Static Analysis Algorithms and Their Logical Verification

Petar Vukmirović

August 15, 2017

Abstract

Static analysis represents a set of techniques used to reason about program behavior without having to execute the program. This survey reviews the most important results and the reach of the state-of-the-art implementations. Algorithms described include a range of code-optimizing algorithms and analysis of heap data structures, whereas the implementation will be discussed using the CompCert compiler as an example. Furthermore, since the analyses are used in safety-critical applications their proofs of correctness are discussed as well.

1 Introduction

Static analysis represents a set of techniques used to reason about program behavior without having to execute the program [AM17]. This kind of techniques is by no means a necessity to produce an executable binary out of the source code, but it can facilitate many parts of this process. Namely, static analysis techniques are widely used in enhancing the performance of the compiled binaries (i.e. optimizing the code). They can also be used to show that the code satisfies certain property devised by a programmer (i.e. verifying the program).

The static analysis techniques are applicable to a wide range of computer science fields. In particular, the applications range from simple ones, such as auto-complete functions in IDEs and text-editors, over more important ones such as inferring properties about source code to assess its quality automatically, to the critical ones, such as optimizations of source code in compilers.

Throughout the years, an extensive corpus of scientific papers with the theme of static analysis has emerged. The role of this survey is to give an overview of those papers, describing their results and emphasizing their impact on further developments in the field. However, since static analysis techniques are used in safety-critical environments, formal proofs of correctness and termination of the static analysis algorithms is discussed as well.

One of the major breakthroughs was the formulation of a general framework to solve many static analysis problems. This framework is known as Kildall's algorithm (Section 3) and all of the concrete static analysis tools will be described as instances of this algorithm.

The concrete static analyses described in this survey are elimination of constants that can be calculated at compile time (constant propagation, Section

4.1), identifying expressions whose results can be saved and reused later (common expression elimination, Section 4.2) and identifying variables that will be used at a later program point (live variables, Section 4.3).

Furthermore, the implementations and proofs of correctness of those algorithms are discussed, following the case of a well-known certified compiler, CompCert (Sections 4.4 and 4.5).

To show that static analysis has more use cases than just facilitation of source code optimization, shape analysis is presented in Section 5. This analysis is able to infer useful properties about heap-residing data structures. Moreover, the kind of shape analysis described here is general and can be extended for many applications.

Lastly, a framework for showing functional correctness instead of only accordance to desired properties is shown in Section 6. This framework is not yet implemented as of time of writing. However, it represents an interesting approach to reduce domain-specific proof effort.

2 Background

Control flow graphs (CFGs). Given a program, the paths that can be taken in a program execution can be visualized by assigning every statement a node and connecting this node with any other that could possibly follow it. This visualization is obtained by creating a graph from nodes and their relations denoted CFG $G = (N, E)$ where N is the set of nodes and $E \subseteq N \times N$ is the node accessibility relation. The assignment of statements to nodes will be left vague intentionally since for different languages different approaches will be taken (e.g. sometimes the whole block can be a node and sometimes an expression will be a node).

Approaches in validation. Most of the algorithms presented in this survey are somehow integrated in compiler toolchains. The compiler authors take different approaches to show that the compiled code is correct with respect to the semantics of the source code. Three major approaches can be distinguished [Ler09]:

Verified compiler is a compiler that is accompanied with the proof that if it compiles the code without errors, the generated binary will correspond to the semantics of the source code. Usually, the correspondence means that high-level code and low-level code have the same *observable behavior*. With observable behavior termination, divergence and the set of input-output operations (i.e. visible trace) is normally assumed.

Translation validation is the approach where unverified compiler's output is checked a posteriori by the verified validator that checks if the binary the compiler has output is compliant with the source semantics.

Proof-carrying code represents the approach where the compiler is unverified, but it generates the proof that compiled code satisfies certain properties (e.g type correctness, memory safety). This proof is usually a proof term that can be checked by a general-purpose proof assistant.

Small-step semantics. The small-step semantics for a programming language explains the effect on the execution environment of each of the programming language constructs (e.g. assignments, memory allocations) and how constructs of the programming language are evaluated (e.g. evaluation of branching conditions, arithmetic expressions). Usually, it is described by the set of rules that explain how expressions are evaluated or how statements alter the execution state and what statement should follow the currently executing one.

The execution state σ models the memory of the executing program and depending on the memory model it can include stack, heap, code segments of the memory or it can be simplified and consist only of a mapping from variable names to values.

The transition relation is usually given as a binary relation on pairs (S, σ) where S is a statement. The transition is defined by the rules for each statement and is effectively defining the statement's behavior [NNH99]. A special case of transition is where the currently executing statement S terminates in this very step so the pair (S, σ) makes transition to a new state σ' without having a new statement S' to execute.

Consider a simple language whose syntax includes an assignment statement and a conditional statement. Also, its memory model simply maps variable names to integers and evaluation of an expression is given by function V. Rules for the two mentioned statements can be defined as follows:

$$\frac{}{(\texttt{x} := e, \sigma) \to \sigma[x \leftarrow V(e, \sigma)]}$$

$$\frac{V(e, \sigma) = 1}{(\texttt{if } e \texttt{ then S}_1 \texttt{ else S}_2, \sigma) \to (\texttt{S}_1, \sigma)}$$

$$\frac{V(e, \sigma) = 0}{(\texttt{if } e \texttt{ then S}_1 \texttt{ else S}_2, \sigma) \to (\texttt{S}_2, \sigma)}$$

Three-valued logic. Three-valued logic is an extension of classical logic that includes the notion of an *unknown* truth value (1/2), in addition to the usual true (1) and false (0) values. This kind of logic is useful when there is a need to express uncertainty about a property of an analyzed program. The logical operators $\wedge$, $\vee$ and $\neg$ are interpreted as follows:

$\wedge$	0	1	1/2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

$\vee$	0	1	1/2
0	0	1	1/2
1	1	1	1
1/2	1/2	1	1/2

$\neg$	
0	1
1	0
1/2	1/2

Table 1: Interpretation of formulas

Furthermore, for $l_1, l_2 \in \{0, 1, 1/2\}$, the *information order* $l_1 \sqsubseteq l_2$ is defined as $l_1 \sqsubseteq l_2 \;\Leftrightarrow\; l_1 = l_2 \vee l_2 = 1/2$.

3 Kildall's Algorithm

The development of high-level programming languages in the 1960s and 1970s has brought many application and language specific static analysis techniques to light. Some of these were rudimentary analyses of straight sequences of instructions, whereas the others took control-flow branching into account [Kil73]. In 1973, a generalization of numerous techniques related to static analysis, that is considered very applicable and attractive even by today's standards was created and is known as Kildall's algorithm.[1] In this section the intuition behind the algorithm will be given as well as pseudocode for it, following the original paper [Kil73]. This algorithm has been widely adopted both academically, where it forms the basis of many advanced static analysis algorithms, and in industry-oriented applications, where it is employed in optimization algorithms of certified compiler CompCert [Ler09] (which is planned to be applied in avionic software).

Kildall's algorithm attaches information to each statement of the program source code to facilitate its further optimization. The input for the algorithm is a control flow graph corresponding to the program to be analyzed, and the output is the optimization information attached to every node of the CFG. Furthermore, the algorithm takes as input a set of special nodes, *entry nodes*, from which the analysis will start. Entry nodes are given the user-predefined initial optimization information. This assumption is reasonable and in general there would be only one entry node (namely, the function or program start point).

The analysis starts by adding the entry nodes with associated optimization information to the working list WL. Then, an arbitrary node w from WL is removed and chosen as the node to be processed. For the node to be processed, the optimization information, which is a function of node properties and currently associated optimization information, is calculated. This information is then propagated to w's successor nodes. If this newly obtained piece of information changes the one currently present at a successor node w', then the node w' with the updated optimization information is added to WL. The analysis is complete when WL is empty.

In other words, the analysis is pushing the processed data to the successors and when no new conclusions can be drawn, the analysis stops. However, the termination of the algorithm can be uncertain if the informal notion of the type of the optimization data is not detailed. Moreover, the operation that changes the data has to be formally defined in such a way that for each node, this operation can be performed finitely many times. To formally state the algorithm, previously described notions will be formalized as it has been done in the original work by Kildall [Kil73].

Let P be the finite set that contains all possible (valid) optimization pools (informally optimization information attached to the node), and $\sqcap$ be the commutative and associative binary operation on P. This operation gives rise to partial ordering $\leq$ defined as

$$\forall x, y \in P,\ x \leq y \ \Leftrightarrow\ x \sqcap y = x$$

Similarly, $<$ can be defined as $\forall x, y \in P,\ x < y \ \Leftrightarrow\ x \sqcap y = x$ and $x \neq y$.

[1] The algorithm was named after its author, Gary Kildall, a famous computer scientist

Additionally, the set P is assumed to contain the zero element $\bot$, such that $\forall x \in P,\ \bot \leq x$. The augmented set P' is defined to be equal to the set P with element $\top$ added, such that $\top \notin P$ and $\forall x \in P,\ x \sqcap \top = x$. Then, it easily follows that $\forall x \in P,\ x < \top$. Now, we can state the pseudocode for the algorithm and discuss the informal termination proof.

Let $\epsilon \subseteq S \times P$, where $S \subseteq N$ is the set of entry nodes, be the given set of initially marked nodes in the sense of the previous informal discussion of the algorithm, and let WL be defined as the subset of $N \times P$ where N is the set of nodes of CFG $G = (N, E)$. Additionally, every node $x \in N$ has its own storage for optimization data P_x. Finally, we set f to be a function that calculates the optimization information based on the node and currently available information, $f : N \times P \to P$, which satisfies the homomorphism property with respect to operation $\sqcap$, $f(x \sqcap y) = f(x) \sqcap f(y)$. The pseudocode is listed in Algorithm 1.

Algorithm 1 Kildall(ϵ, $G = \{N, E\}$)

1: $WL \leftarrow \epsilon$
2: $P_x \leftarrow \top$, for all x
3: **while** $WL \neq \emptyset$ **do**
4: Remove an arbitrary pair $w = (x, P_i)$ from WL
5: **if** $\neg(P_x \leq P_i)$ **then**
6: $P_x \leftarrow P_x \sqcap P_i$
7: $WL \leftarrow WL \cup \{(y, f(x, P_x)) \mid (x, y) \in E\}$
8: **end if**
9: **end while**

The algorithm presented above completely matches the intuitive description that it follows, but it allows for formal reasoning about its termination. Namely, it is obvious that the only step that can lead to divergence is step 7 in which one or more nodes are added to the worklist if the condition $P_x \leq P_i$ is not fulfilled. Unfolding the definition of $\leq$, we get $P_x \neq P_x \sqcap P_i$. However, using the definition of $\leq$ once more $(P_x \sqcap P_i) \sqcap P_x = P_x \sqcap P_i$, which gives us $P_x \sqcap P_i \leq P_x$ and $P_x \sqcap P_i \neq P_x$, so we obtain $P_x \sqcap P_i < P_x$. The last equation that we obtained can be interpreted as decreasing of the expression $P_x \sqcap P_i$ with respect to P_x, which can be done only finitely many times since P is a finite semi-lattice with the minimal element $\bot$. Therefore, step 7 can be performed only finitely many times, which means that the algorithm will eventually terminate.

4 Applications to Compiler Design

The algorithms presented in this section are vital for creating optimizing compilers. The static analysis performed by these algorithms can facilitate generation of code that runs considerably faster than the non-optimized version. The algorithms will be presented by instantiating the semi-lattice $(P, \sqcap)$ used in the general description of Kildall's algorithm and by giving the intuition behind a certain choice of semi-lattice instantiation.

4.1 Constant Propagation Algorithm

The goal of the constant propagation algorithm (CPA) is to determine which variables in the program source code can be calculated at compile time, which gives opportunity to the compiler to remove these calculations from the generated executable and replace them with *propagated* constants.

For example, in the Java code excerpt

```
1  int a = 0;
2  int b = 10;
3  int i = 0;
4  while(i < 10){
5      int c = a + i;
6      i = i + 1;
7  }
```

at line 3 every occurrence of variable `b` can be changed for 10, and, similarly, at line 6, every occurrence of `a` can be substituted with 0 (had they occurred at the mentioned lines).

Let $U = V \times C$ where V is the set of variables and C is the set of values that can be assigned to those variables. Then, we can instantiate set P to be the power-set of U, that is the set 2^U, and $\sqcap$ operation to be set intersection ($\cap$). Then, $\bot$ is defined to be $\emptyset$ and $\top$ is defined to be $U \cup \omega$ where ω is an arbitrary element not in U. It is easy to verify that previous definitions satisfy constraints that are formally put on them.

Now we need to define the optimization function f in such a way that when Kildall's algorithm terminates, f assigns the set of variables that can be evaluated at compile time to each CFG node. Assuming x is an assignment node (if it is not f acts like an identity function): $(v, c) \in f(x, P_i)$ if and only if (a) v is not being assigned to in x and $(v, c) \in P_i$ or (b) v is being assigned to in x, but its value can be calculated using constants in P_i.

In other words, with this instantiation, Kildall's algorithm will calculate the set of constants available up to a certain point in the source code. Based on this information and the information of the node to be processed it will generate (possibly) updated set of constants. Then, this set of constants will be sent to the successors until the estimated constants sets no longer change – which means that the set of constants is known for a certain node.

As an illustration of one run of CPA, a control flow graph for Java code excerpt given at the beginning is shown in Figure 1. Statements that do not change sets of constants are abstracted away and node labels are given next to the node.

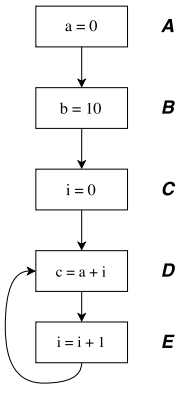


Figure 1: Control flow graph for Java code excerpt

The run of the algorithm is given in Table 2, where each statement that mutates the state of the algorithm is explained. Initially WL equals $[(A, \emptyset)]$, and for every node N, the store P_N equals $\top$, that is $\{a, b, c, i\} \times I$ where I is the set of all integer values in Java.

Line	Update	Explanation
5	$WL \leftarrow []$	$(A, \emptyset)$ removed from WL
9	$P_A \leftarrow \emptyset$	Update of the initial value for A
10	$WL \leftarrow [(B, \{(a, 0)\})]$	Propagate constants from A to all adjacent nodes (just B)
5	$WL \leftarrow []$	$(B, \{(a, 0)\})$ removed from WL
9	$P_B \leftarrow \{(a, 0)\}$	Update of the initial value for B
10	$WL \leftarrow [(C, \{(a, 0), (b, 10)\})]$	Propagate constants from B to all adjacent nodes (just C)
5	$WL \leftarrow []$	$(C, \{(a, 0), (b, 10)\})$ removed from WL
9	$P_C \leftarrow \{(a, 0), (b, 10)\}$	Update of the initial value for C
10	$WL \leftarrow [(D, \{(a, 0), (b, 10), (i, 0)\})]$	Propagate constants from C to all adjacent nodes (just D)
5	$WL \leftarrow []$	D and corresponding set P_i removed from WL
9	$P_D \leftarrow \{(a, 0), (b, 10), (i, 0)\}$	Update of the initial value for D
10	$WL \leftarrow [(E, \{(a, 0), (b, 10), (i, 0), (c, 0)\})]$	Analogous to previous.
5	$WL \leftarrow []$	E and corresponding set P_i removed from WL
9	$P_E \leftarrow \{(a, 0), (b, 10), (i, 0), (c, 0)\})$	Update of the initial value for E
10	$WL \leftarrow [(D, \{(a, 0), (b, 10), (i, 1), (c, 0)\})]$	Notify D that i is changed.
5	$WL \leftarrow []$	D and corresponding set P_i removed from WL
9	$P_D \leftarrow \{(a, 0), (b, 10)\}$	$(i, 0)$ is removed from P_D when merged with incoming information from E
10	$WL \leftarrow [(E, \{(a, 0), (b, 10))\})]$	i could not be calculated from given constants
5	$WL \leftarrow []$	E and corresponding set P_i removed from WL
9	$P_E \leftarrow \{(a, 0), (b, 10)\}$	$(i, 0)$ and $(c, 0)$ are removed from P_E
10	$WL \leftarrow [(D, \{(a, 0), (b, 10))\})]$	Analogous to previous
5	$WL \leftarrow []$	D and corresponding set P_i removed from WL
3	–	Algorithm halts since no changes are possible.

Table 2: One run of CPA

4.2 Common Expression Elimination

Common expression elimination is an optimization technique that relies on the reasonable assumption that there is less performance penalty from fetching already cached expression result, than reevaluating it [Kil73]. The algorithm will be introduced by an example. In the pseudocode excerpt

```
1  x = a + b;
2  y = (a + b) + c;
3  print(x + c);
```

several (sub)expressions can be cached for improving performance. For example, in line 2 of the pseudocode, the `a + b` subexpression does not have to be recalculated. Similarly, expression arising in line 3 is redundant as well as it has the same value as the one in line 2. Again, it is the main task to find an appropriate instantiation for the lattice, as well as to find the optimization function that will find all common expressions.

The interpretation that is convenient for this problem is the interpretation by equivalence classes of expressions. Informally, the idea is to define the notion of expression equivalence as an easily computable approximation of predicate determining whether two equations will evaluate to the same value at the run time. This informal description leaves a lot of space for further specialization of equivalence notion, and the quality of the algorithm depends greatly on how well we are able to determine which expressions evaluate to the same value. It is important that equivalence has to be conservative in the sense that if it determines that two expressions are equal then they necessarily have to evaluate to the same expression.

For the running example, the set of equivalence classes would be

$$\{\{a\}, \{b\}, \{c\}, \{x, a+b\}, \{(a+b)+c, y, x+c\}\}$$

However, the meet operation $\sqcap$ has a more tedious definition. Let C be the set of all members belonging to some class in both of sets of equivalence classes P_1 and P_2. For a $c \in C$ we define $P(c)$ to be the set of classes that c belongs to. Then $P = P_1 \sqcap P_2$ is defined as the set P satisfying $\forall c \in C, P(c) = P_1(c) \cap P_2(c)$. For example, if $P_1 = \{\{a, b\}, \{d, e, f\}\}$ and $P_2 = \{\{a, c\}, \{d, f, g\}\}$, then C is $\{a, d, f\}$ and $P_1 \sqcap P_2$ is $\{\{a\}, \{d, f\}\}$. Now, it is easily shown that this definition of $\sqcap$ satisfies the constraints imposed by Kildall's algorithm.

Given this semi-lattice, the optimization function $f(N, P)$ is defined as the following computation [Kil73]:

1. For every subexpression e in N determine if it is present in any class of P. If it is, then it is redundant.

2. Otherwise, create new class in P which consists of e and all of the expressions in program CFG that have operands equivalent to those of e (i.e. those expressions will evaluate to same value as e).

3. If N contains an assignment `d :=` e, remove from P all expressions containing `d` as a subexpression (intuitively, from this node on, previous value of `d` is invalidated). For each expression e' in P containing e as a subexpression, create e'' with `d` substituted for e, and put e'' in the class of e'.

The most important part of f is step 2, which is called *structuring*. It gets the most out of failing match of (sub)expression e by examining expressions observed so far and checking if any of those have operands equivalent to e's operands. The structuring is what put the expressions $(a + b) + c$ and $x + c$ in the same equivalence class since $+$'s operands are equivalent.

4.3 Live Expressions

Many modern compilers optimize their code by assigning the temporaries and variables to the registers in a way that decreases the need to retrieve the temporaries or the variables from the memory. The optimization consists of speeding up memory access since registers are assumed to provide faster access speeds. However, determining which variables to keep in registers at which point of execution is not an easy task and needs a lot of information about the variable usage to be useful. For each CFG node N we define the set of *live* variables for this node as the variables that might be referenced in a node subsequent to N [Kil73]. Otherwise, the variable is *dead*. Similarly, the expression is called *live* if all variables appearing in the expression are *live*, and *dead* otherwise. This classification of expressions can be used as the hint for further register allocation passes.

Intuitively, to get the needed information we should start from the exit nodes of the CFG. For them the set of live expressions is empty. Furthermore, it is intuitively clear that we can work our way up from the the end nodes processing the assignments in the end nodes. In this way the information about what variables are used in nodes following a certain node are obtained by collecting them from descendants in reverse direction. For this reason, the input CFG used in this analysis will have its edges reversed.

Formally, the carrier set of the lattice is the set of all possible (partial) expressions for the program. Following the algorithm's idea, the propagated data should be the union of the live expressions, so we choose $\sqcap$ operation to be $\cup$, $\bot$ to be the extended set P' and $\top = \emptyset$.

Finally, the optimization function $f(N, P)$ is:

1. If a statement in N is the assignment of e to variable `d` then remove every expression that has `d` as subexpression from P. The reasoning behind this is that since `d` is overwritten here it is considered dead from this moment on.

2. Add all of the expressions appearing in N to the set P. The intuition behind this operation is that those expressions are used here and hence, considered live. The altered P is the result of function f.

When the algorithm terminates, every node will be assigned the set of live expressions for that node.

4.4 Implementation of Verified Algorithms in Coq

CompCert is a formally verified compiler developed by Leroy and his team that compiles large portion of standard C to PowerPC binaries [Ler09]. The goal was a compiler that is not only formally verified, but that is also performing advanced optimizations, which enables it to be used in real-life scenarios, where

high assurance that the compiler is correct is needed (avionics and medical industry, for example).

Most of CompCert is written in the Coq proof assistant, with the executable code obtained by Coq extraction module directly from its specification. However, some parts of the compiler, including some of the optimizations, had to be programmed in OCaml and then integrated with the other code. Since CompCert has end-to-end verification, which means that all of the compiler passes, up from the C source code, to the generated PowerPC binary are formally verified to respect the semantics of original code, this external OCaml code must be verified using translation validation.

As it is usual with the compilers, CompCert preprocesses the source code to the point where the code is simplified enough so that CFG with very simple operations can be created. By simple operations expressions without subexpressions are assumed, as well as reducing switch statements to simpler forms (e.g. jump tables or conditional statements). The following discussion will be based on one of the simpler languages in CompCert pipeline called `RTL`.

Full syntax of `RTL` is given in the CompCert back-end paper [Ler09] that also gives an exhaustive overview of other intermediate languages. We will focus on a subset of RTL from Bertot et al.'s previous work on proving compiler optimizations correct [BGL04]. `RTL` code (C) is a mapping between points in program (which correspond to nodes in CFG) and instructions (I). Each instruction is a tuple of instruction description d and the vector of successor nodes $\overline{s}$. This closely resembles the CFG analogy established in Section 2. Descriptions of the instructions correspond to the operations available at the low level – some examples of instruction descriptions are given below:

$$d ::= \texttt{Inop} \mid \texttt{Iop}\ op\ \overline{r}\ r \mid \texttt{Iload}\ \texttt{mc}\ \texttt{am}\ \overline{r}\ r \mid \texttt{Istore}\ \texttt{mc}\ \texttt{am}\ \overline{r}\ r \mid \cdots$$

The meaning of `Inop` is "no operation", which might be useful when actual code is optimized away. `Iop` performs the operation op on arguments $\overline{r}$ and stores the result in r, where op ranges over various arithmetic, comparison, conditional and other operations. Since `RTL` differentiates between signed and unsigned integers, floating-point numbers and pointers there are appropriate versions of all of the operations that match the argument types. For the same reason, memory load (`Iload`) and store (`Istore`) operations take `mc` arguments, which will determine what chunk of memory the operation will work on. Additionally, various underlying addressing mechanisms (`am`) are supported. According to the memory model, memory is split into blocks and each address consists of block identifier and offset inside a block.

The semantics of `RTL` is given as small-step operational semantics, extended with big-step semantics where it could contribute to easier proofs of semantics preservation. The state S consists of S_r, which maps registers r to their values v, pointer to the current top of the stack S_{stk}, S_{mem}, which maps pointers to blocks, and finally S_{id}, which maps global variables x to their values v. Big-step semantics rule is represented with the line labeled with the function whose results is explained with this rule.

Since the number of rules for `RTL` is overwhelming, a selection of rules that gives a feeling for the way they are described is given:

$$\frac{}{((\texttt{Inop}, [s]), S) \rightarrow (s, S)}$$

$$\frac{\texttt{eval}_{op}(S, op, \overline{r}) = v \qquad \texttt{CheckType}(r_d, v)}{((\texttt{Iop}\ op\ \overline{r}\ r_d, [s]), S) \rightarrow (s, S[r_d := v])}$$

$$\frac{f_C(c_1) = I \qquad (I, S_1) \rightarrow (c_2, S_2) \qquad (c_2, S_2) \xrightarrow{f} (v, S_3)}{(c_1, S_1) \xrightarrow{f} (v, S_3)}$$

The simplest presented rule is the one corresponding to "no operation" which leaves the execution in the same state. The `Iop` rule is more complex since it includes the evaluation meta-function `eval` and checking of return type. Lastly, the big-step semantics rule describes how a function is being evaluated. Namely, if I corresponds to the instruction at c_1 in f and if it terminates with return value v, then we obtain the big-step at the bottom of the third rule.

Now that the syntax, semantics and memory model of `RTL` is described, an approach to showing the correctness of optimizations will be given. The very same version of Kildall's algorithm that is presented in this section is given in the Coq source code of the CompCert. To fully exploit the generic nature of Kildall's algorithm it is given as a special module parametrized by semi-lattice L, which corresponds to the optimization pool, the optimization function *transf*, and the CFG in form of mapping between CFG nodes and their successors.

As previously mentioned, Kildall's algorithm is terminating if we impose some constraints on both the semilattice (e.g. the order relation being well-formed) and the optimization function (e.g. monotone and homomorphism). Concerning this point there is some confusion in CompCert literature written so far by the authors. Paper that primarily focuses on formalizing the static analysis and optimizations [BGL04], notes that the termination is to be expected since "code graph for a function is always finite and the domain of the analysis is a well-founded semilattice". However, closer inspection of recent CompCert source code and very detailed CompCert back-end paper [Ler09] reveals that there is no such constraint. Moreover, since the solver can fail, the code leaves room for special failure value to be returned.

Kildall's algorithm is implemented in a purely functional manner directly in Coq[2] and is completely compliant to the algorithm described above. However, since termination cannot be guaranteed in this implementation, Kildall's algorithm is bounded to N iterations, where N is a large constant. Additionally, the source code file corresponding to Kildall's algorithm implementation is accompanied with the proof that if Kildall's algorithm finds the fixpoint, it is indeed solution to the optimization problem.

The constant propagation optimization has been implemented in a way that closely resembles the description given in this section. Its lattice is defined as set of mappings from register identifiers to their values or a special value that denotes that register value is unknown ($\top$). This set of mappings is indeed a lattice if the domain of the mappings is finite (which is established by using a finite set of registers) and if the codomain is a lattice (which is proven in a separate Coq file) [BGL04].

The analysis is performed by instantiating the Kildall's algorithm with the lattice and optimization function that behaves just like the one in the pseudocode. Based on the analysis the code is changed so that conditionals that evaluate to constant boolean expressions have one of their branches changed to

[2] `github.com/AbsInt/CompCert`

no-ops, arithmetic operations are changed to their immediate forms and constants are included in addressings instead of variable offset. All of this is done so that the general shape of the CFG is not altered (the nodes are neither added or deleted, just changed), which makes this transformation a morphism over CFG [Ler09].

The proof of semantics preservation is done by showing that register values will always correspond to the ones obtained in static analysis, and then a large and tedious case analysis of instructions is performed to show that under this condition instruction semantics is preserved.

Common expression elimination's instantiation of Kildall's algorithm is technically different from the one present in this section, however it follows the same spirit. Namely, the lattice for this optimization is given as the set of pairs (ϕ, ν), where ϕ is partial function that assigns each register an abstract value number, where value number corresponds to a symbolic expression. On the other hand, ν is a set of equations between value numbers of form $x = op(\overline{x})$ which can be interpreted as abstract value x corresponds to op applied to abstract arguments $\overline{x}$ (another type of equations is present, but since it is too technical, it will be omitted).

This set of equations between different value numbers and their symbolic values is technically different, yet analogous to the equation equivalence pool described earlier in the section. In contrast, the transfer function is similar to its pseudocode counterpart. Since `RTL`'s expressions are fairly simple, the structuring part now reduces to simply checking whether all of the arguments are already assigned a number. Another difference is that this transfer function empties equation set ν before a function call since this could be beneficial to the later register allocation phase [Ler09].

Correspondence of the actual program state with the obtained analysis results is given with the following judgment: if R is register state and M is memory state, then R, M satisfies ϕ, ν, written as $R, M \models \phi, \nu$ if there is a valuation V assigning concrete values to abstract value numbers such that: (1) if $\phi(r) = x$, then $R(r) = V(x)$ and (2) if $x = op(\overline{x})$, then $\texttt{eval}_{\texttt{op}}(op, V(\overline{x})) = V(x)$. In other words, the desired property is that the register values correspond to the values assigned by V and that V also respects operation evaluation.

Given this correspondence, the semantic preservation is shown in a way similar to the constant propagation – with the similar laborious case distinction using the new correspondence criterion.

4.5 Verified Register Allocation

In compiler construction, register allocation refers to the problem of allocating program variables to a sparse set of CPU registers. Usually, the number of registers is smaller than the number of variables in use at a certain program point, so some of them have to be stored in memory. The process of storing some of the variables in memory is referred to as *spilling*.

In CompCert, the target language for register allocation pass is `LTL`. It differs from `RTL` only in details, except for the fact that instead of pseudo-register arguments of operations, in `LTL` the arguments are locations that are either stack locations or PowerPC registers [Ler09].

Before the register allocation can be performed, the set of live variables is determined in the same way as it is described in Section 4.3. When two

variables are alive at the same program point, they must be assigned different memory locations so that their values are not corrupted by manipulations on other variables.

This constraint can be modeled as an *interference graph* in which the nodes are variables that are connected by an edge if and only if they are live at the same time. The problem of allocating registers is equivalent to coloring the graph so that no two connected vertices are colored the same color. Moreover, the number of available colors corresponds to the number of available registers.

Substantial amount of effort has been put into finding efficient algorithms that find (approximate) solutions for graph coloring problem. CompCert uses an advanced *Iterated Register Coalescing (IRC)* algorithm formulated by George and Appel. However, at first, this algorithm has been implemented in Caml and then validated a posteriori, in translation-validation fashion by a validator that has been formally verified in Coq.

In later efforts, Blazy, Robillard and Appel [BRA10] have formally verified the implementation of IRC in Coq. This version of the algorithm and corresponding proof of correctness will be discussed in what follows.

The main observation underlying the IRC algorithm is that K-coloring of the graph can be reduced to K-coloring of the graph with a node of $< K$ degree deleted (*low-degree* node). Namely, since the deleted node was of degree $< K$, if we remove it and recursively color the rest, removed node can be put back and colored appropriately.

However, some graphs have no low-degree nodes. To make up for this, the IRC algorithm leaves the possibility of a node not being colored at all if it is not possible, which corresponds to the phenomenon of spilling defined before. Also, if there is a `move` operation from r_s to r_d, then it would be beneficial to assign both r_s and r_d to the same register, which would make `move` command redundant. Edge from r_s to r_d is called *preference edge*. Preference edges will be used as just a hint telling which variables shall be assigned to the same register.

The algorithm consists of four main operations:

Simplify If there is a low-degree node that is not involved in a preference edge (those nodes are called not *move-related*), remove it and recursively color the remaining graph.

Coalesce If there are two move-related nodes that are low-degree after being merged, combine them, remove the combined node and recursively color remaining graph.

Freeze If there is a low-degree move-related node, remove its preference edges and mark it for simplifying.

Spill If none of the previous operations' preconditions was met, remove a node (that will not be colored afterwards) and recursively color the remaining graph.

In the algorithm the operations are performed in the listed order and performed only if their preconditions are met. The reference pseudocode is given in Algorithm 2.

Algorithm 2 IRC(G)

```
1:  Colored ← Precoloring(G)
2:  if Simplify(G) = (n, G') then
3:      Colored ← IRC(G')
4:      Color n and put it in Colored
5:  else
6:      if Coalesce(G) = ((n1, n2), G') then
7:          Colored ← IRC(G')
8:          Color n1 and n2 the same and put them in Colored
9:      else
10:         if Freeze(G) = G' then
11:             Colored ← IRC(G')
12:         else
13:             if Spill(G) = (n, G') then
14:                 Colored ← IRC(G')
15:                 Leave n uncolored
16:             end if
17:         end if
18:     end if
19: end if
20: return Colored
```

To improve the performance of lookup for appropriate node IRC algorithm keeps four worklists:

Name	Property
`spillWL`	high-degree, non-precolored nodes
`freezeWL`	low-degree, move related, non-precolored niodes
`simplifyWL`	low-degree, non-move related, non-precolored nodes
`movesWL`	set of preference edges

Before each operation, as described in pseudocode of Algorithm 2, the algorithm looks up appropriate list to check if there is a node that fulfills a certain precondition. After an operation is (successfully) performed some of the lists might be changed.

The invariant is that all of the worklists maintain their properties after all of the modifying operations of the algorithm. Most of the proving effort goes in this direction. However, since the algorithm is not defined in terms of structural recursion its termination has to be proven.

Termination is proven by giving a function that bounds the number of recursive calls made to the algorithm. This function is accompanied by the proof that no matter what operation of the algorithm is performed, the number of recursive calls has to become smaller.

Soundness of the algorithm is proven by showing that two nodes connected by an interference edge have to be colored by different colors, all colored vertices belong to the original graph and any color belongs to the original pallet (that is to one of available registers).

Finally, the `RTL` code is transformed to `LTL` code by converting pseudo-registers to locations determined by register allocation algorithm. In two cases,

however, the operation is deemed redundant and thus converted to `nop`: first, if source and destination of `move` operation are assigned the same location and second if the result register of the operation is dead.

5 Shape Analysis

The analyses presented so far are mostly useful as precursors for further optimization passes. No attempt has been made to deal with pointers or dynamically allocated memory (heap-allocated memory) and potential problems that arise with them (dereferencing null pointers or memory leakage for example). Shape analysis is a technique used to reason about properties of heap-residing data structures, whose applications range from bug-finding tools to program verifiers [WSR00].

Throughout the years, researchers have approached shape analysis in different ways. In this paper, an advanced approach to shape analysis by Lev-Ami et al. [LRSW00, WSR00] will be discussed, since it allows for generic shape analysis framework that can be used to answer questions discussed here and possibly many others.

This approach is significant because it is general in terms of being programming language and data structure independent. Furthermore, the authors set their goal to create a tool that is as general for shape analysis as `yacc` is for parser generation.

This is made possible because shape analysis is not performed on non-flexible shape graphs, but on structures that are parametrized by the set of predicates on data structure to be analyzed. When there are new questions about a data structure to be answered, new predicates that model the question can be added.

Independence on programming languages is achieved by performing the analysis on CFG, not on the code itself and parameterizing semantics of each statement by predicate-change formulas.

This kind of shape analysis will be introduced bottom-up, by giving examples of the results of the analysis it performs, and then explaining how those results can be obtained.

Consider the following definition of a linked list data structure:

```
1  typedef struct node {
2    struct node* n;
3    int data;
4  }* List;
```

This data structure is unbounded in number of elements it may contain at any point of program execution. Thus, it is hard to create algorithms that reason about properties applicable to all possible lists. To facilitate reasoning about lists, shape analysis uses an abstraction of the list data type in which some nodes of the linked list are summarized in a single node.

More precisely, concrete lists can be represented as two-valued logical structure (over set of predicate symbols with assigned arity) $S = \langle U^S, \iota \rangle$, that consists of set of individuals (nodes) U^S and interpretation of predicates ι that assigns a function $p^S : (U^S)^k \to \{0, 1\}$ for each k-ary predicate p. The predicates that are inherent to pointer semantics, and are called *core* in the works of Lev-Ami et al. are x, n and sm. Their meaning is given in the Table 3, and they represent the basic properties of the heap memory store.

Predicate	Meaning
$x(v)$	Variable `x` points to heap node v
$n(v_1, v_2)$	Heap node v_1's `n` field points to v_2
$sm(v)$	Node v is a summary node (abstract node that possibly corresponds to several concrete nodes)

Table 3: Core predicates

Similarly, the abstract store is represented by three-valued logical structure $S = \langle U^S, \iota \rangle$, which allows predicates to evaluate to the indefinite value $1/2$. This representation of abstract store is useful since when unbounded data structure is represented finitely some information is bound to be lost. Thus, to account for this loss of data, it is convenient to have the ability to evaluate some predicate as unknown or represent set of *equivalent* nodes as a single summarized node.

To increase the precision of the analysis (the reason for gain in precision will be given later), additional *instrumentation* predicates are defined in terms of core predicates. Their definitions and meanings are given in Table 4.

Predicate	Meaning	Defining formula
$r[n, x](v)$	Following field `n`, node v is reachable from `x`	$\exists v_1.\ x(v_1) \wedge n^*(v_1, v)$
$is[n](v)$	v is pointed to by 2 or more nodes	$\exists v_1, v_2.\ n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$

Table 4: Examples of instrumentation predicates (p^* represents reflexive-transitive closure of predicate p)

To show the way shape analysis abstracts properties of concrete data structures, the abstraction of list $[1, 2, 3, 4]$ will be taken. The original data structure is shown in Figure 2, whereas the abstracted structure is shown in Figure 3. The variable is represented with a rectangle, while the node is represented with a circle. Unlabeled edges represent that the value of predicate x for a variable `x` is 1, labeled edge $c(n_1, n_2)$ means that the value of predicate c for arguments n_1 and n_2 is 1. If a unary predicate evaluates to 1 for a given node, it will be represented inside the circle with its name. All other predicates evaluating to 0 are omitted, and the ones evaluating to $1/2$ are shown with dotted lines. The summary node is shown with dotted lines as well.

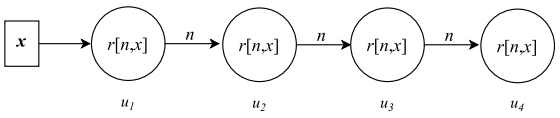


Figure 2: A concrete data structure

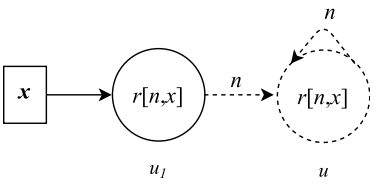


Figure 3: An abstract data structure

This example shows the loss in data structure information that is inevitable when the abstraction is performed. In particular, note that $n(u_1, u)$ evaluates to $1/2$, since nodes u_2, u_3 and u_4 have been summarized in u and it is neither the case that u_1 points to all of u_2, u_3 and u_4 nor that it points to none of them. For the same reason $n(u, u) = 1/2$. In this particular case, the loss of information causes the list in Figure 3 to possibly be interpreted as a cyclic list (since nodes abstracted by u might point to themselves). This is where instrumentation predicates come into play: knowing that $is(u) = 0$, we are sure that no nodes are shared in the list. Therefore, we can conclude that it is acyclic.

Having noticed this, it is already informally clear how to *read off* the data structure properties from three-valued store. However, Wilhelm et al. [WSR00] gives an important theoretical result called the embedding theorem that justifies the intuition.

Let $S_1 = \langle U^{S_1}, \iota_1 \rangle$, $S_2 = \langle U^{S_2}, \iota_2 \rangle$ be two (two- or three-valued) structures and $f : U^{S_1} \to U^{S_2}$ a surjective function. We say f *embeds* S_1 into S_2 (denoted $S_1 \sqsubseteq^f S_2$ iff (1) for every predicate symbol p of arity k, and $u_1, u_2, \ldots u_k \in U^S$ $\iota_1(p)(u_1, u_2, \ldots u_k) \sqsubseteq \iota_2(p)(f(u_1), f(u_2), \ldots, f(u_k))$ and for all $u' \in U^{S_2}$ $|\{u \mid f(u) = u'\}| > 1 \sqsubseteq \iota_2(sm)(u')$. Then we say S_1 can be embedded in S_2, denoted $S_1 \sqsubseteq S_2$.

Conditions (1) and (2) mean that we might lose some information if in the image of f predicate evaluates to $1/2$, but we are sure that if in the image predicate evaluates to a definitive value (0 or 1), then in the original predicate evaluates to the same value. When it is applied in the case of f's domain being concrete two-valued store and f's codomain being abstract three-valued store it enables us to infer properties from abstract presentation of data structure and be sure that they hold for every concrete structure it embeds.

The goal of the analysis is to attach the set of possible structures for every CFG node. To this end, it is necessary to define operational semantics for every C statement. The effect of statement st (with exception of memory allocation which creates new nodes) is given with predicate change formula ϕ_p^{st} which describes how k-ary predicate p's value changes after statement st. The structure that arises after st is re-evaluation of all predicates according to ϕ_p^{st}. Memory allocation statement is special in the sense that it will introduce a new node, but will be taken out of consideration in following description of algorithm.

For example, if statement st is `x = NULL`, then $\phi_x^{st}(v) = 0$ for every individual v. Similarly for st `x->n = t`, $\phi_n^{st}(v_1, v_2) = (n(v_1, v_2) \wedge \neg x(v_1)) \vee (x(v_1) \wedge t(v_2))$, for all individuals v_1, v_2. That is, `n` component stays the same if `x` does not point to the heap node v_1, otherwise it is changed to whatever `t` points to (including `NULL` in which case the predicate will evaluate to 0). The ϕ functions will be used as the *transforming* function for each statement, labeled $[\![st]\!]$.

Since statement transformers might increase the number of individuals in the

structure and interfere with analysis termination, the issue of bounding structure size arises. To bound the structures' size, we define embedding function *abstract* that identifies all individuals that have same values for all unary predicates. That is, for a given structure $S = \langle U, \iota \rangle$, let A be set of S's unary predicates. Then, $abstract(u) = u_{\{p|p\in(A\setminus\{sm\}),p(u)=1\},\{p|p\in(A\setminus\{sm\}),p(u)=0\}}$. Now it is obvious that this embedding bounds the number of individuals for all structures S where A is finite (there can be up to $2^{|A|}$ individuals – each individual can be encoded as binary string of length $|A|$ where bit i is the result of ith predicate).

Now all the elements for the static analysis are present. In the original work of Wilhelm et al. [WSR00], the algorithm is stated with the following set of equations: if we label $StructSet[v]$ the set of structures attached to node v, then equations are given with

$$StructSet[v] = \begin{cases} \bigcup_{w \to v} \{abstract([\![st_v]\!](S)) \mid S \in StructSet[w]\} & \text{if } v \text{ is not a start node} \\ StartStruct & \text{if } v \text{ is a start node} \end{cases}$$

where *StartStruct* is a structure that has no individuals and every predicate evaluates to 1/2.

However, it is obvious that the same result is obtained if we instantiate Kildall's algorithm with the appropriate structure. Namely, we could take the set of all three-value stores, the meet operation $\cup$ and the optimization function can be taken to be the composition of *abstract* and $[\![st]\!]$. Then, Kildall's algorithm would find the fixed point solution to the set of equations.

The authors have extended this simple algorithm with a more advanced pipeline that performs sophisticated analysis to get greater precision. First the *focus* operation is performed to get the set of structures in which every predicate that evaluates to 1/2 is forced to evaluate to a definitive value by a careful case analysis that determines why a predicate evaluates to 1/2. Then, each of the structures that was created is transformed according to the statements and then given to *coerce* operation that is making sure that the transformed structures correspond to C semantics. Thus, instead of a simple transformation three steps are performed to get more precise results.

Suppose that the input to the analysis algorithm is the structure given in Figure 4, and the statement to be analyzed is `x = x->n`. According to previous description, the *focus* operation will make sure that predicates involved in predicate-change formulas (statement transformers) evaluate to definite values. This means that $n(u_1, u)$ will have to evaluate to either 0 (Figure 5a) or 1 (Figure 5b).

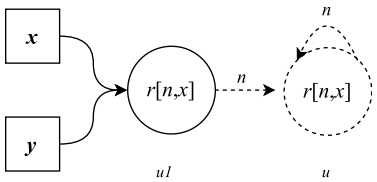
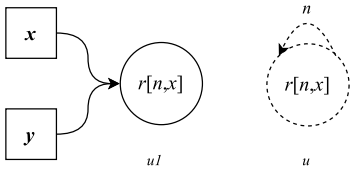


Figure 4: Algorithm input

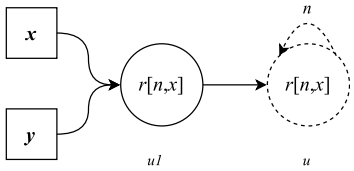
If we take a more informal look at the results from Figures 5a and 5b, we would notice the first abstraction corresponds to the situation where the concrete list had only one element and node u represented elements that are waiting to be garbage-collected, whereas the second one corresponds to the situation where u_1 points to all of the nodes u represents. This is possible only if u is a single node in the concrete list (and as we will see later, *coerce* operation will make sure analysis concludes that).

Lastly, to make this part of the pipeline as precise as possible, the *focus* operation is "unsummarizing" the node u to make sure that the situation where the original list had three or more elements is taken into consideration as well. To this end, new nodes u' and u'' are created out of original u node, and u_1 is set to point to u'. As for the relations between u' and u'' all of them are unknown and evaluating to 1/2. More precise analysis will be obtained later on, from the *coerce* step.

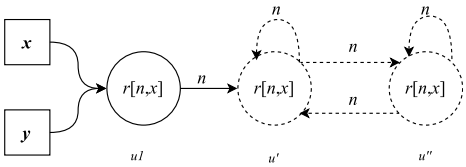
Intuitively the split of the node u corresponds to *zooming in* on it, assuming it represents the original list's tail of length at least two. If that is the case, we can split it in the head about which we can make useful inferences (e.g. u_1 points to head) and the tail that is still a summary node that we do not know much about (i.e. most of the core predicates evaluate to 1/2 at this stage).



(a) Nodes that have to be garbage-collected



(b) List with only two nodes



(c) Node bifurcation

Figure 5: Output of *focus*

The semantics of C assumes that at a given time one variable can point to at most one node, as well as that `n` component of a node can point to up to one node. Similarly, given definition of predicate *is*, it is clear that if it evaluates to 0 for a node, there can be no two pointers pointing to that node. These *hygiene* properties must be respected by each structure.

The *coerce* operation checks each of the generated structures for violations of any of the hygiene properties and simplifies the structures so that they satisfy all of the properties. In this way a correct and more precise analysis is obtained. For example, in the structure from Figure 5b u_1's `n` field points to all of the nodes u represents. However, this is possible only if u is a single node pointing to `NULL`. *coerce* operation will make sure that this result is obtained.

Since the hygiene properties are given as logic formulas, further inferences

can be drawn from the original set of properties if those inferences are logical consequences of original formulas. Thus, even larger set of formulas can be derived. Then, each formula's value is checked against the structure obtained after the statement transform step of the pipeline.

The *coerce* part of the pipeline determines if the structure is *repairable* so that it satisfies the hygiene formulas (or their inferences) and if so, it repairs the structure. If not, this structure cannot represent any concrete data structure and is removed from further analysis. In Figure 6 the structure that is created after repairing the structure from Figure 5c is shown (also, the statement transformer is applied to structure from Figure 5c as an intermediate step).

By adding more instrumentation properties different kinds of analysis can be performed. Authors have augmented the analysis with core predicate $dle(v_1, v_2)$ that is equal to 1 when data fields of v_1 (d_1) and v_2 (d_2) are related with $d_1 \leq d_2$. Using this and a few other order-related instrumentation predicates analysis framework was used to prove partial correctness of sorting algorithms. Moreover, for erroneous programs, analysis was able to pinpoint the source of the error. The errors that analysis uncovered are usually hard to spot with manual debugging. In the case of sorting algorithms, authors managed to uncover the error of not including the first element and memory leakage.

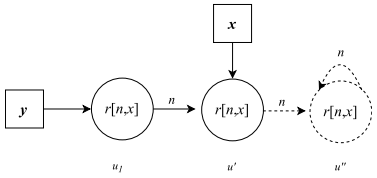


Figure 6: Output of *coerce* (after state transformer) on the structure from Figure 5c

6 Modular Approaches for Verifying Programs

Several static analysis techniques have been discussed. Moreover, their applicability to wide spectrum of problems in software engineering has been shown. However, those tools may not be sufficient for some uses since they are overly conservative (e.g. reporting buffer overflow in a correct program) or overly permissive (e.g. reporting just a warning when a variable is actually not initialized before its use).

A striking example of software for which there is the need for harder guarantees than (overapproximate) bug finders are operating systems kernels, drivers, compilers and similar programs that are usual target of attacks. In his paper about modular verification [App16], Appel argues that a framework can be set up so that apart from security properties (e.g. no buffer overflows, no dangling pointers, no null dereferences), functional properties are shown as well.

Appel's main point is that specification of the software to be verified should be done in a higher-order functional programming language which has a "clean proof theory", that is a language that is easy to formally reason about. Examples of such high-level languages are specification language Gallina of Coq proof assistant and the functional fragment of Isabelle/HOL.

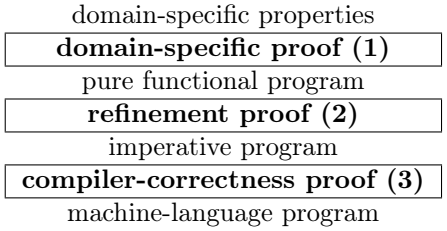
The advantage of this approach is that it is possible to use one language to write the program and formally prove its accordance to the specification in (almost) the same language, largely reducing the proof effort. When the functional program is shown to be correct with respect to specification, it can be compiled to low-level (imperative) language such as C. However, semantic preservation proof from functional programming language to C has to be given as well.

Usually, this is done manually by defining the operational semantics of C in a proof assistant and showing that the C program compiled from functional specification exhibits the same observable behavior. Appel argues that these two verification steps can be done with little interference by describing his experience of working on implementation of crypto algorithm.

Namely, he collaborated with other authors on C implementation of SHA + HMAC and showed that it is possible to formally prove functional program correct with respect to algorithm description isolated from showing correctness of refinement from functional to C program.

Lastly, C program compiled to machine language needs to preserve semantics. This can be again shown manually, or a verified compiler might be used.

The framework described above can be depicted as follows:



Appel argues that most parts of (2) and all of (3) can be automated. Namely, if one could compile functional program directly to imperative program using a verified compiler, there would be no need for any manual proving. So far, CakeML is one complete formally verified ML compiler and CertiCoq is an ongoing project of Gallina to C complier.

With automation of (2) achieved, the only part where the domain-specific proofs are needed is (1). Formally verified compilers exist for quite some time and have been ported to many architectures, so (3) can already be considered automated.

7 Conclusion

Static analysis techniques have started out as algorithms used for analyzing source code of the program to obtain information useful for optimizing the compiled code. However, they became important and widely used in many other practical applications ranging from source code mining to advanced IDE features.

As the need for formally verified compilers emerged, so did the need to make them produce binaries as fast as their non-verified counterparts. To this end, static analysis algorithms used as precursor to optimization passes had to be verified as well.

However, this effort can be greatly reduced by using static algorithms that are part of the same framework. Namely, by having proof that Kildall's algorithm will eventually find the solution given that the input structure satisfies certain constraints, to show that an instantiation of the algorithm is correct, one only has to show that the imposed constraints are satisfied.

That is, by using this *framework* approach, proving effort is greatly modularized since there are parts of the proof that are done once and for all. Modularization of the verification pipeline and separating the common base is an approach that is strongly advocated in logical verification community.

Unfortunately, there are algorithms that are parts of the optimization pipeline that have to be show correct separately (graph coloring algorithm for example). Those proofs however can be studied to see if there is common base that can be used here as well.

It is unreasonable to think that all of the static analysis techniques will be formally verified. For example, there is little need for formally verified autocomplete feature. However, future line of work in this area will probably deal with new optimization algorithms and showing their correctness as those are integrated in verified compiler.

References

[AM17] Michael I. Schwartzbach Anders Møller. Static analysis. *Aarhus University, Denmark*, 2017.

[App16] Andrew W. Appel. Modular verification for computer security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 1–8, 2016.

[BGL04] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, pages 66–81, 2004.

[BRA10] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. Formal verification of coalescing graph-coloring register allocation. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 145–164, 2010.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206, 1973.

[Ler09] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[LRSW00] Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study.

In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000*, pages 26–38, 2000.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[WSR00] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, Arch 25 - April 2, 2000, Proceedings*, pages 1–17, 2000.