A Formally-Verified Alias Analysis

Valentin Robert[1,2] and Xavier Leroy[1]

[1] INRIA Paris-Rocquencourt
[2] University of California, San Diego
`vrobert@cs.ucsd.edu`, `xavier.leroy@inria.fr`

Abstract. This paper reports on the formalization and proof of soundness, using the Coq proof assistant, of an alias analysis: a static analysis that approximates the flow of pointer values. The alias analysis considered is of the points-to kind and is intraprocedural, flow-sensitive, field-sensitive, and untyped. Its soundness proof follows the general style of abstract interpretation. The analysis is designed to fit in the CompCert C verified compiler, supporting future aggressive optimizations over memory accesses.

1 Introduction

Alias analysis. Most imperative programming languages feature *pointers*, or *object references*, as first-class values. With pointers and object references comes the possibility of *aliasing*: two syntactically-distinct program variables, or two semantically-distinct object fields can contain identical pointers referencing the same shared piece of data.

The possibility of aliasing increases the expressiveness of the language, enabling programmers to implement mutable data structures with sharing; however, it also complicates tremendously formal reasoning about programs, as well as optimizing compilation. In this paper, we focus on optimizing compilation in the presence of pointers and aliasing. Consider, for example, the following C program fragment:

```
... *p = 1; *q = 2; x = *p + 3; ...
```

Performance would be increased if the compiler propagates the constant 1 stored in `p` to its use in `*p + 3`, obtaining

```
... *p = 1; *q = 2; x = 4; ...
```

This optimization, however, is unsound if `p` and `q` can alias. Therefore, the compiler is allowed to perform this optimization only if a prior static analysis, called *alias analysis* or *pointer analysis*, establishes that the pointers `p` and `q` differ in all executions of the program.[3]

For another example, consider:

[3] More precisely, the static analysis needed here is called *may-alias analysis* and aims at proving that two pointers are always different. There also exists *must-alias analyses*, which aim at proving that two pointers are always identical, but we will not consider these analyses in this paper.

```
... *p = x; y = *q; ...
```

To give more time to the processor cache to perform the load from `q`, and therefore improve instruction-level parallelism, an instruction scheduling pass would elect to permute the load from `q` and the store to `p`, obtaining:

```
... y = *q; *p = x; ...
```

Again, this optimization is sound only if the compiler can establish that `p` and `q` never alias. Many other optimizations rely on the availability of nonaliasing information. It is fair to say that a precise and efficient alias analysis is an important component of any optimizing compiler.

Compiler verification. Our aim, in this paper, is to equip the CompCert C compiler with a may-alias analysis, in order to enable this compiler to perform more aggressive optimizations over memory accesses. CompCert C is a moderately-optimizing C compiler, producing code for the ARM, PowerPC and x86 architectures [11]. The distinguishing feature of CompCert C is that it is *formally verified* using the Coq proof assistant: a formal, operational semantics is given to every source, intermediate and target language used in CompCert, and a proof of semantic preservation is conducted for every compilation pass. Composing these proofs, we obtain that the assembly code generated by CompCert executes as prescribed by the semantics of the source C program, therefore ruling out the possibility of miscompilation.

When an optimization pass exploits the results of a prior static analysis, proving semantic preservation for this pass requires us to first prove *soundness* of the static analysis: the results of the analysis must, in a sense to be made precise, be a safe over-approximation of the possible run-time behaviors of the program. This paper, therefore, reports on the proof of soundness of an alias analysis for the RTL intermediate language of the CompCert compiler. In keeping with the rest of CompCert, we use the Coq proof assistant both to program the alias analysis and to mechanize its proof of correctness.[4] This work is, to the best of our knowledge, the first mechanized verification of an alias analysis.

The landscape of alias analyses. Like most published may-alias analyses (see Hind [8] for a survey), ours is of the *points-to* kind: it infers sets of facts of the form "this abstract memory location may contain a pointer to that other abstract memory location". Existing alias analyses differ not only on their notions of abstract memory locations, but also along the following orthogonal axes:

– *Intraprocedural vs. interprocedural*: an intraprocedural analysis processes each function of the program separately, making no nonaliasing assumptions about the values of parameters and global variables at function entry. An interprocedural analysis processes groups of functions, or even whole programs, and can therefore infer more precise facts at the entry point of a function when all of its call sites are known.

[4] The Coq development is available at `http://gallium.inria.fr/~varobert/alias/`.

– *Flow-sensitivity*: a flow-sensitive analysis such as Andersen's [1] takes the control flow of the program into account, and is able to infer different sets of facts for different program points. A flow-insensitive analysis such as Steensgaard's [16] maintains a single set of points-to facts that apply to all program points. Consider for example:

```
...   L1: p = &x;   ...   L2: p = &y;   ...
```

  A flow-sensitive analysis can tell that just after `L1`, `p` points only to `x`, and just after `L2`, `p` points only to `y`. A flow-insensitive analysis would conclude that `p` points to either `x` or `y` after both `L1` and `L2`.
– *Field-sensitivity*: a field-sensitive analysis is able to maintain different points-to information for different fields of a compound data structure, such as a C `struct`. A field-insensitive analysis makes no such distinction between fields.
– *Type-based vs. untyped*: many alias analyses operate at the source-language level (e.g. C or Java) and exploit the static typing information of this language (e.g. `struct` declarations in C and `class` declarations in Java). Other analyses ignore static type information, either because it is unreliable (as in C with casts between pointer types or nondiscriminated unions) or because it is not available (analysis at the level of intermediate or machine languages).

These characteristics govern the precision/computational cost trade-off of the analysis, with intraprocedural being cheaper but less precise than interprocedural, flow-insensitive cheaper and less precise than flow-sensitive, and type-based cheaper and more precise than untyped.

The alias analysis that we proved correct in Coq is of the *points-to*, *intraprocedural*, *flow-sensitive*, *field-sensitive*, and *untyped* kind: untyped, because the RTL language it works on is untyped; flow-sensitive, because it instantiates a general framework for dataflow analyses that is naturally flow-sensitive; field-sensitive, for additional precision at moderate extra analysis costs; and intraprocedural, because we wanted to keep the analysis and its proof relatively simple. Our analysis is roughly similar to the one outlined by Appel [2, section 17.5] and can be viewed as a simplified variant of Andersen's seminal analysis [1].

Related work. The literature on may-alias analysis is huge; we refer the reader to Hind [8] for a survey, and only discuss the mechanized verification of these analyses. Many alias analyses are instances of the general framework of abstract interpretation. Bertot [3], Besson *et al.* [4], and Nipkow [15] develop generic mechanizations of abstract interpretation in Coq and Isabelle/HOL, but do not consider alias analysis among their applications. Dabrowski and Pichardie [7] mechanize the soundness proof of a data race analysis for Java bytecode, which includes a points-to analysis, flow-sensitive for local variables but flow-insensitive for heap contents. The analysis is formally specified but its implementation is not verified. Their soundness proof follows a different pattern than ours, relying on an instrumented, alias-aware semantics that is inserted between the concrete semantics of Java bytecode and the static analysis.

Outline. The remainder of this paper is organized as follows. Section 2 briefly introduces the RTL intermediate language over which the alias analysis is conducted. Section 3 explains how we abstract memory locations and execution states. Section 4, then, presents the alias analysis as a forward dataflow analysis. Section 5 outlines its soundness proof. Section 6 discusses a data structure, finite maps with overlapping keys and weak updates, that plays a crucial role in the analysis. Section 7 reports on an experimental evaluation of our analysis. Section 8 concludes and discusses possible improvements.

2 The RTL Intermediate Language

Our alias analysis is conducted over the RTL intermediate language [12, section 6.1]. RTL stands for "register transfer language". It is the simplest of the CompCert intermediate languages, and also the language over which optimizations that benefit from nonaliasing information are conducted. RTL represents functions as a control-flow graph (CFG) of abstract instructions, corresponding roughly to machine instructions but operating over pseudo-registers (also called "temporaries"). Every function has an unlimited supply of pseudo-registers, and their values are preserved across function call. In the following, r ranges over pseudo-registers and l over labels of CFG nodes.

Instructions:	$i ::= \texttt{nop}(l)$	no operation (go to l)
	$\mid \texttt{op}(op, \vec{r}, r, l)$	arithmetic operation
	$\mid \texttt{load}(\kappa, mode, \vec{r}, r, l)$	memory load
	$\mid \texttt{store}(\kappa, mode, \vec{r}, r, l)$	memory store
	$\mid \texttt{call}(sig, (r \mid id), \vec{r}, r, l)$	function call
	$\mid \texttt{tailcall}(sig, (r \mid id), \vec{r})$	function tail call
	$\mid \texttt{cond}(cond, \vec{r}, l_{true}, l_{false})$	conditional branch
	$\mid \texttt{return} \mid \texttt{return}(r)$	function return
Control-flow graphs:	$g ::= l \mapsto i$	finite map
Functions:	$F ::= \{ \texttt{sig} = sig;$	
	$\texttt{params} = \vec{r};$	parameters
	$\texttt{stacksize} = n;$	size of stack data block
	$\texttt{entrypoint} = l;$	label of first instruction
	$\texttt{code} = g \}$	control-flow graph

Each instruction takes its arguments in a list of pseudo-registers $\vec{r}$ and stores its result, if any, in a pseudo-register r. Additionally, it carries the labels of its possible successors. Each function has a stack data block, automatically allocated on function entry and freed at function exit, in which RTL producers can allocate local arrays, structs, and variables whose addresses are taken.

The dynamic semantics of RTL is given in small-step style as a transition relation between execution states. States are tuples $\texttt{State}(\Sigma, g, \sigma, l, R, M)$ of a call stack Σ, a CFG g for the function currently executing, a pointer σ pointing to its stack data block, a label l for the CFG node to be executed, a register state R and a memory state M. (See Leroy [12, section 6.1] for more details on the semantics.)

Register states R map pseudo-registers to their current values: the disjoint union of 32-bit integers, 64-bit floats, pointers, and a special `undef` value. Pointer values $\texttt{Vptr}(b, i)$ are composed of a block identifier b and an integer byte offset i within this block.

Memory states M map (block, offset, memory type) triples to values. (See [14,13] for a complete description of the CompCert memory model.) Distinct memory blocks are associated to 1- every global variable of the program, 2- the stack blocks of every function currently executing, and 3- the results of dynamic memory allocation (the `malloc` function in C), which is presented as a special form of the `call` RTL instruction.

3 Abstracting Memory Locations and Memory States

The first task of a points-to analysis is to partition the unbounded number of memory blocks that can appear during execution into a finite, tractable set of abstract blocks. Since our analysis is intraprocedural, we focus our view of the memory blocks on the currently-executing function, and distinguish the following classes of abstract blocks:

$$\begin{array}{lrl} \text{Abstract blocks:} & \hat{b} ::= & \texttt{Stack} \\ & & \mid \texttt{Globals}(\texttt{Just}\ id) \mid \texttt{Globals}(\texttt{All}) \\ & & \mid \texttt{Allocs}(\texttt{Just}\ l) \mid \texttt{Allocs}(\texttt{All}) \\ & & \mid \texttt{Other} \mid \top \end{array}$$

`Stack` denotes the stack block for the currently-executing function; $\texttt{Globals}(\texttt{Just}\ id)$, the block associated to the global variable id; $\texttt{Allocs}(\texttt{Just}\ l)$, the blocks dynamically allocated (by `malloc`) at point l in the current function; and `Other` all other blocks, including stack blocks and dynamically-allocated blocks of other functions.

The `Stack` and $\texttt{Globals}(\texttt{Just}\ id)$ classes correspond to exactly one concrete memory block each. Other classes can match several concrete blocks. For example, if a call to `malloc` at point l occurs within a loop, several concrete blocks are allocated, all matching $\texttt{Allocs}(\texttt{Just}\ l)$.

To facilitate static analysis, we also introduce summary abstract blocks: $\texttt{Globals}(\texttt{All})$, standing for all the global blocks; $\texttt{Allocs}(\texttt{All})$, standing for all the dynamically-allocated blocks of the current function; and $\top$, standing for all blocks. The inclusions between abstract blocks are depicted in Fig. 1.

Two abstract blocks that are not related by inclusion denote disjoint sets of concrete blocks. We write $\hat{b}_1 \cap \hat{b}_2 = \emptyset$ in this case. If, for instance, the analysis tells us that the pseudo-registers `x` may point to `Stack` and `y` to $\texttt{Allocs}(\texttt{Just}\ 3)$, we know that `x` and `y` cannot alias.

To achieve field sensitivity, our analysis abstracts pointer values not just as abstract blocks, but as abstract locations: pairs $\hat{\ell} = (\hat{b}, \hat{\imath})$ of an abstract block $\hat{b}$

and an abstract offset $\hat{\imath}$, which is either an integer i or $\top$, denoting a statically-unknown offset. We extend the notion of disjointness to abstract locations in the obvious way:

$$(\hat{b}_1, \hat{\imath}_1) \cap (\hat{b}_2, \hat{\imath}_2) = \emptyset \quad \stackrel{\text{def}}{=} \quad \hat{b}_1 \cap \hat{b}_2 = \emptyset \vee (\hat{\imath}_1 \neq \top \wedge \hat{\imath}_2 \neq \top \wedge \hat{\imath}_1 \neq \hat{\imath}_2)$$

For example, the analysis can tell us that `x` maps to the abstract location $(\texttt{Stack}, 4)$ and `y` to the abstract location $(\top, 0)$. In this case, we know that `x` and `y` never alias, since these two abstract locations are disjoint even though the two abstract blocks `Stack` and $\top$ are not.

For additional precision, our analysis manipulates *points-to sets* $\hat{P}$, which are finite sets $\{\hat{\ell}_1, \ldots, \hat{\ell}_n\}$ of abstract locations. For example, the empty points-to set denotes any set of values that can be integers or floats but not pointers; the points-to set $\{(\top, \top)\}$ denotes all possible values; and the points-to set $\{(\texttt{Globals}(\texttt{All}), \top), (\texttt{Other}, \top)\}$ captures the possible values for a function parameter, before the stack block and the dynamic blocks of the function are allocated.

Our points-to analysis, therefore, associates a pair $(\hat{R}, \hat{M})$ to every program point l of every function, where $\hat{R}$ abstracts the register states R at this point by a finite map from pseudo-registers to points-to sets, and $\hat{M}$ abstracts the memory states M at this point by a map from abstract pointers to points-to sets.

4 The Alias Analysis

The alias analysis we consider is an instance of forward dataflow analysis. Given the points-to information $(\hat{R}, \hat{M})$ just "before" program point l, a transfer function conservatively estimates the points-to information $(\hat{R}', \hat{M}')$ "after" executing the instruction at l, and propagates it to the successors of l in the control-flow graph. Kildall's worklist algorithm [9], then, computes a fixed point over all nodes of the control-flow graph. The transfer function is defined by a complex case analysis on the instruction at point l. We now describe a few representative cases.

$$\begin{tikzcd}
 & & & \top & & \\
 & \texttt{Globals(All)} \arrow[urr] & \texttt{Stack} \arrow[ur] & & \texttt{Allocs(All)} \arrow[ul] & \texttt{Other} \arrow[ull] \\
\texttt{Globals(Just } id_1\texttt{)} \arrow[ur] & \cdots & \texttt{Globals(Just } id_n\texttt{)} \arrow[ul] & \texttt{Allocs(Just } l_1\texttt{)} \arrow[ur] & \cdots & \texttt{Allocs(Just } l_n\texttt{)} \arrow[ul]
\end{tikzcd}$$

Fig. 1. Abstract blocks and their inclusion relation

For an arithmetic operation $\mathtt{op}(op, \vec{r}, r, l')$, memory and pseudo-registers other than r are unchanged, therefore the points-to information "after" is $(\hat{R}\{r \leftarrow \hat{P}\}, \hat{M})$. $\hat{P}$ is the abstraction of the result of the operation. Most operations compute integers or floats but not pointers, so we take $\hat{P} = \emptyset$. Other operations form pointers into the stack block or into global variables, where we take $\hat{P} = \{(\mathtt{Stack}, i)\}$ or $\hat{P} = \{(\mathtt{Globals}(\mathtt{Just}\ id), i)\}$ as appropriate. Finally, for `move` instructions as well as pointer addition and pointer subtraction, $\hat{P}$ is determined from the points-to sets $\hat{R}(\vec{r})$ of the argument registers.

For a load instruction $\mathtt{load}(\kappa, mode, \vec{r}, r, l)$, the points-to information "after" is, likewise, of the form $(\hat{R}\{r \leftarrow \hat{P}\}, \hat{M})$, where $\hat{P}$ abstracts the value just loaded. If κ denotes a 32-bit integer-or-pointer quantity, $\hat{P}$ is determined by querying the current abstract memory state $\hat{M}$ at the abstract locations determined by the addressing mode *mode* applied to the points-to sets $\hat{R}(\vec{r})$ of the argument registers. If κ denotes a small integer or floating-point quantity, the RTL semantics guarantee that the result of the load is not a pointer; we therefore take $\hat{P} = \emptyset$.

For a store instruction $\mathtt{store}(\kappa, mode, \vec{r}, r, l)$, pseudo-registers are unchanged but memory is modified. The analysis determines the set L of abstract locations accessed, as a function of *mode* and $\hat{R}(\vec{r})$, then produces the points-to information $(\hat{R}, \hat{M}')$, where $\hat{M}' = \hat{M} \sqcup \{\hat{\ell} \mapsto \hat{R}(r) \mid \hat{\ell} \in L\}$.

Since our abstract pointers correspond, in general, to multiple concrete memory locations, we must perform a *weak update*: the points-to sets associated with $\hat{\ell} \in L$ are not replaced by the points-to set $\hat{R}(r)$, but joined with the latter, using set union. Moreover, we must perform this weak update not only for the abstract locations in L, but also for all the abstract locations that are not disjoint from the locations in L. This weak update is achieved by our definition of the upper bound operation $\sqcup$ over memory maps. The end result is that the new memory map $\hat{M}'$ satisfies the following two properties that are crucial to our soundness proof:

$$\begin{array}{ll} \hat{M}'(\hat{\ell}) \sqsupseteq \hat{M}(\hat{\ell}) & \text{for all abstract locations } \hat{\ell} \\ \hat{M}'(\hat{\ell}) \sqsupseteq \hat{R}(r) & \text{if } \exists \hat{\ell}' \in L,\ \hat{\ell} \cap \hat{\ell}' \neq \emptyset \end{array}$$

As mentioned above, the alias analysis reuses the generic fixed-point solver provided by CompCert [12, section 7.1]. Termination is guaranteed by bounding the total number of iterations and returning $\top$ if no fixpoint is reached within this limit. There is, therefore, no need to prove termination. However, the iteration limit is very high, therefore we must make sure that a fixpoint is reached relatively quickly. For such dataflow analyses, termination is typically ensured by the combination of two facts: the monotonicity of the transfer function and the finite height of the underlying lattice. While our transfer function is monotonic, our lattice of points-to sets does not have a finite height because the sets can grow indefinitely by adding more and more different abstract offsets. To address this issue, we ensure termination by *widening* [6], which accelerates possibly infinite chains by approximation: if the points-to set computed for some memory location or register, at an edge of the control flow graph, contains an accurate memory location (that is, an abstract block with a particular offset) which is a shifted version (that is, the same abstract block and a different offset) of an element of the previous points-to set of that same object, then we widen that points-to set to contain the whole abstract block. This prevents any infinite chain of differing offsets. The lattice quotiented by this widening indeed has a finite height, since the number of abstract blocks to be considered within a function is bounded (the number of global variables and allocation sites is bounded).

5 Soundness Proof

The main contribution of this work is the mechanized proof that the alias analysis is *sound*: namely, that the properties of non-aliasing and flow of pointer values inferred by the analysis are satisfied in every possible execution of the analyzed program. The proof follows the general pattern of abstract interpretation, namely, establishing a correspondence between abstract "things" (blocks, locations, states, etc) and sets of concrete "things", then show that this correspondence is preserved by transitions of the concrete semantics.

The correspondence is presented as relations between abstract and concrete "things", parameterized by an *abstracting function*, called "abstracter" in the Coq development:

```
Definition abstracter := block -> option absb.
```

An abstracter maps every concrete memory block to an abstract memory block, or to `None` if the concrete block is not allocated yet. The abstracter is existentially quantified: the gist of the soundness proof is to construct a suitable abstracter for every reachable concrete execution state.

Given an abstracter, a concrete value belongs to a points-to set if the following predicate holds:

```
Definition valsat (v: val) (abs: abstracter) (s: PTSet.t) :=
  match v with
  | Vptr b o =>
    match abs b with
    | Some ab => PTSet.In (Loc ab o) s
    | None    => PTSet.ge s PTSet.top
    end
  | _         => True
  end.
```

In other words, non-pointer values belong to any points-to set. A pointer value `Vptr b o` belongs to the set `s` if the concrete block `b` is mapped to the abstract block `ab` by the abstracter and if the abstract location `Loc ab o`, or a "bigger" abstract location, appears in `s`. To simplify the proof, we also account for the case where `b` is not mapped by the abstracter, in which case we require `s` to contain all possible abstract locations, i.e. to be at least as large as the $\top$ points-to set.

We extend the `valsat` relation to pseudo-registers and to memory locations. In the following, (`Rhat`, `Mhat`) are the register map and memory map computed by the static analysis at a given program point.

```
Definition regsat (r: reg) (rs: regset) (abs: abstracter) (Rhat: RMap.t) :=
  valsat rs#r abs (RMap.get r Rhat).
```

(The concrete value `rs#r` of register `r` belongs to the points-to set `RMap.get r Rhat` predicted by the analysis.)

```
Definition memsat (b: block) (o: Int.int) (m: mem)
                  (abs: abstracter) (Mhat: MMap.t) :=
  forall v,
  Mem.loadv Mint32 m (Vptr b o) = Some v ->
  match abs b with
   | Some ab => valsat v abs (MMap.get (Loc ab o) Mhat)
   | None    => False
  end).
```

(If, in the concrete memory state, location $(\mathtt{b}, \mathtt{o})$ contains value `v`, it must be the case that `b` is abstracted to `ab` and `v` belongs to the points-to set `MMap.get (Loc ab o) Mhat` predicted by the analysis.)

Not all abstracters are sound: they must map the stack block for the currently-executing function to the abstract block `Stack`, and the blocks for global variables to the corresponding abstract blocks:

```
Definition ok_abs_genv (abs: abstracter) (ge: genv) :=
  forall id b,
    Genv.find_symbol ge id = Some b ->
    abs b = Some (Just (Globals (Just id))).
```

It must also be the case that only valid, already-allocated concrete blocks are abstracted:

```
Definition ok_abs_mem (abs: abstracter) (m: mem) :=
  forall b, abs b <> None <-> Mem.valid_block m b.
```

Piecing everything together, we obtain the following characterization of concrete execution states that agree with the predictions of the static analysis:

```
Inductive satisfy (ge: genv) (abs: abstracter): state -> Prop :=
| satisfy_state: forall cs f bsp pc rs m Rhat Mhat
  (STK:  ok_stack ge (Mem.nextblock m) cs)
  (MEM:  ok_abs_mem abs m)
  (GENV: ok_abs_genv abs ge)
  (SP:   abs bsp = Some (Just Stack))
  (RPC:  (safe_funanalysis f)#pc = (Rhat, Mhat))
  (RSAT: forall r, regsat r rs abs Rhat)
  (MSAT: forall b o, memsat b o m abs Mhat),
  satisfy ge abs (State cs f (Vptr bsp Int.zero) pc rs m)
```

We omit the `ok_stack` predicate, which collects some technical conditions over the call stack `cs`. The `safe_funanalysis` function is the implementation of our alias analysis: it returns a map from program points `pc` to abstract states $(\mathtt{Rhat}, \mathtt{Mhat})$.

In essence, the `satisfy` property says that the abstracter `abs` is sound (premises `MEM`, `GENV`, `SP`) and that, with respect to this abstracter, the values of registers and memory locations belong to the points-to sets predicted by the analysis at the current program point `pc` (premises `RPC`, `RSAT` and `MSAT`).

The main proof of soundness, then, is to show that for every concrete state `st` reachable during the execution of the program, the property `exists abs, satisfy ge abs st` holds:

```
Theorem satisfy_init:
  forall p st,
  initial_state p st ->
  exists abs, satisfy (Genv.globalenv p) abs st.
Theorem satisfy_step:
  forall ge st t st' abs,
  satisfy ge abs st -> step ge st t st' ->
  exists abs', satisfy ge abs' st'.
```

As a corollary, we obtain the soundness of the non-aliasing predictions made on the basis of the results of the analysis:

```
Corollary nonaliasing_sound:
  forall ge abs cs f sp pc rs m Rhat Mhat r1 b1 o1 r2 b2 o2,
  satisfy ge abs (State cs f sp pc rs m) ->
  (safe_funanalysis f)#pc = (Rhat, Mhat) ->
  disjoint (RMap.get r1 Rhat) (RMap.get r2 Rhat) ->
  rs # r1 = Vptr b1 o1 -> rs # r2 = Vptr b2 o2 ->
  Vptr b1 o1 <> Vptr b2 o2.
```

Here, `disjoint` is the decidable predicate stating that two sets of abstract locations are pairwise disjoint, in the sense of the $\hat{l}_1 \cap \hat{l}_2 = \emptyset$ definition above. An optimization that exploits the inferred aliasing information would test whether `disjoint` holds of the points-to sets of two registers `r1` and `r2`. If the test is positive, and since the `satisfy` predicate holds at any reachable state, the corollary above shows that `r1` and `r2` do not alias at run-time, i.e. they cannot contain the same pointer value. In turn, this fact can be used in the proof of semantic preservation for the optimization.

The Coq development consists of about 1200 lines of specifications and 2200 lines of proofs. The proof is entirely constructive: given a suitable abstracter `abs` "before" a transition of the semantics, it is always possible to construct the abstracter `abs'` that satisfies the state after the transition. A large part of the proof is devoted to proving the many required properties of points-to sets and memory maps. A crucial invariant to be maintained is that memory maps $\hat{M}$ maps stay compatible with the inclusion relation between abstract locations: $\hat{M}(\hat{\ell_1}) \subseteq \hat{M}(\hat{\ell_2})$ whenever $\hat{\ell}_1 \sqsubseteq \hat{\ell}_2$. For instance, the points-to set of the abstract block that represents all possible concrete blocks must be a superset of the points-to set of any abstract pointer. We maintain this invariant through the use of dependent types (Coq's subset types).

Additional complications stem from the need to keep the representation of abstract memory states relatively small, eliminating redundant information in order to speed up map updates. We describe our solution in the next section.

6 Maps With Weak Update

Purely-functional finite maps are among the most frequently used data structures in specifications and programs written using proof assistants. The standard signature for total finite maps is of the following form:

```
Module Type Map (K: DecidableType) (V: AnyType).
  Parameter t: Type.
  Parameter init: V.t -> t
  Parameter get: K.t -> t -> V.t
  Parameter set: K.t -> V.t -> t -> t
  Axiom get_init: forall k v, get k (init v) = v
  Axiom get_set_same:
    forall k1 k2 v m, K.eq k1 k2 -> get k1 (set k2 v m) = v
  Axiom get_set_other:
    forall k1 k2 v m, ~K.eq k1 k2 -> get k1 (set k2 v m) = get k1 m
End Map.
```

Here, `K` is the type of map keys, equipped with a decidable equality, and `V` is the type of map values. Three operations are provided: `init`, to create a constant map; `set`, to change the value associated with a given key; and `get`, to obtain the value associated with a key. The semantics of `set` are specified by the familiar "good variable" properties `get_set_same` and `get_set_other` above. Such a signature of total finite maps can easily be implemented on top of an implementation of partial finite maps, such as the AVL maps provided by the Coq library `FMaps`.

To implement the memory maps inferred by our alias analysis, we need a slightly different finite map structure, where the strong update operation `set` is replaced by a weak update operation `add`. During weak update, not only the value of the updated key changes, but also the values of the keys that overlap with / are not disjoint from the updated key. Moreover, the new values of the changed keys are an upper bound of their old value and the value given to `add`. This is visible in the following signature:

```
Module Type OverlapMap (O: Overlap) (L: SEMILATTICE).
  Parameter t: Type.
  Parameter init: t.
  Parameter get: O.t -> t -> L.t.
  Parameter add: O.t -> L.t -> t -> t.
  Axiom get_init: forall x, get x init = L.bot.
  Axiom get_add:
    forall x y s m, L.ge (get x (add y s m)) (get x m).
  Axiom get_add_overlap: forall x y s m,
    O.overlap x y -> L.ge (get x (add y s m)) s.
End OverlapMap.
```

The type of map values, `L.t` is now a *semi-lattice*: a type equipped with a partial ordering `ge`, an upper bound operation `lub`, and a smallest element `bot`. Likewise, the type of keys, `O.t` is a type equipped with a decidable `overlap` relation, which holds when two keys are not disjoint. (Additional operations such as `parent` are included to support the efficient implementation that we discuss next.) Here is the `Overlap` signature:

```
Module Type Overlap.
  Parameter t: Type.
  Parameter eq_dec: forall (x y: t), {eq x y} + {~eq x y}.
  Parameter overlap: t -> t -> Prop.
  Axiom overlap_dec: forall x y, {overlap x y} + {~ overlap x y}.
  Declare Instance overlap_refl: Reflexive overlap.
  Declare Instance overlap_sym: Symmetric overlap.
  Parameter top: t.
  Parameter parent: t -> option t.
  Parameter measure: t -> nat.
  Axiom parent_measure: forall x px,
    parent x = Some px -> measure px < measure x.
  Axiom no_parent_is_top: forall x, parent x = None <-> x = top.
  Axiom parent_overlap: forall x y px,
    overlap x y -> parent x = Some px -> overlap px y.
End Overlap.
```

Note that the `overlap` relation must be reflexive and symmetric, but is not transitive in general. For example, in our application to alias analysis, $(\texttt{Stack}, \top)$ and $(\texttt{Other}, \top)$ do not overlap, but both overlap with $(\top, \top)$.

How, then, to implement the `OverlapMap` data structure? Naively, we could build on top of a regular `Map` data structure, implementing the `add` (weak update) operation by a sequence of `set` (strong updates) over all keys $k_1, \ldots, k_n$ that overlap the given key k. However, the set of overlapping keys is not necessarily finite: if k is `Allocs`(`All`), all keys `Allocs`(`Just` l) overlap. Even if we could restrict ourselves to the program points l that actually occur in the function being analyzed, the set of overlapping keys would still be very large, resulting in inefficient `add` operations.

In practice, during alias analysis, almost all the keys `Allocs`(`Just` l) have the same values as the summary key `Allocs`(`All`), except for a small number of distinguished program points l, and likewise for keys `Globals`(`Just` id). This observation suggests a sparse representation of maps with overlap where we do not store the value of a key if this value is equal to that of its *parent* key.

More precisely, we assume that the client of the `OverlapMap` structure provides us with a *spanning tree* that covers all possible keys. This tree is presented as the `top` element of the `Overlap` structure, representing the root of the tree, and the `parent` partial function, which maps non-`top` keys to their immediate ancestor in the spanning tree. Fig. 2 depicts this spanning tree and the sparse representation in the case where abstract locations are used as keys.

Following these intuitions, we implement the type `t` of `OverlapMap` as a standard, partial, finite map with strong update (written `M` below), as provided by

$$\begin{tikzcd}
(\top, i) \mapsto A \arrow[r] & (\top, \top) \mapsto A \\
(b, i) \mapsto A \arrow[u] \arrow[r] & (b, \top) \mapsto A \arrow[u] \\
(b', i) \mapsto C \arrow[u] \arrow[r] & (b', \top) \mapsto B \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
(\top, i) \arrow[r] & (\top, \top) \mapsto A \\
(b, i) \arrow[r] & (b, \top) \arrow[u] \\
(b', i) \mapsto C \arrow[r] & (b', \top) \mapsto B \arrow[u]
\end{tikzcd}$$

Fig. 2. Sparse maps indexed by abstract locations. Left: the logical view. Arrows represent the inclusion relation between abstract locations. Each location is associated with a value. Right: the concrete representation. Arrows represent the parent relation (spanning tree). Some locations are not associated with a value, because their value is to be looked up in their parent locations.

Coq's `FMap` library for example. We then define the `get` operation as a recursive traversal of the spanning tree, starting at the given key and moving "up" in the tree until a binding is found:

```
Function get (k: O.t) (m: t) {measure O.measure k}: L.t :=
    match M.find k m with
    | Some s => s
    | None => match O.parent k with None => L.bot | Some p => get p m end
    end.
```

The `add` weak update operation, then, can be defined by traversing all the non-default bindings found in the sparse map, and updating those that overlap with the given key:

```
Definition lub_if_overlap (key: O.t) (val: L.t) (k: O.t) (v: L.t): L.t :=
  if O.overlap_dec key k then L.lub val v else v.

Definition add (k: O.t) (v: L.t) (m: t): t :=
  M.mapi (lub_if_overlap k v) (M.add k (get k m) m).
```

Here. `M.mapi` is pointwise application of a function to a finite map: the map returned by `M.mapi` f m maps k to $f\ k\ v$ if and only if m maps k to v.

The initial call to `M.add` is redundant if the key is already present, but necessary when the key is absent, in order to populate the underlying map with the key, at its current value, before performing the traversal. This definition almost satisfies the two "weak good variables" properties `get_add` and `get_add_overlap`: for the latter property, we need to assume that the `O.top` key is bound in the sparse map, otherwise some keys could keep their default `L.bot` value after the weak update. This assumption is easily satisfied by defining the initial map `init` not as the empty sparse map, but as the singleton sparse map `O.top` $\mapsto$ `L.bot`. To make sure that the assumption always holds, we package it along with the sparse maps using a subset type of the form

```
Definition t := { m : M.t | M.In O.top m }
```

This makes it possible to prove the two "weak good variables" properties without additional hypotheses.

This sparse representation of maps, while simple, appears effective for our alias analysis. Two improvements can be considered. One would be to compress the sparse map after each `add` operation, removing the bindings $k \mapsto v$ that have become redundant because k's parent is now mapped to the same value v. Another improvement would be to enrich the data structure to ensure that non-overlapping keys have their values preserved by an `add` update:

```
Conjecture get_add_non_overlap: forall x y s m,
    ~O.overlap x y -> get x (add y s m) = get x m.
```

This property does not hold for our simple sparse representation: assume `x` not bound in the sparse map, its parent `px` bound in the sparse map, `x` non overlapping with `y`, but `px` overlapping with `y`. The value of `px` is correctly updated by the `add y s m` operation, but as a side effect this also modifies the result of `get x` after the `add`.

7 Experimental Evaluation

The first author integrated the alias analysis described here in the CompCert verified compiler and modified its Common Subexpression Elimination pass to exploit the inferred nonaliasing information. CompCert's CSE proceeds by value numbering over extended basic blocks [12, section 7.3]. Without aliasing information, value numbering equations involving memory loads are discarded when reaching a memory write. Using aliasing information, CSE is now able to preserve such equations across memory writes whenever the address of the read is known to be disjoint from that of the write.

This implementation was evaluated on the CompCert test suite. Evaluation consisted in 1- visual examination of the points-to sets inferred to estimate the precision of the analysis, 2- measurements on compilation times, and 3- counting the number of instructions eliminated by CSE as a consequence of the more precise analysis of memory loads enabled by nonaliasing information.

Concerning precision, the analysis succeeds in inferring the expected nonaliasing properties between global and local variables of array or structure types, and between their fields. The lack of interprocedural analysis results in a very conservative analysis of linked, dynamically-allocated data structures, however.

Concerning analysis times, the cost of the analysis is globally high: on most of our benchmarks, overall compilation times increase by 40% when alias analysis is turned on, but a few files in the SPASS test exhibit pathological behaviors of the analysis, more than doubling compilation times.

The additional nonaliasing information enables CSE to eliminate about 1400 redundant loads in addition to the 5600 it removes without this information, a 25% gain. To illustrate the effect, here is an excerpt of RTL intermediate code before (left column) and after (right column) aliasing-aware CSE:

```
  16: x16 = int8signed[currentCodeLen + 0]
  15: x15 = x16 + -8
[...]
   6: int8unsigned[stack(0)] = x10
[...]
   4: x9 = int8signed[currentCodeLen + 0]    →     4: x9 = x16
   3: x7 = x9 + -8                           →     3: x7 = x15
```

The memory store at point 6 was analyzed as addressing offset 0 of the stack block, which cannot alias the global block addressed at point 16. Therefore, when we read that same location at point 4, CSE knows that the result value is the same as that computed at point 16, and therefore reuses the result `x16` of the load at 16. In turn, CSE simplifies the add at point 3, as it knows that the same computation already took place at point 15.

8 Conclusions and Perspectives

An easy simplification that could reduce the cost of alias analysis is to restrict ourselves to points-to sets that are singletons, i.e. a single abstract location instead of a set of abstract locations. The experimental evaluation shows that points-to sets of cardinality 2 or more rarely appear, owing to our use of widening during fixpoint iteration. Moreover, those sets could be collapsed in a single abstract location at little loss of information just by adding a few extra points in the lattice of abstract locations depicted in Fig. 1.

Further improvements in analysis speed appear to require the use of more sophisticated, graph-based data structures, such as the alias graphs used by Larus and Hilfinger [10] and Steensgaard [16], among other authors. It is a challenge to implement and reason upon these data structures in a purely functional setting such as Coq. However, we could circumvent this difficulty by performing validation *a posteriori* in the style of Besson *et al.* [5]: an untrusted alias analysis, implemented in Caml using sophisticated data structures, produces a tentative map from program points to $(\hat{R}, \hat{M})$ abstract states; a validator, written and proved correct in Coq, then checks that this tentative map is indeed a post-fixpoint of the dataflow inequations corresponding to our transfer function.

Concerning analysis precision, a first improvement would be to perform *strong updates* when the location stored into is uniquely known at analysis time, e.g. when the set of accessed abstract locations is a singleton of the form $\{(\texttt{Stack}, i)\}$ or $\{(\texttt{Globals}(\texttt{Just}\ id), i)\}$. In this case, the contents of the memory map for this location can be replaced by the points-to set of the right-hand side of the `store`, instead of being merged with this points-to set.

Another direction is to analyze the offset parts of pointer values more precisely. The flat lattice of integers that we currently use to track offset values could be replaced by integer intervals. More generally, the analysis could be parameterized over an arbitrary abstract domain of integers.

The next major improvement in precision would be to make the analysis interprocedural. Conceptually, the modifications to the abstract interpretation framework are minimal, namely introducing `Stack`(f) and `Allocs`(f, p) abstract blocks that are indexed by the name of the function f where the corresponding stack allocation or dynamic allocation occurs. However, the fixpoint iteration strategy must be changed: in particular, for a call to a function pointer, the points-to set of the function pointer is used to determine the possible successors of the call. In addition, issues of algorithmic efficiency and sparse data structures become much more acute in the interprocedural case.

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (1994)
2. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press (1998)
3. Bertot, Y.: Structural abstract interpretation, a formal study in Coq. In: Language Engineering and Rigorous Software Development. LNCS, vol. 5520, pp. 153–194. Springer (2009)
4. Besson, F., Cachera, D., Jensen, T.P., Pichardie, D.: Certified static analysis by abstract interpretation. In: Foundations of Security Analysis and Design. LNCS, vol. 5705, pp. 223–257. Springer (2009)
5. Besson, F., Jensen, T., Pichardie, D.: Proof-carrying code from certified abstract interpretation to fixpoint compression. Theoretical Computer Science 364(3), 273–291 (2006)
6. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Programming Language Implementation and Logic Programming. LNCS, vol. 631, pp. 269–295. Springer (1992)
7. Dabrowski, F., Pichardie, D.: A certified data race analysis for a Java-like language. In: 22nd International Conference on Theorem Proving in Higher Order Logics. LNCS, vol. 5674, pp. 212–227. Springer (2009)
8. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Program Analysis For Software Tools and Engineering (PASTE'01). pp. 54–61. ACM (2001)
9. Kildall, G.A.: A unified approach to global program optimization. In: 1st symposium Principles of Programming Languages. pp. 194–206. ACM (1973)
10. Larus, J.R., Hilfinger, P.N.: Detecting conflicts between structure accesses. In: Programming Language Design and Implementation (PLDI'88). pp. 21–34. ACM (1988)
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
12. Leroy, X.: A formally verified compiler back-end. J. Automated Reasoning 43(4), 363–446 (2009)
13. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model, version 2. Research report RR-7987, INRIA (Jun 2012)
14. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. J. Automated Reasoning 41(1) (2008)
15. Nipkow, T.: Abstract interpretation of annotated commands. In: Interactive Theorem Proving. LNCS, vol. 7406, pp. 116–132. Springer (2012)
16. Steensgaard, B.: Points-to analysis in almost linear time. In: 23rd symp. Principles of Programming Languages (POPL'96). pp. 32–41. ACM (1996)