Construction of Abstract Domains for Heterogeneous Properties (Position Paper)

Xavier Rival, Antoine Toubhans, Bor-Yuh Evan Chang

HAL Id: hal-01095977

https://hal.science/hal-01095977

Submitted on 16 Dec 2014

Construction of Abstract Domains
for Heterogeneous Properties
(Position Paper) [*]

Xavier Rival[1], Antoine Toubhans[1], and Bor-Yuh Evan Chang[2]

[1] INRIA, ENS, CNRS, Paris, France
[2] University of Colorado, Boulder, Colorado, USA
rival@di.ens.fr, toubhans@di.ens.fr, bec@cs.colorado.edu

Abstract. The aim of static analysis is to infer invariants about programs that are tight enough to establish semantic properties, like the absence of run-time errors. In the last decades, several branches of the static analysis of imperative programs have made significant progress, such as in the inference of numeric invariants or the computation of data structures properties (using pointer abstractions or shape analyzers). Although simultaneous inference of shape-numeric invariants is often needed, this case is especially challenging and less well explored. Notably, simultaneous shape-numeric inference raises complex issues in the design of the static analyzer itself. We study the modular construction of static analyzers, based on combinations of atomic abstract domains to describe several kinds of memory properties and value properties.

Static analysis to infer heterogeneous properties. Static analysis by abstract interpretation [4] utilizes an *abstraction* to over-approximate (non-computable) sets of program states, using computer-representable elements, that stand for *logical properties* of concrete program states. As an example, for numerical properties, the interval abstract domain [4] uses constraints of the form $n \leq x$ and $x \leq p$ to describe possible values of variable x, where n, p are scalars.

To construct a static analyzer capable of inferring sound approximations of program behaviors, one designs an *abstract domain*, which consists of an abstraction, and abstract operations for sound post-condition operators, join and widening:

1. An abstraction is defined by a set of abstract elements $\mathbf{A}$ and a concretization function $\gamma : \mathbf{A} \rightarrow \mathcal{P}(C)$, which maps each abstract property $\mathbf{a}$ into the set of concrete elements $\gamma(\mathbf{a})$ that satisfy it. The set $\mathbf{A}$ of abstract elements will be assumed to be defined by a grammar of admissible logical predicates (e.g., for intervals, $\mathbf{a}(\in \mathbf{A}) ::= \mathbf{a} \wedge \mathbf{a} \mid n \leq x \mid x \leq p$).
2. A post-condition operator is a function $\mathbf{f} : \mathbf{A} \rightarrow \mathbf{A}$ which over-approximates a concrete operation $f : C \rightarrow \mathcal{P}(C)$ encountered in programs (as, e.g., a test).
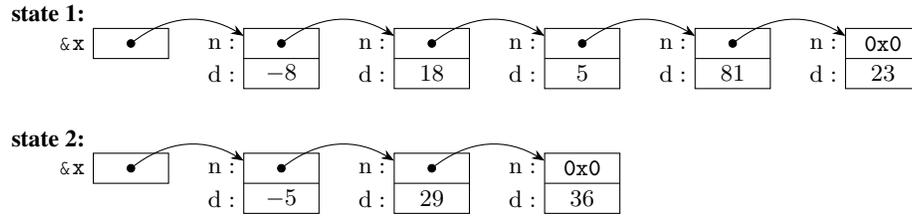
state 1:

state 2:

Fig. 1. Heterogeneous property abstraction

3. Abstract join computes an over approximation of union and widening [4] enforces the termination of abstract iterates for the analysis of loops.

The combination of post-condition operators and widening operators allows us to define a sound static analyzer [4].

In the following, we discuss the design of an abstraction able to handle heterogeneous properties, about both data-structures and values. For instance, Figure 1 shows a couple of concrete states containing lists of numbers that are all positive except for the first one, which belongs to interval $[-10, 0]$: our goal is to engineer abstract domains able to express such properties, yet can be applied to many static analysis problems.

Abstraction of dynamic memory properties. For instance, a memory abstract domain consists of a set of predicates describing memory regions, together with operators for the analysis of memory operations (look-ups, assignments) and widening. XISA [3, 2] relies on points-to predicates, inductive predicates and segment predicates. A simplified version of this abstraction, where the only inductive predicates and segments that are considered are lists boils down to the following:

$$
\begin{array}{lll}
& \text{symbolic variables } \alpha, \alpha', \dots \text{ denote values and addresses} \\
\mathbf{m}(\in \mathbf{M}) ::= \mathbf{m} * \mathbf{m} & \text{separating conjunction of predicates} \\
\quad | \quad \alpha \cdot \mathrm{f} \mapsto \alpha' & \text{cell field f at address } \alpha \text{ containing value } \alpha' \\
\quad | \quad \mathrm{list}(\alpha) & \text{a list at address } \alpha \\
\quad | \quad \mathrm{list}(\alpha') \Rightarrow\!\!* \, \mathrm{list}(\alpha) & \text{a list segment starting at } \alpha \text{ and ending at } \alpha'
\end{array}
$$

The XISA [3] implementation actually represents a larger set of predicates, with arbitrary inductive definitions (including trees, doubly-linked lists and others). Other analysis frameworks utilize other sets of logical properties, such as, e.g., TVLA [9], which is based on reachability predicates.

Adding tracking for value properties, and departing from monolithic abstract domains. Once an abstraction has been defined for memory states, it is natural to extend it with value properties, so as to let the analysis infer constraints over both the structure of data and their values. A straightforward way to achieve this, and to add interval constraints over values is to extend the definition of abstract elements by $\mathbf{m} ::= \dots \mid \mathbf{m} \wedge \alpha \leq n \mid \mathbf{m} \wedge n \leq \alpha \mid \dots$. However, this implies the abstract operations (post-condition operators, join and widening) have to be extended so as to deal with both structures and value properties, at the same time: therefore abstract operations are bound to become overly

complex. Moreover, this approach is awkward, as it does not build upon existing abstract operations of value abstractions such as intervals [4] or octagons [8], which means it will not easily benefit from the efficient algorithms designed to infer such properties (the same also applies to the memory abstraction). Besides, it makes it harder to switch from one value abstraction to another at a later point, hence reducing the flexibility of the analysis.

In the following, we advocate a *modular abstract domain design*, which:
 – separates concerns in the abstract domain designs;
 – reuses existing abstract domains algorithms;
 – allows one to tune distinct parts of the abstractions independently.

Such design has been extensively used in the ASTRÉE static analyzer [1], which makes intensive use of reduced product [5] among other abstract domain combination techniques [6]. This design contributed not only to the precision and efficiency of the analysis, but also to making it easier to extend [6].

Abstraction of value properties, and combined abstract domain. To achieve a modular abstract domain design, we set up a different abstract domain $\mathbf{V}$ that will only track value properties (and not memory layout as the previously defined $\mathbf{M}$ does), and define a new abstract domain $\mathbf{S}$ for states that combines both:

$$
\begin{array}{ll}
\mathbf{m}(\in \mathbf{M}) ::= \ldots & \text{defined as before} \\
\mathbf{v}(\in \mathbf{V}) ::= \mathbf{true} \mid \mathbf{v} \wedge \mathbf{v} \mid \alpha \leq n \mid n \leq \alpha & \text{value predicates} \\
\mathbf{s}(\in \mathbf{S}) ::= \mathbf{m} \wedge \mathbf{v} & \text{conjunction of sub-properties}
\end{array}
$$

In essence, $\mathbf{S}$ defines a *reduced product* [5] of the memory abstraction $\mathbf{M}$ and value abstraction $\mathbf{V}$. As such, it completely separates memory and value abstraction concerns, which makes the abstract domain fully modular [11]. Indeed, both sub-components can be implemented in distinct ML modules, and $\mathbf{S}$ is defined as a ML functor. In practice, this functor should ensure that the symbolic variables used in the value abstraction are consistent with the memory cell contents and addresses symbols defined in the memory abstraction (thus it implements a co-fibered abstract domain [12], which essentially generalizes the notion of reduced product).

Both concrete states of Figure 1 can be abstracted by $\alpha \mapsto \alpha_0 * \alpha_0 \cdot \mathrm{n} \mapsto \alpha_1 * \alpha_0 \cdot \mathrm{d} \mapsto \alpha_0' * \mathrm{lpos}(\alpha_1) \wedge \alpha = \&\mathrm{x} \wedge -10 \leq \alpha_0' \wedge \alpha_0' \leq 0$, where inductive definition lpos describes all lists of positive numbers.

Separate combination of memory abstractions. So far, we combined abstract domains capturing distinct sets of properties. Yet, this abstract domain decomposition approach can be pushed further. As an example, ASTRÉE [1] relies on a decomposition of the numerical abstract domain into simpler abstractions that handle specific sets of properties. Likewise, a similar approach can be applied to the memory abstraction part. One approach to do this is to split concrete heaps and apply distinct memory abstractions to *disjoint* regions [11]:

$$
\begin{array}{ll}
\mathbf{m}(\in \mathbf{M}) ::= \mathbf{m_0} * \mathbf{m_1} & \text{where } \mathbf{m_0} \in \mathbf{M_0} \wedge \mathbf{m_1} \in \mathbf{M_1} \\
\mathbf{m_0}(\in \mathbf{M_0}) ::= \ldots & \text{defines a 1st memory abstract domain, e.g., for lists} \\
\mathbf{m_1}(\in \mathbf{M_1}) ::= \ldots & \text{defines a 2nd memory abstract domain, e.g., for arrays}
\end{array}
$$

This construction allows one to apply parsimoniously expensive memory abstractions to the memory regions that require them, while lighter weight abstractions can be used for simpler structures. This results in better control of the analysis complexity. A cost is that the analyzer now has to resolve memory fragments across sub-domains, and to also select which memory fragment is the most adequate to account for each memory allocation.

Reduced product of memory abstractions. Likewise, one can design a reduced product [5] of memory abstract domains [10]:

$$\mathbf{m}(\in \mathbf{M}) ::= \mathbf{m_0} \wedge \mathbf{m_1} \quad \text{where } \mathbf{m_0} \in \mathbf{M_0} \wedge \mathbf{m_1} \in \mathbf{M_1}$$

Such a composed abstraction is adequate when considering *overlaid* data structures [7] (such as lists or trees of objects with a common field pointing to class methods) and separates the concerns of analyzing each aspects of the structures. In turn, it imposes on the analysis the burden to let logical predicates represented in one sub-domain be usable to refine the computations done in the other sub-domain.

Modular abstract domain design. A modular abstract domain significantly simplifies the design of static analyzers while offering additional flexibility and control. The cost for this benefit is the innovation needed to design these more complex and general abstract domain combinators, but this cost is quickly amortized with the ability to reuse these combinators to realize arbitrary static analyzer configurations.

References

1. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.
2. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
3. B.-Y. E. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *SAS*, 2007.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the astrée static analyzer. In *ASIAN*, 2006.
7. O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *CAV*, 2011.
8. A. Miné. The octagon abstract domain. *HOSC*, 19(1):31–100, 2006.
9. M. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.
10. A. Toubhans, B.-Y. E. Chang, and X. Rival. Reduced product combination of abstract domains for shapes. In *VMCAI*, 2013.
11. A. Toubhans, B.-Y. E. Chang, and X. Rival. An abstract domain combinator for separately conjoining memory abstractions. In *SAS*, 2014.
12. A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *SAS*, 1996.