Entangled abstract domains for higher-order programs

Shuying Liang Matthew Might
University of Utah
{liangsy,might}@cs.utah.edu

Abstract
Relational abstract domains are a cornerstone of static analysis for first-order programs. We explore challenges in generalizing relational abstract domains to higher-order program analysis. We find two reasonable, orthogonal and complementary interpretations of relational domains in a higher-order setting. The first technique, locally relational abstract domains, are relational abstract domains that travel with the environments found within closures. These abstract domains record invariants discovered within a given scope. The second technique, globally entangled abstract domains, allow relational abstract domains to quantify over the concrete constituents of an abstract resource. This approach enables the discovery of interprocedural, interstructural and intrastructural program invariants. We develop the techniques for a lambda calculus enriched with structs. We structurally abstract the concrete semantics; we develop a logic for both the local and global generalizations; and then we integrate both logics into the abstraction. By restricting the logics, existing relational abstract domains (or their entangled, higher-order generalizations) are recoverable. To demonstrate the applicability of the framework, higher-order variants of both octagon and polyhedral domains are formulated as such restrictions.

1. Introduction: Relating higher-order values
Control-flow analysis is no longer enough for higher-order programs. In most instances, control-flow analysis (CFA) amounts to a “lambda-flow” analysis [8, 18, 19]. That is, control-flow analysis discovers where values (commonly abstracted as lambda terms) may flow. It does not discover what values may flow, or more precisely, the relationships between values. Despite these limitations, the bulk of the literature on higher-order program analysis focuses on improving the analytic precision rather than the semantic depth of the invariants discovered.

What higher-order program analysis needs is an analog to relational abstract domains.

Outline In this work, we grapple with a shortcoming of higher-order program analysis: imprecise, non-relational abstract domains. In bringing relational abstraction to abstract interpretation [3, 4] of higher-order programs, we find two distinct, orthogonal yet complementary generalizations of the concept:

1. locally relational abstract domains that travel with closed environments and mimic traditional relational abstract domains; and
2. globally “entangled” abstract domains that generalize relational abstract domains with universal quantification over concretization.¹

We also formulate the entangled equivalents of well-known relational abstract domains: entangled octagon domains [16] and entangled polyhedral domains [5].

1.1 Example: Capturing array bounds
Before adapting relational abstract domains to higher-order program analysis, we pause to motivate by example why inferring relational invariants is useful in a higher-order setting. Array bounds analysis is a classic application of relational abstract domains for flat, first-order languages. While most higher-order languages insert run-time array-bounds checks for security, reasoning about arrays is still important for efficiency, correctness and stability. Consider the expression:

\[
(f \ a)
\]

where the variable \( a \) is captured from an outer lexical scope. An optimizing compiler—or an analysis concerned with proving error-freedom—will ask, “What is the relationship between the length of the array \( arr \) and the value of \( i \)?”

While the previous question is sensible, many other questions one might ask in a flat-environment, first-order setting are ambiguous in a higher-order setting. Consider, for example, the question, “What is the relationship between the variable \( i \) and the array bound to \( a \)?” In a flat, first-order setting, there is but one copy of a variable, and they all exist in the same scope.

In a higher-order setting, there could be multiple closures over \( \lambda \) living at the same time, each capturing its own binding to \( i \). If the same capturing happens to \( a \) as well, the possible meanings of this question are multiplied by the number of such capturings. Entangled domains provide a means of specifying, precisely, which instances of \( a \) and \( i \) are of concern.

1.2 Example: Capturing intrastructural relationships
Classical CFAs miss low-hanging fruit when fields within a struct or an object are related to one another. Consider the constructor for a vector in a 3D engine:

¹ Quantification over all constituents of an abstract value is what “entangles” them.
(define (make-3d-vector vx vy vz)
  (struct [x vx]
    [y vy]
    [z vz]
    [norm (+ (* vx vx) (* vy vy) (* vz vz))]))

With the vector’s norm being a frequently used value, the 3d-vector struct caches it upon construction. It is not unreasonable to expect an analyzer to discover that:

(\let* ((\vx (3d-vector-x \v))
  \vy (3d-vector-y \v))
  \vz (3d-vector-z \v))
  (sqrt (+ (* vx vx) (* vy vy) (* vz vz))))

is equivalent to:

(sqrt (3d-vector-norm \v))

Yet, standard higher-order flow analyses cannot infer a relationship between the fields within a struct. A state-of-the-art CFA will infer that the fields x, y, z and norm are numeric, but it will not detect the relationship between them.

2. The setting: Enriched, a-normalized lambda calculus

We conduct our investigation of entangled domains from the perspective of the A-Normal Form (ANF) lambda calculus [6]. A-Normal Form fixes the order of evaluation and atomizes complex calculations, which makes it a popular intermediate format for functional compilation.

We prefer ANF’s atomization of complex expressions because it simplifies the number of invariants one might establish at any program point. To allow for rich, intrastructural relationships, the grammar for our extension of ANF (Figure 1) includes integers, primitives, structs and arrays.

3. Concrete semantics

In this section, we’ll present a standard CESK-style semantics for ANF, but with a pointer refinement that threads continuations through the store [20].

Figure 2 contains the concrete state-space for this machine. Another standard addition in preparation for static analysis is a context. This component will contain an ever-increasing program history that, under abstraction, will set the time component, \(t\). This component will will determine succession: \(\text{tick} : \Sigma \rightarrow \text{Time}\).

For allocating arrays/objects, the opaque function \text{alloc} selects a fresh location:

\(\text{alloc} : \Sigma \rightarrow \text{Loc}\).

The function \(\mathcal{A} : \mathcal{AExp} \times \text{Env} \times \text{Store} \rightarrow D\) evaluates atomic expressions:

\(\mathcal{A}(z, \rho, \sigma) = z\)

\(\mathcal{A}(\#t, \rho, \sigma) = \text{true}\)

\(\mathcal{A}(\#f, \rho, \sigma) = \text{false}\)

\(\mathcal{A}(v, \rho, \sigma) = \sigma(\rho(v))\)

\(\mathcal{A}(\text{lam}, \rho, \sigma) = (\text{lam}, \rho)\)

\(\mathcal{A}((\text{op} \, x_1 \ldots x_n), \rho, \sigma) = \sigma(\mathcal{A}(x_1, \rho, \sigma), \ldots, \mathcal{A}(x_n, \rho, \sigma))\)

\(\mathcal{A}((\text{R} \, x_1 \ldots x_n), \rho, \sigma) = (\mathcal{A}(x_1, \rho, \sigma), \ldots, \mathcal{A}(x_n, \rho, \sigma)) \in \mathcal{R}(R)\)

\(\mathcal{A}((\text{struct-ref} \, x \, v), \rho, \sigma) = \sigma(\text{fieldp}(\ell, v))\)

where \(\ell = \mathcal{A}(x, \rho, \sigma)\)

\(\mathcal{A}((\text{array-ref} \, x \, x'), \rho, \sigma) = \sigma(\text{eemplp}(\ell, z))\)

where \(\ell = \mathcal{A}(x, \rho, \sigma)\)

\(z = \mathcal{A}(x', \rho, \sigma)\)

\(\zeta \in \Sigma = \mathcal{E}xp \times \text{Env} \times \text{Store} \times \text{Kont} \times \text{Time}\)

\(\rho \in \text{Env} = \text{Var} \rightarrow \text{Addr}\)

\(\sigma \in \text{Store} = \text{Addr} \rightarrow \text{D}\)

\(\kappa \in \text{Cont} ::= \text{letk}(v, e, \rho, a)\)

| \text{haltk} |

\(d \in \text{D} = \text{Clo} + \text{Bas} + \text{Loc} + \text{Kont}\)

\(\text{clo} \in \text{Clo} = \text{Lam} \times \text{Env}\)

\(\ell \in \text{Loc}\) is an infinite set of struct locations

\(\text{bas} \in \text{Bas} = \mathbb{Z} + \{\text{true}, \text{false}\}\)

\(a \in \text{Addr} ::= \text{bindp}(v, t)\)

| \text{contp}(\text{lam}, t) |

| \text{eemplp}(\ell, z) |

| \text{fieldp}(\ell, v) |

\(t \in \text{Time}\) is an infinite set of contexts.

---

Figure 2. Concrete CESK-style state-space for ANF.

---

3.1 Concrete semantics

The concrete semantics for our dialect of ANF is small-step transition relation through the state-space \(\Sigma\):

\(\Rightarrow \subseteq \Sigma \times \Sigma\).

A few helper functions aid in the definition of this relation. For the interpretation of relations and operators, we use the functions \(\mathcal{R}\) and \(\mathcal{O}\):

\(\mathcal{R} : \text{Rel} \rightarrow \mathcal{P}(D^*)\)

\(\mathcal{O} : \text{Op} \rightarrow (D^* \rightarrow D)\)

We assume the natural definitions of these functions, and will repurpose them shortly for the definition of relational logics. For manipulating time, the opaque function \text{tick} determines succession:

\(\text{tick} : \Sigma \rightarrow \text{Time}\).

For allocating arrays/objects, the opaque function \text{alloc} selects a fresh location:

\(\text{alloc} : \Sigma \rightarrow \text{Loc}\).

The function \(\mathcal{A} : \mathcal{AExp} \times \text{Env} \times \text{Store} \rightarrow D\) evaluates atomic expressions:

\(\mathcal{A}(z, \rho, \sigma) = z\)

\(\mathcal{A}(\#t, \rho, \sigma) = \text{true}\)

\(\mathcal{A}(\#f, \rho, \sigma) = \text{false}\)

\(\mathcal{A}(v, \rho, \sigma) = \sigma(\rho(v))\)

\(\mathcal{A}(\text{lam}, \rho, \sigma) = (\text{lam}, \rho)\)

\(\mathcal{A}((\text{op} \, x_1 \ldots x_n), \rho, \sigma) = \sigma(\mathcal{A}(x_1, \rho, \sigma), \ldots, \mathcal{A}(x_n, \rho, \sigma))\)

\(\mathcal{A}((\text{R} \, x_1 \ldots x_n), \rho, \sigma) = (\mathcal{A}(x_1, \rho, \sigma), \ldots, \mathcal{A}(x_n, \rho, \sigma)) \in \mathcal{R}(R)\)

\(\mathcal{A}((\text{struct-ref} \, x \, v), \rho, \sigma) = \sigma(\text{fieldp}(\ell, v))\)

where \(\ell = \mathcal{A}(x, \rho, \sigma)\)

\(\mathcal{A}((\text{array-ref} \, x \, x'), \rho, \sigma) = \sigma(\text{eemplp}(\ell, z))\)

where \(\ell = \mathcal{A}(x, \rho, \sigma)\)

\(z = \mathcal{A}(x', \rho, \sigma)\)

\(\zeta \in \Sigma = \mathcal{E}xp \times \text{Env} \times \text{Store} \times \text{Kont} \times \text{Time}\)

\(\rho \in \text{Env} = \text{Var} \rightarrow \text{Addr}\)

\(\sigma \in \text{Store} = \text{Addr} \rightarrow \text{D}\)

\(\kappa \in \text{Cont} ::= \text{letk}(v, e, \rho, a)\)

| \text{haltk} |

\(d \in \text{D} = \text{Clo} + \text{Bas} + \text{Loc} + \text{Kont}\)

\(\text{clo} \in \text{Clo} = \text{Lam} \times \text{Env}\)

\(\ell \in \text{Loc}\) is an infinite set of struct locations

\(\text{bas} \in \text{Bas} = \mathbb{Z} + \{\text{true}, \text{false}\}\)

\(a \in \text{Addr} ::= \text{bindp}(v, t)\)

| \text{contp}(\text{lam}, t) |

| \text{eemplp}(\ell, z) |

| \text{fieldp}(\ell, v) |

\(t \in \text{Time}\) is an infinite set of contexts.
where invoked and binds each argument to the corresponding variable:

\[
\begin{align*}
\text{invoked} & : \text{expr} \\
\text{binds each argument} & : \text{binding}
\end{align*}
\]

To allocate a struct, the transition first allocates a location, and then installs the values of its fields at the corresponding addresses:

\[
\begin{align*}
(\text{let} ((v' (\text{struct} (v_1 \ldots v_n))) e')), f, x & \in \text{Exp} \\
& \Rightarrow (e', \rho', \sigma', \kappa', t')
\end{align*}
\]

where

\[
\begin{align*}
v' & : \text{Exp} \\
\rho' & : \text{Env} \\
\sigma' & : \sigma[a \mapsto \ell, a'_i \mapsto A(x_i, \rho, \sigma)] \\
\kappa' & : \ell = \text{alloc}(\varsigma)
\end{align*}
\]

**4. First attempt: Structural abstraction**

On the road to richer abstract domains for higher-order program analysis, we first pause at classical higher-order program analysis to observe its shortcomings with respect to encoding relational invariants. The subsequent three sections develop the machinery (locally relational and globally entangled abstract domains) necessary to rectify these shortcomings, and the following section integrates all three.

Applying the systematic abstraction of Van Horn and Might \[11, 20\] can transform the concrete state-space into an abstract state-space immediately suitable for an intensional static analysis (Figure 3). Classical control-flow analyses may be cast as instantiations of this state-space with choices for the set of abstract times and allocation strategies determining which classical CFA is recovered (see \[20\] for more details).
the abstraction over these sets, which can be encoded as a single
of locations. In essence, this tweaking takes place by modifying
state-space—the set of times, the set of basic values and the set
flow analysis have focused on tweaking the semantic leaves of this
co-live with each other. Traditional relational domains expect
higher-order program, there can be many bindings to each variable
a relational abstract domain is not straightforward, because in a
namespaces do this (at a conceptual level) by constructing the di-
binding environments (\(\hat{\rho}\)), since for any binding environment there is a finite number
of variables in scope; and
1. Locally relational abstract domains: Traditional relational ab-
stract domains may accompany abstract binding environments
2. Globally entangled abstract domains: By allowing abstract do-
ments to quantify over the concrete constituents of an abstract
value, we can formulate a direct product with a standard ab-
interpretation. Entangled abstract domains allow accumu-
knowledge to pass interprocedurally.

For the purpose of generality, we phrase relational domains in terms of restrictive logics. The next section defines a logic suitable for describing invariants that hold intraprocedurally. The following section defines a logic suitable for describing entangled interprocedural invariants.

6. A logic for locally relational abstract domains
A local logic allows the encoding of invariants holding within a spe-
cific program scope. Propositions within this logic will travel with
abstracted environments. Existing relational domains are phrased
as restricted (finite) subsets of this logic. Sets of propositions in the
local logic are members of the set \(LProps\) (Figure 4).

Since these propositions will be distributed throughout the ab-
state-space, it is problematic to allow them to reason about mutable
addresses, hence the omission of terms for referencing ar-
ays and structs. (It is problematic because modifying a mutable
value would imply a crawl through the entire state-space for modifi-
cation or removal of propositions involving the impacted value.) To
reason about mutable addresses and structures, we require the en-
tangled logic of Section 7, which uses addresses directly for ground
terms.

To give a semantics to this logic, we define the three-part satis-
relation (\(|=\)), in which a local environment paired with a
store may justify a proposition:

\[\rho, \sigma \models \phi \iff \rho \models \phi \text{ for all } \phi \in \Theta\]
\[\rho, \sigma \models \phi_1 \land \phi_2 \iff \rho, \sigma \models \phi_1 \text{ and } \rho, \sigma \models \phi_2\]
\[\rho, \sigma \models \phi_1 \lor \phi_2 \iff \rho, \sigma \models \phi_1 \text{ or } \rho, \sigma \models \phi_2\]
\[\rho, \sigma \models \lnot \phi \text{ iff it is not the case that } \rho, \sigma \models \phi\]

\[\rho, \sigma \models \phi \iff \rho \models \phi \iff \rho, \sigma \models \phi \text{ for all } \phi \in \Theta\]

\[\rho, \sigma \models \phi_1 \lor \phi_2 \iff \rho, \sigma \models \phi_1 \text{ or } \rho, \sigma \models \phi_2\]

\[\rho, \sigma \models \lnot \phi \text{ iff it is not the case that } \rho, \sigma \models \phi\]

\[\rho, \sigma \models \phi \iff \rho \models \phi \iff \rho, \sigma \models \phi \text{ for all } \phi \in \Theta\]

\[\rho, \sigma \models \phi_1 \land \phi_2 \iff \rho, \sigma \models \phi_1 \text{ and } \rho, \sigma \models \phi_2\]

\[\rho, \sigma \models \lnot \phi \text{ iff it is not the case that } \rho, \sigma \models \phi\]

\[\rho, \sigma \models \phi \iff \rho \models \phi \iff \rho, \sigma \models \phi \text{ for all } \phi \in \Theta\]

\[\rho, \sigma \models \phi_1 \lor \phi_2 \iff \rho, \sigma \models \phi_1 \text{ or } \rho, \sigma \models \phi_2\]

\[\rho, \sigma \models \lnot \phi \text{ iff it is not the case that } \rho, \sigma \models \phi\]

\[\rho, \sigma \models \phi \iff \rho \models \phi \iff \rho, \sigma \models \phi \text{ for all } \phi \in \Theta\]

\[\rho, \sigma \models \phi_1 \land \phi_2 \iff \rho, \sigma \models \phi_1 \text{ and } \rho, \sigma \models \phi_2\]

\[\rho, \sigma \models \lnot \phi \text{ iff it is not the case that } \rho, \sigma \models \phi\]

\[\rho, \sigma \models \phi \iff \rho \models \phi \iff \rho, \sigma \models \phi \text{ for all } \phi \in \Theta\]

\[\rho, \sigma \models \phi_1 \lor \phi_2 \iff \rho, \sigma \models \phi_1 \text{ or } \rho, \sigma \models \phi_2\]

\[\rho, \sigma \models \lnot \phi \text{ iff it is not the case that } \rho, \sigma \models \phi\]

\[\rho, \sigma \models \phi \iff \rho \models \phi \iff \rho, \sigma \models \phi \text{ for all } \phi \in \Theta\]

\[\rho, \sigma \models \phi_1 \land \phi_2 \iff \rho, \sigma \models \phi_1 \text{ and } \rho, \sigma \models \phi_2\]

\[\rho, \sigma \models \lnot \phi \text{ iff it is not the case that } \rho, \sigma \models \phi\]

\[\rho, \sigma \models \phi \iff \rho \models \phi \iff \rho, \sigma \models \phi \text{ for all } \phi \in \Theta\]

\[\rho, \sigma \models \phi_1 \lor \phi_2 \iff \rho, \sigma \models \phi_1 \text{ or } \rho, \sigma \models \phi_2\]

\[\rho, \sigma \models \lnot \phi \text{ iff it is not the case that } \rho, \sigma \models \phi\]

\[\rho, \sigma \models \phi \iff \rho \models \phi \iff \rho, \sigma \models \phi \text{ for all } \phi \in \Theta\]
\[ \Theta \in GProps = \mathcal{P}(GProp) \]

\[ \psi \in GProp \equiv \varphi \]
\[ \mid \forall x : \ell :: \psi \]
\[ \mid \forall x : a :: \psi \]

\[ \varphi \in GForm \equiv R(tg_1, \ldots, tg_n) \]
\[ \mid \varphi \lor \varphi \]
\[ \mid \varphi_1 \land \varphi_2 \]
\[ \mid \neg \varphi \]

\[ tg \in GTerm \equiv : x \mid d \mid a \]
\[ \mid op(tg_1, \ldots, tg_n) \]
\[ \mid tg_{array}[tg_{index}] \]
\[ \mid tg.v \]

Figure 5. An entangled logic for global (interprocedural) invariants. (\(x\) comes from a set of meta-variables.)

where the term interpretation function \(\mathcal{I}_g : LTerm \rightarrow D\) roughly mimics the argument evaluation function \(A\) from the semantics:

\[ \mathcal{I}_g(v) = \sigma(p(v)) \]
\[ \mathcal{I}_g(d) = d \]
\[ \mathcal{I}_g[\text{op}(tl_1, \ldots, tl_n)] = \mathcal{O}(\text{op})(\mathcal{I}_g(tl_1), \ldots, \mathcal{I}_g(tl_n)) \]

7. A logic for globally entangled abstract domains

To capture interprocedural invariants, or invariants between multiple bindings to the same address, an analysis requires a logic that can describe relationships between the values at addresses. For this purpose, we develop an “entangled” logic whose propositions are members of GProp (Figure 5). We term this logic entangled since it quantifies over the constituents of abstract values, thereby entangling invariants over them.

We provide a semantics for this logic through a two-part satisfaction relation, \((\models)\), which judges a proposition against a concrete store:

\[ \sigma \models \Theta \text{ iff } \sigma \models \psi \text{ for all } \psi \in \Theta \]
\[ \sigma \models \forall x : \ell :: \psi \text{ iff } \sigma \models \{ \ell/x \} \psi \text{ for each } \ell \text{ such that } \alpha(\ell) = \ell \]
\[ \sigma \models \forall x : a :: \psi \text{ iff } \sigma \models \{ a/x \} \psi \text{ for each } \ell \text{ such that } \alpha(\ell) = a \]
\[ \sigma \models \varphi_1 \lor \varphi_2 \text{ iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \]
\[ \sigma \models \varphi_1 \land \varphi_2 \text{ iff } \sigma \models \varphi_1 \text{ or } \sigma \models \varphi_2 \]
\[ \sigma \models \neg \varphi \text{ iff it is not the case that } \sigma \models \varphi \]

\[ \sigma \models R(tg_1, \ldots, tg_n) \text{ iff } (\mathcal{J}_g(tg_1), \ldots, \mathcal{J}_g(tg_n)) \in R(R), \]

where the term interpretation function \(\mathcal{J}_g\) evaluates terms into denotable values:

\[ \mathcal{J}_g(a) = \sigma(a) \]
\[ \mathcal{J}_g(d) = d \]
\[ \mathcal{J}_g[\text{op}(tg_1, \ldots, tg_n)] = \mathcal{O}(\text{op})(\mathcal{J}_g(tg_1), \ldots, \mathcal{J}_g(tg_n)) \]
\[ \mathcal{J}_g[tg.v] = \sigma(\text{fieldp}(\mathcal{J}_g(tg), v)) \]
\[ \mathcal{J}_g[tg.array[tg.index]] = \sigma(\text{elemp}(\mathcal{J}_g(tg_1), \mathcal{J}_g(tg_2))). \]

8. Second attempt: An entangled abstract state-space

With the two adaptations of relational abstract domains available for a higher-order analysis, we can determine an analysis by specifying an abstraction function. The high-level structure of the abstract state-space stays identical to Figure 3, but the internal structure of both abstract environments and abstract stores changes to include sets of propositions:

\[ \hat{\rho} \in \hat{Env} = (\text{Var} \rightarrow \hat{Addr}) \times LProps \]
\[ \hat{\sigma} \in \hat{Store} = (\hat{Addr} \rightarrow \hat{D}) \times GProps \]

We assume the natural definitions and point-wise, element-wise and member-wise lifting for partial orders over the structure of this abstract state-space. (Sets of propositions are ordered by implication.)

We encode specific relational and entangled abstract domains as restrictions on these logics. As such, the state-level abstraction function, \(\alpha_\Theta : \Sigma \rightarrow \hat{\Sigma}\), is parameterized by filtering sets—\(\theta\) and \(\Theta\). When abstract environments and abstract stores are abstracted into sets of propositions, any propositions not in these filter sets are discarded. In the forthcoming formulations of octagon and polyhedral domains, these filters sets are defined inductively. The abstraction function \(\alpha_\Theta\) is structural over the state-space, and it is composed of a family of simpler abstraction functions (Figure 6).

9. Example: Local octagons and entangled octagons

To tie this work back to first-order work on relational abstract domains, we explore higher-order adaptations of a couple of well-known abstract domains. We begin with Miné’s precise, efficient octagon domain [16]. Miné’s octagon domain encodes constraints between program variables \(x\) and \(y\) of the form \(\pm x \pm y \leq c\), where \(c\) is a constant between \(-\infty\) and \(\infty\), inclusive. For example, to encode that \(x = y\), the abstraction would assert both \(x - y \leq 0\) and \(y - x \leq 0\).

To import the octagon abstract domain locally, we can abstract with respect to a restricted subset of LProp:

\[ \text{LProp}_{\text{octagon}} \equiv v_1 - v_2 \leq z \]
\[ \mid v_1 + v_2 \leq z \]
\[ \mid -v_1 - v_2 \leq z. \]

We can entangle the octagon domain by allowing these assertions between sets of concrete addresses:

\[ \text{GProp}_{\text{octagon}} \equiv \varphi \]
\[ \mid \forall x : a :: \psi \]

\[ \varphi \in \text{GProp}_{\text{octagon}} \equiv x_1 - x_2 \leq z \]
\[ \mid x_1 + x_2 \leq z \]
\[ \mid -x_1 - x_2 \leq z. \]

Thus, for completeness, the instantiated octagonal abstract map for ANF is:

\[ \text{LProp}_{\text{octagon}}, \text{GProp}_{\text{octagon}}. \]

10. Example: Local polyhedra and entangled polyhedra

Classical polyhedral domains bound program state as a conjunction of linear inequalities over program variables. It is straightforward to restrict the local logic to produce locally polyhedral abstract
contribution of this work is to resolve the conflicts in merging the
for ANF is:
Again, for completeness, the instantiated polyhedral abstract map
descends from the branch initiated by Cousot and Halbwach’s study
This work descends from the line of work set in motion by the
11. Related work
This work descends from the line of work set in motion by the
relational branch of abstract interpretation with the higher-order
program analysis branch of abstract interpretation.
As the closest relative of this work, Might’s logic-flow analysis
(LFA) was a first attempt to integrate a propositional abstraction
into analysis of programs in continuation-passing style [10].
Our formulation of higher-order relational abstract domains is
largely inspired by LFA’s failure. LFA places no restrictions on
sets of propositions, which leads to non-termination without crude
widening operations, nor do abstract environments travel with local
propositions. As such, LFA cannot be termed a proper or full gen-
eralization of relational abstract domains in a higher-order setting.
Given the information available, our approach seems distinct from
the Cousots’ excursion into higher-order program analysis [2],
where the emphasis is on relational abstraction of procedures rather
than structures.
Aside from LFA, excursions beyond enhancing precision in
control-flow analysis have been limited thus far to environment
analysis [7, 9, 13, 19], the analog of shape analysis [1, 17] for
higher-order programs [12]. Even though environment analysis rea-
sors about the substructural equivalence of environments trapped
inside closures, it cannot express even the simplest relations be-
tween two values, e.g., linear inequalities. Yet environment analysis
plays an important supporting role for relational analysis. In
practice, it is difficult to assert and maintain quantified proposi-
tions without incorporating environment analysis [13–15] and
shape analysis [12]. Reusing an abstract address means that, in or-
der to preserve propositions quantifying over this address, all the
propositions must still hold. With an environment/shape analysis
such as anodization [12], the most recent reuse of an abstract ad-
dress lives apart from all prior uses. As a result, anodized addresses
have time to be initialized, to pass through conditional statements
and acquire invariants prior to merging with all previous instances.
With respect to this work, environment and shape analysis are
orthogonal abstractions, which can be combined via direct prod-
uct [12].
12. Conclusion
In this work, we grapple with a shortcoming of higher-order
program analysis: imprecise, non-relational abstract domains, by
proposing two reasonable, orthogonal yet complementary interpre-
tations of relational domains that are able to discover invariants in-
traprocedurally, interprocedurally, intrastructurally and interstruc-
turally in a higher-order setting.
Acknowledgments
This material is based on research sponsored by DARPA under
agreement number FA8750-12-2-0106. The U.S. Government is
authorized to reproduce and distribute reprints for Governmental
purposes notwithstanding any copyright notation thereon.
References
[1] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of
pointers and structures. In PLDI ’90: Proceedings of the ACM
SIGPLAN 1990 Conference on Programming Language Design and
Implementation, PLDI ’90, pages 296–310, 1990.
[2] P. Cousot and R. Cousot. Relational abstract interpretation of higher-
order functional programs. JTSPEFL ’91, Bordeaux. BIGRE, 74:33–
36, October 1991.
lattice model for static analysis of programs by construction or approx-
imation of fixpoints. In Conference Record of the Fourth ACM Sym-
analysis frameworks. In POPL ’79: Proceedings of the 6th ACM

\[
\alpha_{\phi}(c, \rho, \sigma, t) = (c, \alpha_{\phi}(\rho), \alpha_{\phi}(\sigma), \alpha_{\phi}(\sigma), \alpha_{\phi}(t))
\]
\[
\alpha_{\phi}(\rho) = (\alpha(\rho), \alpha_{\phi}(\rho) \cap \Theta)
\]
\[
\alpha(\rho) = \lambda v.\alpha(\rho(v))
\]
\[
\alpha_{\phi}(\rho) = \{ \phi : \rho, \sigma \models \phi \}
\]
\[
\alpha_{\phi}(\sigma) = (\alpha_{\phi}(\sigma), \alpha(\sigma) \cap \Theta)
\]
\[
\alpha(\sigma) = \{ \psi : \sigma \models \psi \}
\]
\[
\alpha_{\phi}(\text{letk}(v, e, \rho, \alpha)) = \text{letk}(v, e, \alpha_{\phi}(\rho), \alpha(\alpha))
\]
\[
\alpha_{\phi}(\text{haltk}) = \text{haltk}
\]
\[
\alpha_{\phi}(\text{lam}(\rho)) = (\text{lam}, \alpha_{\phi}(\rho))
\]
\[
\alpha(\text{bindp}(v, t)) = \text{bindp}(v, \alpha(\alpha(t)))
\]
\[
\alpha(\text{contp}(\text{lam}(\alpha(t))) = \text{contp}(\text{lam}, \alpha(\alpha(t)))
\]
\[
\alpha(\text{fieldp}(\ell, v)) = \text{fieldp}(\alpha(\text{loc}), v)
\]
\[
\alpha(\text{elem}(\ell, z)) = \text{fieldp}(\alpha(\text{loc}), \alpha(\ell))
\]
\[
\alpha(t) \text{ determines by context-sensitivity}
\]
\[
\alpha(\ell) \text{ determines by object polyvariance}
\]
\[
\alpha(\text{bas}) \text{ determines precision for basic values.}
\]

Figure 6. A family of abstraction maps that integrates locally re-
LProp_{poly} ::= z_1 v_1 + \cdots + z_n v_n \leq z_{\text{bound}}.
and, with nominally more effort, entangled polyhedral domains:
\[
\text{GProp}_{\text{poly}} ::= \varphi
\]
\[
\varphi \in \text{GForm}_{\text{poly}} ::= z_1 x_1 + \cdots + z_n x_n \leq z_{\text{bound}}.
\]
Again, for completeness, the instantiated polyhedral abstract map
for ANF is:
\[
\text{LProp}_{\text{poly}} \circ \text{GProp}_{\text{poly}}.
\]
11. Related work
This work descends from the line of work set in motion by the
Cousots’ original work on abstract interpretation [3, 4]. It also
descends from the branch initiated by Cousot and Halbwach’s study
of (polyhedral) relational abstract domains [5]. Jones’s initiated the
second branch from which this work descends with early results in
control-flow analysis [8], Van Horn et al. [20] provide a modern
treatment of this branch through systematic abstraction. The core
contribution of this work is to resolve the conflicts in merging the


