SparrowBerry: A Verified Validator for an Industrial-Strength Static Analyzer

Sungkeun Cho, Jeehoon Kang, Joonwon Choi, and Kwangkeun Yi

Seoul National University
{skcho,jhkang,jwchoi,kwang}@ropas.snu.ac.kr

Abstract. In this article we present SparrowBerry: a verified validator for Sparse Sparrow, an industrial-strength static analyzer for the C language. Sparse Sparrow is a sound, global, yet scalable static analyzer whose design is proven correct by the abstract interpretation and our general sparse analysis frameworks. However, it does not necessarily mean that the implementation, which has lots of engineering, is also correct conforming to the design. To solve this problem, we attach a verified validator to the analyzer: the validator checks if the analysis result from Sparse Sparrow is indeed a sound abstract semantics of the input C program. The validator is extracted from our 20 K-line Coq proof for the correctness of the underlying "vanilla" abstract interpreter of Sparse Sparrow. We have demonstrated the feasibility of this verified validator by experiments with realistic benchmarks.

1 Introduction

Motivation. Sparse Sparrow [9–12, 15, 17–20] is our industrial-strength static analyzer for C, whose design is proven correct in the abstract interpretation [7, 8] and the general sparse analysis [18] frameworks. Sparse Sparrow chases C's whole semantic behavior by estimating numbers, pointers, dynamic memory allocations, procedure calls, etc. in order to find safety errors such as buffer overrun and null dereference. Sparse Sparrow globally analyzes the whole program as one unit starting from the input C program's main procedure, yet it is scalable too: built on top of the *general sparse analysis framework* [18], Sparse Sparrow can globally analyze up to million lines of C code with a practically useful precision.

Though the design is correct, guaranteeing that the analysis result is always a sound approximation of the input program's all executions, the implementation that achieved a realistic cost-accuracy performance is fairly complex and thus can easily lead to unsound analysis results. Sparse Sparrow consists of 150,000 lines of OCaml code with lots of optimizations, among which one key optimization is to make the analysis sparse [18]. For example, we had to economically and safely estimate the data dependencies between abstract locations across procedural boundaries. For this engineering, we used the static single assignment (SSA) transformation [22] with binary decision diagrams [5], etc.

We thus have developed a verified validator that checks whether the the analysis results from Sparse Sparrow is correct. The analysis result is supposed, in design, to be a fixpoint of a correct abstract semantic function. Our validator SparrowBerry has a proven-correct abstract semantic function. Given the analysis result from Sparse Sparrow, SparrowBerry checks if the result is actually a fixpoint of the correct abstract semantic function. We mechanized the correctness proof of the abstract semantic function in Coq and SparrowBerry is extracted by Coq from this mechanization. Our trust base is largely the Coq system and analysis-result translations from Sparse Sparrow to SparrowBerry.

Overview. Figure 1 shows an overview of the validation process using SparrowBerry. Sparse Sparrow analyzes an input program and returns an analysis result (a fixpoint of $\hat{F}$ in OCaml). Given an input program and an analysis result, SparrowBerry checks whether the analysis result is correct (a post-fixpoint of $\hat{F}$ in Coq). If it is, the analysis result is guaranteed to be a sound approximation of the concrete semantics of the input program; otherwise, it guarantees nothing.

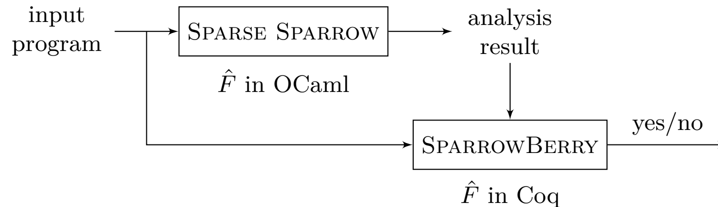


Fig. 1. The overview of validation using SparrowBerry. The $\hat{F}$ is the analysis function (an abstract semantic function).

Contributions. Our contributions are as follows.

- We present a formally verified validator SparrowBerry for Sparse Sparrow, our industrial-strength sound static analyzer for full C. By this combination, we acquire trusted real-world static analysis results for C. To the best of our knowledge, no previous static analyzer achieves this: trusted analysis results from an industrial-strength safety-error-detecting static analyzer for full C.
- We experimentally demonstrate SparrowBerry's feasibility in validating Sparse Sparrow's results for realistic C benchmarks.

Expression	e ::=	`literal`(l)	(literal)
	|	`bop`(b, e, e)	(binary operation)
	|	`uop`(u, e)	(unary operation)
	|	lv	(value-of operator)
	|	`&`lv	(address-of operator)
l-value	lv ::=	v	(variable)
	|	`*`e	(dereference)
	|	e`[`e`]`	(array access)
	|	e`.`m	(struct access)
Allocation	$alloc$ ::=	`[`e`]`	(array allocation)
	|	`{`x`}`	(struct allocation)
Command	c ::=	lv `:=` e	(assignment)
	|	lv `:=` `alloc`$(label, alloc)$	(allocation)
	|	`assume`(e)	(assumption)
	|	`call`(`list` f, `list` e)	(function call)
	|	`return`$_f$	(function return)
	|	`libcall`($lcall$, `list` e)	(library call)
		...	

Fig. 2. Excerpt of the syntax of the input of Sparse Sparrow

Outline. Section 2 presents the target static analyzer Sparse Sparrow. Section 3 explains the correctness proof of our validator which is mechanized in Coq. Section 4 discusses the proof mechanization issues. Section 5 presents the experimental results. Section 6 concludes with discussions and related works.

2 Target Analyzer Sparse Sparrow

We introduce Sparse Sparrow, the target analyzer of the validation. It analyzes C programs and detects safety errors, e.g. buffer overrun, divided by zero, and null pointer dereference. In this section, we introduce only some key features of it. Sect. Appendix A has more details of the analysis.

Input and Output. The input C code to Sparse Sparrow is represented as a control-flow-graph (CFG) whose nodes are basic blocks. A basic block contains a list of commands. Figure 2 presents an excerpt of the syntax of the input program [16]. Note that the *control-related* commands, such as `if`, `goto`, and function call, are desugared to control flow edges and basic blocks with their conditions explicit by `assume` command.

The output of Sparse Sparrow is a map from program points to abstract memories. An abstract memory is in turn a map from abstract locations to abstract values.

Soundness. Based on the abstract interpretation framework [7], SPARSE SPARROW is sound in design. It is guaranteed that the output abstract memories over-approximate every reachable concrete memory for each program point. Over-approximation means for every location and its value of the concrete memory, the abstract value of the corresponding abstract location in the abstract memory over-approximates the concrete value. Based on this result, the analyzer alarms possible safety errors.

Analyzed Semantics. SPARSE SPARROW faithfully chases whole C's semantic behavior, including numbers, pointers, procedure calls, dynamic allocations, etc. SPARSE SPARROW can track semantic properties across function calls throughout the whole program.

Scalability. SPARSE SPARROW is scalable, based on the global sparse analysis framework [18]. The sparse analysis framework reduces the analysis cost by saving unnecessary abstract memory propagation during the analysis. In the ordinary *dense* analysis, values are propagated along control flows. In the sparse analysis, unnecessary propagation is eliminated by exploiting the *data dependency* [18]. For example of the sparse analysis, in Fig. 3, the value at the location `x` defined at the program point ⓐ is not propagated to ⓑ, but directly to ⓒ. This is because `x` defined at ⓐ is not used in ⓑ but in ⓒ. In this way, SPARSE SPARROW succeeded in globally analyzing million lines of C code.

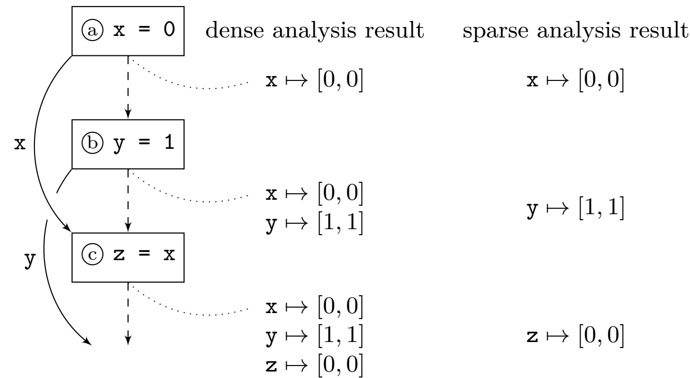


Fig. 3. The figure shows the missing entries on the sparse analysis result. Solid lines represent the data dependency, while dashed lines represent the control flow. In the sparse analysis, an abstract value is propagated only to the program points where it is used.

3 Correctness Proof of Validator SparrowBerry

In this section, we present the design of the validator SparrowBerry and prove its correctness. The validator and its soundness proof are mechanized in Coq. See our Coq development[1] for the full details.

Validation. The result of Sparse Sparrow is expected to be sound with respect to the abstract interpretation framework. Formally, for all program p, the analysis result Sparse Sparrow(p) is expected to be a post-fixpoint of the underlying abstract semantics of Sparse Sparrow.

However, the underlying abstract semantics of Sparse Sparrow exists only in design. Sparse Sparrow is an OCaml program that implements the design with lots of engineering for performance optimization. The implementation correctness is not known.

Instead of proving the implementation itself, we develop a verified validator that checks the analysis result's correctness. The validator's correctness is proven in Coq.

The validator SparrowBerry checks whether the input analysis result is a post-fixpoint of its own abstract semantic function $\hat{F}_p : \hat{\mathbb{S}} \to \hat{\mathbb{S}}$, i.e.

$$\textsc{SparrowBerry}(p, \hat{s}) = \hat{F}_p(\hat{s}) \sqsubseteq \hat{s} \ .$$

As usual in the abstract interpretation framework, the abstract semantic function $\hat{F}_p$ should "over-approximate" the concrete semantic relation F_p (Lemma 2) and satisfies some natural properties (Lemma 1, 3). Note that the abstract semantic function $\hat{F}_p$ is specific to the validator, and that of Sparse Sparrow is irrelevant to the soundness of the validator.

Densifying sparse analysis result. The validator SparrowBerry targets on dense analysis results, so the sparse analysis result from our Sparse Sparrow cannot be directly checked by SparrowBerry. This is because sparse analysis eliminates unnecessary propagation, making some entries empty. To address this, we have to fill entries by exploiting *helper data dependency* [18]. This process degrades the scalability of SparrowBerry by increasing the sizes of analysis results. See Sect. 4.2 for more details on this process.

3.1 Correctness

The validator SparrowBerry is *sound* in the sense that if the validation succeeds, the analysis result indeed over-approximates the concrete semantics. In this section, we state the soundness formally and present a sketch of proof, which is mechanized in Coq.

We use the proof technique of [3, 4, 13]. The soundness proof has two features. First, the soundness is represented by a relation between concrete values

[1] `http://ropas.snu.ac.kr/sparrowberry/`

and abstract values; collecting semantics does not appear in the soundness definition as in the abstract interpretation. Second, the soundness is proved by a simple induction on the number of execution steps; which is showing that the relation between concrete and abstract values is preserved on all of the program executions.

This proof approach reduces the proof burden of the abstract interpretation framework. First, semantic domains are not required to have hand-to-define-in-coq structures such as complete partial order or complete lattice. This is because the least fixpoints are no longer considered: for the concrete semantics, only reachable states in finite steps are considered by induction; for the abstract semantics, we validate a post-fixpoint instead of the least fixpoint. Second, the abstract semantic function $\hat{F}_p$ of SparrowBerry is not required to be monotonic. Third, the requirement of the Galois connection is reduced to Lemma 3.

Formally, the validator SparrowBerry is sound as follows:

Theorem 1 (Soundness of SparrowBerry). *For all program p, abstract state $\hat{s}$, and reachable state $s \in [\![p]\!]$,* SparrowBerry$(p, \hat{s}) =$ `true` *implies* $s \ \mathrm{R}_{\mathrm{State}} \ \hat{s}$.

Before presenting a sketch of proof, we elaborate the meaning of the soundness theorem.

Abstraction relation. We introduce the *abstraction relation* between concrete and abstract objects such as values, memories, and states. Simply put, a concrete object c and an abstract object a are abstractly related if a over-approximates c. For example, the interval $[1, 5]$ over-approximates 3, so we may define the abstract relation $R_{\mathbb{Z}}$ on integers so that

$$3 \ \mathrm{R}_{\mathbb{Z}} \ [1, 5] \ .$$

In this way, the abstract relation $\mathrm{R}_{\mathrm{State}}$ on states is designed to satisfy the auxiliary lemmas in the soundness proof.

The relation $\mathrm{R}_{\mathrm{state}}$ is the abstract relation on the concrete and the abstract states, which will be defined later in this section. Before presenting a sketch of proof, we elaborate the meaning of the soundness theorem.

Over-approximation of concrete semantics. For an input program p, let $F_p \subseteq \mathbb{S} \times \mathbb{S}$ be the operational semantic relation of the small-step concrete semantics for the program p. The semantics $[\![p]\!] \in 2^{\mathbb{S}}$ is defined as follows:

$$[\![p]\!] ::= \{s \mid s_0 \ F_p^* \ s\}$$

where the initial state s_0 is defined as a pair of the global entry node and the empty memory. An analysis result $\hat{s}$ over-approximates the concrete semantics of a program p if for all reachable state $s \in [\![p]\!]$, $s \ \mathrm{R}_{\mathrm{State}} \ \hat{s}$.

Now we are ready to prove the soundness.

Proof (Theorem 1). Since $s \in [\![p]\!]$, there exists n such that

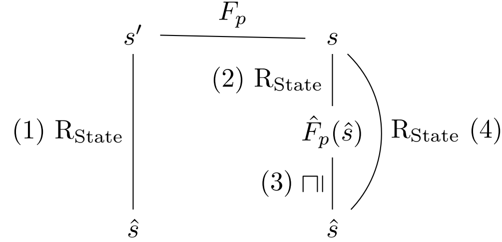
$$s_0 \; F_p^n \; s \; .$$

We prove by induction on n.

- Case $n = 0$.
  We have $s = s_0$ since $s_0 \; F_p^0 \; s$. Lemma 1 proves this initial case.
- Case $n = m + 1$.
  Since $s_0 \; F_p^{m+1} \; s$, there exists $s' \in \mathbb{S}$ such that

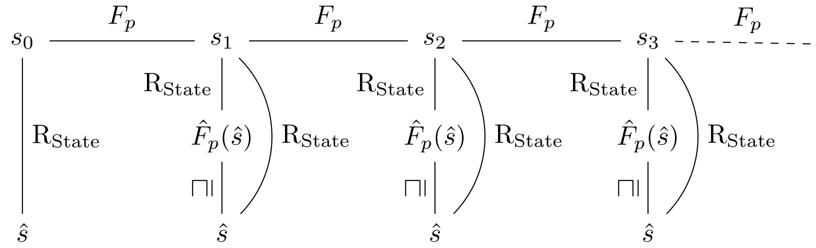
  $$s_0 \; F_p^m \; s' \text{ and } s' \; F_p \; s \; .$$

  For the rest of the proof, see the diagram below. The item numbers match with the corresponding propositions in the diagram.

  1. By induction hypothesis, we have $s' \; \mathrm{R}_{\mathrm{State}} \; \hat{s}$.
  2. By Lemma 2 on the relation between the concrete and the abstract semantics, we have $s \; \mathrm{R}_{\mathrm{State}} \; \hat{F}_p(\hat{s})$.
  3. We have SparrowBerry$(p, \hat{s}) =$ `true`. By the definition of Sparrow-Berry, we have $\hat{F}_p(\hat{s}) \sqsubseteq \hat{s}$.
  4. By Lemma 3 on the relation between the abstract relation and the order of the abstract state, we have $s \; \mathrm{R}_{\mathrm{State}} \; \hat{s}$.

  The figure below summarizes the structure of the proof.



□

It remains to prove Lemma 1, 2, and 3. They all are stated and proved in Coq, but we omit the proofs here. In the following diagrams, solid lines are the premises and dashed lines are the conclusions of the Lemmas.

The initial concrete state is a pair of an initial program point and an empty concrete memory. Lemma 1 means that the initial concrete state has abstract relations with all abstract memories.

Lemma 1. *For all $\hat{s} \in \hat{\mathbb{S}}$, we have $s_0 \; \mathrm{R}_{\mathrm{State}} \; \hat{s}$.*

$$\begin{array}{c} s_0 \\ \vdots \; \mathrm{R}_{\mathrm{State}} \\ \hat{s} \end{array}$$

The abstract semantic function $\hat{F}_p$, implemented in Coq, is designed to abstract the one-step concrete execution F. For the soundness, it is not required to be the same with that of the untrusted fixpoint solver. However, for the completeness, the abstract semantic function $\hat{F}_p$ should be more precise than that of the untrusted fixpoint solver. See Sect. 6 for more discussions on the completeness.

Lemma 2 means that the abstract execution over-approximates the concrete execution. It is the only condition that the abstract semantic function should satisfy.

Lemma 2. *For all concrete state $s, s' \in \mathbb{S}$ and abstract state $\hat{s} \in \hat{\mathbb{S}}$, $s \; F_p \; s'$ and $s \; \mathrm{R}_{\mathrm{State}} \; \hat{s}$ implies $s' \; \mathrm{R}_{\mathrm{State}} \; \hat{F}_p(\hat{s})$.*

$$\begin{array}{ccc} s & \xrightarrow{F_p} & s' \\ \mid \; \mathrm{R}_{\mathrm{State}} & & \vdots \; \mathrm{R}_{\mathrm{State}} \\ \hat{s} & & \hat{F}_p(\hat{s}) \end{array}$$

Lemma 3 means the abstraction relation should be consistent to the order relation of abstract states. It is straightforward from the intuitive definitions.

Lemma 3. *For all $s \in \mathbb{S}$ and $\hat{s} \in \hat{\mathbb{S}}$, $s \; \mathrm{R}_{\mathrm{State}} \; \hat{s}$ and $\hat{s} \sqsubseteq \hat{s}'$ implies $s \; \mathrm{R}_{\mathrm{State}} \; \hat{s}'$.*

$$\begin{array}{c} s \\ \mathrm{R}_{\mathrm{State}} \; \mid \\ \hat{s} \\ \sqcap\!| \; \mid \\ \hat{s}' \end{array} \quad (s \; \mathrm{R}_{\mathrm{State}} \; \hat{s}' \text{ dashed})$$

3.2 Abstraction Relation

As mentioned, we define abstraction relations between concrete and abstract domains. A concrete object c and an abstract object a are abstractly related if a soundly over-approximates c.

The relation is defined constructively from primitive domains such as integer and location domains to complex ones such as array, record, memory, and state domains. Since the construction of the relations is quite standard, we only present examples on interval, memory, and state domain in the paper.

Interval. We define the abstraction relation $\mathrm{R_{Int}} \subseteq \mathbb{Z} \times \hat{\mathbb{Z}}$ on the set of integers and the interval domain. Note that the set $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty, -\infty\}$ is lifted from $\mathbb{Z}$ and the interval domain $\hat{\mathbb{Z}} = \mathbb{Z}_\infty \times \mathbb{Z}_\infty$ is the product of two sets of lifted integers for lower and upper bounds. Formally, the abstraction relation $\mathrm{R_{Int}} \subseteq \mathbb{Z} \times \hat{\mathbb{Z}}$ on integers is defined as follows:

$$z \; \mathrm{R_{Int}} \; [z_1, z_2] \triangleq z_1 \leq z \leq z_2 \ .$$

The order is obviously defined.

Memory. We define the abstraction relation $\mathrm{R_{Mem}} \subseteq \mathrm{Mem} \times \hat{\mathrm{Mem}}$ on concrete and abstract memory domains in a pointwise manner. Note that a concrete memory is a map from locations to values and an abstract memory is a top-lifted partial map from sets of locations to abstract values (see Sect. A for more details of domains). Formally, the abstraction relation $\mathrm{R_{Mem}} \subseteq \mathrm{Mem} \times \hat{\mathrm{Mem}}$ on the memories is defined as follows:

$$\begin{aligned} m \; \mathrm{R_{Mem}} \; \hat{m} \triangleq\ & \forall l \in \mathrm{Loc}, v \in \mathrm{Val} : m(l) = v \rightarrow v \; \mathrm{R_{Val}} \; \hat{m}\,(\{l\}) \\ & \vee\ \hat{m} = \top \ . \end{aligned}$$

State. A concrete state (pp, m) is a pair of a program point pp and a memory m. An abstract state is a map from program points of the input program to abstract memories. A concrete state is abstractly related to an abstract state if the concrete memory is abstractly related to the corresponding abstract memory of the program point of the concrete state. Formally, we define the abstraction relation $\mathrm{R_{State}} \subseteq \mathrm{State} \times \hat{\mathrm{State}}$ on state domains as follows:

$$(pp, m) \; \mathrm{R_{State}} \; \hat{s} \triangleq m \; \mathrm{R_{Mem}} \; \hat{s}(pp) \ .$$

4 Proof Mechanization

4.1 Semantic Domains in Coq

In this section, we introduce domain module types and functors defined in SparrowBerry.

Lattice. Figure 4 shows the `Lattice` module type defined in Coq. The `Lattice` module type is defined by the definitions of partial ordered set, join and meet operations, bottom and top values, and their related properties. The definition of partial ordered set is omitted since it is trivial [21].

The `Lattice` module type is based on Pichardie's work [21]. Most of the specifications in the module type are exactly the same as his work. However, there are two main differences between theirs and ours: (1) one is that more specific lemmas that should be proved for making modules of the module type are added to avoid the repetition of some useful definitions and proofs, e.g.

```
Module Type Lattice.
  Include Poset.

  Parameter join : binop t.
  Parameter join_spec_left  : forall x y : t, order x (join x y).
  Parameter join_spec_right : forall x y : t, order y (join x y).
  Parameter join_spec_least : forall x y u : t,
            order x u -> order y u -> order (join x y) u.
  Parameter join_comm : forall x y : t, eq (join x y) (join y x).

  Parameter meet : binop t.
  Parameter meet_spec_left     : forall x y : t, order (meet x y) x.
  Parameter meet_spec_right    : forall x y : t, order (meet x y) y.
  Parameter meet_spec_greatest : forall x y u : t,
            order u x -> order u y -> order u (meet x y).
  Parameter meet_comm : forall x y : t, eq (meet x y) (meet y x).

  Parameter bottom : t.
  Parameter bottom_spec    : forall x : t, order bottom x.
  Parameter join_bottom_id : forall x, eq x (join x bottom).

  Parameter top : t.
  Parameter top_spec : forall x : t, order x top.
  Parameter top_must : forall x, order top x -> x = top.
  Parameter meet_top_id : forall x, eq x (meet x top).
End Lattice.
```

Fig. 4. The `Lattice` module type

`top_must`; (2) another is that it does not include the lemmas related to the termination of the analysis. Note that the termination guarantee is necessary to make a verified analyzer, not a verified validator like SparrowBerry.

There is an additional lemma `top_must`, the meaning of which is that if a value is bigger than or equal to top then the value is *syntactically* equal to the top. This lemma is satisfiable in the lattices we are dealing with, while it is not satisfiable in lattices in general. This stronger condition for the lattice makes the proof simple since we can easily use `rewrite` tactic, which substitutes a term in a proposition to another syntactically equivalent term in Coq.

Countable. The `Countable` module type plays an important role in the `Map` module type. Figure 5 shows the module type `Countable`. By the definition of the countable set, all of the elements in the set should be mapped onto natural numbers, and the mapping should be an one-to-one function. The `represent` function mimics the mapping. A difference between the `represent` function and the mapping is that `represent` is the *partial* function that returns optional natural numbers. By means of the difference, finite lists of a set can be defined as countable sets without using the notion of dependent type in Coq. There are two lemmas that `represent` function should satisfy: the function and its inverse function preserve the equality of `t` type that is the type of the elements in a countable set. Hence, it guarantees that different values of the countable set are mapped onto different natural numbers each other.

```
Module Type Countable.
  Include Equiv.

  Parameter represent : t -> option nat.
  Parameter represent_preserve_eq : forall (i j : t),
              eq i j -> represent i = represent j.
  Parameter represent_inv_preserve_eq : forall (i j : t) (n : nat),
              represent i = Some n -> represent j = Some n -> eq i j.
End Countable.
```

Fig. 5. The `Countable` module type

Set. Figure 6 shows the `Set` functor constructing set domains. The element type of `Set` (`ind_t`) is defined by two cases: (1) one is a finite set of the elements typed `t` using the Coq standard library `MSetWeakList`; (2) another is top. The input of the functor is an `Equiv`-typed module which is used to make decidability module `Dec`. Finally the set module `DSet` is generated by the decidability module.

```
Module DSetEquiv (Import equiv : Equiv) <: Equiv.
  Module Dec <: DecidableType.DecidableType.
    Definition t : Type := equiv.t.
    Definition eq : t -> t -> Prop := fun x y => equiv.eq x y.
    ...
  End Dec.
  Module DSet := MSetWeakList.Make Dec.

  Inductive ind_t : Type :=
  | T_top : ind_t
  | T_value : DSet.t -> ind_t.
  Definition t : Type := ind_t.
  ...
End DSetEquiv.
```

Fig. 6. The `Set` functor

Map. Our map domain is implemented by the trie datatype. It is similar to radix search tree which Appel et al. used for implementing maps [1], but simpler and less efficient; when the key of a map is a sequence of elements, each edge in trie is labeled with a single element of the key, while those in radix search tree can be labeled with subsequences of the key. Figure 7 shows the definition of the `Map` functor. The inputs of the functor are two modules which represent keys and values of the map. The binary form of the key value guides the root to the node in which the corresponding value is stored.

In general, functions on `Map` have two phases: (1) one is to translate a key value to a binary number; (2) another is to operate some functionalities using the binary number. For example, the addition on `Map` is defined by the functions `add` and `add_pos` in Fig. 7. First in the `add` function, key `k` is translated to a binary number by `represent` and `P_of_succ_nat` functions. Note that `P_of_succ_nat` is a standard function translating natural numbers to binary numbers in Coq. If the key is not representable by `represent`, it returns `None`. Using the translated binary number `p`, the `add_pos` function finds the node corresponding to the input key.

In our map design, key values are stored in nodes, although it is not essential for designing maps. The design choice helps making the fold function that is using key values in its operation. The key values do not need to be stored if they can be restored from the binaries which are the positions the values are stored. However, to restore the original key values, the reverse function of `represent` should be defined for every module typed `Countable`. To simply solve the problem, we just store the original key values in the map datatype and they are used by the fold function directly.

```
Module DMapEquiv (Import key : Countable) (Import val : Equiv) <: Equiv.
  Inductive ind_t :=
  | T_leaf: ind_t
  | T_node: ind_t -> option (key.t * val.t) -> ind_t -> ind_t.
  Definition t : Type := ind_t.
  ...

  Fixpoint add_pos (p:positive) (k:key.t) (v:val.t) (m:t) :=
    match p,m with
      | xH,T_leaf               => T_node T_leaf (Some (k,v)) T_leaf
      | xH,T_node ls oldv rs    => T_node ls     (Some (k,v)) rs
      | xO p',T_leaf            => T_node (add_pos p' k v T_leaf) None T_leaf
      | xO p',T_node ls oldv rs => T_node (add_pos p' k v ls)     oldv rs
      | xI p',T_leaf            => T_node T_leaf None (add_pos p' k v T_leaf)
      | xI p',T_node ls oldv rs => T_node ls     oldv (add_pos p' k v rs)
    end.
  Definition add : key.t -> val.t -> t -> option t := fun k v m =>
    match key.represent k with
      | None => None     (* key is invalid *)
      | Some n => let p := P_of_succ_nat n in
                  Some (add_pos p k v m)
    end.
  ...
End DMapEquiv.
```

Fig. 7. The `Map` functor

4.2 Translating Sparse Analysis Results into Dense Ones

Because SparrowBerry targets on dense analysis results, sparse one from the Sparse Sparrow must be transformed into its dense version. Without this densification SparrowBerry will always fail to validate the result. For example, in Fig. 8, SparrowBerry fails to validate the sparse abstract memory because of the edge from the program point ⓐ to ⓑ: the value of x should be propagated down the control flows.

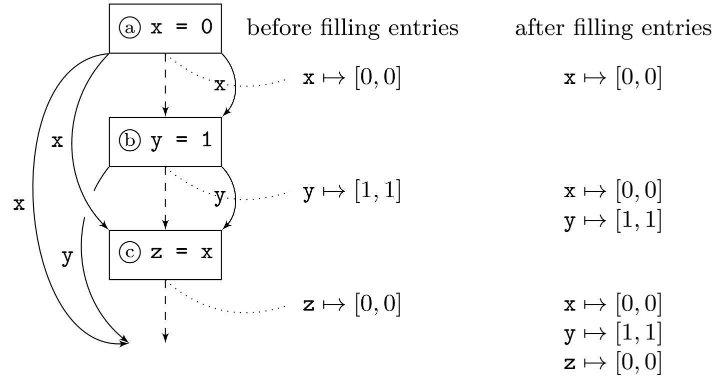


Fig. 8. The figure shows the abstract memories before and after filling the missing entries. Solid lines represent the helper data dependency, while dashed lines represent the control flow. The dense analysis result is restored from the sparse analysis result.

To address this, we fill missing entries by exploiting helper data dependency [**?**]. As our example of filling missing entries, in Fig. 8, the value at the location x defined at the program point ⓐ is propagated to ⓑ, ⓒ, and anywhere the definition reaches to. The global sparse analysis framework ensures that in this way, we can obtain the dense analysis result from the sparse analysis result [**?**]. This process degrades the scalability of SparrowBerry by increasing the sizes of analysis results to about 50 times in our benchmarks.

5 Experiments

In this section, we evaluate SparrowBerry. The validation of SparrowBerry has three phases: (1) filling missing entries on sparse abstract result from Sparse Sparrow; (2) translating the data type of an input program and the analysis result into being appropriate for SparrowBerry; (3) validating the analysis result by SparrowBerry. We measured times and peak memory usages for each phase. All experiments were conducted on a Linux 2.6 system running on a single core of Intel 2.40 GHz box with 24 GB of main memory.

Programs	Size		Analysis		Validation				
					Fill.	Trs.	Val.	Total	
	LOC	Blocks	Time	Mem	Time	Time	Time	Time	Mem
spell-1.0	2 K	583	1	39	10	3	5	18	83
gzip-1.2.4a	7 K	4,152	6	65	408	576	1,058	2,042	1,287
bc-1.06	13 K	4,731	20	97	1,038	1,661	2,823	5,522	1,950
tar-1.13	20 K	8,586	28	110	6,432	3,009	5,754	15,195	6,409
make-3.76.1	27 K	9,094	81	176	15,465	9,503	16,819	41,787	10,804

Table 1. The table shows the validation costs, time and memory, of 5 software packages. **Blocks** reports the number of basic blocks in each input program. **Analysis**, **Fill.**, **Trs.**, and **Val.** are phases for analysis, filling, translating, and validating, respectively, and **Total** reports the total costs for the whole validation process. Units for **Time** and **Mem** are seconds and MB, respectively.

SparrowBerry have succeeded in validating 5 software packages from GNU open-source project: spell, gzip, bc, tar, and make. Table 2 shows the experiment results. The results show that the validations of the analysis results of Sparse Sparrow come at a price. For the 5 benchmarks, the whole validation process takes more time by 19–544 times and memory by 2–61 times and all phases contribute to increment of the time.

The main reason for the time and memory overhead is the densifying process. As explained in Sect. 4.2, by the filling, each abstract sparse memory gets bigger than the original one by 36–85 times in average. These big abstract memories obviously delay not only the filling but also the translation and the validation; the number of locations that have to be translated and validated increases heavily.

6 Discussion

6.1 Trust Base

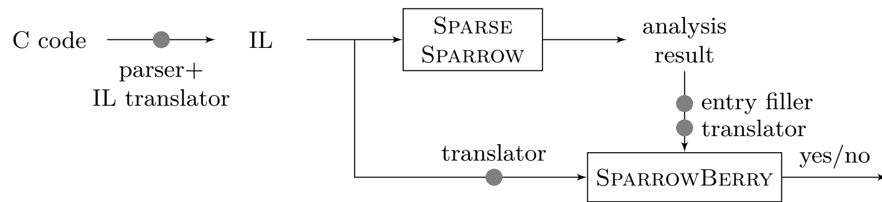


Fig. 9. Trust base of the validation

SparrowBerry trusts the following components: language parser, translators, entry filler, the Coq proof assistant, and its extractor. We give the details of the entry filler and the translators only; others are obvious and not interesting.

First, we have to fill missing entries of the sparse analysis result. This is because SparrowBerry is designed to check the result whose entries are fully filled while Sparse Sparrow gives a sparse result. Details of the entry filling was already explained in Sect. 4.2. Since the process is considered direct, we employ it undoubtedly.

In addition, a target program and the analysis result of the program must be translated to fit for SparrowBerry to validate the result. These translations are considered to be simple, since translations are almost like the one-to-one correspondence.

6.2 Completeness

While the validator SparrowBerry is sound by Theorem 1, it is not formally guaranteed to be *complete*. To be complete with respect to Sparse Sparrow, SparrowBerry should succeed in validating every correct analysis result of Sparse Sparrow. Considering the structure of SparrowBerry, the lack of formal completeness is inevitable. For a formal guarantee of completeness, the analyzer should be written and its properties should be proved in Coq. Note that the validator only depends on its own concrete, abstract semantics and abstraction relations.

SparrowBerry may fail in validation if the abstract semantic function underlying SparrowBerry is less precise than that of Sparse Sparrow. In this case, it is possible that a sound analysis result from Sparse Sparrow is not a post-fixpoint of SparrowBerry, and SparrowBerry says `false`.

However, experiment results indicate that SparrowBerry can be considered to be "complete" enough. As shown in Sect. 5, we experimented with 5 well-known benchmarks and the validator said `true` for all cases. Since we attained the results from our analyzer Sparse Sparrow, we may regard that SparrowBerry is complete, at least experimentally.

6.3 Related Work

There have been various approaches to implement verified static analyzers [3, 6, 13] but all their target languages are not realistic compared to ours. Bertot [3] and Leroy [13] gaved tutorials on how to design verified analyzer for simple imperative languages. Since their tutorials are giving full explanations of soundness proofs of the analyzers, they contributed greatly to our soundness proof design of SparrowBerry. As explained in Sect. 3, the proof strategy reduces the proof burden. Cachera and Pichardie [6] designed verified static analyzer for a simple functional language. The work is focusing on the termination guarantee of a verified analyzer; they formalized lattice domain including widenning and

narrowing in Coq. But in SparrowBerry, we do not have to prove the widening and narrowing operators, since we just checks whether each analysis result is a post-fixpoint.

Besson et al. [4] developed a verified validator that validates the interval analysis results of bytecode programs, but the benchmarks they used in their experiments are simple; the benchmarks are array manipulation-intensive Java programs the sizes of which are about 1 KB.

Acknowledgement

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2012-0000468).

References

1. A. W. Appel, R. Dockins, and X. Leroy. A list-machine benchmark for mechanized metatheory. *Journal of Automated Reasoning*, 49(3):453–491, 2012.
2. G. Barthe, D. Demange, and D. Pichardie. A formally verified ssa-based middle-end. In H. Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 47–66. Springer Berlin Heidelberg, 2012.
3. 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.
4. F. Besson, T. Jensen, and D. Pichardie. A PCC architecture based on certified abstract interpretation. In *Proc. of 1st International Workshop on Emerging Applications of Abstract Interpretation (EAAI'06)*, ENTCS. Springer-Verlag, 2006.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986.
6. D. Cachera and D. Pichardie. A certified denotational abstract interpreter. In *International Conference on Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 9–24, Edimburgh, Royaume-Uni, 2010. Springer-Verlag.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
8. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.
9. Y. Jhee, M. Jin, Y. Jung, D. Kim, S. Kong, H. Lee, H. Oh, D. Park, and K. Yi. Abstract interpretation + impure catalysts: Our Sparrow experience. Presentation at the Workshop of the 30 Years of Abstract Interpretation, San Francisco, `ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf`, January 2008.
10. Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *SAS*, 2005.

11. Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *ISMM*, 2008.
12. W. Lee, W. Lee, and K. Yi. Sound non-statistical clustering of static analysis alarms. In *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 299–314, Berlin, Heidelberg, 2012. Springer-Verlag.
13. 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.
14. G. C. Necula. Proof-carrying code. In P. Lee, F. Henglein, and N. D. Jones, editors, *POPL*, pages 106–119. ACM Press, 1997.
15. H. Oh. Large spurious cycle in global static analyses and its algorithmic mitigation. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, volume 5904 of *Lecture Notes in Computer Science*, pages 14–29, Seoul, Korea, December 2009. Springer-Verlag.
16. H. Oh. *Spatial, Temporal, and Contextual Localization Techniques for Scalable Global Static Analysis*. PhD thesis, Seoul National University, Feb. 2012.
17. 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.
18. 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.
19. H. Oh and K. Yi. An algorithmic mitigation of large spurious interprocedural cycles in static analysis. *Software - Practice and Experience*, 40(8):585–603, 2010.
20. H. Oh and K. Yi. Access-based localization with bypassing. In *APLAS 2011: 9th Asian Symposium on Programming Languages and Systems*, volume 7078 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2011.
21. D. Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *Proc. of the 1st International Conference on Foundations of Informatics, Computing and Software (FICS'08)*, Shangai, China, 2008.
22. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr. 1991.

A Details of SPARSE SPARROW

In this section, we present the main parts of the analysis of SPARSE SPARROW: abstract memory, state, semantics of program, and the soundness. You can find the full description of the analysis in Oh's thesis [16].

Memory and State The domains of the state and the memory are defined as follows.

$$\begin{array}{lll} \mathbb{S} ::= \text{Label} \times \mathbb{M} & \hat{\mathbb{S}} ::= \text{Label} \xrightarrow{\text{fin.}} \hat{\mathbb{M}} & \text{(state)} \\ \mathbb{M} ::= \mathbb{L} \xrightarrow{\text{fin.}} \mathbb{V} & \hat{\mathbb{M}} ::= \hat{\mathbb{L}} \xrightarrow{\text{fin.}} \hat{\mathbb{V}} & \text{(memory)} \end{array}$$

Concrete and abstract memories are maps from concrete and abstract locations to concrete and abstract values, respectively. A concrete state is a pair of a label

and a memory and an abstract state is a map from labels to abstract memories. Note that the labels are partitioning indices.

The powerset of the concrete states and the set of abstract states are Galois-connected:

$$2^{\mathbb{S}} = 2^{\text{Label}\times\mathbb{M}} \quad \underset{\alpha_1}{\overset{\gamma_1}{\longleftrightarrow}} \quad \text{Label} \overset{\text{fin.}}{\to} 2^{\mathbb{M}} \quad \underset{\alpha_2}{\overset{\gamma_2}{\longleftrightarrow}} \quad \text{Label} \overset{\text{fin.}}{\to} \hat{\mathbb{M}} = \hat{\mathbb{S}} \ .$$

We define the required Galois-connection between the powerset $2^{\mathbb{M}}$ of the concrete memories and the set $\hat{\mathbb{M}}$ of the abstract memory as follows:

$$\begin{aligned}
\alpha_{\mathbb{M}} &\in 2^{\mathbb{M}} \to \hat{\mathbb{M}} \\
\alpha_{\mathbb{M}} &::= \lambda n.\lambda \hat{l}.\alpha_{\mathbb{V}}(\{m(l) \in \mathbb{V} \mid m \in n \text{ and } \phi(l) = \hat{l}\}) \\
\gamma_{\mathbb{M}} &\in \hat{\mathbb{M}} \to 2^{\mathbb{M}} \\
\gamma_{\mathbb{M}} &::= \lambda \hat{m}.\{m \in \mathbb{M} \mid \forall l \in \mathbb{L}, m(l) \in \gamma_{\mathbb{V}}(\hat{m}(\phi(l)))\} \ .
\end{aligned}$$

To make them Galois-connected indeed, i.e. $2^{\mathbb{M}} \underset{\alpha_{\mathbb{M}}}{\overset{\gamma_{\mathbb{M}}}{\longleftrightarrow}} \hat{\mathbb{M}}$, we require two conditions. First, we require the set $\hat{\mathbb{L}}$ of abstract locations is a partition of the set $\mathbb{L}$ of locations, i.e. there exists $\phi : \mathbb{L} \overset{\text{fin.}}{\to} \hat{\mathbb{L}}$. We also require the powerset of the concrete values and the set of the abstract values are Galois-connected: $2^{\mathbb{V}} \underset{\alpha_{\mathbb{V}}}{\overset{\gamma_{\mathbb{V}}}{\longleftrightarrow}} \hat{\mathbb{V}}$.

Semantics and Soundness For an input program p, let $F_p \subseteq \mathbb{S} \times \mathbb{S}$ be the small-step concrete operational semantics for the program p. The semantics $[\![p]\!] \in 2^{\mathbb{S}}$ is inductively defined as follows:

$$[\![p]\!] ::= \{s \mid s_0 \ F_p^* \ s\} \ .$$

Let $\hat{F}_p : \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ be the abstract semantic function that satisfies,

$$\alpha_{\mathbb{S}} \circ F_p \sqsubseteq \hat{F}_p \circ \alpha_{\mathbb{S}} \ .$$

We also require that $\hat{\mathbb{V}}$ has the bottom element $\bot$. Then $\hat{\mathbb{S}}$ also has the bottom element $\bot$, and assume that $\alpha_{\mathbb{S}}(\emptyset) = \bot$. Finally we have the soundness, i.e. the least fixpoint of the abstract semantic function over-approximates the concrete semantics: for all program p, we have $[\![p]\!] \subseteq \gamma_{\mathbb{S}}(\text{lfp } \hat{F}_p)$.