# 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, allows 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)
```

a classical flow analysis could determine that closures over the lambda terms $\lambda_{42}$ and $\lambda_{13}$ may flow to `f` and an array allocated from expression 12 may flow to the reference `a`. Suppose then that $\lambda_{42}$ is:

```
(λ (arr)
 (array-ref arr i)) ; arr[i] in most languages
```

where the variable `i` 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 $\lambda_{42}$ living at the same time, each capturing its own binding to (and value of) `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 time component, $Time$. This component will contain an ever-increasing program history that, under abstraction, will set the *context* in *context*-sensitivity. Structs and arrays are encoded by a base location $\ell$, so that the address of the field named $v$ is $\mathbf{fieldp}(\ell, v)$ and the address of index $z$ is $\mathbf{elemp}(\ell, z)$. Structs and arrays behave identically, so to save space, we present the semantics only for structs.

### 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 $\mathcal{O}$:

$$\mathcal{R} : \mathsf{Rel} \to \mathcal{P}\,(D^*)$$
$$\mathcal{O} : \mathsf{Op} \to (D^* \to D)$$

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 $tick$ determines succession:

$$tick : \Sigma \to Time.$$

For allocating arrays/objects, the opaque function $alloc$ selects a fresh location:

$$alloc : \Sigma \to Loc.$$

The function $\mathcal{A} : \mathsf{AExp} \times Env \times Store \rightharpoonup D$ evaluates atomic expressions:

$$\begin{aligned}
\mathcal{A}(z, \rho, \sigma) &= z \\
\mathcal{A}(\texttt{\#t}, \rho, \sigma) &= true \\
\mathcal{A}(\texttt{\#f}, \rho, \sigma) &= false \\
\mathcal{A}(v, \rho, \sigma) &= \sigma(\rho(v)) \\
\mathcal{A}(lam, \rho, \sigma) &= (lam, \rho) \\
\mathcal{A}([\![(op\ \text{æ}_1 \ldots \text{æ}_n)]\!], \rho, \sigma) &= \mathcal{O}\langle \mathcal{A}(\text{æ}_1, \rho, \sigma), \ldots, \mathcal{A}(\text{æ}_n, \rho, \sigma)\rangle \\
\mathcal{A}([\![(R\ \text{æ}_1 \ldots \text{æ}_n)]\!], \rho, \sigma) &= \langle \mathcal{A}(\text{æ}_1, \rho, \sigma), \ldots, \mathcal{A}(\text{æ}_n, \rho, \sigma)\rangle \in \mathcal{R}(R) \\
\mathcal{A}([\![\texttt{(struct-ref æ v)}]\!], \rho, \sigma) &= \sigma(\mathbf{fieldp}(\ell, v)) \\
&\quad \text{where } \ell = \mathcal{A}(\text{æ}, \rho, \sigma) \\
\mathcal{A}([\![\texttt{(array-ref æ æ')}]\!], \rho, \sigma) &= \sigma(\mathbf{elemp}(\ell, z)) \\
&\quad \text{where } \ell = \mathcal{A}(\text{æ}, \rho, \sigma) \\
&\quad\ \ z = \mathcal{A}(\text{æ}', \rho, \sigma)
\end{aligned}$$

***Function return*** Atomic expressions in tail position represent function return. Their transition pops the current continuation frame and resumes execution at the point of the function call within it:

$$\overbrace{(\text{æ}, \rho, \sigma, \mathbf{letk}(v, e, \rho', a), t)}^{\varsigma}$$
$$\Rightarrow (e, \rho'[v \mapsto a'], \sigma[a' \mapsto \mathcal{A}(\text{æ}, \rho, \sigma)], \kappa, t'), \text{ where}$$
$$a' = (v, t') \quad \kappa = \sigma(a)$$
$$\kappa = \sigma(a) \quad t' = tick(\varsigma).$$

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Exp} \times Env \times Store \times Kont \times Time \\
\rho \in Env &= \mathsf{Var} \to Addr \\
\sigma \in Store &= Addr \to D \\
\kappa \in Cont &::= \mathbf{letk}(v, e, \rho, a) \\
&\quad\ \ | \ \mathbf{haltk} \\
d \in D &= Clo + Bas + Loc + Kont \\
clo \in Clo &= \mathsf{Lam} \times Env \\
\ell \in Loc &\text{ is an infinite set of struct locations} \\
bas \in Bas &= \mathbb{Z} + \{true, false\} \\
a \in Addr &::= \mathbf{bindp}(v, t) \\
&\quad\ \ | \ \mathbf{contp}(lam, t) \\
&\quad\ \ | \ \mathbf{elemp}(\ell, z) \\
&\quad\ \ | \ \mathbf{fieldp}(\ell, v) \\
t \in Time &\text{ is an infinite set of contexts.}
\end{aligned}$$

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

$$
\begin{aligned}
f, æ \in \mathsf{AExp} &::= v \mid lam \mid z \mid \texttt{\#f} \mid \texttt{\#t} \\
&\mid (prim\ æ_1 \ldots æ_n) \\
&\mid (\texttt{struct-ref}\ æ\ v) \\
&\mid (\texttt{array-ref}\ æ_{\text{array}}\ æ_{\text{index}}) \\
lam \in \mathsf{Lam} &::= (\lambda\ (v_1 \ldots v_n)\ e) \\
ce \in \mathsf{CExp} &::= (f\ æ_1 \ldots æ_n) \\
&\mid (\texttt{if}\ æ\ e_{\text{true}}\ e_{\text{false}}) \\
&\mid (\texttt{array-alloc}\ æ_{\text{size}}) \\
&\mid (\texttt{struct}\ (v_1\ æ_1) \ldots (v_n\ æ_n)) \\
&\mid (\texttt{struct-set!}\ æ_{\text{struct}}\ v\ æ_{\text{value}}) \\
&\mid (\texttt{array-set!}\ æ_{\text{array}}\ æ_{\text{index}}\ æ_{\text{value}}) \\
&\mid (\texttt{let}\ ((v\ æ))\ e) \\
e \in \mathsf{Exp} &::= ce \\
&\mid æ \\
&\mid (\texttt{let}\ ((v\ ce))\ e) \\
prim \in \mathsf{Prim} &= \mathsf{Op} + \mathsf{Rel} \\
op \in \mathsf{Op} &= \{\texttt{+}, \texttt{-}, \texttt{*}, \ldots\} \\
R \in \mathsf{Rel} &= \{\texttt{=}, \texttt{<=}, \texttt{<}, \ldots\} \\
v \in \mathsf{Var} &\text{ is a set of identifiers} \\
z \in \mathbb{Z} &\text{ is the set of integers.}
\end{aligned}
$$

**Figure 1.** An enriched A-normal form lambda calculus.

***Tail function call*** Tail function call looks up the closure to be invoked and binds each argument to the corresponding variable:

$$
\begin{aligned}
\overbrace{(\llbracket(f\ æ_1 \ldots æ_n)\rrbracket, \rho, \sigma, \kappa, t)}^{\varsigma} &\Rightarrow (ce, \rho', \sigma', \kappa, t'), \text{ where} \\
(\llbracket(\lambda\ (v_1 \ldots v_n)\ e)\rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma) \\
a_i' &= \mathbf{bindp}(v_i, t') \\
\rho'' &= \rho'[v_i \mapsto a_i'] \\
\sigma' &= \sigma[a_i' \mapsto \mathcal{A}(æ_i, \rho, \sigma)] \\
t' &= tick(\varsigma).
\end{aligned}
$$

***Non-tail call*** Non-tail function call looks up the closure to be invoked and binds each argument to the corresponding variable:

$$
\overbrace{(\llbracket(\texttt{let}\ ((v'\ (f\ æ_1 \ldots æ_n)))\ e')\rrbracket, \rho, \sigma, \kappa, t)}^{\varsigma} \Rightarrow (ce, \rho'', \sigma', \kappa', t'),
$$

$$
\begin{aligned}
\text{where } (\overbrace{\llbracket(\lambda\ (v_1 \ldots v_n)\ e)\rrbracket}^{lam}, \rho') &= \mathcal{A}(f, \rho, \sigma), \quad a_i' = \mathbf{bindp}(v_i, t') \\
\rho'' &= \rho'[v_i \mapsto a_i'], \quad a_\kappa = \mathbf{contp}(lam, t') \\
\sigma' &= \sigma[a_i' \mapsto \mathcal{A}(æ_i, \rho, \sigma), a_\kappa \mapsto \kappa], \\
\kappa' &= \mathbf{letk}(v', e', \rho, a_\kappa) \quad t' = tick(\varsigma).
\end{aligned}
$$

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

$$
\overbrace{(\llbracket(\texttt{let}\ ((v'\ (\texttt{struct}\ (v_1\ æ_1) \ldots (v_n\ æ_n))))\ e')\rrbracket, \rho, \sigma, \kappa, t)}^{\varsigma}
$$

$$
\begin{aligned}
&\Rightarrow (e', \rho', \sigma', \kappa, t'), \\
\text{where } t' = tick(\varsigma), &\quad a = \mathbf{bindp}(v', t') \\
a_i' = \mathbf{fieldp}(\ell, v_i), &\quad \rho' = \rho[v' \mapsto a] \\
\sigma' = \sigma[a \mapsto \ell, &\quad a_i' \mapsto \mathcal{A}(æ_i, \rho, \sigma)] \\
\ell = alloc(\varsigma).&
\end{aligned}
$$

***Remaining cases*** Concrete transition rules for the remaining cases (array allocation and access, array/struct mutation, and tail variants thereof) are straightforward given the four above. Arrays behave like structs, with numbered indices instead of field names and a distinguished `size` field at the location storing the size of the array. We omit them to save space and focus the presentation on the topic of concern: incorporating relational and entangled abstractions into higher-order program analysis.

## 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).

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} \times \widehat{Time}$$
$$\hat{\rho} \in \widehat{Env} = \mathsf{Var} \to \widehat{Addr}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \hat{D}$$
$$\hat{\kappa} \in \widehat{Cont} ::= \mathbf{letk}(v, e, \hat{\rho}, \hat{a}) \mid \mathbf{haltk}$$
$$\hat{d} \in \hat{D} = \mathcal{P}\left(\widehat{Clo} + \widehat{Bas} + \widehat{Loc} + \widehat{Kont}\right)$$
$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{Env}$$
$\hat{\ell} \in \widehat{Loc}$ is a finite set of struct locations
$$\widehat{bas} \in \widehat{Bas} = \hat{\mathbb{Z}} + \{true, false\}$$
$$\hat{a} \in \widehat{Addr} ::= \mathbf{bindp}(v, \hat{t}) \mid \mathbf{contp}(lam, \hat{t}) \mid \mathbf{fieldp}(\hat{\ell}, v) \mid \mathbf{elemp}(\hat{\ell}, \hat{z})$$
$\hat{t} \in \widehat{Time}$ is a finite set of contexts
$\hat{z} \in \hat{\mathbb{Z}}$ is a partitioning abstraction of the integers.

**Figure 3.** A structural abstract state-space for ANF.

Classical approaches to improving reasoning ability in control-flow analysis have focused on tweaking the semantic leaves of this state-space—the set of times, the set of basic values and the set of locations. In essence, this tweaking takes place by modifying the abstraction over these sets, which can be encoded as a single function:

$$\alpha : (Bas \to \widehat{Bas}) \cup (Time \to \widehat{Time}) \cup (Loc \to \widehat{Loc}).$$

While tweaking these domains (and their allocation) can improve precision (or speed), no amount of tweaking can force these domains to capture invariants that relate variables and addresses to one another.

***Why classical CFAs fall short: An example*** A simple example, involving the multiplication of a number by itself, illustrates this limitation concisely. Consider the analysis of the expression `(* x x)`. If we choose signs—$\{-, 0, +\}$—as the abstract domain representing integers, a classical CFA will fail to determine that the result is always non-negative. It fails because the analysis does not see an abstract number multiplied by *itself*; it sees two abstract numbers multiplied together. Relational abstract domains *can* encode that a value is the square of another value. While classical CFAs could be patched to handle this specific case, this is a band-aid; relational abstract domains are the cure.

## 5. Enriching abstract domains with relational knowledge

To capture deeper invariants—invariants that relate variables or program addresses to one another—the abstract state-space must be enriched. First-order program analyses for languages with flat namespaces do this (at a conceptual level) by constructing the direct product of an intensional abstraction with a relational abstraction [5].

Constructing the direct product of a higher-order analysis with a relational abstract domain is not straightforward, because in a higher-order program, there can be many bindings to each variable that co-live with each other. Traditional relational domains expect a finite number of bindings (in fact, just one) to program variables. To adapt relational abstract domains, two options emerge:

1. Locally relational abstract domains: Traditional relational abstract domains may accompany abstract binding environments ($\hat{\rho}$), since for any binding environment there is a finite number of variables in scope; and
2. Globally entangled abstract domains: By allowing abstract domains to quantify over the concrete constituents of an abstract value, we can formulate a direct product with a standard abstract interpretation. Entangled abstract domains allow accmulated knowledge to pass interprocedurally.

For the purpose of generality, we phrase relational domains in terms of restrictable 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.

$$\theta \in LProps = \mathcal{P}(\mathsf{LProp})$$
$$\phi \in \mathsf{LProp} ::= R(tl_1, \ldots, tl_n) \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi$$
$$tl \in \mathsf{LTerm} ::= v \mid d \mid op(tl_1, \ldots, tl_n)$$

**Figure 4.** A logic for environment-local environments.

## 6. A logic for locally relational abstract domains

A local logic allows the encoding of invariants holding within a specific 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 abstract state-space, it is problematic to allow them to reason about mutable addresses, hence the omission of terms for referencing arrays and structs. (It is problematic because modifying a mutable value would imply a crawl through the entire state-space for modification or removal of propositions involving the impacted value.) To reason about mutable addresses and structures, we require the entangled logic of Section 7, which uses addresses directly for ground terms.

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

$$\rho, \sigma \models \theta \text{ iff } \rho, \sigma \models \phi \text{ for all } \phi \in \theta$$
$$\rho, \sigma \models \phi_1 \wedge \phi_2 \text{ iff } \rho, \sigma \models \phi_1 \text{ and } \rho, \sigma \models \phi_2$$
$$\rho, \sigma \models \phi_1 \vee \phi_2 \text{ iff } \rho, \sigma \models \phi_1 \text{ or } \rho, \sigma \models \phi_2$$
$$\rho, \sigma \models \neg\phi \text{ iff it is not the case that } \rho, \sigma \models \phi$$
$$\rho, \sigma \models R(tl_1, \ldots, tl_n) \text{ iff } \langle \mathcal{I}^{\rho}_{\sigma}(tl_1), \ldots, \mathcal{I}^{\rho}_{\sigma}(tl_n) \rangle \in \mathcal{R}(R),$$

$$\Theta \in \mathit{GProps} = \mathcal{P}(\mathsf{GProp})$$

$$\begin{aligned}
\psi \in \mathsf{GProp} ::= {} & \varphi \\
\mid {} & \forall x : \hat{\ell} :: \psi \\
\mid {} & \forall x : \hat{a} :: \psi
\end{aligned}$$

$$\begin{aligned}
\varphi \in \mathsf{GForm} ::= {} & R(tg_1, \ldots, tg_n) \\
\mid {} & \varphi_1 \vee \varphi_2 \\
\mid {} & \varphi_1 \wedge \varphi_2 \\
\mid {} & \neg\varphi
\end{aligned}$$

$$\begin{aligned}
tg \in \mathsf{GTerm} ::= {} & x \mid d \mid a \\
\mid {} & op(tg_1, \ldots, tg_n) \\
\mid {} & tg_{\text{array}}[tg_{\text{index}}] \\
\mid {} & tg.v
\end{aligned}$$

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

where the term interpretation function $\mathcal{I}^{\rho}_{\sigma} : \mathsf{LTerm} \rightharpoonup D$ roughly mimics the argument evaluation function $\mathcal{A}$ from the semantics:

$$\begin{aligned}
\mathcal{I}^{\rho}_{\sigma}(v) &= \sigma(\rho(v)) \\
\mathcal{I}^{\rho}_{\sigma}(d) &= d \\
\mathcal{I}^{\rho}_{\sigma}[\![op(tl_1, \ldots, tl_n)]\!] &= \mathcal{O}(op)\langle \mathcal{I}^{\rho}_{\sigma}(tl_1), \ldots, \mathcal{I}^{\rho}_{\sigma}(tl_n)\rangle
\end{aligned}$$

## 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:

$$\begin{aligned}
\sigma \models \Theta &\text{ iff } \sigma \models \psi \text{ for all } \psi \in \Theta \\
\sigma \models \forall x : \hat{\ell} :: \psi &\text{ iff } \sigma \models \{\ell/x\}\,\psi \text{ for each } \ell \text{ such that } \alpha(\ell) = \hat{\ell} \\
\sigma \models \forall x : \hat{a} :: \psi &\text{ iff } \sigma \models \{a/x\}\,\psi \text{ for each } \ell \text{ such that } \alpha(a) = \hat{a} \\
\sigma \models \varphi_1 \wedge \varphi_2 &\text{ iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \varphi_1 \vee \varphi_2 &\text{ iff } \sigma \models \varphi_1 \text{ or } \sigma \models \varphi_2 \\
\sigma \models \neg\varphi &\text{ iff it is not the case that } \sigma \models \varphi \\
\sigma \models R(tg_1, \ldots, tg_n) &\text{ iff } \langle \mathcal{J}_{\sigma}(tg_1), \ldots, \mathcal{J}_{\sigma}(tg_n)\rangle \in \mathcal{R}(R),
\end{aligned}$$

where the term interpretation function $\mathcal{J}_{\sigma}$ evaluates terms into denotable values:

$$\begin{aligned}
\mathcal{J}_{\sigma}(a) &= \sigma(a) \\
\mathcal{J}_{\sigma}(d) &= d \\
\mathcal{J}_{\sigma}[\![op(tg_1, \ldots, tg_n)]\!] &= \mathcal{O}(op)\langle \mathcal{J}_{\sigma}(tg_1), \ldots, \mathcal{J}_{\sigma}(tg_n)\rangle \\
\mathcal{J}_{\sigma}[\![tg.v]\!] &= \sigma(\mathbf{fieldp}(\mathcal{J}_{\sigma}(tg), v)) \\
\mathcal{J}_{\sigma}[\![tg_1[tg_2]]\!] &= \sigma(\mathbf{elemp}(\mathcal{J}_{\sigma}(tg_1), \mathcal{J}_{\sigma}(tg_2))).
\end{aligned}$$

## 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:

$$\begin{aligned}
\hat{\rho} \in \widehat{Env} &= (\mathsf{Var} \to \widehat{Addr}) \times \mathit{LProps} \\
\hat{\sigma} \in \widehat{Store} &= (\widehat{Addr} \to \hat{D}) \times \mathit{GProps}
\end{aligned}$$

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, $\alpha^{\theta}_{\Theta} : \Sigma \to \hat{\Sigma}$, 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 $\alpha^{\theta}_{\Theta}$ 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:

$$\begin{aligned}
\mathsf{LProp}_{\text{octagon}} ::= {} & v_1 - v_2 \leq z \\
\mid {} & v_1 + v_2 \leq z \\
\mid {} & -v_1 - v_2 \leq z.
\end{aligned}$$

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

$$\begin{aligned}
\mathsf{GProp}_{\text{octagon}} ::= {} & \varphi \\
\mid {} & \forall x : \hat{a} :: \psi
\end{aligned}$$

$$\begin{aligned}
\varphi \in \mathsf{GForm}_{\text{octagon}} ::= {} & x_1 - x_2 \leq z \\
\mid {} & x_1 + x_2 \leq z \\
\mid {} & -x_1 - x_2 \leq z.
\end{aligned}$$

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

$$\alpha^{\mathsf{LProp}_{\text{octagon}}}_{\mathsf{GProp}_{\text{octagon}}}.$$

## 10. Example: Local polyhedra and entangled polyhedra

Classical polyhedral domains bound program state as a conjuction of linear inequalities over program variables. It is straightforward to restrict the local logic to produce locally polyhedral abstract

$$\alpha^{\theta}_{\Theta}(e,\rho,\sigma,\kappa,t) = (e, \alpha^{\theta}_{\sigma}(\rho), \alpha^{\theta}_{\Theta}(\sigma), \alpha^{\theta}_{\sigma}(\kappa), \alpha(t))$$

$$\begin{aligned}
\alpha^{\theta}_{\sigma}(\rho) &= (\alpha(\rho), \alpha_{\sigma}(\rho) \cap \theta) \\
\alpha(\rho) &= \lambda v.\alpha(\rho(v)) \\
\alpha_{\sigma}(\rho) &= \{\phi : \rho, \sigma \models \phi\}
\end{aligned}$$

$$\begin{aligned}
\alpha^{\theta}_{\Theta}(\sigma) &= (\alpha^{\theta}(\sigma), \alpha(\sigma) \cap \Theta) \\
\alpha^{\theta}(\sigma) &= \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \left\{ \alpha^{\theta}_{\sigma}(\sigma(a)) \right\} \\
\alpha(\sigma) &= \{\psi : \sigma \models \psi\}
\end{aligned}$$

$$\begin{aligned}
\alpha^{\theta}_{\sigma}(\mathbf{letk}(v,e,\rho,a)) &= \mathbf{letk}(v, e, \alpha^{\theta}_{\sigma}(\rho), \alpha(a)) \\
\alpha^{\theta}_{\sigma}(\mathbf{haltk}) &= \mathbf{haltk}
\end{aligned}$$

$$\alpha^{\theta}_{\sigma}(lam,\rho) = (lam, \alpha^{\theta}_{\sigma}(\rho))$$

$$\begin{aligned}
\alpha(\mathbf{bindp}(v,t)) &= \mathbf{bindp}(v, \alpha(t)) \\
\alpha(\mathbf{contp}(lam,t)) &= \mathbf{contp}(lam, \alpha(t)) \\
\alpha(\mathbf{fieldp}(\ell,v)) &= \mathbf{fieldp}(\alpha(loc), v) \\
\alpha(\mathbf{elemp}(\ell,z)) &= \mathbf{fieldp}(\alpha(loc), \alpha(z))
\end{aligned}$$

$\alpha(t)$ determines by context-sensitivity
$\alpha(\ell)$ determines by object polyvariance
$\alpha(bas)$ determines precision for basic values.

**Figure 6.** A family of abstraction maps that integrates locally relational abstract domains with globally entangled abstract domains into a structural abstraction.

domains:

$$\mathsf{LProp}_{\mathrm{poly}} ::= z_1 v_1 + \cdots + z_n v_n \leq z_{\mathrm{bound}},$$

and, with nominally more effort, entangled polyhedral domains:

$$\begin{aligned}
\mathsf{GProp}_{\mathrm{poly}} ::= {} & \varphi \\
\mid {} & \forall x : \hat{a} :: \psi
\end{aligned}$$

$$\varphi \in \mathsf{GForm}_{\mathrm{poly}} ::= z_1 x_1 + \cdots + z_n x_n \leq z_{\mathrm{bound}}.$$

Again, for completeness, the instantiated polyhedral abstract map for ANF is:

$$\alpha^{\mathsf{LProp}_{\mathrm{poly}}}_{\mathsf{GProp}_{\mathrm{poly}}}.$$

## 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 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 generalization 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 reasons about the substructural equivalence of environments trapped inside closures, it cannot express even the simplest relations between 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 propositions without incorporating environment analysis [13–15] and shape analysis [12]. Reusing an abstract address means that, in order 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 address 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 product [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 interpretations of relational domains that are able to discover invariants intraprocedurally, interprocedurally, intrastructurally and interstructurally 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. JTASPEFL '91, Bordeaux. *BIGRE*, 74:33–36, October 1991.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM*

*SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, 1979.

[5] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, 1978.

[6] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 237–247, June 1993.

[7] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 329–341, 1998.

[8] Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128, 1981.

[9] Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[10] Matthew Might. Logic-flow analysis of higher-order programs. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–198, 2007.

[11] Matthew Might. Abstract interpreters for free. In *SAS 2010: Proceedings of the 17th Static Analysis Symposium*, SAS'10, pages 407–421, 2010.

[12] Matthew Might. Shape analysis in the absence of pointers and structure. In *VMCAI 2010: International Conference on Verification, Model-Checking and Abstract Interpretation*, pages 263–278, January 2010.

[13] Matthew Might and Olin Shivers. Environment analysis via Delta-CFA. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 127–140, 2006.

[14] Matthew Might and Olin Shivers. Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 13–25, 2006.

[15] Matthew Might and Olin Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming*, 18(Special Double Issue 5-6):821–864, 2008.

[16] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100–100, March 2006.

[17] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[18] Olin Shivers. Control flow analysis in Scheme. *SIGPLAN Not.*, 23(7): 164–174, June 1988.

[19] Olin G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[20] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, 2010.