Pointer Analysis, Conditional Soundness, and Proving the Absence of Errors

Christopher L. Conway[1], Dennis Dams[2], Kedar S. Namjoshi[2], and Clark Barrett[1]

[1] New York University, Dept. of Computer Science
{cconway,barrett}@cs.nyu.edu
[2] Bell Laboratories, Alcatel-Lucent
{dennis,kedar}@research.bell-labs.com

Abstract. It is well known that the use of points-to information can substantially improve the accuracy of a static program analysis. Commonly used algorithms for computing points-to information are known to be sound only for memory-safe programs. Thus, it appears problematic to utilize points-to information to verify the memory safety property without giving up soundness. We show that a sound combination is possible, even if the points-to information is computed separately and only conditionally sound. This result is based on a refined statement of the soundness conditions of points-to analyses and a general mechanism for composing conditionally sound analyses.

1 Introduction

It is well known that information about pointer relationships is essential for effective analysis and optimization of C programs [2,18]. Such information can be provided by a variety of algorithms that compute an approximation of the *points-to* relations of a program (e.g., [3,12,27]). For variables `x` and `y`, `x` *points to* `y` if there is some execution of the program such that the value of `x` is either the address of `y` or, if `x` and `y` are aggregate objects (such as arrays or structures), the value of an element of `x` is an address within the extent of `y`.

A (may) points-to analysis is sound if the relation it computes over-approximates the true points-to relation of the program. Typical analysis algorithms are known to be sound only for "well-behaved" programs, i.e., programs with behavior that is well-defined by the C standard [22]. For instance, typical points-to analysis algorithms consider the points-to sets of pointer values `x` and `x+1` to be the same. This is justified if `x+1` does not "overflow" the bounds of the object pointed to by `x`. However, if the expression does overflow (i.e., the program is not "well-behaved"), the object pointed to by `x+1` is undefined.

In extending the Orion static analyzer [11] to verify memory safety, we found that performing the analysis without access to points-to information resulted in an overwhelming number of false alarms. In principle, a single, combined analysis can be defined that computes memory safety and points-to information simultaneously. Since the memory safety information being computed in one

M. Alpuente and G. Vidal (Eds.): SAS 2008, LNCS 5079, pp. 62–77, 2008.
© Springer-Verlag Berlin Heidelberg 2008

"half" is available to the points-to "half," the points-to information is kept sound even for ill-behaved executions. Conversely, the memory safety component has access to up-to-date points-to information, enabling a higher precision analysis.

However, such a fine-grained combination of analyses may not be scalable. We would like to treat existing scalable points-to analyses as plug-in components. Moreover, one may wish to perform the memory safety analysis in a "bottom-up" fashion, computing a general summary for each function on any possible input—in this case, separately computed points-to information helps limit the possible values of pointer parameters and global variables.

These considerations lead to the central questions addressed in this paper: Is it possible to obtain a sound combination of independent points-to and memory safety analyses, especially as the first obtains sound results assuming memory safety? More generally, under what conditions can conditionally sound analyses be combined? What guarantees can be made for the combination?

This paper makes several contributions:

– We formalize the notion of conditional soundness, show how to compose conditionally sound analyses, and derive the conditional soundness guarantee of the composition. Although we describe conditional soundness in the specific context of points-to analysis and a particular kind of memory safety, we believe our framework can be used to refine the soundness results of a variety of static analyses, e.g., analyses that are sound assuming sequential consistency or numerical analyses that are sound assuming the absence of integer overflows. Conditional soundness can be formulated in terms of the (unconditional) preservation of a class of temporal safety properties using Cousot and Cousot's power construction [7,8,9]. Our formulation, while more specialized, is simpler (e.g., it is state-based rather than path-based) and captures the behavior of several interesting analyses more directly.
– We show that a set of points-to analyses similar to and sharing the soundness properties of commonly-used flow-sensitive and insensitive analyses—such as those of Emami, Ghiya, and Hendren [16]; Wilson and Lam [28]; Andersen [3]; Steensgaard [27]; and Das [12]—provide results that are sound for any *memory-safe execution* of a program. This statement is both stronger and more precise than the traditional statement that such analyses are sound for "well-behaved" programs.
– This more precise characterization of a points-to analysis, along with the combination theorem for conditional analyses, shows that the combination of an independent points-to analysis with a memory safety analysis is conditionally sound. The soundness result guarantees that the *absence* of errors can be proved. Conversely, for a program with memory errors, at least one representative error—but not necessarily all errors—along any unsafe execution will be detected.

Motivating Example. Figure 1(a) is an example of a program which is *not* well-behaved: there is an off-by-one error at label `L1` and an off-by-one-thousand error at label `L3` when `c` is not `0`. Assume that the functions `ok` and `bad` are analyzed in a bottom-up fashion, without reference to the actual parameters

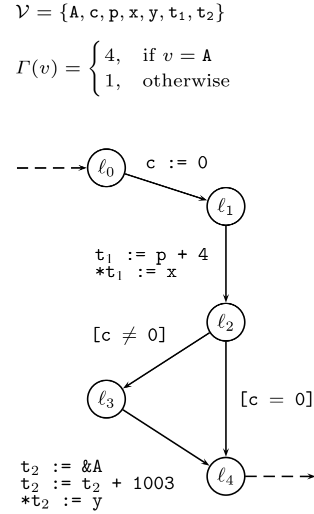
```
int A[4], c;

void bad(int *p, int x, int y) {
L0: c = 0;
L1: p[4] = x;
L2: if( c!=0 ) {
L3:     A[1003] = y;
L4: }
}

void ok(int *q, int n) {
L5: q[0] = n;
}

void main() {
    ok(A,0);
    bad(A,1,0);
}
```

(a)



(b)

Fig. 1. An unsafe C program

supplied in `main`. Without any points-to information regarding `q`, the only safe assumption is that the expression `q[0]` at label `L5` can alias any location in memory—a conservative memory safety analysis would be forced to assume that the behavior of the program is undefined from this point on. But this is not the case: the function `ok` is memory-safe so long as `q` points to an array of at least one element. Points-to information is necessary to obtain a precise memory safety analysis.

A typical points-to analysis (e.g., Andersen's [3]) will determine that `p` and `q` both point to `A` and not to `c`, `n`, `x`, or `y`. Using points-to information, a memory safety analysis can (correctly) infer that `q[0]` is an in-bounds location at `L5` and, thus, the execution of `ok` is well-defined. Further, it can (correctly) detect that `p[4]` is out-of-bounds at `L1` and emit a useful error report.

In many implementations `p[4]` will alias `c` at `L1`—so that `c` is set to `1`, making `L3` reachable—but `p` will not point to `c` according to the points-to relation. Since `c` is initialized to `0` in `bad` and—according to the points-to relation—no expression aliasing `c` is subsequently assigned to, the analysis is likely to (incorrectly) infer that the error at `L3` is unreachable. Thus, it may seem that relying on the points-to relation will lead a static analyzer to miss real errors. Note, though, that the reachability of the error at `L3` is due solely to the unsafe assignment at `L1`: the points-to relation can be relied upon up to the first occurrence of a memory safety error.

This line of reasoning is not specific to the example: as we will show, it applies to any conditionally sound analysis, enabling such an analysis to detect at least one error along any erroneous execution.

Note: Full proofs of all theorems in this paper are given in a technical report [5].

2 Program Analysis and Conditional Soundness

To present program analysis in a formal setting, we use the framework of abstract interpretation [6]. A full syntax of program statements is given in the next section. For now, we are concerned only with the relationship between concrete and abstract interpretations. We omit any discussion of techniques (such as widening and extrapolation) which serve to make program analyses finite and computable—we are concerned solely with issues of soundness.

Let C be a distinguished set of *concrete states*. A *domain* (D, γ) is a pair, where D is a set of *abstract states* and $\gamma : D \to 2^C$ is a *concretization function*. When the meaning is clear, we overload D to refer both to a domain and to its underlying set of states and use γ_D to refer to the concretization function. We lift γ_D to sets of states: $\gamma_D(D') = \bigcup_{d \in D'} \gamma_D(d)$, where $D' \subseteq D$. We say a set $D' \subseteq D$ *over-approximates* $C' \subseteq C$ iff $\gamma_D(D') \supseteq C'$.

We define the soundness of a program interpretation in terms of a *collecting semantics*. Given a (concrete or abstract) domain D, we will define a *semantic operator* $[\![\cdot]\!]$ which maps a program $\mathcal{P}$ to a set $[\![\mathcal{P}]\!] \subseteq D$ of *reachable states*. The semantics $[\![\mathcal{P}]\!]$ is defined inductively in terms of *semantic interpretations over* D: a set $\mathcal{I}[\mathcal{P}] \subseteq D$ of *initial states* and a *transfer function* $\mathcal{F}[\mathcal{P}] : D \to 2^D$. We lift $\mathcal{F}[\mathcal{P}]$ to sets of states: $\mathcal{F}[\mathcal{P}](D') = \bigcup_{d \in D'} \mathcal{F}[\mathcal{P}](d)$, where $D' \subseteq D$.

An *analysis* $\mathcal{A}$ is represented as a tuple $(D, \mathcal{I}, \mathcal{F})$, where D is a domain and $\mathcal{I}$ and $\mathcal{F}$ are semantic interpretations over D. We use $D_\mathcal{A}$, $\mathcal{I}_\mathcal{A}$, and $\mathcal{F}_\mathcal{A}$ to denote the constituents of an analysis $\mathcal{A}$ and $\gamma_\mathcal{A}$ to denote the concretization function of the domain $D_\mathcal{A}$.

Definition 1. *Let* $\mathcal{A} = (D, \mathcal{I}, \mathcal{F})$ *be an analysis. The* k-reachability predicate $R^k_\mathcal{A}[\mathcal{P}]$ for a program $\mathcal{P}$ w.r.t. $\mathcal{A}$ *holds if a state is reachable in* $\mathcal{A}$ *in exactly* k *steps. We define* $R^k_\mathcal{A}[\mathcal{P}]$ *inductively as a subset of* D*:*

$$R^0_\mathcal{A}[\mathcal{P}] = \mathcal{I}[\mathcal{P}] \qquad\qquad R^k_\mathcal{A}[\mathcal{P}] = \mathcal{F}[\mathcal{P}](R^{k-1}_\mathcal{A}[\mathcal{P}]), \quad k > 0$$

The semantics $[\![\cdot]\!]_\mathcal{A}$ w.r.t. $\mathcal{A}$ *maps a program* $\mathcal{P}$ *to a subset of* D*, the* reachable states in $\mathcal{P}$ w.r.t. $\mathcal{A}$*:*

$$[\![\mathcal{P}]\!]_\mathcal{A} = \bigcup_{k \geq 0} R^k_\mathcal{A}[\mathcal{P}]$$

To judge the soundness of an analysis, we need a concrete semantics against which it can be compared. The *concrete domain* $D_\mathcal{C}$ is given by the pair (C, γ_C), where γ_C is the trivial concretization function: $\gamma_C(c) = \{c\}$. We assume that a *concrete analysis* $\mathcal{C} = (D_\mathcal{C}, \mathcal{I}_\mathcal{C}, \mathcal{F}_\mathcal{C})$ is given. The concrete analysis uniquely defines a *concrete semantics* $[\![\cdot]\!]_\mathcal{C}$.

Definition 2. *An analysis* $\mathcal{A}$ *is* sound *iff for every program* $\mathcal{P}$*,* $[\![\mathcal{P}]\!]_\mathcal{A}$ *over-approximates* $[\![\mathcal{P}]\!]_\mathcal{C}$ *(i.e.,* $\gamma_\mathcal{A}([\![\mathcal{P}]\!]_\mathcal{A}) \supseteq [\![\mathcal{P}]\!]_\mathcal{C}$*).*

Conditional Soundness. So far, we have defined a style of analysis which is *unconditionally sound*, mirroring the traditional approach to abstract interpretation. However, as we have noted, points-to analysis is sound only under certain

assumptions about the behavior of the program analyzed. To address this, we introduce the notion of *conditional soundness with respect to a predicate* θ. An analysis will be θ*-sound* if it over-approximates the concrete states of a program that are reachable via *only* θ*-states*. We first define a semantics restricted to θ.

Definition 3. *Let $\mathcal{A} = (D, \mathcal{I}, \mathcal{F})$ be an analysis and θ a predicate on D (we view the predicate θ, equivalently, as a subset of D). The* θ-restricted k-reachability predicate $R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$ for program $\mathcal{P}$ w.r.t. $\mathcal{A}$ *holds if a state is reachable in $\mathcal{A}$ in exactly k steps via* only θ states. *$R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$ is defined inductively:*

$$R^0_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}] = \mathcal{I}[\mathcal{P}] \qquad R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}] = \mathcal{F}[\mathcal{P}](\theta \cap R^{k-1}_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]), \quad k > 0$$

The θ-restricted semantics $[\![\cdot]\!]_{\mathcal{A}}{\downarrow_\theta}$ w.r.t. $\mathcal{A}$ *maps a program $\mathcal{P}$ to a subset of D, the* θ-reachable states in $\mathcal{P}$ w.r.t. $\mathcal{A}$*:*

$$[\![\mathcal{P}]\!]_{\mathcal{A}}{\downarrow_\theta} = \bigcup_{k \geq 0} R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$$

Note that $R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$ may include non-θ states—neither $\mathcal{I}[\mathcal{P}]$ nor the range of $\mathcal{F}[\mathcal{P}]$ are restricted to θ—but those states will not yield successors in $R^{k+1}_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$. The θ-restricted semantics give us a lower bound for the approximation of a θ-sound analysis.

Definition 4. *Let $\mathcal{A}$ be an analysis and θ a predicate on C. $\mathcal{A}$ is* θ-sound *iff for every program $\mathcal{P}$, $[\![\mathcal{P}]\!]_{\mathcal{A}}$ over-approximates $[\![\mathcal{P}]\!]_{\mathcal{C}}{\downarrow_\theta}$.*

Note that an unconditionally sound analysis is also θ-sound for *any* θ. More generally, any θ-sound analysis is also φ-sound, for any φ stronger than θ.

This notion of conditional soundness does not just give us a more precise statement of the behavior of certain analyses—it provides us with a sufficient condition to show an analysis *proves the absence of error states*.

Theorem 1. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ a θ-sound analysis. If there are no reachable non-θ states in $\mathcal{P}$ w.r.t. $\mathcal{A}$, then there are no reachable concrete non-θ states in $\mathcal{P}$ (i.e., if $\gamma_A([\![\mathcal{P}]\!]_{\mathcal{A}}) \subseteq \theta$, then $[\![\mathcal{P}]\!]_{\mathcal{C}} \subseteq \theta$)).*

In Section 4, we will show that points-to analysis is SafeDeref-sound, where SafeDeref is a predicate that captures memory safety.

Parameterized Analysis. Having defined a precise notion of conditional soundness, we now consider how the results of a θ-sound analysis can be used to refine a second analysis. Suppose that $\mathcal{A}$ is an analysis and we have already computed the set of reachable states $[\![\mathcal{P}]\!]_{\mathcal{A}}$. We may wish to use the information present in $[\![\mathcal{P}]\!]_{\mathcal{A}}$ to refine a second analysis over a different domain B. For example, we could use the reduced product construction [7] to form a new domain over a subset of $D_{\mathcal{A}} \times B$ including only those states (a, b) where a is in $[\![\mathcal{P}]\!]_{\mathcal{A}}$ and the states a and b "agree" (e.g., $\gamma_{\mathcal{A}}(a) \cap \gamma_B(d) \neq \emptyset$).

Traditional methods for combining analyses take a "white box" approach—e.g., Cousot and Cousot [7] assume that the state transformers are available and can be combined in a mechanical way; Lerner et al. [24] assume that analyses can be run in parallel, one step at a time. In contrast, we will assume that any prior analysis is a black box: we have access to its result (in the form of a set of reachable abstract states), its domain (which allows us to interpret the result), and some (possibly conditional) soundness guarantee. This naturally models the use of off-the-shelf program analyses to provide refinement advice.

We will define such a refinement in terms of a parameterized analysis which produces a new, refined analysis from the results of a prior analysis. An *analysis generator* $\widetilde{\mathcal{G}}$ is a tuple $(D, E, \widetilde{\mathcal{I}}, \widetilde{\mathcal{F}})$ where: D and E are domains (the *input* and *output* domains, respectively) and $\widetilde{\mathcal{I}}$ and $\widetilde{\mathcal{F}}$ are *parameterized interpretations* mapping a set of states $D' \subseteq D$ to semantic interpretations $\widetilde{\mathcal{I}}\langle D'\rangle$ and $\widetilde{\mathcal{F}}\langle D'\rangle$ over E. We denote by $\widetilde{\mathcal{G}}\langle D'\rangle$ the analysis over E defined by the parameterized interpretations on input D': $\widetilde{\mathcal{G}}\langle D'\rangle = (E, \widetilde{\mathcal{I}}\langle D'\rangle, \widetilde{\mathcal{F}}\langle D'\rangle)$. As one might expect, the soundness of $\widetilde{\mathcal{G}}\langle D'\rangle$ depends on the input D'.

Definition 5. *An analysis generator $\widetilde{\mathcal{G}}$ with input domain D is* sound *iff for every set of states $D' \subseteq D$, $\widetilde{\mathcal{G}}\langle D'\rangle$ is θ-sound with $\theta = \gamma_D(D')$.*

Given an analysis generator, it is natural to consider the analysis formed by composing the generator with an analysis over its input domain. If $\mathcal{A}$ is an analysis and $\widetilde{\mathcal{G}}$ is an analysis generator with input domain $D_{\mathcal{A}}$ (i.e., the input domain of $\widetilde{\mathcal{G}}$ is the underlying domain of $\mathcal{A}$), the *composed analysis* $\widetilde{\mathcal{G}} \circ \mathcal{A}$ is defined by providing the result of $\mathcal{A}$ as a parameter to $\widetilde{\mathcal{G}}$ (i.e., $\widetilde{\mathcal{G}} \circ \mathcal{A} = \widetilde{\mathcal{G}}\langle [\![\mathcal{P}]\!]_{\mathcal{A}}\rangle$). An important property of the composed analysis is preservation of soundness.

Theorem 2. *If $\widetilde{\mathcal{G}}$ is sound and $\mathcal{A}$ is θ-sound, then the composed analysis $\widetilde{\mathcal{G}} \circ \mathcal{A}$ is θ-sound.*

In the remainder of this paper, we will apply Theorem 2 to the problem of verifying memory safety using points-to information. In Section 3, we define the concrete semantics for a little language that captures the pointer semantics of C. In Section 4, we define a memory safety property SafeDeref and a set of SafeDeref-sound points-to analyses similar to the points-to analyses found in the literature. In Section 5, we define a memory safety analysis parameterized by points-to information. These, together with Theorem 2, allow us to prove the absence of memory safety errors.

Note that since the points-to analysis is SafeDeref-sound, we could in theory use it to prove the absence of memory safety errors. However, the points-to domain does not track memory safety information with adequate precision to detect errors without an impractical number of false alarms. The value of Theorem 2 is that it allows for the definition of a specialized memory safety analysis which uses the points-to analysis to increase its precision while remaining SafeDeref-sound.

$$n \in \mathbb{Z} \qquad \mathtt{x}, \mathtt{y} \in \mathit{Vars}$$

$$\begin{aligned} L \in \mathit{Lvals} &::= \mathtt{x} \mid \mathtt{*x} \\ E \in \mathit{Exprs} &::= L \mid n \mid \mathtt{x} \oplus \mathtt{y} \mid \mathtt{x} \trianglelefteq \mathtt{y} \mid \mathtt{\&x} \\ S \in \mathit{Stmts} &::= L \mathtt{\ :=\ } E \mid \mathtt{[}E\mathtt{]} \end{aligned}$$

Fig. 2. Grammar for a minimal C-like language

3 Concrete Semantics

To make precise statements about program analyses requires a concrete program semantics. We will define the semantics of the little language presented in Fig. 2. The language eliminates all but those features of C that are essential to the question at hand. The semantics of the language is chosen to model the requirements of ANSI/ISO C [22] without making implementation-specific assumptions. Undefined or implementation-defined behaviors are modeled with explicit nondeterminism. Note that an ANSI/ISO-compliant C compiler is free to implement undefined behaviors in a specific, deterministic manner. By modeling undefined behaviors using non-determinism, the soundness statements made about each analysis apply to *any* standard-compliant compilation strategy.

The most important features of C that we exclude here are fixed-size integer types, narrowing casts, dynamic memory allocation, and functions.[1] We also ignore the "strict aliasing" rule [22, §6.5]. Each of these can be handled, at the cost of a higher degree of complexity in our definitions.

The syntactic classes of variables, lvalues, expressions, and statements, are defined in Fig. 2. We use n to represent an integer constant and `x` and `y` to represent arbitrary variables. We use $\oplus$ to represent an arbitrary binary arithmetic operator and $\trianglelefteq$ to represent a relational operator. Pointer operations include arithmetic, indirection (`*`), and address-of (`&`). Statements include assignments and tests (`[`E`]`, where E is an expression).

Variables in our language are viewed as arrays of memory cells. Each cell may hold either an unbounded integer or a pointer value. The only type information present is the allocated size of each variable—the "type system" merely maps variables to their sizes and provides no safety guarantees.

A *program* $\mathcal{P}$ is a tuple $(\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$, where: $\mathcal{V} \subseteq \mathit{Vars}$ is a finite set of *program variables*; $\Gamma : \mathcal{V} \to \mathbb{N}$ is a *typing environment* mapping variables to their allocated sizes; $\mathcal{L}$ is a finite set of *program points*; $\mathcal{S} \subseteq \mathit{Stmts}$ is a finite set of *program statements* whose variables are from $\mathcal{V}$; $\tau \subseteq \mathcal{L} \times \mathcal{S} \times \mathcal{L}$ is a *transition*

[1] The omission of dynamic allocation in the discussion of points-to analysis and memory safety may seem an over-simplification. However, it is not essential to our purpose here. Points-to analyses typically handle dynamic allocation by treating each allocation site as if it were the static declaration of a global array of unknown size.

relation; and $en \in \mathcal{L}$ is a distinguished *entry point*. In the following, we assume a fixed program $\mathcal{P} = (\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$.

Example 1. Figure 1(b) gives a fragment of the program representation for the code in Fig. 1(a), corresponding to the function `bad` . We have introduced temporaries $\mathtt{t}_1$ and $\mathtt{t}_2$ in order to simplify expression evaluation and compressed multiple statements onto a single transition when they represent a single statement in the source program.

In order to reason about points-to and memory safety analyses, we need a memory model on which to base the concrete semantics. The unit of memory allocation is a *home* in the set $\mathbb{H}$. Each home h represents a contiguous block of memory cells, e.g., a statically declared array. A *location* $h[i]$ represents the cell at integer *offset* i in home h. The set of locations with homes from $\mathbb{H}$ is denoted $\mathbb{L}$. The function $\mathbf{size} : \mathbb{H} \to \mathbb{N}$ maps a home to its allocated size. When $0 \le i < \mathbf{size}(h)$, location $h[i]$ is *in bounds*; otherwise it is *out of bounds*. Memory locations contain values from the set $\mathit{Vals} = \mathbb{Z} \cup \mathbb{L}$. A *memory state* is a partial function $m : \mathbb{L} \to \mathit{Vals}$. The set of all memory states is denoted $\mathbb{M}$. The set of concrete states C is the set of pairs (p, m) where $p \in \mathcal{L}$ represents the program position and m is a memory state.

An *allocation for* $\mathcal{V}$ is an injective function $\mathbf{home} : \mathcal{V} \to \mathbb{H}$ such that $\mathbf{size}(\mathbf{home}(\mathtt{x})) = \Gamma(\mathtt{x})$ for all $\mathtt{x} \in \mathcal{V}$. Given such an allocation, the *lvalue* of $\mathtt{x} \in \mathcal{V}$ is $\mathbf{lval}(\mathtt{x}) = \mathbf{home}(\mathtt{x})[0]$. We write $m(\mathtt{x})$ for $m(\mathbf{lval}(\mathtt{x}))$ and $m[\mathtt{x} \mapsto v]$ for $m[\mathbf{lval}(\mathtt{x}) \mapsto v]$, where m is a memory state. We say a location $h[i]$ is *within* a variable $\mathtt{x}$ if $h = \mathbf{home}(\mathtt{x})$ and $h[i]$ is in bounds.

Figure 3 defines the concrete interpretations $\mathcal{E}$ and **post** of, respectively, expressions and statements. We present here only the most interesting cases. Complete definitions are given in a technical report [5]. Note that both $\mathcal{E}$ and **post** result in *sets* of, respectively, values and concrete states—the set-based semantics is needed as undefined operations may have a nondeterministic result. $\mathcal{E}$ returns the distinguished value $\perp$ in the case where an expression is not just ill-defined, but erroneous (e.g., reading an out-of-bounds memory location)—in this case the next state can have *any* memory state at *any* program point.

We now define the concrete interpretation of a program.

Definition 6. *The* concrete semantics $[\![\mathcal{P}]\!]_{\mathcal{C}}$ *of a program* $\mathcal{P} = (\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$ *is defined by the analysis* $\mathcal{C} = (D_{\mathcal{C}}, \mathcal{I}_{\mathcal{C}}, \mathcal{F}_{\mathcal{C}})$*, where*

$$\mathcal{I}_{\mathcal{C}}[\mathcal{P}] = \{(en, m) \mid \forall l \in \mathbb{L}.\ m(l) \text{ is not a location}\}$$
$$\mathcal{F}_{\mathcal{C}}[\mathcal{P}](p, m) = \bigcup_{(p,S,p') \in \tau} \mathbf{post}(m, p', S)$$

Example 2. Figure 4(a) gives a subset of the reachable concrete states of the program in Fig. 1(b). At ℓ_0, `p` is $\mathtt{A}[0]$ (the base address of the array `A`), `x` is 1, and `y` is 0. At ℓ_1, due to the assignment to out-of-bounds location $\mathtt{A}[4]$, the next state is undefined: *every* program point is reachable with *any* memory state.

$$\mathcal{E}(m, \mathtt{x}) = \begin{cases} \mathbb{Z}, & \text{if } m(\mathtt{x}) \text{ is undefined} \\ \{m(\mathtt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}(m, *\mathtt{x}) = \begin{cases} \bot, & \text{if } m(\mathtt{x}) \text{ is undefined, not a location, or out of bounds} \\ \mathbb{Z} & \text{if } m(m(\mathtt{x})) \text{ is undefined} \\ \{m(m(\mathtt{x}))\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}(m, p, \mathtt{x} \texttt{ := } E) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } \mathcal{E}(m, E) = \bot \\ \{(p, m[\mathtt{x} \mapsto v]) \mid v \in \mathcal{E}(m, E)\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}(m, p, *\mathtt{x} \texttt{ := } E) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } m(\mathtt{x}) \text{ is undefined, not a location, or out of bounds;} \\ & \text{or if } \mathcal{E}(m, E) = \bot \\ \{(p, m[m(\mathtt{x}) \mapsto v]) \mid v \in \mathcal{E}(m, E)\}, & \text{otherwise.} \end{cases}$$

Fig. 3. The concrete interpretation

4 Pointer Analysis

The goal of pointer analysis is to compute an over-approximate points-to set for each variable in the program, i.e., the set of homes "into" which a variable may point in some reachable state.

A *points-to state* is a relation between variables. We denote the set of points-to states by *Pts*. When it is convenient, we treat a points-to state also as a relation between variables and memory locations: for points-to state *pts*, variables $\mathtt{x}, \mathtt{y}$, and location $h[i]$, we say $(\mathtt{x}, h[i])$ is in *pts* when $(\mathtt{x}, \mathtt{y})$ is in *pts* and $h[i]$ is within $\mathtt{y}$ (i.e., $h[i]$ is in bounds and $h = \mathbf{home}(\mathtt{y})$). We write $pts(\mathtt{x})$ for the *points-to set* of the variable x in *pts*, i.e., the set of variables y (alt. locations l) such that $(\mathtt{x}, \mathtt{y})$ (alt. $(\mathtt{x}, l)$) is in *pts*.

The concretization function γ_{Pts} takes a points-to state to the set of concrete states where at *most* its points-to relationships hold. Say that variable x *points to* y in memory state m if there exist locations l_1, l_2 such that l_1 is within x, l_2 is within y, and $m(l_1) = l_2$. Then m is in $\gamma_{Pts}(pts)$ iff for all $\mathtt{x}, \mathtt{y}$ such that x points to y in m, the pair $(\mathtt{x}, \mathtt{y})$ is in *pts*. Note that there may be other pairs in *pts* as well—the points-to relation is over-approximate. Note also that *only in-bounds location values must agree with the points-to state*; out-of-bounds locations are unconstrained.

Figure 5 defines the interpretations $\mathcal{E}_{Pts}$ and $\mathbf{post}_{Pts}$ for a selection of, respectively, expressions and statements in the points-to domain. (Complete definitions are given in a technical report [5].) The interpretations are chosen to match those used by common points-to analyses. A key feature is the treatment of the indirection operator *, which assumes that its argument is within bounds. Without this assumption, the interpretation would have to use the "top" points-to state (i.e., all pairs of variables) for the result of any indirect assignment.

We lift *Pts* to the set $\mathcal{L} \times Pts$ in the natural way.

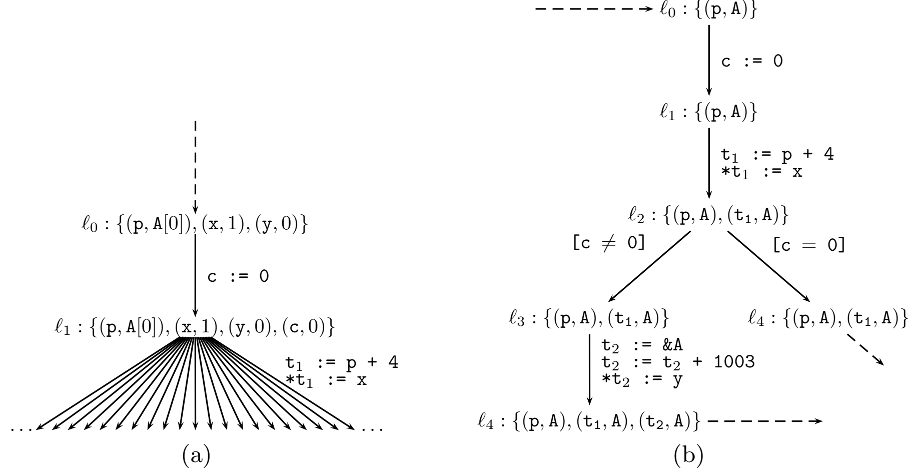


Fig. 4. Concrete and points-to semantics for the program in Fig. 1(b)

$$\begin{aligned}
\mathcal{E}_{Pts}(pts, \mathtt{x}) &= pts(\mathtt{x})\\
\mathcal{E}_{Pts}(pts, \mathtt{*x}) &= \{\mathtt{z} \in \mathcal{V} \mid \exists \mathtt{y} \in \mathcal{V} : pts(\mathtt{x}, \mathtt{y}) \wedge pts(\mathtt{y}, \mathtt{z})\}\\
\mathbf{post}_{Pts}(pts, \mathtt{x} \;\mathtt{:=}\; E) &= pts \cup \{(\mathtt{x}, \mathtt{y}) \mid \mathtt{y} \in \mathcal{E}_{Pts}(pts, E)\}\\
\mathbf{post}_{Pts}(pts, \mathtt{*x} \;\mathtt{:=}\; E) &= \bigcup_{(\mathtt{x},\mathtt{y}) \in pts} \mathbf{post}_{Pts}(pts, \mathtt{y} \;\mathtt{:=}\; E)
\end{aligned}$$

Fig. 5. Abstract interpretation over points-to states

Definition 7. *A* flow- and path-sensitive points-to analysis **Pts** *is given by the tuple* $(Pts, \mathcal{I}_{Pts}, \mathcal{F}_{Pts})$*, where*

$$\begin{aligned}
\mathcal{I}_{Pts}[\mathcal{P}] &= \{(en, \emptyset)\}\\
\mathcal{F}_{Pts}[\mathcal{P}](p, pts) &= \bigcup_{(p,S,p') \in \tau} (p', \mathbf{post}_{Pts}(pts, S))
\end{aligned}$$

Example 3. Figure 4(b) shows a subset of the reachable points-to states for the program in Fig. 1(b). At ℓ_0, p points to A. The transition from ℓ_1 to ℓ_2 causes $\mathtt{t}_1$ to point to A as well. The presence of an out-of-bounds array access has no effect on the points-to state: the analysis assumes that evaluating $\mathtt{*t}_1$ is safe.

Definition 8. *Let* SafeDeref *be the predicate that holds in a concrete state* (p, m) *if, for every transition* (p, S, p') *in* τ *where* S *includes an expression of the form* $\mathtt{*x}$*,* $m(\mathtt{x})$ *is an in-bounds location.*

Theorem 3. *The points-to analysis* **Pts** *is* SAFEDEREF*-sound.*

We can extract more traditional flow-sensitive, global, and flow-insensitive pointer analyses from $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ as follows.

- A *flow-sensitive, program-point-sensitive (path-insensitive) analysis* is derived by assigning to each program point p the least points-to state (by subset inclusion) $pts^\sharp$ such that, if (p, pts) is in $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$, then $pts \subseteq pts^\sharp$.
- A *flow-sensitive, global (program-point-insensitive) analysis* is derived by assigning to every program point the least points-to state (by subset inclusion) $pts^\sharp$ such that, if (p, pts) is in $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ for *any* program point p, then $pts \subseteq pts^\sharp$.
- A *flow-insensitive analysis* is derived by replacing τ in Definition 7 with the relation $\tau^\sharp$, where the edge (p, S, q) is in $\tau^\sharp$ whenever some edge (t, S, u) is in τ, for *any* program points t and u. Intuitively, if a statement occurs anywhere in the program, then it may occur between any two program points—the interpretation ignores the control-flow structure of the program.
- *Flow-insensitive, program-point-sensitive* and *flow-insensitive, global* combinations can be defined as above, substituting the flow-insensitive semantics for $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$.

Theorem 4. *Each of the flow-, path-, and program-point-sensitive and insensitive variations of the points-to analysis is* SAFEDEREF*-sound.*

Note 1. The flow-sensitive, program-point-sensitive analysis yields a points-to relation similar to that of Emami et al. [16]. The flow-insensitive, global analysis procedure yields a points-to relation similar to that of Andersen [3]. The Steensgaard [27] and Das [12] relations add additional approximation to the global relation. We claim (but do not prove formally here) that these procedures approximate $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ and, thus, are *at least* SAFEDEREF-sound.

By the definition of conditional soundness, it is possible some condition θ weaker than SAFEDEREF exists such that some or all of the above analyses are θ-sound. It is our belief that this is not the case: no realistic points-to analysis is θ-sound for any θ weaker than SAFEDEREF. A proof of this proposition is beyond the scope of this paper.

In summary, we have shown that a set of points-to analyses which share the assumptions of widely used analyses from the literature are sound for all memory-safe executions. This claim is both stronger and more precise than any correctness claims the authors have encountered: our points-to analyses (and, by extension, those cited above) compute a relation which is conservative not only for "well-behaved" (i.e., memory-safe) programs, but for all well-behaved *executions*, even the well-behaved executions of ill-behaved programs

We have shown that, if we can prove the absence of non-SAFEDEREF states in $[\![\mathcal{P}]\!]_{\mathcal{C}}$, the points-to analyses we have defined above will be sound. It remains to describe an analysis parameterized by points-to information which can perform a precise memory safety analysis.

5 Checking Memory Safety

We wish to define an analysis procedure that will soundly prove the absence of non-SafeDeref states in the concrete program. Note that the only attributes of a location value that are relevant to the property SafeDeref are its offset and the size of its home; if we can precisely track these attributes, we can ignore the home component of a location (i.e., which variable it is within) so long as we have access to over-approximate points-to information.

Note 2. In our description of the analysis, we will omit the merging, widening, and covering operations necessary to make the reachability computation tractable. In our implementation of a memory safety analysis in Orion, we constrain integer values and pointer offsets using a relational abstract domain (e.g., convex polyhedra [10]) and use merging and widening to efficiently over-approximate the semantics given below.

Our analysis will track *abstract values* from the set $\widehat{Vals}$. An abstract value is either an integer or an *abstract location*, a pair (i, n) representing a location at offset i in a home of size n. Each abstract value $\hat{v}$ represents a set of concrete values, according to the abstraction function $\alpha : Vals \to \widehat{Vals}$. For integer values, α is the identity (i.e., $\alpha(n) = n$); for concrete location values, α preserves the offset and size (i.e., $\alpha(h[i]) = (i, \mathbf{size}(h))$). An abstract location (i, n) is *in bounds* if it represents only in bounds concrete locations (i.e., $0 \leq i < n$); otherwise it is *out of bounds*. An *abstract memory state* is a partial function $b : \mathbb{L} \to \widehat{Vals}$. We denote by B the set of abstract memory states.

The concretization function $\gamma_B : B \to 2^C$ takes an abstract memory state b to the set of concrete memories abstracted by b. A concrete memory m is in $\gamma_B(b)$ iff for all l either $m(l)$ and $b(l)$ are both undefined or $\alpha(m(l)) = b(l)$.

Figure 6 defines the interpretations $\mathcal{E}_B$ and $\mathbf{post}_B$ for a selection of, respectively, expressions and statements with respect to B. (Complete definitions are given in a technical report [5].) Note that the interpretations rely on points-to information. In the limiting case, where no points-to information is available (i.e., the points-to relation includes all pairs), the expression `*x` can take the value of any location abstracted by $b(\mathtt{x})$. As in the concrete interpretation $\mathcal{E}_B$ returns the value $\perp$ in the case where expression evaluation is (potentially) erroneous.

We lift B to the domain $\mathcal{L} \times B$ in the natural way.

Definition 9. *The analysis generator $\widetilde{\mathcal{B}}$ maps a set of points-to states Q to the memory safety analysis $\mathcal{B}\langle Q\rangle$ defined by the parameterized interpretations*

$$\widetilde{\mathcal{I}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}] = \{(en, b) \mid \forall l \in \mathbb{L} : b(l) \text{ is undefined}\}$$
$$\widetilde{\mathcal{F}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}](p, b) = \bigcup_{(p,S,p')\in\tau} \; \bigcup_{(p,pts)\in Q} \mathbf{post}_B(b, pts, p', S)$$

Theorem 5. *The analysis generator $\widetilde{\mathcal{B}}$ is sound.*

$$\mathcal{E}_B(b, pts, \mathtt{x}) = \begin{cases} \mathbb{Z}, & \text{if } b(\mathtt{x}) \text{ is undefined} \\ \{b(\mathtt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}_B(b, pts, *\mathtt{x}) = \begin{cases} \bot, & \text{if } b(\mathtt{x}) \text{ is undefined, not a location, or out of bounds} \\ \widehat{Vals}, & \text{if } b(l) \text{ is undefined for some } l \text{ in } pts(\mathtt{x}) \text{, where } \alpha(l) = b(\mathtt{x}) \\ \{b(l) \mid pts(\mathtt{x}, l),\ \alpha(l) = b(\mathtt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}_B(b, pts, p, \mathtt{x} \ \mathtt{:=}\ E) = \begin{cases} \mathcal{L} \times B, & \text{if } \mathcal{E}_B(b, pts, E) = \bot \\ \{(p, b[\mathtt{x} \mapsto \hat{v}]) \mid \hat{v} \in \mathcal{E}_B(b, pts, E)\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}_B(b, pts, p, *\mathtt{x} \ \mathtt{:=}\ E) = \begin{cases} \mathcal{L} \times B, & \text{if } b(\mathtt{x}) \text{ is undefined, not a location, or out of bounds; or if } \mathcal{E}_B(b, pts, E) = \bot \\ \{(p, b[l \mapsto \hat{v}]) \mid pts(\mathtt{x}, l),\ \alpha(l) = b(\mathtt{x}),\ \hat{v} \in \mathcal{E}_B(b, pts, E)\}, & \text{otherwise} \end{cases}$$

Fig. 6. Abstract interpretation over B

Corollary 1. *If a points-to analysis* $\mathcal{Q}$ *is* SafeDeref*-sound, the composed memory safety analysis* $\widetilde{\mathcal{B}} \circ \mathcal{Q}$ *is* SafeDeref*-sound.*

Combining Corollary 1 with Theorems 3 and 4, we can compose $\widetilde{\mathcal{B}}$ with any of the points-to analyses described in Section 4 and the resulting analysis will be SafeDeref-sound. Recall from Theorem 1 that SafeDeref-soundness guarantees the detection of error states. If any non-SafeDeref state exists in $[\![\mathcal{P}]\!]_{\mathcal{C}}$, then a non-SafeDeref state is represented by the composed semantics; if only SafeDeref states are reachable in the composed analysis then no concrete non-SafeDeref state is reachable—the absence of error states can be proved.

6 Related Work

Methods for combining analyses have been described in the abstract interpretation community, starting with Cousot and Cousot [7]. The focus has been on exploiting mutual refinement to achieve the most precise combined analyses, as in Gulwani and Tiwari [19] and Cousot et al. [9]. The power domain of Cousot and Cousot [7, §10.2] provides a general model for analyses with conditional semantics. We believe our notion of conditional soundness provides a simpler model which captures the behavior of a variety of interesting analyses.

Pointer analysis for C programs has been an active area of research for decades [21,16,28,3,27,17,12,20,23]. The correctness arguments for points-to algorithms are typically stated informally—each of the analyses has been developed for the purpose of program transformation and understanding, not for use in a sound verification tool. Although Hind [21] proposes the use of pointer analysis in verification, the authors are not aware of any prior work that formally addresses the soundness of verification using points-to information.

Adams et al. [1] explored the use of Das' algorithm to prune the search space for a typestate checker and to generate initial predicates for a software model checker. In both cases, the use of the points-to information is essentially heuristic—the correctness of the overall approach does not depend on the points-to analysis being sound.

Dor, Rodeh, and Sagiv [15] describe a variation on traditional points-to analyses intended to improve precision for a sound, inter-procedural memory safety verifier. A proof of soundness is given in Dor's thesis [14]. However, the proof is not explicit about the obligations of the points-to analysis. We provide a more general framework for reasoning about verification using conditionally sound information.

Bruns and Chandra [4] provide a formal model for reasoning about pointer analysis based on transition systems. The focus of their work is primarily complexity and precision, rather than soundness.

Dhurjati, Kowshik, and Adve [13] define a program transformation which preserves the soundness of a flow-insensitive, equality-based points-to analysis (e.g., those of Steensgaard [27] and Lattner [23]) even for programs with memory safety errors. The use of an equality-based analysis is necessary to achieve an efficient implementation, but it limits the use of the technique in applications where a more precise analysis may be necessary, e.g., in verification. The soundness results we describe here are equally applicable to flow-sensitive, flow-insensitive, equality-based and subset-based pointer analyses.

Our abstraction for memory safety analysis is very similar to the formal models used in CCured [26] and CSSV [15]. Miné [25] describes a combined analysis for embedded control systems which incorporates points-to information. His analysis makes implementation-specific (i.e., unsound in general) assumptions about the layout of memory.

7 Conclusion

This work grew out of a simple, but puzzling question: is it possible to utilize the results of an analysis (points-to) whose soundness is dependent on a property (memory-safety) in a sound analysis for the same property? There seemed to be a circularity that could make a sound combination impossible.

Studying this question, we were led to a more precise statement of the soundness properties of points-to analysis and to the definition of conditional soundness. The final result shows that the combination is sound enough to correctly prove the absence of errors, although it may not be strong enough to point out every possible error.

We have concentrated here on points-to and memory safety analysis, but our conditional soundness framework is by no means restricted to these domains. For example, some static analyses are sound only assuming sequential consistency, that integer overflow does not occur, or that the program is free of floating point exceptions. The soundness claims of such analyses could be refined using the methods we have described in this paper.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. 0341685. Additional support was provided by NSF Grant No. 0644299.

References

1. Adams, S., Ball, T., Das, M., Lerner, S., Rajamani, S.K., Seigle, M., Weimer, W.: Speeding up dataflow analysis using flow-insensitive pointer analysis. In: Static Analysis Symposium, Madrid, Spain, pp. 230–246 (September 2002)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1988)
3. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (May 1994)
4. Bruns, G., Chandra, S.: Searching for points-to analysis. In: Foundations of Software Engineering, Charleston, South Carolina, pp. 61–70 (November 2002)
5. Conway, C.L., Dams, D., Namjoshi, K.S., Barrett, C.: Points-to analysis, conditional soundness, and proving the absence of errors. Technical Report TR2008-910, New York University, Dept. of Computer Science (2008)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, Los Angeles, California, pp. 238–252 (1977)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages, San Antonio, Texas, pp. 269–282 (1979)
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: European Symposium on Programming, Edinburgh, Scotland, pp. 21–30 (April 2005)
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: Asian Computing Science Conference (ASIAN), Tokyo, Japan (December 2006)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages, Tucson, Arizona (January 1978)
11. Dams, D., Namjoshi, K.S.: Orion: Building blocks for program analyzers. In: Formal Methods for Components and Objects, Amsterdam, The Netherlands (November 2005)
12. Das, M.: Unification-based pointer analysis with directional assignments. In: Programming Language Design and Implementation, Vancouver, British Columbia, pp. 35–46 (2000)
13. Dhurjati, D., Kowshik, S., Adve, V.: SAFECode: enforcing alias analysis for weakly typed languages. In: Programming Language Design and Implementation, Ottawa, Canada, pp. 144–157 (June 2006)
14. Dor, N.: Automatic Verification of Program Cleanness. PhD thesis, Tel Aviv University (December 2003)
15. Dor, N., Rodeh, M., Sagiv, M.: CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In: Programming Language Design and Implementation, San Diego, California, pp. 155–167 (July 2003)
16. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Programming Language Design and Implementation, pp. 242–256 (June 1994)

17. Foster, J.S., Fähndrich, M., Aiken, A.: Flow-insensitive points-to analysis with term and set constraints. Technical Report UCB/CSD-97-964, University of California, Berkeley (August 1997)
18. Ghiya, R., Lavery, D.M., Sehr, D.C.: On the importance of points-to analysis and other memory disambiguation methods for C programs. In: Programming Language Design and Implementation, Snowbird, Utah, pp. 47–58 (June 2001)
19. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Programming Language Design and Implementation, Ottawa, Canada (June 2006)
20. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: Programming Language Design and Implementation, Snowbird, Utah, pp. 24–34 (June 2001)
21. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: Program Analysis for Software Tools and Engineering, Snowbird, Utah (June 2001)
22. ISO Standard - Programming Languages - C, ISO/IEC 9899:1999 (December 1999)
23. Lattner, C.: Macroscopic Data Structure Analysis and Optimization. PhD thesis, University of Illinois at Urbana-Champaign (May 2005)
24. Lerner, S., Grove, D., Chambers, C.: Composing dataflow analyses and transformations. In: Principles of Programming Languages, Portland, Oregon, pp. 270–282 (2002)
25. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Languages, Compilers, and Tools for Embedded Systems, Ottawa, Canada (2006)
26. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. In: Principles of Programming Languages, Portland, Oregon, pp. 128–139 (January 2002)
27. Steensgaard, B.: Points-to analysis in almost linear time. In: Principles of Programming Languages, St. Petersburg Beach, Florida, pp. 32–41 (January 1996)
28. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: Programming Language Design and Implementation, San Diego, California, pp. 1–12 (June 1995)