ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 NRF ERC

ROSAEC MEMO
2014-03
November 24, 2014

Towards Scalable Translation Validation of Static Analyzers

Jeehoon Kang, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, Kwangkeun Yi

November 24, 2014

Abstract

Static analyzers, which have been successfully deployed in real world to statically find software errors, are complex pieces of software whose reliability is very hard to establish by testing. Testing is not so effective because analysis results are hard to validate *manually* for the following reasons: *(i)* even valid outputs can contain false alarms (or even false negatives if the analyzer is *deliberately unsound*), and *(ii)* internal data such as an abstract state of an input program are too big to manually inspect.

In this paper, we claim that the translation validation approach is a *scalable* and *effective* method to establish reliability of such software. To demonstrate scalability, we developed a verified validator for a real-world static analyzer, which uses various complex algorithms. To demonstrate effectiveness, using our validator we validated the analysis results of the analyzer for 16 open-source programs, during which we effectively identified and fixed 13 bugs using the validation results.

1 Introduction

Formal verification of functional correctness of software can be practically useful only when the following conditions are met: *(i)* the verification effort should not be too costly; and *(ii)* reliability of the software should be either very important or hard to establish by other means such as testing. The former condition is hard to be met for many applications because, for instance, their specifications are too complex to formalize, or their implementations are too big to formalize. However, there are some large applications such as compilers that have simple specifications. Also, if the *translation validation* approach [Pnueli et al.(1998)Pnueli, Siegel, and Singerman] is applicable, instead of target programs, we can verify their (possibly much smaller) validators that validate runtime results of the target programs. Similarly, the latter condition is also not the case for many applications. However, there are some safety-critical programs such as flight control systems whose reliability is extremely important. Or, sometimes reliability is hard to establish only by testing because, for instance, the programs are highly concurrent, or it is not easy to determine which behaviours are faulty.

Static analyzers based on abstract interpretation are ideal examples of such software that satisfy the two conditions. First of all, reliability of a static analyzer is hard to establish only by testing, especially when it is designed to be unsound (usually due to unavailable library code and for the sake of precision, *i.e.*, reduction of false alarms). The reason is that whether an analysis result is valid is not evident because both valid and invalid results can contain false alarms and moreover, if the analyzer is unsound, even can miss true alarms. Also, it is infeasible to detect bugs by manually examining intermediate data such as an abstract state of the analyzed program because such data are simply too big unless the analyzed program is very small. However, reliability of a static analyzer is important, especially when it is applied to safety-critical software, because reliability of analyzed software depends on that of the analyzer. This argument is still valid even if the analyzer is deliberately unsound, although the importance gets slightly weakened.

Second, the formal verification cost of a static analyzer based on abstract interpretation is not too expensive. First, such an analyzer has a simple specification for its main part: the resulting abstract state should include all possible concrete states that can occur during executions of the analyzed program. Second, the verification cost can be reduced by using translation validation. The idea is to develop a validator (to some extent specialized for a particular analyzer[1]) that checks whether a given analysis result is valid or not, and then formally verify that the validator is correct. By validating each analysis result using the verified validator, we can trust the result if it passes the validator; or otherwise detect bugs of the analyzer. In this way we can greatly reduce the verification effort because the validator is much smaller and simpler than the analyzer. The validation algorithm would typically amount to checking whether some derived abstract state is a prefixed point of a given abstract semantic function.

However, a downside of translation validation is that the runtime cost of validation might become too expensive. If we implement a naive validator in order to reduce its verification effort, then the runtime cost of the validator may be much larger than that of the analyzer. Indeed, an early version of our validator was 100 times slower than the associated highly-optimized analyzer. Thus there is a trade-off between *development scalability* and *runtime scalability*. By the former we mean scalability regarding development cost (*i.e.*, implementation and verification cost) of a validator, and by the latter regarding runtime cost of a validator.

In this paper, we demonstrate that translation validation is a scalable method for static analyzers based on abstract interpretation by developing a verified validator[2] for a real-world analyzer, Sparrow, which is designed to be unsound. In this work, our contributions are summarized as follows.

- We gave a formal specification of Sparrow. It is a variant of the usual soundness statement that properly captures the deliberate unsoundness of Sparrow.
- We struck a balance between development and runtime scalability. By optimizing the naive validator, we could be able to reduce the runtime cost of the validator to be on average twice as high as that of Sparrow.
- We developed a simple method to identify bugs of the analyzer from a validation result when the validation fails. Using this method, we found and fixed 13 bugs of Sparrow.

2 Overview

In this section, we will first introduce our framework for translation validation of static analyzers. Then we present several approaches that have different development and runtime scalability, and discuss their pros and cons in a general setting. In the following sections, as a case study, we discuss in detail how we develop our validator for Sparrow.

2.1 Framework for Translation Validation

In our framework, as depicted in Figure 1, a static analyzer consists of two parts: a main analyzer and an alarm generator. The former, given an input program P, produces a core abstract state $\hat{S}_c$, from which together with the input program P the latter produces alarms for potential program errors. A program P consists of function blocks, which in turn consist of basic blocks In Figure 2 is shown example basic blocks and control flow edges. An abstract state $\hat{S}$ in general is a map from basic blocks to abstract memories, where $\hat{S}(b)$ is considered as an over-approximation of all concrete memories that can occur when an execution reaches the

[1] Developing a general validator is undecidable [Rice(1953)].

[2] Formal verification in the theorem prover Coq is available at `http://sf.snu.ac.kr/jeehoon.kang/assets/sparrowberry.tar.gz`.

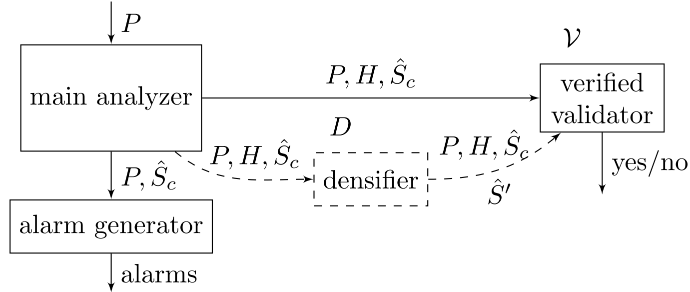

Figure 1: Framework for Translation Validation of Static Analyzers.

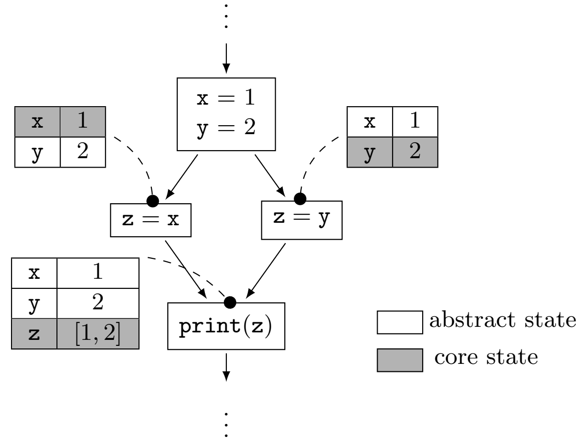

Figure 2: Basic Blocks, Control Flow Edges, and (Core) Abstract States

block b. A core (abstract) state $\hat{S}_c$ is a substate of an abstract state such that $\hat{S}_c(b)$ defines only those variables that are used in the block b. In Figure 2, the abstract state $\hat{S}(b)$ is attached to each block b, of which the shaded part denotes the core state $\hat{S}_c(b)$ that has only used variables in the block b.

It is for generality of our framework that we consider core states $\hat{S}_c$ as input to alarm generators, instead of whole states $\hat{S}$. One can easily see that core states are just enough information for alarm generators to correctly produce alarms because core states define all used variables. Thus requiring a whole state as output of main analyzers is too restrictive because some analyzer may not compute a whole state at all. Indeed, it is the case for many analyzers including Sparrow [Oh et al.(2012)Oh, Heo, Lee, Lee, and Yi, Oh et al.(2013)Oh, Heo, Park, Kang, and Yi, Choi et al.(1991)Choi, Cytron, and Ferrante, Ramalingam(2002), Reif and Lewis(1977), Wegman and Zadeck(1991), Dhamdhere et al.(1992)Dhamdhere, Rosen, and Zadeck, Chase et al.(1990)Chase, Wegman, and Zadeck, Tok et al.(2006)Tok, Guyer, and Lin, Cytron and Ferrante(1995), Johnson and Pingali(1993), Hardekopf and Lin(2009), Hardekopf and Lin(2011)].

Among the two components of static analyzers, only main analyzers are worth being verified due to their complexity. Alarm generators are usually so simple that they are unlikely to have software bugs. Thus considering their verification effort, it might not be so useful to verify alarm generators.

In order to verify a main analyzer, the first step is to formalize its specification. As we have discussed in the introduction, an analyzer has a clear specification: a core state $\hat{S}_c$ of a

program P is *valid* if the following holds:

$$\forall(b,m) \in [\![P]\!].\ \ m|_b^{\texttt{use}} \in \gamma(\hat{S}_c(b))\ .$$

Here, by $(b,m) \in [\![P]\!]$ we mean that m is a concrete memory that can occur when an execution of P reaches the basic block b. By $m|_b^{\texttt{use}}$ we denote the concrete memory m restricted to the variables used in the block b. Finally, by $\gamma(\hat{S}_c(b))$ we denote the set of concretized memories of the core abstract memory $\hat{S}_c(b)$.

The second step is to implement and verify a validator for a main analyzer. As shown in Figure 1, a validator first takes input and output of the analyzer (*i.e.*, an input program P and its core state $\hat{S}_c$) together with hints H for why $\hat{S}_c$ is valid. (Ignore dashed parts in Figure 1 for now.) Then it determines whether $\hat{S}_c$ is a valid abstraction of the program P. A standard way of proving that an abstract state $\hat{S}$ of P is valid is to show that $\hat{S}$ is a prefixed point of a sound abstract semantic function $\hat{F}_P$ of P (*i.e.*, $\hat{F}_P(\hat{S}) \sqsubseteq \hat{S}$). From this, it is not hard to see that a core state $\hat{S}_c$ is valid w.r.t. P if there exists a (whole) abstract state $\hat{S}$ such that $\hat{S}$ is valid w.r.t. P and $\hat{S}$ *extends* $\hat{S}_c$ (denoted $\hat{S}_c \subseteq_P^{\texttt{use}} \hat{S}$) in the following sense: $\hat{S}(b)$ and $\hat{S}_c(b)$ agree on the used variables of each block b.

2.2 Trade-off Between Development and Runtime Scalability

From now on we discuss how to strike a balance between development cost and runtime cost of a validator.

Densifier Verification Approach An obvious approach, which we call *densifier verification* approach, is to validate $\hat{S}_c$ directly and then verify the validation. This verification amounts to showing that if the validation succeeds, there exists a valid $\hat{S}'$ that extends $\hat{S}_c$, formulated as follows:

$$\mathcal{V}(P,H,\hat{S}_c) \Rightarrow \exists \hat{S}'.\ (\hat{F}_P(\hat{S}') \sqsubseteq \hat{S}') \wedge (\hat{S}_c \subseteq_P^{\texttt{use}} \hat{S}').$$

One of the important benefits of this approach is that the computation of $\hat{S}'$ is a part of the correctness proof and never needs to be performed at runtime. Thus this would greatly reduce runtime cost of the validator because, as we will discuss later, the computation cost of $\hat{S}'$ is high.

However, in this approach we may have to pay high cost of verification for the following reasons. In order to validate $\hat{S}_c$, we first need a hint H such as semantic dependence graph for technical reasons (see Section 4 for details). Then, we have to check whether the dependence graph H is correct and the core state $\hat{S}_c$ is valid. In order to prove that the validation is valid, we have to show the existence of a function D, which we call *densifier*, that constructs the extended state $\hat{S}'$ from a core state $\hat{S}_c$, and then show that if the validation $\mathcal{V}(P,H,\hat{S}_c)$ succeeds, $D(P,H,\hat{S}_c)$ is a valid abstract state of P. In other words, we have to *verify* the densifier D rather than just *validate* its result. The verification would involve complicated soundness proofs of the dependence graph checker and the densifier, but the validation would not.

Moreover, this approach may require the validator to be tightly coupled with the analyzer. Given a program, there are many dependence graph that correctly approximates the actual semantic dependence in the program. Thus the validator has to know exactly how the analyzer generates a dependence graph and then check whether a given graph is generated as such. As a result, we may have to revise the implementation and verification of the validator as the analyzer gets improved.

Densifier Validation Approach The opposite approach, which we call *densifier validation* approach, is to just validate results of the densifier D instead of verifying the densifier itself. As depicted as dashed parts in Figure 1, we first compute $D(P,H,\hat{S}_c)$, say $\hat{S}'$, and then just

validate the result $\hat{S}'$. We also have to verify the validation, which amounts to showing that if the validation succeeds, $\hat{S}'$ is a valid abstract state of P and $\hat{S}'$ extends $\hat{S}_c$, formulated as follows:

$$\mathcal{V}(P, H, \hat{S}_c, \hat{S}') \Rightarrow (\hat{F}_P(\hat{S}') \sqsubseteq \hat{S}') \wedge (\hat{S}_c \subseteq_P^{\mathtt{use}} \hat{S}').$$

It is important to note that $\hat{S}'$ is given as an argument to the validator, whereas in the densifier verification approach we have to prove that such $\hat{S}'$ exists.

This approach benefits from usual merits of translation validation over verification. First, we do not need to implement the densifier D in theorem provers such as Coq. Writing programs in theorem provers is in general much more difficult than doing so in usual programming languages because of, for example, termination checking and lack of libraries. Second, more importantly, showing that a given state $\hat{S}'$ is a prefixed point is much easier than showing the existence of such $\hat{S}'$.

Moreover, the validator is loosely coupled with the analyzer for the following reasons. The validation of $\hat{S}'$ uses a control flow graph rather than a dependence graph, and unlike the latter, the former is much less dependent on the implementation of the analyzer. As a result, we may not have to often revise and reverify the validator as the analyzer gets improved. Note that we may have to revise the densifier, which however does not need to be verified.

A downside of the densifier validation approach, however, is that runtime cost of the validation might be problematic. Though each block typically uses only tiny fraction of whole variables, the densifier extends the core state $\hat{S}_c$ to $\hat{S}'$ in such a way that $\hat{S}'$ includes the whole variables. Thus the densified state $\hat{S}'$ is much larger than the core state $\hat{S}_c$, which may cause high runtime cost of the densification and the validation. Indeed, an early version of our validator, which used this approach, was more than 100 times slower than the analyzer Sparrow.

Hybrid Approach Another approach that we used, which we call *hybrid* approach, is to use the both densifier validation and verification approaches. The idea is to first split the densifier D into two parts D_1 and D_2 and then to *validate* the former and *verify* the latter. The first part, which amounts to validating D_1, is to compute $D_1(P, H, \hat{S}_c)$, say $\hat{S}'$, and validate the result $\hat{S}'$. The second part, which amounts to verifying D_2, is to show that if the validation of $\hat{S}'$ succeeds, there exists $\hat{S}''$ (which will be instantiated with $D_2(P, H, \hat{S}_c, \hat{S}')$ in the proof) such that $\hat{S}''$ is valid and extends $\hat{S}_c$, formulated as follows:

$$\mathcal{V}(P, H, \hat{S}_c, \hat{S}') \Rightarrow \exists \hat{S}''.\ (\hat{F}_P(\hat{S}'') \sqsubseteq \hat{S}'') \wedge (\hat{S}_c \subseteq_P^{\mathtt{use}} \hat{S}'').$$

An important point in this hybrid approach is that one has to split D in an appropriate way. To maximize its benefit, D has to be split into D_1 and D_2 in such a way that D_1 has low runtime cost and high verification cost, while D_2 has high runtime cost and low verification cost. Our validator, to be presented later, uses this hybrid approach. In the development, we could be able to keep the verification cost to be twice as high as that of its earlier version using the densifier validation approach. On the other hand, we reduced its runtime cost to be on average twice as high as the analyzer Sparrow's, rather than 100 times.

We will further discuss how we develop our validator using the hybrid approach in Section 4. The other two contributions of our work, *(i)* how we formalize the deliberate unsoundness of the analyzer Sparrow and *(ii)* how we identify bugs from unsuccessful validation results, will be discussed in Section 3 and 5 respectively. Finally, we conclude by discussing our experiment results in detail in Section 6 and discussing future work and related work in Section 7.

3 Formal Specifications of Unsound Analyzers

3.1 Our Model of Unsound Analyzers

Soundness and preciseness are two valuable properties of a static analyzer. The former means that the analyzer raises alarms for all bugs of certain types. The latter means that the analyzer raises a relatively small number of false alarms. However, it is hard to achieve both properties for a static analyzer that is to analyze a large class of programs. For example, suppose an analyzer analyzes an open program. When an unknown library function is called, the only way for an analyzer to be sound is to give the top($\top$) abstract memory after the function call, which denotes that any location can contain any value. The reason is because the unknown function can potentially do anything it likes, such as overwriting the whole memory. However, this way the analyzer loses preciseness. The imprecise approximation caused by the top memory may be propagated throughout the whole program, which may result in a huge number of false alarms.

In practice, many analyzers give up soundness in order to increase precision. For example, one can treat any unknown library function call as no operation.

A natural question would be whether such an unsound analyzer guarantees nothing about its result. Even for an unsound analyzer, would it not be possible to talk about the degree of unsoundness? Since an unsound analyzer is also software, clearly it will have its own specification, though it may be complicated.

To answer the question, we model an unsound analyzer as one that transforms a given input program and performs a sound analysis for the transformed program. The degree of unsoundness can be also simply modelled as how much different the two programs are. In case of the above example, all unknown library function calls in an input program P will be replaced by the no-op statement in the transformed program P'. Clearly, what the unsound analyzer does for P amounts to a sound analysis for P'.

We successfully formalized the specification of the analyzer Sparrow using this model. We believe that many other unsound analyzers can be modelled in this way.

3.2 Case Study: Sparrow

We illustrate the three unsound features of Sparrow and how to model them using our approach.

As in the above example, the first unsound feature of Sparrow is to handle unknown library calls as no operations. For example, suppose we analyze the following C code fragment.

```
10:  int p[2] = {0, 0};
11:  unknown_f(p + sizeof(int));
12:  assert(p[0] == 0);
```

Suppose that we do not have the source code for the function `unknown_f` invoked at line `11`. Then, in principle, the `assert` at line `12` may fail since `unknown_f` may be able to update the variable `p[0]` by accessing the argument pointer backward. Thus, a sound analyzer should raise an alarm at line `12`. However, this is very likely to be a false alarm because accessing a pointer backward is rare. For the sake of precision, the analyzer Sparrow treats the function call as no operation. To model this unsoundness, we can simply comment out the line `11` in the transformation.

The second unsound feature is to treat an update of the value in an unknown location as no operation. During the analysis, one can reach a statement such as `*p = v;` that writes a value to a location `p`. At that time, if the location `p` is approximated as the top value, the assignment is treated as no operation because otherwise a huge number of false alarms may be generated for the same reason as before. We can also simply model this unsound feature by commenting out the assignment statement in the transformation. Note that we have to use the analyzer's analysis result to perform this transformation because we have to determine whether a location is approximated as the top value or not.

The last unsound feature is to treat float and double values as integers. The reason for this feature is that the main goal of Sparrow is to find buffer-overrun bugs. While float and double values are hard to analyze soundly, they are unlikely to influence on buffer-overruns in practice. This feature can be easily modelled by syntactically changing the type of float and double variables to int.

In the current stable version of Sparrow that we verified, we model its unsound features by simple transformations as we have seen above. However, in an experimental version that employs more advanced techniques such as context-sensitive analysis, modelling its unsound features may require more complex transformations. See Section 7 for details.

Using this model of unsoundness, we can easily adapt the framework shown in Figure 1 to allow unsound analyzers. The main analyzer needs to simply pass to the validator the transformed program P' instead of the input program P.

4 Translation Validation for Sparrow

In this section, we will discuss how we develop our validator for Sparrow. We first look into an example program and its core state from Sparrow. Then we apply three approaches discussed in Section 2 to translation validation for Sparrow, and discuss their pros and cons.

Example from Sparrow Given an example program P, Sparrow produces the core abstract state $\hat{S}_c$, as shown in Figure 3 (a). In the figure, the comment in a basic block b represents the abstract (approximated) pre-state $\hat{S}_c(b)$ right before executing b, where each line is considered as a block. For expository purposes, in this section we assume that all concrete values are integers and an abstract memory at each block is a partial map from variables to intervals (for example, $\hat{S}_c(\texttt{11}) = [\texttt{x} \mapsto [1,1]]$). For an abstract memory $\hat{m}$, by $\gamma(\hat{m})$ we mean the set of concrete memories approximated by $\hat{m}$.

The core state $\hat{S}_c$ contains enough information to generate alarms. This is because for each block b, the core state $\hat{S}_c(b)$ has all those variables that are used in the block b. For example, $\hat{S}_c(\texttt{14})$ has the variables `x` and `y` that are used in the block `14` (`assert(x<y)`) in Figure 3 (a).

The core state $\hat{S}_c$ is valid w.r.t. the program P since there exists a whole state $\hat{S}$, presented in Figure 3 (b), that extends $\hat{S}_c$ and is a prefixed point of a sound abstract semantic function $\hat{F}_P$. Here, $\hat{F}_P$ is defined as follows. First, an abstract semantic function $\hat{f}_b$ for each block is defined to soundly estimate the concrete execution f_b of the block b, formulated as follows:

$$\forall m \forall \hat{m}.\ m \in \gamma(\hat{m}) \Rightarrow f_b(m) \in \gamma(\hat{f}_b(\hat{m}))$$

where m and $\hat{m}$ represent a concrete memory and an abstract memory respectively. Second, $\hat{F}_P(\hat{S})(b)$ is defined as the join of all abstract memories at those block b' from which b is reachable via a control flow edge (denoted $b' \to b$), formulated as follows:

$$\hat{F}_P(\hat{S})(b) = \bigsqcup_{b' \to b} \hat{f}_{b'}(\hat{S}(b')).$$

Densifier Verification Approach In theory [Oh et al.(2012)Oh, Heo, Lee, Lee, and Yi, Oh et al.(2013)Oh, Heo, P it is possible to validate the core state $\hat{S}_c$ in densifier verification approach. More specifically, using a semantic dependence graph (generated as a hint by Sparrow) we can validate $\hat{S}_c$ without actually computing the whole state $\hat{S}$. The semantic dependence graph H of the program P is depicted in Figure 3 (c). By a dashed line from `10` to `12` with label `x` (denoted $\texttt{10} \overset{\texttt{x}}{\dashrightarrow} \texttt{12}$) we mean the variable `x` defined at `10` is used at `12`. Assuming the semantic dependence graph is correct, the validation for the block `12` amounts to checking whether $\hat{f}_{\texttt{10}}(\hat{S}_c(\texttt{10}))(\texttt{x}) \sqsubseteq \hat{S}_c(\texttt{12})(\texttt{x})$. Generally, we can validate a core state by checking whether $\hat{F}^c_P(\hat{S}_c) \sqsubseteq \hat{S}_c$, where $\hat{F}^c_P(b)$ is

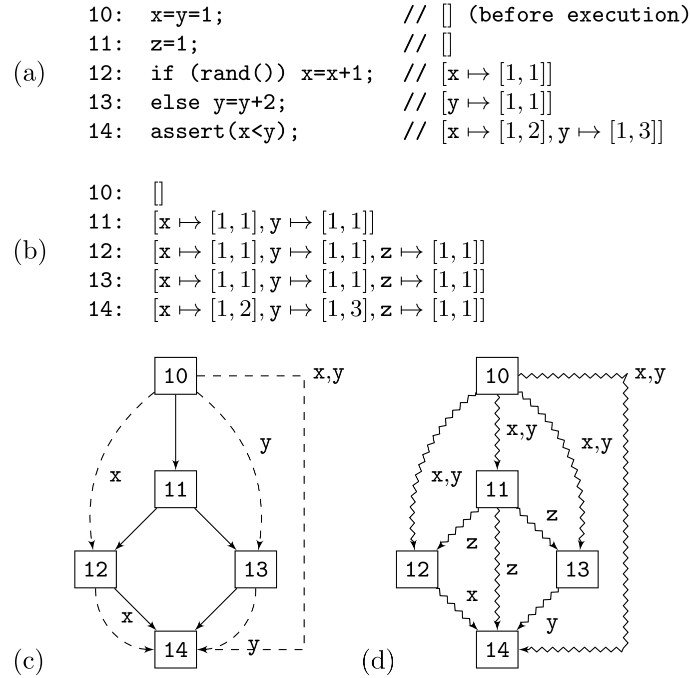

Figure 3: (a) a program P and a core abstract state $\hat{S}_c$ from Sparrow. (b) a whole abstract state $\hat{S}$ that extends $\hat{S}_c$ and is a prefixed point of an abstract semantic function. (c) the semantic dependence graph H of P from Sparrow. Bold lines represent control flow and dashed lines represent data flow of the labeled variable. (d) the extended semantic dependence graph H' of P. Squiggly lines represent extended semantic dependence.

defined as follows:

$$\hat{F}^c_P(\hat{S}_c)(b) = \bigsqcup_{b' \overset{x}{\dashrightarrow} b} \hat{f}_{b'}(\hat{S}_c(b'))|_x$$

where $\hat{m}|_x$ denotes the submemory of the abstract memory $\hat{m}$ that only has x.

In addition, we have to validate the dependence graph as well as the core state $\hat{S}_c$ since the dependence graph H is from an untrusted source, Sparrow. A dependence edge $b' \overset{x}{\dashrightarrow} b$ is valid if x defined at the block b' reaches the block b following edges in the control flow graph without being redefined by another definition of x. For example, the dependence edge $\texttt{10} \overset{\texttt{y}}{\dashrightarrow} \texttt{14}$ is valid since `y` is defined at `10`, and reaches `11`, `12`, and then `14` without being redefined.

If the dependence graph H and the core state $\hat{S}_c$ are validated, there exists $\hat{S}$ that extends $\hat{S}_c$ and is a prefixed point of the abstract semantic function $\hat{F}_P$.

However, we have to pay high cost of verification in this approach. We have to verify the complex implementation of Sparrow. This means the relevant parts of the implementation should be specified and proved in the verification of the validator. Also, we have to revise the implementation and verification of the validator as the analyzer gets improved on the semantic dependence graph generation.

Densifier Validation Approach In this approach we densify the core state $\hat{S}_c$ into a whole state $D(P, H, \hat{S}_c)$ and validate it. As discussed in Section 2, this approach has clear advantage that we do not need to verify the densifier D and we just need to check whether the whole state $D(P, H, \hat{S}_c)$ is a prefixed point of the abstract semantic function $\hat{F}_P$ and extends the core state $\hat{S}_c$.

Densification D fills out what is missing in the core state $\hat{S}_c$ using an extended semantic dependence graph. In Figure 3 (d) is depicted the extended semantic dependence graph H' of the program P shown in (a). By a squiggly line from `10` to `12` with the label `x`,`y` (denoted $\texttt{10} \overset{\texttt{x,y}}{\rightsquigarrow} \texttt{12}$) we mean the variables `x` and `y` defined at the block `10` reaches the block `12` without being redefined. The extended dependence is different from the ordinary dependence in that for the extended semantic dependence, the variable is not necessarily used in the destination block. For example, $\texttt{10} \overset{\texttt{y}}{\rightsquigarrow} \texttt{12}$ since `y` defined at `10` reaches `12`, but not $\texttt{10} \overset{\texttt{y}}{\dashrightarrow} \texttt{12}$ since `y` is not used in `12`.

Densification D is defined as follows [Oh et al.(2013)Oh, Heo, Park, Kang, and Yi]:

$$D(P, H, \hat{S}_c)(b) = \bigsqcup_{b' \overset{x}{\rightsquigarrow} b} \hat{f}_{b'}(\hat{S}_c(b'))|_x \ .$$

Here, the extended semantic dependence graph H' is used in $b' \overset{x}{\rightsquigarrow} b$ and can be derived from the ordinary semantic dependence graph H. Note that the densification D is similar to the abstract semantic function $\hat{F}^c_P$ used to directly validate the core state $\hat{S}_c$, with the difference that D uses the extended semantic dependence rather than the ordinary one.

The problem of this approach is that the runtime cost of the validation is impractically high for Sparrow. An early version of our validator, which used this approach, was more than 100 times slower than Sparrow. This is because the densified state $\hat{S}$ is much larger than the core state $\hat{S}_c$. For example, $\hat{S}$ is 85 times larger than $\hat{S}_c$ for the benchmark program gzip-1.2.4a.

Hybrid Approach Before explaining in detail, let us recap the hybrid approach. The approach is divided into two parts. The first part is the validation of D_1, *i.e.*, to compute $D_1(P, H, \hat{S}_c)$, say $\hat{S}'$, and validate the result $\hat{S}'$. The second part is the verification of D_2, *i.e.*, to show that if the validation of $\hat{S}'$ succeeds, there exists $\hat{S}''$ (which will be instantiated with $D_2(P, H, \hat{S}_c, \hat{S}')$ in the proof) such that $\hat{S}''$ is valid and extends $\hat{S}_c$.

The key design choice in the hybrid approach is how to split the densifier D into the part to be validated (D_1) and the other part to be verified (D_2). To maximize the benefit, D has to be split in such a way that D_1 has low runtime cost and D_2 has low verification cost.

We choose to split D into D_1 and D_2 in such a way that D_1 only densifies locally inside the scope of each function, and D_2 densifies among functions. Here, by inside the scope of a function f we mean considering only semantic dependence edges inside the function f. For example, consider the program P and its core state $\hat{S}_c$ depicted in Figure 4 (a). A semantic dependence graph for P is depicted in Figure 4 (c). In the semantic dependence graph, $\texttt{10} \overset{\texttt{x}}{\dashrightarrow} \texttt{12}$ and $\texttt{10} \overset{\texttt{y}}{\dashrightarrow} \texttt{11}$ are inside the scope of `main` and $\texttt{20} \overset{\texttt{y}}{\dashrightarrow} \texttt{21}$ is inside the scope of `foo`. The densified state $D_1(P, H, \hat{S}_c)$, say $\hat{S}'$, is shown in Figure 4 (b).

The result $D_1(P, H, \hat{S}_c)$ of local densification can be validated using only the dependence edges among functions. This is because we already locally densified using edges inside the scope of each function. We explain the general way to validate a locally densified state with an example. Consider the locally densified state $\hat{S}'$ shown in Figure 4 (b) of the program P shown in (a). The semantic dependence edges among functions are shown in Figure 4 (d). Around the function call from `main` to `foo`, we check whether:

- $\hat{f}_{\texttt{11}}(\hat{S}'(\texttt{11}))(\texttt{y}) \sqsubseteq \hat{S}'(\texttt{20})(\texttt{y})$, for the dependence edge of the function call;
- $\hat{f}_{\texttt{21}}(\hat{S}'(\texttt{21}))(\texttt{y}) \sqsubseteq \hat{S}'(\texttt{12})(\texttt{y})$, for the dependence edge of the function return; and
- $\hat{f}_{\texttt{11}}(\hat{S}'(\texttt{11}))(\texttt{x}) \sqsubseteq \hat{S}'(\texttt{12})(\texttt{x})$, for those variables that are not accessed in the callee `foo`.

The verification of the local densification D_2 in this approach is easier than that of the full densification D. You may wonder why the verification cost of is reduced, while the verification task is largely similar. This is because the edges among functions are regularly drawn. More specifically, edges are always drawn in pairs of a call and its corresponding return with the same variables labeled, as shown in Figure 4 (d). Thus the densifier D_2 is simpler than D and its verification is not too expensive. In the development, we could be able to keep the verification cost to be twice as high as that of its earlier version using the densifier validation approach.

At the same time, we reduced the runtime cost to be on average twice as high as the analyzer Sparrow's, rather than 100 times (the early version's runtime overhead). The validator in this approach is much faster because an intermediate state $\hat{S}'$ has less variables (abstract locations) than a fully densified state $\hat{S}''$ [Oh et al.(2011)Oh, Brutschy, and Yi].

5 Identifying Bugs from Validation Failures

We explain the details of how we identify bugs in the analyzer Sparrow from unsuccessful validation results. We could effectively fixed 13 tricky bugs of the analyzer Sparrow.

5.1 Finding Reasons of Validation Failures

We explain how to identify the reasons of validation failures. Note that a validation fails if *(i)* the densified state $\hat{S}'$ is not a prefixed point of the abstract semantic function, or *(ii)* it does not extend the input core state $\hat{S}_c$.

There are three reasons of validation failures: bugs in the analyzer, bugs in the densifier, and mismatches between the abstract semantic functions used in the analyzer and the validator.

For Sparrow, the reason of a validation failure can be identified as follows. When $\hat{S}'$ does not extend $\hat{S}_c$, the reason is obviously bugs in the densifier because it should densify $\hat{S}_c$ with keeping intact the values of the used variables. On the other hand, when $\hat{S}'$ is not a prefixed point, we have to further investigate the failure. We will explain with the following examples.

- We find a bug of the analyzer using the following code as follows:

```
10:  x = 1;  // [x ↦ ⊥] (before execution)
11:  y = x;  // [x ↦ ⊥]
```

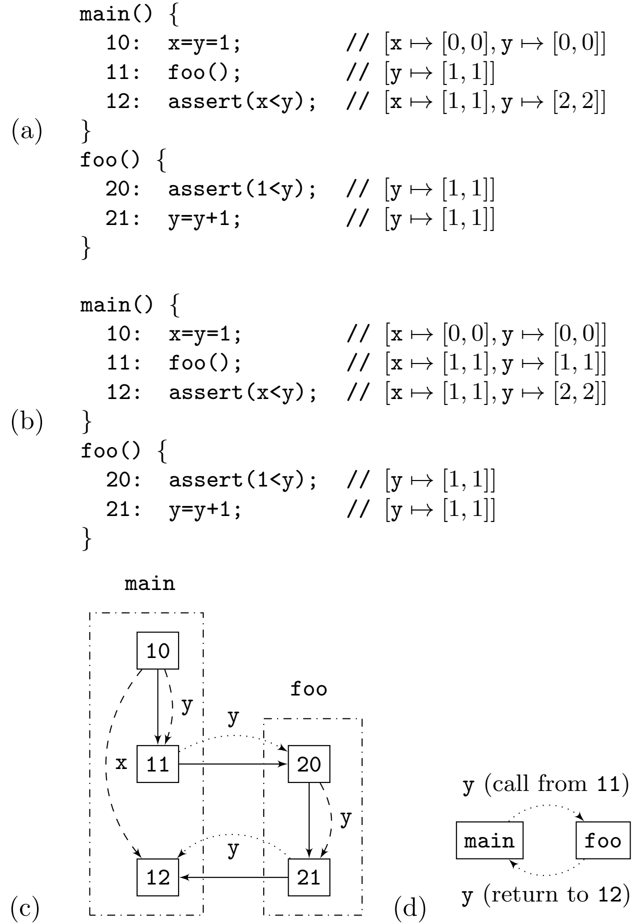

Figure 4: (a) an example program P and the core state $\hat{S}_c$ from Sparrow. (b) densified state $\hat{S}' = D_1(P, H, \hat{S}_c)$. (c) the semantic dependence graph H of P for $\hat{S}_c$ from Sparrow. (d) the semantic dependence edges among functions.

Note that the comment in a line l represents the abstract pre-state right before executing l. Our validator's abstract semantic function calculates that the output memory of the line 10 is $[\mathtt{x} \mapsto [1,1]]$. Thus the validation fails because $[1,1] \not\sqsubseteq \bot$. In this case, we can deduce that the analyzer has a bug in *(i)* its abstract semantic function, or *(ii)* its semantic dependence generation. In the former case, the analyzer incorrectly calculates that the output memory of the line 10 is $[\mathtt{x} \mapsto \bot]$. In the latter case, the analyzer ignores to deliver the value of x from the line 10 to the line 11.

- We find a bug of the densifier using the following code as follows:

```
10:  x = 1;  // [x ↦ ⊥]
11:  y = 1;  // [x ↦ ⊥]
```

Our validator's abstract semantic function calculates that the output memory of the line 10 is $[\mathtt{x} \mapsto [1,1]]$ and the validation fails. Unlike the previous example, The failure of this example is due to a bug in the densifier. In this example, the core state at 11 does not have x since x is not defined in 11. Thus it is the densifier which should have filled x in the abstract state in 11. On the other hand, in the previous example, the core state at 11 already should have x.

- We find a mismatch between the abstract semantic functions of the analyzer and the validator as follows:

```
10:  x = 1;  // [x ↦ [0,0]]
11:  y = x;  // [x ↦ [1,1]]
             // (validator:  [x ↦ [0,1]])
```

Suppose that the validator calculates that the output memory of the line 10 is $[\mathtt{x} \mapsto [0,1]]$. Then the validation fails even though the analyzer's result is sound. This is because the abstract semantic function of the validator is less precise than that of the analyzer. In this case, we have to modify the semantics of the validator and the corresponding part of the soundness proof.

5.2 Subtle Bug Found

Using our validator and the previous simple method, we effectively found and fixed 13 bugs of Sparrow on the way of validating 16 real-world benchmark programs. These subtle bugs had not been detected for years in spite of extensive testing using large benchmark programs. We discuss the subtleties of some of the bugs we found to illustrate how effective the validation approach is in debugging static analyzers.

We present an example that illustrates such subtle false-negative bugs that are specially hard to detect.

```
10: void f(){
11:   int *p = malloc (4);
12:   *p = 0;
13:   h(p);
14: }
15:
16: void g(){
17:   int x = 10;
18:   h(&x);
19: }
20:
21: void h(int *q){
22:   int array[5];
23:   array[*q] = 0;
24: }
```

Table 1: Bugs of the target analyzer found by the validator: **Category** summarize the origin of a bug by three sorts - dependence graph, semantics, and parser. **Description** briefly describes the bugs.

Category	Description
dependence graph	Dynamic locations were not included in a definition set when arrays are declared. Graph edges were not drawn correctly when weak-update occurs. Graph edges were not drawn correctly when an encoded library function is called. Graph edges for fields were not drawn correctly. Return edges should be definition points.
semantics	Field values should be top if the struct itself is top. Local variables should not be removed on an exit node in some cases. Field values should not be declared as dynamic values. Typing errors on abstract interval operations. 0 and null worked inconsistently in some cases. Values from address-taken locations should not be removed on exit nodes. Weak update conditions for local variables were incorrect.
parser	Functions and local variables should be treated individually, even if their names are same.

Using the above example, we explain one of the bugs we found that caused a true alarm not to be raised. Here, in order to be sound, semantic dependence edges to the entry point of the function `h` at line `21` should be drawn from the call sites of `h` at line `13` for `p` and at line `18` for `&x`, respectively. However, Sparrow only drew the former dependence edge but not the latter one. More specifically, Sparrow did not draw such a dependence edge when the address passed is taken from a variable.

To see the problem, let us consider the set of possible values of `*q` at the entry point of `h` at line `21`. It is not hard to see that it should contain 0 and 10 because `h` is invoked at line `13` and at line `18`. However, Sparrow infers that `*q` can be only 0 due to the missing edge from the line `18`. As a result, it does not raise any buffer overrun alarm at line `23`, which is unsound.

6 Experiment

In this section, we evaluate our validator and confirm its stability, scalability, and effectiveness as a debugger. During the experiment, we used the machine with Linux 3.0 operating system equipped with a quad-core of Intel 3.07GHz box with 24GB of main memory.

Stability Our validator succeeded to validate 16 real-world benchmarks. The benchmarks are open-source software, mostly from GNU projects. They include well-known applications such as make, screen, and lsh. It is impossible to formally guarantee that the validator always succeeds for correct analysis result. However, we regard that our current validator achieve the stability in some degree. This is because the number and the variety of benchmarks represent general properties of real-world applications.

Scalability Our validator generally takes less time and consumes less memory than the analyzer. Table 2 presents overall elapsed time and memory consumption of the analyzer and the validator. Averagely, the validator is faster by 2.5 times and consumes less memory by 0.8 times than the analyzer. This result directly demonstrates the runtime scalability of the validator.

Effectiveness as a Debugger While trying to validate with real-world benchmarks, we found 13 major bugs of the target analyzer. Table 1 presents the bugs we found. The origin of bugs largely fall into three categories: dependence graph, semantics, and parser. We only discuss the bugs by dependence graph, which is a core component of the analyzer. Dependence graph type of bugs can be generated due to insufficient graph nodes or edges. An incorrect dependence graph results in invalid abstract states, thus leads to validation fails. Note that

Table 2: Performance of the validator: times (in seconds) and memory consumptions (in megabytes) are represented for all benchmarks with respect to the analyzer and the validator. The performance is evaluated for the analyzer that bugs are fixed by validation ($\textbf{Analyzer}_{\textsf{fixed}}$). **LOC** shows the number of lines of code, calculated with `wc`. The validator has largely three phases: **Trs** reports the data translation time. **Dns** reports the densification time. Lastly, **Val** reports the time for whole validations, including the prefixed point validation. $\textbf{Cmp}_{\textsf{Time}}$ indicates how much the validator is faster than the analyzer. Similarly, $\textbf{Cmp}_{\textsf{Mem}}$ indicates how less the validator consumes memory than the analyzer.

Programs	LOC	$\textbf{Analyzer}_{\textsf{Fixed}}$		Validator					$\textbf{Cmp}_{\textsf{Time}}$	$\textbf{Cmp}_{\textsf{Mem}}$
		Time	**Mem**	**Trs**	**Dns**	**Val**	**Time**	**Mem**		
spell-1.0	2K	1.2	46	0.1	0.2	0.1	0.4	4	3.34 x	0.09 x
gzip-1.2.4a	7K	13	126	1	3	1	5	37	2.76 x	0.29 x
combine-0.3.3	11K	24	196	2	3	1	6	28	3.99 x	0.14 x
bc-1.06	13K	40	165	4	23	7	34	337	1.20 x	2.04 x
tar-1.13	20K	149	408	10	33	7	50	242	3.06 x	0.59 x
coan-4.2.2	22K	137	724	16	36	9	61	406	2.30 x	0.56 x
less-382	23K	280	479	45	133	24	201	718	1.43 x	1.50 x
make-3.76.1	27K	497	1299	30	106	10	146	496	3.49 x	0.38 x
cflow-1.3	34K	15	94	1	3	1	5	30	2.75 x	0.32 x
wget-1.9	35K	275	1041	24	51	8	83	458	3.43 x	0.44 x
screen-4.0.2	45K	1772	2899	184	389	28	600	1814	3.03 x	0.63 x
asn1c-0.9.21	50K	927	2185	76	320	96	493	2878	1.95 x	1.31 x
judy-1.0.5	87K	466	677	20	58	59	136	198	3.44 x	0.29 x
gsasl-1.6.1	91K	3493	754	828	342	82	1252	116	2.79 x	0.15 x
openssh-5.8p1	102K	4303	5485	1050	5060	650	6760	7308	0.66 x	1.33 x
lsh-2.0.4	111K	1714	2655	472	1972	461	2905	6768	0.62 x	2.55 x

drawing a dependence graph involves a lot of optimizations, so it is quite complicated. The fact that about 40% of bugs are due to dependence graph indicates that complex implementations are more likely to induce bugs, and thus finding these bugs are meaningful.

7 Discussion & Related Work

Translation Validation Our validator is regarded as a translation validation in that the analyzer *translates* a source program into the abstract analysis result. Instead of checking semantic equivalence of two programs, our validator checks that the analysis result is a prefixed point of an abstract semantics.

Formal Verification of Static Analysis A verified validation approach has practical merits in comparison with formal verification of static analysis [Bertot(2009), Leroy(2011), Blazy et al.(2013)Blazy, Laporte, Maroneze, and Pichardie]. Because of the complicated nature (deliberate unsoundness, highly-enginneered optimizations, and etc.) of a realistic static analyzer, implementing a verified validator is pragmatic than verifying such a static analyzer.

Efforts for the Verification A project took 6 man-months, of which the most efforts (5 man-months) are done for proving the validator in Coq. The rest one man-month were used to debug the target analyzer once the validations failed.

Specification of Unsoundness for Complex Analyzers As seen in Section 3, we invented a simple model where some unsoundness is specified easily by syntactically changing the input program. However, our simple model does not hold when the analyzer employs more precise optimizations such as context-sensitivity. One problem can be simulated with following example code.

```
10:  int* p;
11:  void f() {
12:    // p → ⊤
13:    h();
14:  }
15:  void g() {
16:    // p → null
17:    h();
18:  }
19:  void h() {
20:    *p = 1;
21:  }
```

Suppose the analyzer now is 1-CFA context-sensitive, so it can determine the caller of a function in which the analyzer now reaches. In the example, when the analyzer reaches the line `19`, it remembers what function called `h`, either `f` or `g` at the line `12` or `16`, respectively. In other words, there are two abstract memories at the line `19`, one with the context `f` and the other with `g`. When `h` is called from `f`, according to our simple model, we should comment out the line `19`, since `p` is approximated as the top value at the line `11`. However, we should not do this because `p` is not approximated as the top value when `g` calls `h`.

We have to develop more complex transformation when we meet a new optimization of the analyzer. For the example above, we may solve this problem by reproducing the main body of `h`, naming it `h'`, and letting `g` to call `h'` instead of `h`. In this case, we can comment out `*p = 1;` in `h` since now `h` is called only from `f`, where `p` is approximated by the top value.

References

[Bertot(2009)] Y. Bertot. Structural abstract interpretation, a formal study in Coq. In A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, revised tutorial lectures*, volume 5520 of *Lecture Notes in Computer Science*, pages 153–194. Springer, 2009. URL `http://hal.inria.fr/inria-00329572/`.

[Blazy et al.(2013)Blazy, Laporte, Maroneze, and Pichardie] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Proc. of the 20th Static Analysis Symposium (SAS 2013)*, Lecture Notes in Computer Science. Springer-Verlag, 2013. To appear.

[Chase et al.(1990)Chase, Wegman, and Zadeck] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 296–310, 1990.

[Choi et al.(1991)Choi, Cytron, and Ferrante] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66, 1991.

[Cytron and Ferrante(1995)] R. K. Cytron and J. Ferrante. Efficiently computing ϕ-nodes on-the-fly. *ACM Trans on Programming Languages and Systems*, 17:487–506, May 1995.

[Dhamdhere et al.(1992)Dhamdhere, Rosen, and Zadeck] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '92, pages 212–223, New York, NY, USA, 1992. ACM.

[Hardekopf and Lin(2009)] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 226–238, 2009.

[Hardekopf and Lin(2011)] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 289–298, 2011.

[Johnson and Pingali(1993)] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 78–89, 1993.

[Leroy(2011)] X. Leroy. Proving a compiler: Mechanized verification of program transformations and static analyses. `http://gallium.inria.fr/~xleroy/courses/Eugene-2010/`, June 2011. Oregon Programming Languages Summer School.

[Oh et al.(2011)Oh, Brutschy, and Yi] H. Oh, L. Brutschy, and K. Yi. Access analysis-based tight localization of abstract memories. In *VMCAI 2011: 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2011.

[Oh et al.(2012)Oh, Heo, Lee, Lee, and Yi] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[Oh et al.(2013)Oh, Heo, Park, Kang, and Yi] H. Oh, K. Heo, D. Park, J. Kang, and K. Yi. Global sparse analysis framework. Technical Memorandum ROSAEC-2013-014, Research On Software Analysis for Error-free Computing Center, Seoul National University, March 2013.

[Pnueli et al.(1998)Pnueli, Siegel, and Singerman] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64356-7. URL `http://dl.acm.org/citation.cfm?id=646482.691453`.

[Ramalingam(2002)] G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.

[Reif and Lewis(1977)] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–118, 1977.

[Rice(1953)] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[Tok et al.(2006)Tok, Guyer, and Lin] T. B. Tok, S. Z. Guyer, and C. Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the International Conference on Compiler Construction*, pages 17–31, 2006.

[Wegman and Zadeck(1991)] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans on Programming Languages and Systems*, 13:181–210, April 1991.