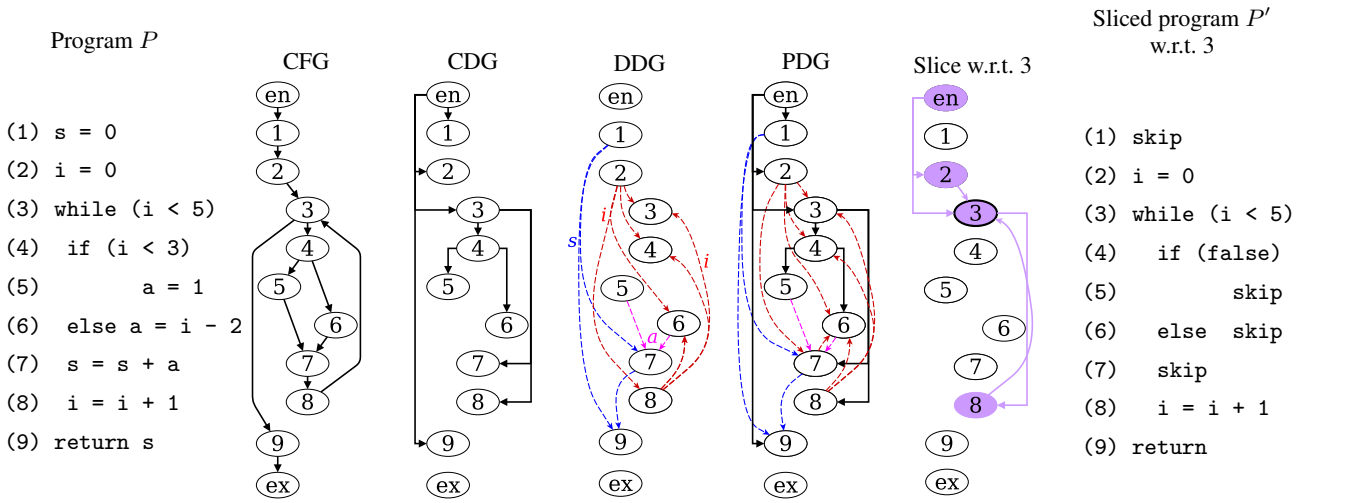

Figure 1. Example program P with its CFG, its control dependence graph (CDG), its data dependence graph (DDG), the resulting program dependence graph (PDG), the slice w.r.t. criterion 3, and the resulting sliced program.

All results presented in this paper have been mechanically verified using the Coq proof assistant. The complete development is available online [1]. This paper makes the three following contributions:

1. a validation algorithm for program slicing, formally verified in Coq,
2. a fully executable program slicer, integrated in the CompCert C compiler,
3. experiments showing that our validator is efficient and scales on large realistic programs.

The remainder of this paper is organized as follows. First, Section 2 briefly introduces main concepts related to program slicing and PDG. Then, Section 3 explains the architecture of our formally verified validator for program slicing. Section 4 details the construction of sliced programs. The next two sections are devoted to the formal verification of our program slicer: Section 5 details our validator, and Section 6 explains our main correctness theorem and the simulation relation it relies on. In Section 7, we describe the experimental evaluation of the implementation of our validator. Related work is discussed in Section 8, followed by concluding remarks.

2. Program Slicing Based on Program Dependence Graph

A sliced program is a simplified version of the program which has been sliced. We represent a program by its control flow graph (CFG), which maps program points to statements. A sliced program is defined with respect to a *slicing criterion*, which we consider as a program point, referred to as ℓ. Given a slicing criterion ℓ, we define a slice $\mathtt{SL}_\ell$ as the set of program points made of ℓ and all the nodes that depend on ℓ. A dependence is here a data dependence or a control dependence. Equivalently, $\mathtt{SL}_\ell$ is a set of program points which preserve the + values of the variables used in ℓ.

Instead of constructing the final sliced program by removing the nodes that are not in the slice, we follow the choice made in [23] and replace the statements associated to these nodes with empty statements. In so doing, program slicing preserves the CFG, thus facilitating the correctness proof of program slicing, without affecting the precision of the computed slices. Empty statements are then removed by further compilation passes performed by the CompCert compiler.

A widely-used technique to compute program slices is via the construction of the PDG [13], which is the composition of two subgraphs, one containing control dependences (CDG) and another one containing data dependences (DDG). The former is related to conditional branches (e.g. the condition of an `if` statement controls its branches) and the latter is related to the current values of variables obtained via def/use paths.

Our PDG is constructed from the CFG using algorithms relying on compiler techniques. Control dependences are computed using Lengauer and Tarjans's [16] post-dominance tree algorithm and also Muchnick's [18] algorithm to construct the CDG from the post-dominance tree. Data dependences are computed using a reaching definition analysis [20] based on a dataflow iterator, and they rely on the def/use [20] paths associated to program statements: each statement which uses a program variable depends on every statement whose definition may reach it.

Rather than formalizing in Coq the construction of the PDG and post-dominance trees, we chose to write these algorithms in OCaml and to formally verify in Coq a validator of their results. The main advantage of this approach is that it simplifies the correctness proof of program slicing, thus avoiding to reason on sophisticated data structures such as the PDG and dominance trees.

Figure 1 presents an example program P along with its CFG, CDG, DDG and PDG. The PDG is a graph having the same nodes as the CDG and the DDG. The edges of a PDG consist of both data dependence edges (the DDG edges) and control dependence edges (the CDG edges). The last two columns in Figure 1 illustrate how the PDG is used to compute a slice: by performing a backward traversal of the PDG starting at the slicing criterion (in the figure, program point 3), we obtain the slice, which is here the set of colored nodes $\mathtt{SL}_3 = \{\mathtt{en}, 2, 3, 8\}$. The entry node `en` is included in the slice due to control dependences, while nodes 2 and 8 are included due to data dependences.

The slice is then used to compute the final sliced program. The structure of the CFG of this final sliced program (i.e. the number and labels of nodes as well as the edges) is the same as the structure of the CFG of the initial program. As explained previously, preserving the CFG facilitates the correctness proof of program slicing. The final sliced program is presented in the last column in Figure 1. It is computed from the slice by replacing every statement not in the slice either with an empty statement (written as `skip`) or with an empty conditional statement.

More precisely, the condition of an empty statement is either `true` or `false` (because of the preservation of the CFG). In Fig-

ure 1, as there is no dependence from the condition `i<3` (node 4) to the slicing criterion (node 3), the `if` statement does not belong to the slice. Thus, we replace the `if` statement by the `if (false) skip skip` statement. We could have chosen `true` as a condition, and we explain in Section 4.2 the reason why we chose `false`.

In terms of efficiency, Tip [25] mentions the advantage of PDG-based approaches over other approaches, such as dataflow equations. The major benefit of the PDG is that, once it has been computed for a given program, several slices (based on different slicing criteria) can be computed efficiently for the same program. Every new slice only requires a backward traversal of the PDG. This avoids the most expensive part of the slicing computation, which is the construction of the PDG. We benefit from this fact in our application of program slicing.

When computing a slice, the steps used to compute control and data dependences do not depend on the slicing criterion. Thus, they are performed only once per program, independently of the number of slices to be computed. Only the last steps (backward traversal of the PDG and replacement of sliced statements) need to be performed for each slice.

3. Architecture of our Formally Verified Program Slicer

Typical program slicers take as input a program P and a slicing criterion ℓ and produce a sliced program P'. The architecture of a PDG-based program slicer is presented in Figure 2. In this slicer, there are two main components: the slice calculator, which computes a slice $\texttt{SL}_\ell$, and a slice builder, which uses $\texttt{SL}_\ell$ to actually modify the program P, producing the sliced program P'.

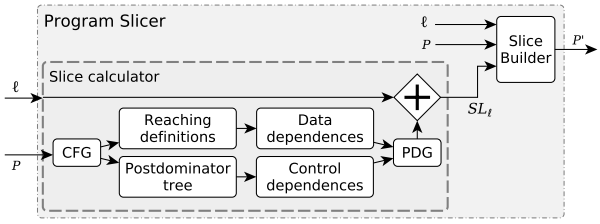

Figure 2. Architecture of a PDG-based program slicer.

Our formally verified program slicer is presented in Figure 3 as a diagram and in Figure 4 as pseudocode. As shown in Figure 3, we modified the architecture of Figure 2 by adding a slice validator between the slice calculator and the slice builder. We then formally verify both the slice validator and the slice builder, leaving the slice calculator to untrusted OCaml code. From the inputs P (the program to be sliced) and ℓ (the slicing criterion), we compute the slice $\texttt{SL}_\ell$, validate it, and then build the sliced program P'.

The correctness proof of our program slicer requires the slice calculator to compute extra information from the slice and related to relevant variables and next observable vertices. This information, which we denote as $\texttt{Extra}_\ell$, improves the performance of the validator and does not constrain the program slicer. This information is used for two purposes: *a posteriori* validation and efficient construction of a CFG-preserving slice. It is less difficult to compute than the PDG. Our slice validator is modeled as a function taking as input this information and a slice and returning a boolean value. We proved in Coq that when the validator returns true, then the slice is correct.

$\texttt{Extra}_\ell$ is composed of three mappings: $\texttt{RV}_\ell$, $\texttt{NObs}_\ell$ and $\texttt{DObs}_\ell$, which correspond respectively to relevant variables (for the slicing criterion ℓ), next observable vertices (for the slicing criterion ℓ) and distance between next observable vertices, that are detailed

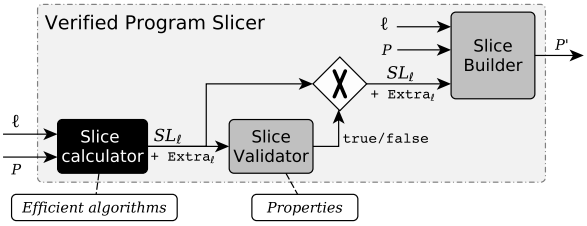

Figure 3. Architecture of the validated program slicing. The black block is untrusted code, gray blocks are formalized in Coq.

respectively in Section 5 and Section 4. Using $\texttt{SL}_\ell$ plus these three mappings, the validator is able to verify the set of properties that every valid slice must respect. These properties are detailed in Section 5. The validated slice is then given to the formally verified slice builder (along with $\texttt{Extra}_\ell$), where it is used to compute the actual program slice.

The pseudocode in Figure 4 further details the operations performed in Figure 3, given a program `P` and a slice criterion ℓ. First, the PDG is constructed from `P`. Second, the slice $\texttt{SL}_\ell$ is computed (line 2), and this is the most expensive computation. Then, $\texttt{Extra}_\ell$ is computed from the slice (lines 3 and 4). The next step is the validation of the slice: after the calls to the two validators called `obs_validator` and `rvs_validator`, the slice is either validated as a correct slice or rejected. The last step is the construction of the executable sliced program from a validated slice.

```
Definition slicer (P : program) (ℓ : node) :=
1   let PDG := build_pdg(P) in
2   let SL := compute_slice(ℓ,PDG) in
3   let (NObs,DObs) := compute_obs(P,ℓ,SL) in
4   let RV := compute_rv(P,ℓ,SL) in
5   if obs_validator(P,ℓ,SL,NObs,DObs) then
6     if rvs_validator(P,SL,RV) then
7       slice_builder(P,SL,DObs) (* produces P' *)
8     else Error("invalid RVs")
9   else Error("invalid NObs/DObs")
```

Figure 4. Pseudocode of the program slicer (italics indicate OCaml code).

Our program slicer operates over the RTL (Register Transfer Language) intermediate language of the CompCert compiler (as shown on the top of Figure 5). Our program slicer takes as input a RTL program and generates a sliced RTL program. We chose RTL mainly because this language is the most adapted for representing programs by their control flow graphs. RTL programs have the same expressiveness as C programs. Our program slicer can slice programs with unstructured loops and `goto` statements. Moreover, we benefit from the inlining optimization also performed at the RTL level. Thus, our program slicer currently handles only non-recursive programs. This limitation could be lifted by assuming any function call may modify all the memory or formalising an interprocedural slicing technique.

The RTL language is defined in CompCert via a small-step semantics $\sigma \rightarrow \sigma'$. In this paper, for clarity reasons, we simplify the semantic states σ and define them as pairs (ℓ, E) consisting of a program point ℓ and an environment E mapping variables to values. $E(x)$ denotes the value of variable x in environment E. Variables model temporaries (i.e. pseudo-registers, where infinitely many pseudo-registers are available) and the memory seen as a single block. In the sequel of this paper, we use the notation $\sigma.l$ to denote the program point of state σ. The set of reachable states of a program is denoted by $\texttt{reach}(P) = \sigma_0 \rightarrow^* \sigma'$, where σ_0

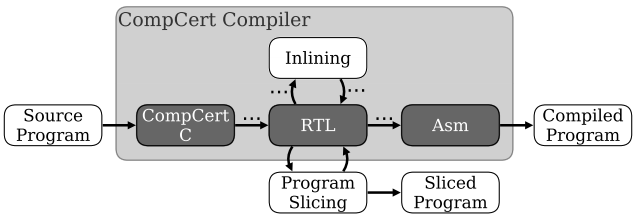

Figure 5. Integration of our program slicer into the CompCert compiler

indicates an initial state, and $\rightarrow^*$ is the reflexive transitive closure of the $\rightarrow$ relation.

4. Computing a Sliced Program

This section explains how executable sliced programs are generated. First, it details our slice calculator that we wrote in OCaml. Then, it details our slice builder that we wrote in Coq. Our slice builder generates a sliced program from a slice; this slice is computed from an initial program and a slicing criterion.

4.1 Slice Calculator

Our algorithm to compute control dependences is based on Muchnick's CDG construction [18]. It requires as input a tree of immediate post-dominators. We recall that n' post-dominates n if n' is present in every CFG path from n to the program exit. n' is said to *immediately* post-dominate n if $n \neq n'$ and no other node which post-dominates n is also post-dominated by n'. The uniqueness of the immediate post-dominance relation leads to a tree structure.

To compute this tree, Lengauer and Tarjans's algorithm provides an efficient solution [16]. This graph-based algorithm is widely used and available on several libraries, such as the OCamlGraph library [10] that we used. Moreover, our data structures are based on native machine integers for extra efficiency. Note that our control dependence algorithm, in line with common practice, augments the CFG with the insertion of unique entry and exit nodes (*en* and *ex* in Figure 1) and two edges, from *en* to *ex* and from *en* to the CFG entry point. This ensures a single post-dominance tree, otherwise it may be a forest of disjoint trees.

The computation of data dependences relies on def-use chains. More precisely, we perform a dataflow-based reaching definitions analysis, using a dataflow framework à la Kildall. This framework is an adaptation of CompCert's, using the same style of Patricia trees [22], but based on native machine integers (instead of Coq's standard arbitrary-precision trees). This not only improves the CPU time, but it also reduces memory consumption. Finally, we apply a liveness analysis to avoid propagating definitions for variables which will never be used. This is especially important for scalability. Once we have the reaching definitions, we transform them into data dependences by traversing the CFG and adding an edge to the DDG whenever a program point uses a variable which has a reachable definition at that program point. Data dependencies between memory accesses are currently managed in a very conservative way: our analysis assumes any read or write may access any part of the memory. A more fine grained reasoning could be achieved by relying on alias information [14]: the alias analysis would be performed before the computation of the def-use sets, and the sets of program variables considered in the dependences would include references to specific memory locations. Less coarse def-use sets would lead to less dependence edges in the PDG, thus improving precision.

Finally, the PDG itself is composed of the the edges of the CDG and the edges of the DDG. The PDG is reused for computing all the slices of the program it represents. The slice itself is computed by starting at the slicing criterion (which is always in the slice) and traversing the PDG backwards, capturing all direct (and, by path transitivity, indirect) dependences. We then use the slice to build an executable sliced program.

4.2 Slice Builder

Given a slicing criterion ℓ and a program P, the slice calculator computes a slice $\mathtt{SL}_\ell$ (see Figure 2). In a simple imperative language without `gotos`, a slice builder can be simply defined as a function which iterates over each program statement, and removes those whose program points are not in $\mathtt{SL}_\ell$. For `if` (resp. `while`) statements, this implies removing their branches (resp. bodies) as well.

However, a slice builder for an unstructured language such as ours with `gotos` is more complex. The main difficulty stems from the slicing of conditional statements: arbitrary control flow means that it is not possible to always select the `true` or the `false` branch. For instance, consider the C program (and its CFG) shown at the left of Figure 6.

```
L1:
  if (x < y)
    if (y < z)
      goto L1;
    else goto L2;
  else
    goto L1;
L3:
  exit;
L2:
  print(1);
  ...;
```

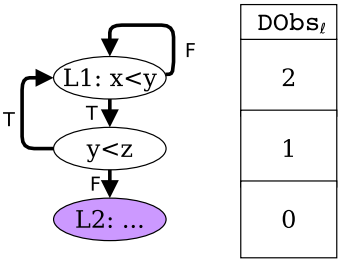

Figure 6. Example of unstructured flow leading to non-trivial slicing of conditional branches. The slicing criterion ℓ is `L2`. $\mathtt{DObs}_\ell$ is the next observable distance associated to each CFG node, used to build the sliced program.

This program contains a loop that starts at program point `L1` and has program point `L2` as its only exit. L2 is reached when the first condition (`x < y`) is true and the second one (`y < z`) is false. Program point `L3` is unreachable in this program. Let us slice this program with respect to program point `L2`. In this example, the program is such that `L2` has no dependency from any statement in the loop, so the entire loop can be sliced away. A consequence is that the variables `x`, `y` and `z` are sliced away and do not belong to the slice. Thus, they can not be used in the conditions of the `if` statements anymore.

A naive slice builder which simply removed the condition of the `if` and its branches would produce an incorrect slice: from program point `L1` we would execute the statements at program point `L3` and thus terminate execution, without ever reaching program point `L2`. A less brutal, but equally naive slice builder, might try instead to replace condition of the `if` with a fixed static value, either `true` or `false`. By systematically choosing always the same branch, the final result will be an infinite loop in `L1`.

A correct slice builder needs to statically identify which branch should be taken in every sliced conditional statement, in order to avoid infinite loops. In the example in Figure 6, this should result in replacing `if (x < y)` with `if (true)` and `if (y < z)` with `if (false)`. With such a replacement, the resulting slice can then reach program point `L2` as intended.

To identify the branch to be taken, we use the fact that the chosen branch must enable the program to reach a node that is

in the slice. In other words, we must avoid creating infinite loops of nodes that are not in the slice. Nodes in the slice are called *observable* nodes (a terminology related to its proof by simulation, see Section 6), and the node chosen by the slice builder is the *next observable* node. Note that there can be several sliced nodes between a sliced conditional and its next observable node. It is therefore not sufficient to examine each immediate successor. The general solution requires searching along the CFG paths for the next successor in the slice. Once the next successor is found, the branch taken in (cycle-free) path used to reach it can be safely chosen in the sliced program.

Instead of recomputing the next observable node on-the-fly for each sliced conditional, we compute a partial function $\texttt{DObs}_\ell$ mapping each program point to the *distance* (in number of edges in the CFG) to its next observable node. This distance indicates which branch to take: the successor having the shortest distance will safely avoid infinite loops.

For instance, consider the slicing of the conditional node `L1` in Figure 6. Its next observable distance is 2, as its next observable node is `L2` which is at the distance 2 to `L1`. The successor of the `true` branch of node `L1` is `y<z`, which has a next observable distance of 1. The successor of the `false` branch is itself, whose distance is 2. Choosing the successor closest to its next observable node (i.e. choosing the `then` branch) is the way to avoid generating in the slice the infinite loop in `L1`. The same situation (that is, the choice between distances 0 and 2) happens when considering the sliced conditional in `y<z`, this time taking the `else` branch.

To compute $\texttt{DObs}_\ell$, first we compute the (partial) function $\texttt{NObs}_\ell$, which uses the CFG and the slice $\texttt{SL}_\ell$ to map each program point to its next observable node. A node in the slice is its own next observable node. The $\texttt{NObs}_\ell$ function is partial because some nodes (such as 9 and *ex* in Figure 1) have no next observable node.

The traversal performed during computation of $\texttt{NObs}_\ell$ is made more efficient by the fact that, for any adjacent nodes $n \to n'$, if n is not in the slice, then $\texttt{NObs}_\ell(n) = \texttt{NObs}_\ell(n')$. This allows us to collect several nodes during the traversal and update all of them at once when their next observable node is reached. Also notice that all successors of a node not in the slice have the same next observable node, which ensures that our algorithm works correctly.

Then, using the CFG and $\texttt{NObs}_\ell$, $\texttt{DObs}_\ell$ is computed via a simple backward BFS starting at the nodes in the slice. Note that both mappings will also be used to validate the correctness of the slicing.

```
Definition slice_builder (P : program)
           (SL : list vertex)
           (DObs : vertex → N) :=
1 foreach program point l' in P:
2   let s := P.code[l'] in
3   if l' ∈ SL:
4     P'.code[l'] := s
5   else:
6     if s is an unconditional statement:
7       P'.code[l'] := skip
8     else: (* s is a condition *)
9       let ([l_true, l_false] := successors(l')) in
10      let (b := shortest_DObs(DObs, l_true, l_false) in
11      P'.code[l'] :=
          constant_predicate(b, l_true, l_false)
```

Figure 7. Pseudocode of a slice builder algorithm, which uses a slice and the next observable distances to produce an executable sliced program.

The pseudocode of the slice builder algorithm is presented in Figure 7. For each program point l', it tests whether l' belongs to the slice. If so, it preserves the statement associated to l' (line 4). Otherwise, it replaces this statement with:

- a `skip`, if the statement is not a condition (line 7);
- a constant predicate otherwise (lines 9-11). The `shortest_DObs` function uses $\texttt{DObs}_\ell$ to choose whether the returned predicate is `true` or `false`.

5. Validation of Slices

As explained before, we do not directly program and prove correct in Coq the slice calculator algorithm that we described in the previous section. We only prove in Coq the previous `slice_builder` function (introduced in Figure 7) and develop a verified validation algorithm that given a slicing criterion ℓ checks the result $(\texttt{SL}_\ell, \texttt{NObs}_\ell, \texttt{DObs}_\ell)$ and then use it with `slice_builder` to transform the input program.

Amtoft [2] and Ranganath *et al.* [23] describe a pen-and-paper proof of correctness for program slicing based on the notions of relevant variables. Our validation algorithm uses this information as an extra hint for a faster validation. The notion of relevant variables is related to data dependences. From the program slicing point of view, there are two kinds of program variables: relevant variables, which are those whose values may affect the slicing criterion (and therefore must be preserved), and other variables, whose values are not guaranteed to be preserved in the program slice.

Given a slicing criterion ℓ, the set of relevant variables is defined for each program point n and written as $\texttt{RV}_\ell(n)$. For instance, in Figure 1, $\texttt{RV}_\ell(6) = \{3\}$. The sets $\texttt{RV}_\ell$ make precise, at the program point n, which variables are semantically preserved by the program slicing. $\texttt{RV}_\ell(n)$ includes the variables used in the slicing criterion ℓ.

We instrument our program slicer to make it produce the extra information $\texttt{RV}_\ell$ and check the result $(\texttt{SL}_\ell, \texttt{RV}_\ell, \texttt{NObs}_\ell, \texttt{DObs}_\ell)$. The extra information $\texttt{RV}_\ell$ is easily computed by an untrusted backward dataflow analysis.

The validation algorithm works as follows: for each control flow edge from node n to node n' that generates a set U of used variables and a set D of defined variables, we check the property $\texttt{SL}_\ell, \texttt{RV}_\ell, \texttt{NObs}_\ell, \texttt{DObs}_\ell \vdash n \xrightarrow{D,U} n'$ defined in Figure 8. There are two rules. One must hold when n belongs to the computed slice, the other holds when n does not belong to it.

In the first case, n must be its own next observable node and as a consequence the nearest observable is at distance 0 from n. The set of relevant variables at node n is defined by a backward dataflow constraint that removes all variables defined in n from the set $\texttt{RV}_\ell(n')$ of the successor node n', and adds the variables that are used in n. Variables in U must be preserved at node n because their values have an influence on the result of the instruction while variables in D are dead at n and do not affect the execution.

In the second case, n is not its own next observable. Its successor n' shares the same next observable node (which may be undefined or n' itself). The variables that are defined in n are not preserved ($D \cap \texttt{RV}_\ell(n) = \emptyset$) but $\texttt{RV}_\ell(n)$ must at least contain the preserved variables from n'. The distance $\texttt{DObs}_\ell(n')$ of the next node n' is not strictly less than $\texttt{DObs}_\ell(n) - 1$ (if it is defined) but this bound is reached for at least one successor.

To intuitively convince oneself of the correctness of the validator, it suffices to consider what would happen if the slice did not contain a necessary node, that is, instead of having $n \in \texttt{SL}$, we would have $n \notin \texttt{SL}$, for some n which should have been in the slice. In this case, we would either have a missing control dependence (which would entail the violation of constraints related to

$$\frac{\texttt{NObs}_\ell(n) = n \quad \texttt{DObs}_\ell(n) = 0 \quad U \cup \texttt{RV}_\ell(n')\backslash D \subseteq \texttt{RV}_\ell(n)}{\texttt{SL}_\ell, \texttt{RV}_\ell, \texttt{NObs}_\ell, \texttt{DObs}_\ell \vdash n \xrightarrow{D,U} n'} \; n \in \texttt{SL}_\ell$$

$$\frac{\begin{array}{c}\texttt{NObs}_\ell(n) \neq n \qquad\qquad \texttt{NObs}_\ell(n) = \texttt{NObs}_\ell(n') \\ n \neq \ell \qquad D \cap \texttt{RV}_\ell(n) = \emptyset \qquad \texttt{RV}_\ell(n') \subseteq \texttt{RV}_\ell(n) \\ \texttt{DObs}_\ell(n) = d \implies d > 0 \wedge d - 1 \leq \texttt{DObs}_\ell(n') \wedge \exists n'', n \to n'' \wedge \texttt{DObs}_\ell(n'') < d\end{array}}{\texttt{SL}_\ell, \texttt{RV}_\ell, \texttt{NObs}_\ell, \texttt{DObs}_\ell \vdash n \xrightarrow{D,U} n'} \; n \notin \texttt{SL}_\ell$$

Figure 8. Validation constraints on each control flow edge

$\texttt{NObs}_\ell$ or $\texttt{DObs}_\ell$, e.g. $\texttt{DObs}_\ell(n) \neq \texttt{DObs}_\ell(n')$), or a missing data dependence (which would entail the violation of constraints related to $\texttt{RV}_\ell$, e.g. $D \cap \texttt{RV}_\ell(l)(n) \neq \emptyset$).

A slice $\texttt{SL}_\text{l}$ is validated for correctness but not precision, that is, it may contain extra nodes, as long as the other sets $\texttt{DObs}_\ell(l)$ and $\texttt{NObs}_\ell(l)$ also contain the corresponding extra nodes. This happens when the computed slice is imprecise, albeit correct.

The validation is performed in one linear pass of the CFG. As we show in Section 7, this computation is negligible compared to the slice computation.

6. Correctness of Program Slicing

The correctness of program slicing is expressed by the following theorem. It states that all program variables used in the slicing criterion as well as in any program point in the slice are preserved. In other words, whenever we execute both the original and sliced programs up to a same program point ℓ, the values of all variables used in ℓ are equal in both programs.

Theorem 1 (Correctness of Program Slicing)**.** *Let P be a program and ℓ a program point of P. Let $\texttt{SL}_\ell$ be the be the slice of P with respect to the slicing criterion ℓ, and P' the corresponding sliced program. Then, for any reachable state $\sigma \in \texttt{reach}(P)$ such that its program point $\sigma.\ell$ is in the slice, the same program point can be reached in P', with the same values for all variables used in $\sigma.\ell$:*

$$\forall \sigma \in \texttt{reach}(P), \sigma.\ell \in \texttt{SL}_\ell \quad \Rightarrow$$
$$\exists \sigma' \in \texttt{reach}(P') \mid \sigma'.\ell = \sigma.\ell \wedge \forall x \in U_{\sigma.\ell}, \sigma.E(x) = \sigma'.E(x)$$

The proof of this theorem relies on a weak simulation relating the sets $\texttt{RV}_\ell$, $\texttt{NObs}_\ell$ and $\texttt{SL}_\ell$, without relying directly on the control and data dependences used to compute the PDG. We present here the main parts of the Coq formalization of this proof. As explained in [2], this simulation enables slicing away loops unrelated to the slicing criterion.

The main simulation lemma uses the validated sets of relevant variables and next observable nodes (and their validated properties) to show the existence of matching states in the sliced program. Matching states have the same next observable nodes and the same values for all relevant variables at the current program point. Because the set of used variables in the slicing criterion is included in its set of relevant variables, this simulation leads to Theorem 1.

Lemma 1 (Simulation of Program Slicing)**.** *Let P be a program and ℓ a program point of P. Let P' be the sliced program from P with respect to program point ℓ, such that P' was validated successfully. Then, we have the following simulation relation between reachable states of both programs: for each step $\sigma_1 \to \sigma_2$ of the execution of P, and each state σ_1' of P' that matches with σ_1, there exists a state σ_2' that matches with σ_2 and that is reached after zero, one or several steps from σ_1'.*

$\forall \sigma_1, \sigma_2 \in \texttt{reach}(P), \sigma_1' \in \texttt{reach}(P')$*, if* $\sigma_1 \to \sigma_2$ *and* $\sigma_1 \sim \sigma_1'$*, there exists* σ_2' *such that* $\sigma_1' \to^* \sigma_2'$ *and* $\sigma_2 \sim \sigma_2'$*.*

This simulation matches each step in the original program to zero, one or several steps in the sliced program. As detailed in the rest of this section, this is due to the fact that our program slicer may generate empty conditional statements. The four kinds of proof diagrams of Figure 9 show the different state configurations used during the simulation. Colored states indicate that the current program point is in the slice. The horizontal line between σ_1 and σ_1', as well as the arrow from σ_1 to σ_2, are the hypotheses of the simulation.

In Figure 9, we consider two matching states, $\sigma_1 \in \texttt{reach}(P)$ and $\sigma_1' \in \texttt{reach}(P')$. An execution step $\sigma_1 \to \sigma_2$ is called observable if $\sigma_1.l \in \texttt{SL}_\ell$, and silent otherwise. A silent step corresponds to an execution step from an empty statement or from an empty conditional statement in the sliced program.

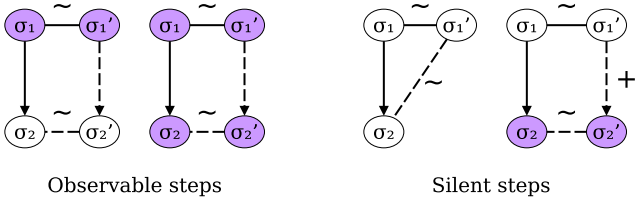

Figure 9. Simulation diagrams for the weak simulation used to prove correctness of the slicing. Continuous lines represent hypotheses, dashed lines represent conclusions. Nodes in the slice are colored.

Observable steps are synchronized (one-to-one), and independent of whether the arrival node is in the slice. Silent steps, on the other hand, depend on whether the arriving node ($\sigma_1'.l$) is in the slice. When this is not the case, no step is performed in the sliced program (it waits for the original program to reach its next observable state). Otherwise, the sliced program performs one or several steps (depending on how many steps the original program has performed while the sliced program was waiting) until it reaches the same program point. The number of steps in this case is equal to the next observable distance.

Figure 10 presents an example of the evolution of a few execution steps. Note that the execution is often alternating between nodes in the slice and nodes not in the slice, therefore between observable and silent steps, and whenever execution re-enters the slice there is a resynchronization of both programs. The agreement relation between relevant variables in both program states and the careful slicing of conditional statements ensure that this resynchronization is always possible (e.g. by preventing the introduction of infinite loops in the sliced program).

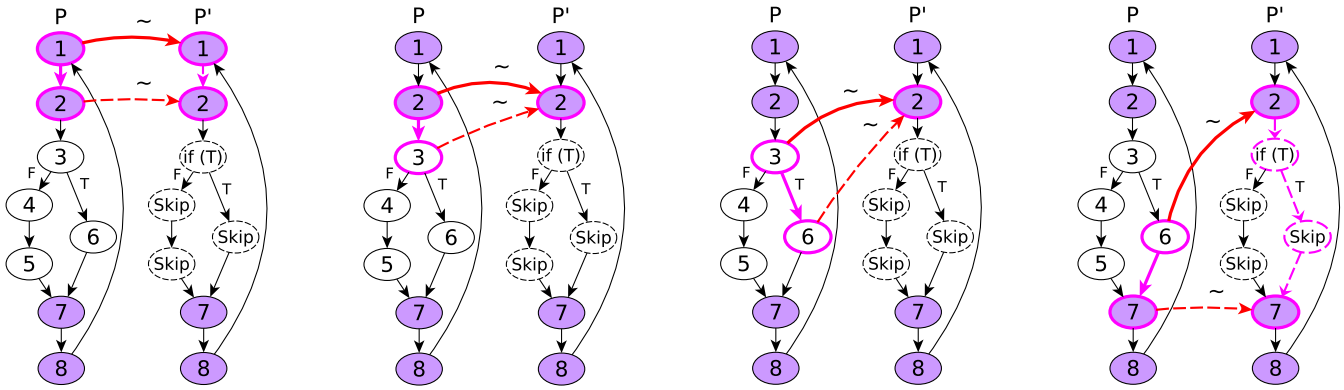

Figure 10. Example of the evolution of a simulation between a program P and its sliced version P'. Thick solid arcs represent hypotheses, dashed arcs represent conclusions. Nodes in the slice are colored.

7. Experimental Evaluation

One of the contributions of this paper is the experimental evaluation of our program slicer, to ensure that it is efficient and that it scales for large programs. Thanks to the separation between algorithm and proof enabled by our program slicer validator, we are able to perform a very efficient slicing computation. Another benefit of this separation is the possibility of quickly modifying the algorithms and data structures used in the computation, without having to adapt or redo the proofs.

Our formal development consists of about 5,000 lines of Coq specifications and 7,000 lines of proofs. It also contains about 3,000 lines of OCaml code, including the PDG construction and accessory code. Our development is integrated into the latest version (i.e. version 2.4) of the CompCert C compiler [9].

To evaluate our program slicer, we chose a set of C programs containing several thousands of lines of code. The benchmarks come from several different sources: `arcode`, `lzw` and `lzss` were taken from the CompCert benchmarks; `bzip2`, `gzip` and `oggenc` come from the SMCC archive by Stephen McCamant; `hmmer` and `mcf` come from the SPEC2006 benchmarks; and `nsichneu` and `papabench` come from WCET-related reference benchmarks.

Figure 11 presents the sizes of our benchmarks, both in number of lines of C code (#Cloc column) and in sizes of the RTL program: number of vertices in the CFG (#RTL column, also counting the number of RTL statements) and edges (#Edges column). Note that this is the final size of the RTL program after several transformations performed by the CompCert compiler, such as function inlining and dead code elimination. This explains why some programs with similar number of lines of C code result in RTL programs with very different sizes. The size of the biggest C program of our benchmark is almost 50,000 LOC. The biggest CFG of our benchmark consists of about 90,000 vertices and 100,000 edges.

For each program, we measure the time necessary to compute and validate one program slice. In order to obtain interesting slices, we choose loop headers as slicing criteria. The reason for this choice is that loop headers provide challenging slicing points, since they entail control dependences. More precisely, for each program to be sliced, we choose as slicing criterion its last loop header, that is the statement closest to the exit node in the CFG of the program. By choosing the last loop header, we thus obtain larger slices. Indeed, the last loop in the RTL representation may be nested inside other loops, and also often depends on several preceding statements.

The construction of the PDG is the most time consuming step of our program slicer. Thus, in order to show the differences between

Program	#Cloc	#RTL	#Edges
mcf	1,378	1,747	1,933
lzss	1,107	1,848	2,063
lzw	1,087	1,871	2,085
arcode	1,180	2,212	2,457
nsichneu	2,361	4,280	4,904
hmmer	6,870	10,572	11,809
papabench	1,370	11,435	12,135
oggenc	48,178	16,451	17,813
gzip	4,386	92,566	101,756
bzip2	5,026	93,929	101,556

Figure 11. Benchmark programs and their sizes in C source lines of code (#Cloc), RTL statements (#RTL) and CFG edges (#Edges).

the construction of the PDG and the other steps, we build slices twice. We first build a slice and measure the time to build it, including the construction of the PDG. Then, we build this slice again and reuse the PDG that we built previously. This only serves to illustrate the effect of the reuse of the PDG.

In Figure 12, we present the computation times (in seconds) taken for the different steps of the program slicer. The #SL column indicates the size of the slice. It is roughly proportional to the size of the RTL code for most programs, except for `nsichneu`. The precision in this case is due to the fact that `nsichneu` contains a single loop with a statically predictable iteration count.

The following three columns in Figure 12 (called CDG, DDG and Obs) contain the computation times related to the slice construction: control dependences, data dependences and next observable vertices. The computation of data dependences is the most expensive step, mainly because there is a great deal of RTL variables in programs. These computation times show that the time necessary to compute next observable vertices is insignificant with respect to the PDG construction.

To illustrate the benefit of using a PDG to compute slices, we present in columns called 1st Slice and 2nd Slice the times taken to compute a slice, first including the PDG construction and then reusing the previously built CFG. In both columns, the slice construction time includes a PDG traversal to compute the slice, the computation of next observable vertices and the slice builder yielding the final executable sliced program.

Figure 12 also shows the times related to the different validation steps performed during program slicing: computation of relevant variables (RVs column), validation of next observable vertices (Val

Obs column) and validation of relevant variables (Val RVs column). The T_{val} column sums these values to obtain the time due to validation. Note that the displayed times are related to the first slice only. From the table, we can see that computing and validating relevant variables is much cheaper than computing control dependencies. In large programs, such as `bzip2`, validation takes less than 1/10 of the time used for computing data dependences.

Imprecision in the larger slices is mainly due to lack of pointer analysis and dynamic (e.g. input-dependent) data dependencies. For instance, several compression algorithms execute loops whose number of iterations depend on the input, not known statically.

The Total column indicates the entire time taken for program slicing, from the construction of PDG to the validation of slices. The last column displays the percentage of time used by the validator with respect to the total execution time. For large programs, the time taken during validation is not significant ($\leq 5\%$ for the two largest programs, `gzip` and `bzip2`), which confirms that it scales for realistic programs.

8. Related Work

Ranganath, Amtoft *et al.* [23, 2] developed paper-and-pencil proofs of program slicing, introducing the notion of observable vertices. Their main concern was to deal with generalized programs, having several or no end nodes. Ranganath *et al.* proved slicing soundness by weak bisimulation, dealing with infinite behaviors. Amtoft extended the proof, obtaining a smaller slice by using a weak simulation.

Our work is integrated in the formally verified compiler CompCert [17]. It is a realistic compiler than relies on several standard program transformations and analyses but currently, CompCert does not handle neither control nor data dependence analysis. CompCertSSA [4] is an extension of CompCert with a Single Static Assignement (SSA) form. This representation is computed thanks to a control dependence analysis that builds a dominance frontier [11], and using a dominator tree. However, the whole transformation is performed with an efficient external tool and is *a posteriori* validated. The validation algorithm is dedicated to dominance frontier and could not be reused in the context of this work for the validation of our PDG.

In [7], program slicing is used to improve the precision of a WCET-oriented loop bound estimation by reducing the number of program states considered by the loop bound estimation without affecting the number of loop iterations. The formalization focused on the properties related to loop bound iterations (e.g. using an instrumented semantics with iteration counters) and due to relatively limited size of the WCET-oriented benchmarks, it did not focus on scalability and efficiency of the validator. The axiomatization in this present paper is more abstract and the implementation has been redesigned for efficiency. We also explain a different correctness theorem for program slicing, which does not involve iteration counters or relevant variables.

There have been several works on mechanized verification of static analyses. They are mostly all based on classical dataflow frameworks: Klein and Nipkow instantiate such a framework for inference of Java bytecode types [15]; Coupet-Grimal *et al.* Delobel [12] and Bertot *et al.* [5] for compiler optimizations, and Cachera *et al.* [8] for control flow analysis. We reuse the Compcert framework for our reachable definition analysis. We are among the first to give an empirical evaluation of the efficiency of the extracted analysis. Other verified analyses [6, 21] are based on a widening and narrowing fixpoint iteration scheme that is not directly related to the technique we use in this paper.

The construction of PDG requires a post-dominance analysis. Zhao and Zdancewic [32] have formalized in Coq a fast dominance computation based on Cooper-Harvey-Kennedy algorithm [11]. This is an interesting and non-trivial proof effort but we believe that a direct verification of the PDG construction would still require an important extra effort, even if we were reusing their work. Indeed, we would need to reason on the (post-)dominator tree they produce. Their work is integrated in the Vellvm verified compiler [33, 31] which performs verified program transformations inside the LLVM compiler. As far as we know, none of these verified transformations manipulate dominator trees.

Wasserrab [27], in the context of non-interference of information flow proposes an Isabelle/HOL formalization of the proof of program slicing described by Amtoft [2]. His slicer operates over an abstract language-independent layer, that is then instantiated on the operational semantics of the Jinja framework. This proof does not concern the actual algorithm used to compute the PDG itself. Indeed, Wasserrab formalizes in a relational (non-constructive way) a PDG and its use to produce program slices [28]. This axiomatization does not enable the direct production of executable code and therefore does not take into account efficiency aspects of an executable instantiation of the specification. For instance, control dependences are formalized in terms of paths in the CFG, using existential quantification, meaning that a direct translation of the specification into executable code would be unbearably inefficient (e.g. it would be necessary to examine all possible paths in the CFG of the program).

Our formalization, on the other hand, focuses on the components that are essential for the proof: relevant variables and next observable vertices. Our approach is centered on the executability and efficiency aspects of these components, and we deliberately avoid formalizing the PDG in order to take advantage of efficient PDG-building algorithms without the burden of their proof. Our axiomatization of relevant variables and next observable vertices based on local reasoning leads to a constructive algorithm for the validation of a program slice, which is not present in the the works of Amtoft and Wasserrab.

9. Conclusion

We presented a validator for program slicing, formally verified in Coq. This validator enables the computation of a program slice using efficient untrusted code, with the final result being validated in a fraction of the time necessary to compute the slice. We implemented a PDG-based program slicer and then validated its result on a series of realistic C benchmarks. The experimental results confirm the efficiency of this approach, while the formal proof ensures that the computed slice is correct.

As a further work, we would like to combine our formally verified program slicer with the Verasco abstract interpreter [14]. Our program slicer would benefit from the Verasco alias analysis to perform a more fine-grained data dependency analysis of memory accesses, while Verasco would be able to extract program slices in order to improve its precision when reporting memory safety alarms to the user.

Acknowledgments

This work is supported by Agence Nationale de la Recherche, grant ANR-11-INSE-003.

References

[1] *Companion website*. http://www.irisa.fr/celtique/ext/slicing.

[2] T. Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Information Processing Letters*, 106(2):45–51, 2008.

[3] J. B. Barros, D. da Cruz, P. R. Henriques, and J. S. Pinto. Assertion-based slicing and slice graphs. *Formal Aspects of Computing*, 24(2):217–248, 2012.

		Construction Times (s)					Validation Times (s)					
Program	#SL	CDG	DDG	Obs	1st Slice	2nd Slice	RVs	Val Obs	Val RVs	T_{val}	Total	T_{val}/Total
nsichneu	3	0.15	1.17	0.05	1.37	0.05	0.01	0.05	0.01	0.06	1.54	4%
lzw	871	0.03	0.05	0.01	0.09	0.01	0.02	0.00	0.02	0.02	0.15	14%
lzss	897	0.03	0.07	0.01	0.12	0.02	0.01	0.00	0.02	0.02	0.16	12%
arcode	1,034	0.04	0.04	0.01	0.10	0.01	0.02	0.00	0.01	0.02	0.16	12%
mcf	1,379	0.03	0.10	0.01	0.14	0.02	0.01	0.00	0.03	0.03	0.21	16%
hmmer	7,932	0.82	4.59	0.11	5.54	0.19	0.13	0.04	0.90	0.94	6.87	14%
papabench	9,816	0.95	0.73	0.09	1.81	0.25	0.12	0.05	0.09	0.14	2.23	6%
oggenc	12,996	1.84	10.18	0.22	12.27	0.24	0.34	0.10	0.99	1.09	14.03	8%
gzip	65,473	50.61	125.62	3.56	180.66	3.77	3.91	1.10	2.35	3.45	191.37	2%
bzip2	79,799	52.73	96.37	1.46	150.98	1.68	2.91	0.55	7.19	7.74	164.35	5%

Figure 12. Computation times for the construction and validation of program slices. Two identical slices are performed to illustrate the reuse of the PDG. The columns in the left are associated to the slice construction, while the columns in the right are associated to its validation. The percentage of time spent in validation is presented in the last column.

[4] G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end - static single assignment meets CompCert. In *European Symposium on Programming (ESOP)*, volume 7211 of *LNCS*, pages 47–66. Springer, 2012.

[5] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Proc. of TYPES 2006*, volume 3839 of *LNCS*, pages 66–81. Springer, 2006.

[6] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Static Analysis Symposium (SAS)*, pages 324–344, 2013.

[7] S. Blazy, A. Maroneze, and D. Pichardie. Formal verification of loop bound estimation for WCET analysis. In *Verified Software: Theories, Tools, Experiments (VSTTE 2013)*, LNCS, pages 281–303. Springer, 2013.

[8] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[9] T. CompCert development team. The CompCert formally verified compiler, version 2.4, 2008-2014. http://compcert.inria.fr.

[10] S. Conchon, J. Filliâtre, and J. Signoles. Designing a generic graph library using ML functors. In *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007*, pages 124–140, 2007.

[11] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical report, Rice university, 2000.

[12] S. Coupet-Grimal and W. Delobel. A uniform and certified approach for two static analyses. In *Proc. of TYPES 2004*, volume 3839 of *LNCS*, pages 115–137, 2004.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[14] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. Formal verification of a C static analyzer. In *42nd symposium Principles of Programming Languages (POPL)*, 2015. To appear.

[15] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[16] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[17] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[18] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[19] G. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, 2000.

[20] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[21] T. Nipkow. Abstract interpretation of annotated commands. In *Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 116–132. Springer, 2012.

[22] C. Okasaki and A. Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[23] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[24] T. Reps and W. Yang. The semantics of program slicing and program integration. In *TAPSOFT'89*, pages 360–374. Springer, 1989.

[25] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[26] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages (POPL)*, pages 17–27. ACM Press, 2008.

[27] D. Wasserrab. *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, Oct. 2010.

[28] D. Wasserrab and A. Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in Isabelle/HOL. In *TPHOLs*, volume 5170 of *LNCS*, pages 294–309. Springer, 2008.

[29] M. Weiser. Program slicing. In S. Jeffrey and L. G. Stucki, editors, *Int. Conf. on Software Engineering (ICSE)*, pages 439–449. IEEE Computer Society, 1981.

[30] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

[31] J. Zhao, S. Nagarakatte, M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 175–186, New York, NY, USA, 2013. ACM.

[32] J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Certified Programs and Proofs - Second International Conference, CPP 2012*, pages 27–42. Springer, 2012.

[33] J. Zhao, S. Zdancewic, S. Nagarakatte, and M. Martin. Formalizing the LLVM intermediate representation for verified program transformation. In *39th symposium Principles of Programming Languages (POPL'12)*, pages 427–440. ACM, 2012.