Abstracting Faceted Execution

Kristopher Micinski*, David Darais†, Thomas Gilray‡
*Syracuse University {kkmicins}@syr.edu
†University of Vermont {david.darais}@uvm.edu
‡University of Alabama at Birmingham {gilray}@uab.edu

***Abstract*—Faceted execution is a linguistic paradigm for dynamic information-flow control with the distinguishing feature that program values may be *faceted*. Such values represent multiple versions or facets at once, for different security labels. This enables policy-agnostic programming: a paradigm permitting expressive privacy policies to be declared, independent of program logic. Although faceted execution prevents information leakage at runtime, it does not guarantee the absence of failure due to policy violations. By contrast with static mechanisms (such as security type systems), dynamic information-flow control permits arbitrarily expressive and dynamic privacy policies but imposes significant runtime overhead and delays discovery of any possible violations.**

In this paper, we present the two different abstract interpretations for faceted execution in the presence of first-class policies. We first present an abstraction which allows one to reason statically about the *shape* of facets at each program point. This abstraction is useful for statically proving the absence of runtime errors and eliminating runtime checks related to facets. Reasoning statically about the contents of faceted values, however, is complicated by the presence of first-class security labels, especially because *abstract labels* may conflate more than one runtime label. To address these issues, we also develop a more precise abstraction that relies on an analysis tracking singleton heap abstractions. We present an implementation of our coarse abstraction in Racket and demonstrate its performance on several sample programs. We conclude by showing how our precise domain can be used to verify information-flow properties.

I. Introduction

Digital systems are used to manage sensitive data more than ever before. As these systems continue to grow in complexity, so do their privacy policies. In the wild, these policies are dynamic, imperfect and evolve over time, yet we still lack the tools to design software robust to policy evolution. Developers face daunting (re)engineering efforts in order to ensure systems correctly implement their stated policies—the program logic to implement these policies typically being scattered throughout a codebase, making it hard to gain confidence in its correctness. Unlike program crashes—which may simply cause downtime—bugs in policy code is likely to have privacy implications for millions of users.

There is a vibrant research community built around reasoning about the information-flow security of data. These efforts have culminated in successful programming languages (e.g., Jif [35]), analysis techniques (e.g., self-composition and product programs [6], [5]), and core formalisms (e.g., the dependency core-calculus and decentralized label model [34]). However, all of these solutions assume a single static privacy policy will hold forever. As we are reminded—often daily—this is essentially never the case.

Policy-agnostic programming [48] allows developers to write code that interacts with sensitive data without any additional logic to enforce the data's privacy policy—the implementation of such logic is often error-prone and changes rapidly. Instead, a dynamic monitor is used to ensure the system respects the data's privacy policy, regardless of the functional program logic. This allows data to be guarded by privacy policies that are written independently but remain as expressive as the underlying language itself.

In this paper, we present a static analysis methodology for programs written in the policy-agnostic style. Our technique works by analyzing the program using a semantics built on faceted execution—a dynamic monitor for policy-agnostic programming. Faceted execution [3], [4] represents privileged data via decision trees, where each node is a policy or privilege level (keyed on a *label*), and branches represent views of the data from two perspectives: one where the nodes policy grants the privilege, and one where it doesn't.

As computation progresses, faceted execution normally must propagate data from both views, versions, or privilege levels, ensuring that protected data is never leaked to an unprivileged context other than by means of an explicit observation (and evidence that the policy holds). We leverage this faceted semantics and design a static analysis using abstract interpretation [11] of an abstract machine [43]. The result is a static analysis for policy-agnostic programs which manipulate data in a system with dynamic authorization policies.

A core technical problem in designing our abstract interpretation is the choice of abstract domain for faceted values. As we show, the natural structural abstraction for facets is unsound, as our abstract interpretation must necessarily approximate the set of (unbounded) runtime policies in a finite way. Instead, we present a sound but imprecise abstract domain for facets which merges the two branches of a facet into a single branch representing an approximation of both values. Additionally, we observe that we can distinguish branches as long as the label guarding the facet is approximated by a singleton abstraction. Therefore, we present a more precise representation of facets whose labels can be shown to be representing exactly one concrete label.

We implemented our abstract interpreter in Racket, scaling our core formalism to handle *k*-ary lambdas, built-ins, `let` bindings, and conditionals. Our precise abstract domain for facets relies on abstract counting—a technique to ascertain whether abstractions of labels in our programs are singletons in the analysis. Abstract counting [32] is known to be imprecise using a global heap analysis, so we present a frontier semantics for retaining high-precision abstract counting while maintaining an efficient analysis overall. Our analysis then uses a lazy count-based facet collapse, so that we can soundly move from a high-precision analysis when possible to a sound but less precise abstraction when necessary.

Specifically, this paper makes the following contributions: first, we present a novel formulation of faceted execution as a small-step semantics. We develop a sound abstract domain for faceted values that is precise for the abstract labels guarding its facets but imprecise for its underlying values—we call this a branch-insensitive abstraction. Next, we present a precise abstract domain for faceted values that retains its sensitivity to distinct branches but is only sound for singleton abstract labels. We present an implementation of our coarse abstract domain in Racket and detail its performance on five benchmark programs. We conclude by discussing how our precise abstract domain may be applied to verify information-flow properties.

II. Background

To introduce our setting we present the implementation of Battleship, a small guessing-based board game, using a policy-agnostic programming paradigm. In this game, each player has a grided board on which they place tiles (or "ships"). The players hide their boards from each other as play progresses in rounds. Each turn, a player guesses the position of a ship on the other players board. If the guess is successful, that tile is removed from the board. Play ends once one player's board has no remaining tiles, at which point that player loses.

We implement game boards as lists of cons cells representing the (x, y) coordinates of ships. Board creation yields an empty list, and adding a piece is implemented using cons:

```
(define (makeboard) '())
(define (add-piece board x y)
  (cons (cons x y) board))
```

Next, we define mark-hit, which takes a player's board and removes a piece if the guessed coordinate is present. We return a pair of the updated board and a boolean indicating whether the guess was a hit:

```
(define (mark-hit board x y)
  (if (null? board)
    (cons board #f)
    (let* ([fst (car board)]
           [rst (cdr board)])
      (if (and (= (car fst) x)
               (= (cdr fst) y))
        (cons rst #t)
        (let ([rst+b (mark-hit rst x y)])
          (cons (cons fst (car rst+b))
                (cdr rst+b)))))))
```

Although mark-hit will operate on sensitive data (game boards), it is written without any special machinery to maintain the secrecy of board. Protecting data w.r.t. policies is instead handled automatically and implicitly by a runtime monitor. When Alice and Bob want to play, they both create a *label* to protect their game board. A label is a dynamically allocated predicate that takes an argument, a credential—only if the predicate returns true for the credential should the value's secret branch be observable. Alice's label is used to annotate whatever data she wants kept secret. Supposing Alice chooses to be player 1, she will use the following label:

```
(define alice-label (label [x] (= 1 x)))
```

Bob would use a similar label (but for player 2 instead of player 1). At runtime, the label form creates a label ℓ_A and returns it to the binding for alice-label. When Alice wants to protect a value, she creates a *facet*, annotated with her label and two *branches*. The positive (left) branch represents the value as it should appear to her, and the negative (right) to everyone else:

```
(define alice-board ⟨ alice-label : (add-pieces
    (makeboard) x1 y1 … xn yn) ⋄ ★ ⟩)
```

$$\begin{tikzcd}[column sep=small] & \ell_{\text{Family}} \arrow[dl, dash] \arrow[dr, dash] & & \\ p_1 & & \ell_{\text{Friends}} \arrow[dl, dash] \arrow[dr, dash] & \\ & p_2 & & p_3 \end{tikzcd}$$

In the above example, Alice uses ★ (lazy failure) to represent that others parties should fail if attempting to observe her data. In other applications, she might choose a public default value to reveal to others. She may even want to create a *nested* facet. For example, in a social-networking application she may want a nested facet consisting of two labels: ℓ_{Friends} and ℓ_{Family}. She would present three views of her social-media profile: p_1 to her family, containing her phone number, p_2 to her friends showing her interests, and p_3 to everyone else, showing only name and email.

As gameplay progresses, Alice and Bob both make guesses, and a driver calls the function mark-hit with each of their (faceted) game boards. However, because Alice and Bob's game boards are both facets, mark-hit cannot be immediately applied, as the argument board is a facet. Faceted execution "splits" the execution of the function on faceted arguments, running it first on the positive branch, then again on the negative branch. Finally, the results from each branch are merged again to produce a faceted value:

$$\begin{tikzcd}[column sep=tiny] & (\text{mark-hit } \langle \ell_A ? v^+ \diamond v^- \rangle \;\; x \;\; y) \arrow[dl] \arrow[dr] & \\ (\text{mark-hit } v^+ \; x \; y) = v' \arrow[dr] & & (\text{mark-hit } v^- \; x \; y) = v'' \arrow[dl] \\ & \langle \ell_A ? v' \diamond v'' \rangle & \end{tikzcd}$$

Because the applied function could be stateful, faceted execution also records the current privilege level in a *program counter* when splitting evaluation over two branches of a faceted value. The program counter is used to build new label-guarded facets when writes are made to the store inside a privileged context. We expand upon these subtleties in Section III, where we present a full semantics.

To introspect on faceted values (e.g., the return value from mark-hit) we must *observe* them. Facet observation is performed via the obs form in our semantics, which takes a label, argument to the label, and potentially-faceted value:

$$\begin{aligned} &\text{env} = \{\ell_A \mapsto \langle \lambda\texttt{x. (= x 1)}\rangle\} \\ &\qquad \texttt{(obs alice-label } 1\ \langle \ell_A \ ?\ v^+ \diamond v^- \rangle\texttt{)} \rightsquigarrow^* \\ &\qquad\qquad \texttt{(if (= } 1 \texttt{ 1) } v^+ \ v^- \texttt{)} \rightsquigarrow^* v^+ \end{aligned}$$

$\texttt{(obs l v}^1\ \texttt{v}^2\texttt{)}$ first executes the predicate associated with l using the argument $\texttt{v}^1$. Then, obs will remove all of the facets from $\texttt{v}^2$, selecting either the positive or negative branch based on whether the label's predicate returns true or false. For example, $\text{env}(\ell_A)(1) \rightsquigarrow^*$ true, so obs select the right side of the Alice's facet.

Analyzing Faceted Execution. Faceted execution will ensure at runtime that no secret information leaks from Alice to Bob. However, faceted execution cannot guarantee sensible results (e.g., observing $\star$ results in failure). We want to be able to write code independent of the policy, while retaining the robustness of dynamic policies. Faceted execution achieves this at runtime, but doesn't tell us anything about the policy statically. Additionally, faceted execution is a fairly heavy-weight dynamic monitor, imposing significant performance overhead, especially on code that uses many-faceted values. Ideally, we want the best of both worlds: the flexibility of dynamic policies with the ability to verify many of them up-front. In the case that we cannot verify a policy ahead of time—as all analyses must concede some degree of imprecision—we can gracefully degrade to normal faceted execution.

Static analysis has dynamic monitoring to fall back on, where it fails, so whatever can be learned statically will be of benefit to optimization and verification with no improvement being the worst case. This case may arise when complex and dynamic use of faceted values thwarts analysis by requiring it to see through many layers of abstraction. Neither the failures or successes of static analysis will lead to an increased permissiveness in the compiled application: failures can fall back on dynamic monitoring, successes correspond to true proofs that the application code must behave as indicated by the analysis. Analysis is at odds with dynamic monitoring in that all relevant code should be available or an analysis cannot help but yield no information. For example, a system may allow for security or privacy policies to be updated live, without rebuilding the application code; in this case, static analysis can only verify properties that are fully independent of any information-control policies. Static analysis can only verify specific properties of code that is available at analysis-time.

For this example, our static analysis (developed in Section IV) tells us that every call to `obs` selects only the positive view of a faceted value. This could be used to gain confidence in the program's security despite the use of dynamic policies. It also enables an optimization: as the program never violates the policy, the negative branch of the facets don't need to be computed at all. Inside of a compiler (for programs written in policy-agnostic languages) this result could be used to eliminate the machinery needed to manage faceted applications.

III. Semantics of Faceted Execution

Figure 1 gives the syntax for our source language. λ_{FE} extends the lambda calculus with mutable references and three new forms unique to faceted execution: label creation, facet creation, and facet observation. Our implementation (described in Section V) also includes `let` bindings, conditionals, various builtins, k-ary lambdas, and sequencing.

The `label` form dynamically generates and returns a new label each time it is evaluated. Such labels uniquely address a policy predicate comprised of a policy variable x and a policy body e (closed by parameter x and the current environment). When a policy is later invoked, the body evaluates to a boolean that indicates whether to observe the positive or negative branch of a facet. Facets are created with the form $\langle e_1 \ ? \ e_2 \ \diamond \ e_3\rangle$ where the label (address) returned by e_1 is allowed to be an expression (as label addresses are first-class). The expressions e_2 and e_3 are the facet's positive and negative branches, respectively. To introspect on a faceted value, the expression $\texttt{obs}[e_1 @ e_2](e_3)$ observes the label e_1 of faceted value e_3. The expression e_2 is evaluated to a key value and passed to the policy predicate bound to the label bound to e_1—if the policy predicate returns true, all facets guarded by label e_1 under e_3 are replaced by their positive branch, or negative branch when the predicate returns false.

$$\begin{array}{llll}
c \in & \text{const} & ::= () \mid \texttt{true} \mid \texttt{false} \mid \ldots & \textit{constants} \\
x \in & \text{var} & \triangleq \langle \textit{identifiers} \rangle & \textit{variables} \\
e \in & \text{exp} & ::= c \mid x & \textit{constants} \mid \textit{variables} \\
 & & \mid \ \lambda x.\ e \mid e(e) & \textit{function creation} \mid \textit{application} \\
 & & \mid \ \texttt{ref}(e) \mid\ !e \mid e \leftarrow e & \textit{reference creation} \mid \textit{read} \mid \textit{write} \\
 & & \mid \ \texttt{label}[x](e) & \textit{label creation} \\
 & & \mid \ \langle e \ ? \ e \diamond e \rangle & \textit{facet creation} \\
 & & \mid \ \texttt{obs}[e @ e](e) & \textit{observing a faceted value}
\end{array}$$

Fig. 1. Syntax of λ_{FE}

We formalize the concrete semantics of λ_{FE} as a big-step reduction relation presented in Figure 2. Our presentation primarily follows that of Austin et al. [3]. The reduction relation $\Downarrow^{E}_{pc}$ takes an environment (ρ), store (σ), and term (e), to produce a final value and store. The relation is parameterized by a current *program counter* (pc) which is a set of *branches*: positive or negative labels. As evaluation splits to evaluate the positive and negative branches of facets, the program counter remembers which branches were taken. The program counter is used for two reasons. First, it avoids doing redundant work by selecting the left branch of a facet such as $\langle \ell \ ? \ v^{+} \diamond v^{-}\rangle$ when $+\ell \in pc$, rather than splitting. Second, it is used during store update to form facets that remember the label of the privileged information along the branch.

The rules for constants (Const), variables (Var), and lambda (Lam) are standard. However, many other rules must be extended to account for faceted values. For example, application (App) must handle faceted values being applied. Consider the application $(\ell \ ? \ \lambda x.\ x \diamond \lambda x.\ 0)(1)$. To handle this, the App rule calls out to an application relation $\Downarrow^{A}_{pc}$. This reduction splits execution in the case that a facet with label ℓ is applied to a value (AppFacetSplit), as long as $\{+\ell, -\ell\} \cap pc = \varnothing$, indicating that execution has not yet split on ℓ (so we cannot be sure if we have permission for label ℓ yet). The application rule first considers the positive branch, applying $\langle \lambda x.\ x, \rho\rangle$ to 1 while extending pc to record the fact that the positive branch was taken. The AppBase rule applies unfaceted values to arguments in the expected way. Next, the negative branch is evaluated under the extended program counter $pc \cup \{-\ell\}$, being careful to thread through the store produced by the evaluation of the positive branch. After both branches are evaluated, the rule facets the results again using label ℓ, in this case producing $\langle \ell \ ? \ 1 \diamond 0\rangle$. To avoid creating redundant facets, the application rules do not split execution when a branch is already present in pc. Instead, the rules AppFacetLeft and AppFacetRight select the appropriate branch of the facet to apply. Last, AppStar handles the application of $\star$, a value representing lazy failure, useful as a default value for store updates.

$$\begin{array}{rrcl}
\alpha \in & \text{vaddr} & \triangleq & \ldots \\
\ell \in & \text{label} & \triangleq & \ldots \\
bv \in & \text{bval} & ::= & c \mid \alpha \mid \ell \mid \langle \lambda x.\ e, \rho \rangle \mid \star \\
v \in & \text{fval} & ::= & bv \mid \langle \ell\ ?\ v \diamond v \rangle
\end{array}
\qquad
\begin{array}{rrcl}
b \in & \text{branch} & ::= & +\ell \mid -\ell \\
pc \in & \text{PC} & \triangleq & \wp(\text{branch}) \\
\rho \in & \text{env} & \triangleq & \text{var} \rightharpoonup \text{val} \\
\sigma \in & \text{store} & \triangleq & \text{vaddr} \uplus \text{label} \rightharpoonup \text{val}
\end{array}$$

(Selected rules only...) $\boxed{\rho, \sigma, e \Downarrow^{E}_{pc} \sigma, v}$

$$\textsc{Lam}\quad \frac{}{\rho, \sigma, \lambda x.\ e \Downarrow^{E}_{pc} \sigma, \langle \lambda x.\ e, \rho \rangle}
\qquad
\textsc{Read}\quad \frac{\rho, \sigma, e \Downarrow^{E}_{pc} \sigma', v \qquad v' = \text{read}(pc, \sigma', v)}{\rho, \sigma, !\, e \Downarrow^{E}_{pc} \sigma', v'}$$

$$\textsc{Facet}\quad \frac{\rho, \sigma, e_1 \Downarrow^{E}_{pc} \sigma', \ell \qquad \rho, \sigma', e_2 \Downarrow^{E}_{pc} \sigma'', v_1 \qquad \rho, \sigma'', e_3 \Downarrow^{E}_{pc} \sigma''', v_2}{\rho, \sigma, \langle e_1\ ?\ e_2 \diamond e_3 \rangle \Downarrow^{E}_{pc} \sigma''', \langle\!\langle \ell\ ?\ v_1 \diamond v_2 \rangle\!\rangle}$$

$$\textsc{App}\quad \frac{\rho, \sigma, e_1 \Downarrow^{E}_{pc} \sigma', v_1 \qquad \rho, \sigma', e_2 \Downarrow^{E}_{pc} \sigma'', v_2 \qquad \sigma'', v_1(v_2) \Downarrow^{A}_{pc} \sigma''', v}{\rho, \sigma, e_1(e_2) \Downarrow^{E}_{pc} \sigma''', v}$$

$$\textsc{Write}\quad \frac{\rho, \sigma, e_1 \Downarrow^{E}_{pc} \sigma', v_1 \qquad \rho, \sigma', e_2 \Downarrow^{E}_{pc} \sigma'', v_2 \qquad \sigma''' = \text{write}(pc, \sigma'', v_1, v_2)}{\rho, \sigma, e_1 \leftarrow e_2 \Downarrow^{E}_{pc} \sigma''', ()}$$

$$\textsc{Obs}\quad \frac{\begin{array}{c}\rho, \sigma, e_1 \Downarrow^{E}_{pc} \sigma', \ell \qquad \rho, \sigma', e_2 \Downarrow^{E}_{pc} \sigma'', v_2 \\ \rho, \sigma'', e_3 \Downarrow^{E}_{pc} \sigma''', v_3 \qquad \sigma''', \sigma'''(\ell)(v_2) \Downarrow^{A}_{pc} \sigma'''', b\end{array}}{\rho, \sigma, \mathtt{obs}[e_1 @ e_2](e_3) \Downarrow^{E}_{pc} \sigma'''', \text{obs}(\ell, b, v_3)}$$

(Selected rules only...) $\boxed{\sigma, v(v) \Downarrow^{A}_{pc} \sigma, v}$

$$\textsc{AppBase}\quad \frac{\rho[x \mapsto v], \sigma, e \Downarrow^{E}_{pc} \sigma', v'}{\sigma, \langle \lambda x.\ e, \rho \rangle(v) \Downarrow^{A}_{pc} \sigma', v'}
\qquad
\textsc{AppStar}\quad \frac{}{\sigma, \star(v) \Downarrow^{A}_{pc} \sigma, \star}$$

$$\textsc{AppFacetSplit}\quad \frac{\begin{array}{c}\{+\ell, -\ell\} \cap pc = \varnothing \\ \sigma, v_1(v_3) \Downarrow^{A}_{pc \cup \{+\ell\}} \sigma', v_1' \qquad \sigma', v_2(v_3) \Downarrow^{A}_{pc \cup \{-\ell\}} \sigma'', v_2'\end{array}}{\sigma, \langle \ell\ ?\ v_1 \diamond v_2 \rangle(v_3) \Downarrow^{A}_{pc} \sigma'', \langle\!\langle \ell\ ?\ v_1' \diamond v_2' \rangle\!\rangle}$$

$$\boxed{\text{obs} \in \text{label} \times \text{bool} \times \text{val} \rightharpoonup \text{val}}$$

$$\begin{array}{rcl}
\text{obs}(\ell, b, bv) & \triangleq & bv \\
\text{obs}(\ell, \mathtt{true}, \langle \ell\ ?\ v_1 \diamond v_2 \rangle) & \triangleq & v_1 \\
\text{obs}(\ell, \mathtt{false}, \langle \ell\ ?\ v_1 \diamond v_2 \rangle) & \triangleq & v_2 \\
\text{obs}(\ell, b, \langle \ell'\ ?\ v_1 \diamond v_2 \rangle) & \triangleq & \langle \ell'\ ?\ \text{obs}(\ell, b, v_1) \diamond \text{obs}(\ell, b, v_2) \rangle \ \textit{where}\ \ \ell \neq \ell'
\end{array}$$

Fig. 2. Concrete Big-step Semantics (Selected Rules)

The FACET rule creates a faceted value by calling out to the $\langle\!\langle \cdot\ ?\ \cdot \diamond \cdot \rangle\!\rangle$ meta-operator, which *canonicalizes* a facet. Facet canonicalization ensures that all facets exist in a normal form, and prevents the creation of facets such as $\langle \ell\ ?\ \langle \ell\ ?\ a \diamond b \rangle \diamond \langle \ell \diamond c \diamond d \rangle \rangle$, collapsing this instead to $\langle \ell\ ?\ a \diamond d \rangle$. Sets of labels can also be given where $\langle\!\langle \varnothing\ ?\ v_1 \diamond v_2 \rangle\!\rangle = v_1$ and $\langle\!\langle \{k\} \cup pc\ ?\ v_1 \diamond v_2 \rangle\!\rangle = \langle\!\langle k\ ?\ \langle\!\langle pc\ ?\ v_1 \diamond v_2 \rangle\!\rangle \diamond v_2 \rangle\!\rangle$. This technique was first used by Austin et al. [3] to optimize the runtime of faceted programs, since the semantics is otherwise doing redundant work. We omit the definition of canonicalization due to space: the interested reader can refer to Figure 6 in [3].

Labels are created with the LABEL rule. This rule extends the store with a new closure, and binds it to a fresh label address ℓ, returning ℓ. Because the label may be produced under a non-empty program counter, the label address is faceted under the current pc, with a default value of $\star$. Unless the label escapes its enclosing context, it is essentially an unfaceted value, as the subsequent OBS rule will unfacet ℓ using the current pc when checking the policy associated with the label.

Labels in our semantics are store-allocated, rather than lexically scoped. This is important to retain the security of program values. To understand why, consider the example on the right,

```
let l = label[x](false) in
let fv = ⟨l ? 100 ⋄ 200⟩ in
let l = label[x](x) in
obs[l@true](fv)
```

which rebinds the label ℓ. In our semantics, this will result in the faceted value $\langle \ell^1\ ?\ 100 \diamond 200 \rangle$, not 200—ℓ^1 is the label address generated by the first use of the LABEL rule (on line 1). If the semantics for label introduction rebound the current l from the lexical environment, instead of dynamically generating a fresh one and returning it as a first-class address, the program would be able to circumvent the label originally associated with the facet by simply rebinding to a more permissive policy (e.g., $\lambda x.\ true$).

The OBS rule evaluates syntax $\mathtt{obs}[e_1 @ e_2](e_3)$, which introspects on a faceted value e_3. It first evaluates e_1 to a label ℓ, e_2 to a value v_2, and e_3 to a possibly-faceted value v_3, and calls out to the meta-operator $\text{obs}(\ell, b, v_3)$. This meta-operator performs the observation, returning the base value in the case its argument v_3 is a base value, and the appropriate branch (based on b) if v_3 is a facet whose label is ℓ. Otherwise (as the facet being observed may be farther down the tree), obs recurs to both its branches, rebuilding facets upon its return.

Our semantics allows references, reads, and writes via the REF, READ, and WRITE rules respectively. The REF rule creates a new reference cell in the store and initializes it with the result of the expression e. Crucially, REF must remember the current pc upon creating a faceted value. For example, consider the evaluation of $(\lambda x.\ \mathtt{ref}(x))(\langle \ell\ ?\ 1 \diamond 0 \rangle)$. During the positive branch of $\langle \ell\ ?\ 1 \diamond 0 \rangle$, we have that $pc = \{+\ell\}$, via the APPFACETSPLIT and APPBASE rules. Under this pc, the REF rule creates a reference to $\langle \ell\ ?\ 1 \diamond \star \rangle$, as simply returning a reference to 1 would strip away the label ℓ and would permit the exfiltration of sensitive data.

The READ and WRITE rules are similar, remembering that if they modify the store they must do so in a way that respects pc. READ uses the metafunction read, which takes pc as an argument and uses it to return the correct branch of a faceted reference cell. For example, $\text{read}(\{+\ell\}, \sigma, \langle \ell\ ?\ 1 \diamond \star \rangle)$ returns 1 and $\text{read}(\{-l\}, \sigma, \langle \ell\ ?\ 1 \diamond \star \rangle)$ returns $\star$. The WRITE rule is similar, using the value presently in the cell as the default value in the case of facet construction.

The Projection Property and Noninterference. Austin et al. [3], [4] demonstrate how faceted execution simulates multiple concrete runs, one for each combination of branches in $\wp(\text{branch})$. This is done via a *projection property*. Projection interprets every $q \in \wp(\mathit{branch})$ as projection of $\langle \ell\ ?\ v^{+} \diamond v^{-} \rangle$ to v^{+} if $+\ell \in q$ and to v^{-} if $-\ell \in q$. This is extended to environments and stores in the expected way. We say that two sets of branches pc and q are *consistent* when they do not contradict on any labels, i.e., $\neg\exists \ell.(+l \in pc \wedge -l \in q) \vee (-l \in pc \wedge +l \in q)$.

Theorem III.1 (Projection Theorem)**.** *Suppose* $\rho, \rho, e \Downarrow^{E}_{pc} \sigma', v$*. Then for any* $q \in \wp(\mathit{branch})$ *such that* pc *and* q *are consistent,* $q(\rho), q(\sigma), q(e) \Downarrow^{E}_{pc \setminus q} q(\sigma'), q(v)$.

As our semantics is largely similar to that in Austin et al. [4] (which updates the projection theorem to include support for first-class labels), we elide the proof of the projection theorem. The projection theorem can be used to immediately prove termination-insensitive noninterference, as shown in [3], [4].

Small-Step Semantics of Faceted Execution. As a first step towards abstraction, we reformulate our big-step semantics in the small-step style using an abstract machine. We

$$
\begin{array}{lll}
a \in \text{atom} ::= c \mid x \mid \lambda x.\ e & \kappa \in \text{context} \triangleq \text{frame}^* & \varsigma \in \text{config} ::= E\langle e,\rho,pc,\sigma,\kappa\rangle \quad \textit{eval} \\
e \in \text{exp} ::= a \mid \mathtt{ref}(a) \mid {!a} \mid a \leftarrow a \mid a(a) & fr \in \text{frame} ::= \langle \ell\ ?\ \square \diamond E\langle e,\rho,pc\rangle\rangle \mid \langle \ell\ ?\ \square \diamond A\langle v,v,pc\rangle\rangle & \quad\mid A\langle v,v,pc,\sigma,\kappa\rangle \quad \textit{apply} \\
\quad\mid \mathtt{label}[x](e) \mid \langle a\ ?\ e \diamond e\rangle \mid \mathtt{obs}[a@a](a) & \quad\mid \langle \ell\ ?\ v \diamond \square\rangle \mid O\langle \ell,\square,v\rangle \mid \text{HALT} & \quad\mid T\langle v,\sigma,\kappa\rangle \quad \textit{return}
\end{array}
$$

$$
\mathcal{A}[\![\cdot]\!] : \text{atom} \times \text{env} \rightharpoonup \text{value} \qquad \mathcal{A}[\![c]\!](\rho) \triangleq c \qquad \mathcal{A}[\![x]\!](\rho) \triangleq \rho(x) \qquad \mathcal{A}[\![\lambda x.\ e]\!](\rho) \triangleq \langle \lambda x.\ e,\rho\rangle
$$

$$\boxed{\varsigma \rightsquigarrow \varsigma}$$

$$
\begin{array}{rcll}
E\langle a,\rho,pc,\sigma,\kappa\rangle & \rightsquigarrow & T\langle v,\sigma,\kappa\rangle & \textit{where } v = \mathcal{A}[\![a]\!](\rho) \\
E\langle \mathtt{ref}(a),\rho,pc,\sigma,\kappa\rangle & \rightsquigarrow & T\langle \alpha,pc,\sigma[\alpha\mapsto v'],\kappa\rangle & \textit{where } v = \mathcal{A}[\![a]\!](\rho) \quad v' = \langle\langle pc\ ?\ v \diamond \star\rangle\rangle \quad \alpha \notin \text{dom}(\sigma) \\
E\langle !\,a,\rho,pc,\sigma,\kappa\rangle & \rightsquigarrow & T\langle \text{read}(pc,\sigma,v),\sigma,\kappa\rangle & \textit{where } v = \mathcal{A}[\![a]\!](\rho) \\
E\langle a_1 \leftarrow a_2,\rho,pc,\sigma,\kappa\rangle & \rightsquigarrow & T\langle (),\text{write}(pc,\sigma,v_1,v_2),\kappa\rangle & \textit{where } v_1 = \mathcal{A}[\![a_1]\!](\rho) \quad v_2 = \mathcal{A}[\![a_2]\!](\rho) \\
E\langle a(a),\rho,pc,\sigma,\kappa\rangle & \rightsquigarrow & A\langle v_1,v_2,pc,\sigma,\kappa\rangle & \textit{where } v_1 = \mathcal{A}[\![a_1]\!](\rho) \quad v_2 = \mathcal{A}[\![a_2]\!](\rho) \\
E\langle \mathtt{label}[x](e),\rho,pc,\sigma,\kappa\rangle & \rightsquigarrow & T\langle \ell,\sigma[\ell\mapsto v],\kappa\rangle & \textit{where } v = \langle\langle pc\ ?\ \langle \lambda x.\ e,\rho\rangle \diamond \star\rangle\rangle \quad \ell \notin \text{dom}(\sigma) \\
E\langle a\ ?\ e_1 \diamond e_2,\rho,pc,\sigma,\kappa\rangle & \rightsquigarrow & E\langle e_1,\rho,pc\cup\{+\ell\},\sigma,\kappa'\rangle & \textit{where } \ell = \mathcal{A}[\![a]\!](\rho) \quad \kappa' = \langle \ell\ ?\ \square \diamond E\langle e_2,\rho,pc\rangle\rangle :: \kappa \\
E\langle \mathtt{obs}[a_1@a_2](a_3),\rho,pc,\sigma,\kappa\rangle & \rightsquigarrow & A\langle \sigma(\ell),v_2,\rho,pc,\sigma,\kappa'\rangle & \textit{where } \ell = \mathcal{A}[\![a_1]\!](\rho) \quad v_2 = \mathcal{A}[\![a_2]\!](\rho) \quad v_3 = \mathcal{A}[\![a_3]\!](\rho) \quad \kappa' = O\langle \ell,\square,v_3\rangle :: \kappa \\
A\langle \langle \lambda x.\ e,\rho\rangle,v,pc,\sigma,\kappa\rangle & \rightsquigarrow & E\langle e,\rho[x\mapsto v],pc,\sigma,\kappa\rangle & \\
A\langle \star,v,pc,\sigma,\kappa\rangle & \rightsquigarrow & T\langle \star,\sigma,\kappa\rangle & \\
A\langle\langle \ell\ ?\ v_1 \diamond v_2\rangle,v,pc,\sigma,\kappa\rangle & \rightsquigarrow & A\langle v_1,v,pc,\sigma,\kappa\rangle & \textit{where } +\ell \in pc \\
A\langle\langle \ell\ ?\ v_1 \diamond v_2\rangle,v,pc,\sigma,\kappa\rangle & \rightsquigarrow & A\langle v_2,v,pc,\sigma,\kappa\rangle & \textit{where } -\ell \in pc \\
A\langle\langle \ell\ ?\ v_1 \diamond v_2\rangle,v,pc,\sigma,\kappa\rangle & \rightsquigarrow & A\langle v_1,v,pc,\sigma,\kappa'\rangle & \textit{where } \{+\ell,-\ell\}\cap pc = \varnothing \quad pc' = pc\cup\{+\ell\} \quad \kappa' = \langle \ell\ ?\ \square \diamond A\langle v_2,v,pc\rangle\rangle :: \kappa
\end{array}
$$

$$
\begin{array}{rcl}
T\langle v,\sigma,\langle \ell\ ?\ \square \diamond E\langle e,\rho,pc\rangle\rangle :: \kappa\rangle & \rightsquigarrow & E\langle e,\rho,pc\cup\{-\ell\},\sigma,\langle \ell\ ?\ v \diamond \square\rangle :: \kappa\rangle \\
T\langle v,\sigma,\langle \ell\ ?\ \square \diamond A\langle v_1,v_2,pc\rangle\rangle :: \kappa\rangle & \rightsquigarrow & A\langle v_1,v_2,pc\cup\{-\ell\},\sigma,\langle \ell\ ?\ v \diamond \square\rangle :: \kappa\rangle \\
T\langle v_1,\sigma,\langle \ell\ ?\ v_2 \diamond \square\rangle :: \kappa\rangle & \rightsquigarrow & T\langle \langle\!\langle \ell\ ?\ v_1 \diamond v_2\rangle\!\rangle,\sigma,\kappa\rangle \\
T\langle b,\sigma,O\langle \ell,\square,v\rangle :: \kappa\rangle & \rightsquigarrow & T\langle \text{obs}(\ell,b,v),\sigma,\kappa\rangle
\end{array}
$$

Fig. 3. Concrete Small-step Syntax and Semantics

assume that expressions in our language have been converted to A-Normal Form [16], shown in the top of Figure 3. Our atomic expressions include constants, variables, and lambdas, which are evaluated using $\mathcal{A}[\![\cdot]\!]$.

The top of Figure 3 also shows the configurations of our abstract machine. Configurations include environments (ρ), stores (σ), and program counters (pc)—all with the same structure as in Figure 2. Additionally, configurations include stacks, which are lists of frames.

The E and A configurations in our small-step semantics correspond to the reduction relations $\Downarrow^{E}_{pc}$ and $\Downarrow^{A}_{pc}$ in Figure 2 respectively. Starting from evaluation of both E and A, computation terminates with a value in the T configuration, which inspects the continuation and handles it appropriately.

The small-step semantics are shown in Figure 3. Stack frames track work left to be done once reaching a value. For example, in the AppFacetSplit rule, the reduction $\Downarrow^{A}_{pc}$ first evaluates the positive branch of the facet using the $\Downarrow^{E}_{pc\cup\{l\}}$ reduction, before then evaluating the negative branch. This corresponds to the rule for $A\langle\langle l\ ?\ v_1 \diamond v_2\rangle ...\rangle$ in the small-step semantics, which first performs the application of v_1, extending pc with $+\ell$. However, this rule uses the $l\ ?\ \square \diamond A\langle v_2,v,pc\rangle$ frame to remember to apply v_2 before forming a result using v, and tracking l to ensure that $\{-l\}$ is added to pc.

Expression evaluation occurs within the E configuration, which defers to the other configurations when it encounters possibly-faceted values. The evaluation of atomic expressions, reference creation, label creation, reads, and writes are all analogous to the big-step semantics, and immediately produce a T state. Application defers to A, which applies the argument v_2 to the possibly-faceted function v_1. The A configuration reduces v_1 to a base value before finally applying it, building continuations along the way to explore negative branches.

Facet creation defers to E to evaluate the positive branch of the facet while extending pc with $+\ell$, remembering to go back and evaluate the negative branch using the $\langle \ell\ ?\ \square \diamond E\langle e_2,\rho,pc\rangle\rangle$ frame (which must remember ℓ and pc so that e_2 can be run with $\{-\ell\}\cup pc$). Observation first evaluates each of the label, parameter, and value to observe to values. It then applies $\sigma(\ell)$ (as ℓ is store-allocated) to the parameter. This application must result in a boolean, and when it does so, the $O\langle \ell,\square,v_3\rangle$ frame will be indicate that an observation should be performed on v_3, reducing the ℓ facet in v_3 to its positive or negative branch.

As previously mentioned, the A configuration performs applications. The base case defers to the E rule using the closure's body and extending the environment for the binding. In the case that v_1 is a facet, the A configuration will recursively pull apart facets, via subsequent A forms, and remembers the negative branch in a continuation. As in the big-step semantics, if a facet with label ℓ is applied and either $+\ell \in pc$ or $-\ell \in pc$, the appropriate branch of the facet is taken to avoid redundant splitting.

The T rule decides what to do with a value based on the last frame in the stack. The $\langle \ell\ ?\ \square \diamond E\langle e,\rho,pc\rangle\rangle$ frame explores the negative branch e of a facet during facet creation, and pushes the $\langle \ell\ ?\ v \diamond \square\rangle$ frame onto the stack. This frame remembers to create a facet from ℓ, v, and the value in the value position of the T frame. As mentioned previously, $\ell\ ?\ v \diamond A\langle v_1,v_2,pc\rangle$ remembers to jump back to evaluate the application of a negative branch, remembering to create the facet upon completion. The $O\langle \ell,\square,v\rangle$ frame performs an observation, by calling out to the $\text{obs}(\ell,b,v_3)$ meta-operator.

We conclude this section by the statement of a theorem that stipulates our small-step semantics simulates our big-step semantics. The full statement of this theorem and its corresponding proof may be found in a companion tech report [30]. Our theorem, small-step simulation, establishes a mapping from each evaluation in the big-step semantics to a corresponding sequence of steps in our small-step semantics (nothing is said about the other direction). Given this theorem, our big-step semantics is approximated by our small-step semantics, which is in turn approximated by the abstract interpretation developed in section IV. Taken together, this gives us soundness for safety properties.

Theorem III.2 (Small-step Simulation)**.** *The small-step semantics simulates the big-step semantics, that is, the following are mutually true:*

1) *For all ρ, σ, e, σ' and v.*
   If: $\rho, \sigma, e \Downarrow^{E}_{pc} \sigma', v$
   Then for all: κ
   $E\langle e, \rho, pc, \sigma, \kappa\rangle \rightsquigarrow^{*} T\langle v, \sigma', \kappa\rangle$
2) *For all: σ, v_1, v_2, σ' and v:*
   If: $\sigma, v_1(v_2) \Downarrow^{P}_{pc} \sigma', v$
   Then for all: κ
   $A\langle v_1, v_2, pc, \sigma, \kappa\rangle \rightsquigarrow^{*} T\langle v, \sigma', \kappa\rangle$

Proof. We omit a detailed proof here and defer its details to a tech report. The proof proceeds by mutual induction on each of the big-step derivations along with a lemma establishing simulation for atomic values. □

IV. An Abstract Semantics for Faceted Execution

We develop a static analysis of this faceted execution semantics using the framework of *abstracting abstract machines* (AAM): a general approach to developing *abstract interpretations* of abstract-machine semantics [44]. Abstract interpretation is a well-explored set of tools and techniques for approximating the fixed points of a semantic function over an infinite lattice either by structurally finitizing the lattice (formalizing a Galois connection between the infinite and finite lattice) or by accelerating convergence to a fixed point (using a widening operator), or both [11], [12]. The heart of the AAM approach is the use of small-step transitions, preparatory store-allocating transformations shown in section IV-C that break direct recursion in the state space, and the systematic derivation of Galois connections for higher-order machine components (such as abstract stores) by composing Galois connections for lower-order components (such as abstract addresses and abstract first-order values).

The problem with structurally abstracting a traditional operational interpreter for the λ-calculus lies in the recursive nature of closures: closures include environments which can include closures, and so forth to an arbitrary depth. It should be no surprise as this feature is precisely what makes universal computation through higher-order recursion possible with only variable reference, lambda, and application. All static analyses, however, must voluntarily concede precision in order to achieve guarantees of computability (and bounded complexity).

The AAM methodology is to prepare a small-step machine for abstraction by first store-allocating all values (not just explicit ref cells). This means that binding environments map variables to store/heap addresses, and stores map these addresses to values (e.g., closures, ref addresses, base values). At this point, the address set, along with the domains for base values, can be finitized: abstracted to a finite set of *abstract addresses* at which an approximation of multiple concrete values become conflated during analysis. This imprecision in the store, where a single abstract address can map to many possible concrete values, goes hand-in-hand with nondeterminism in the small-step transition relation (e.g., multiple closures can be bound to `f` at a call site `f(…)`).

$$
\begin{array}{l}
\mathrm{setup} \triangleq \begin{array}[t]{l}
\mathrm{mk\text{-}pol} \;\; = \lambda a.\ \mathtt{label}^{\widehat{\ell}}[x](x \overset{?}{=} a) \\
\mathrm{alice\text{-}pol} = \mathrm{mk\text{-}pol}(\textsc{alice}) \;;\ \mathrm{alice\text{-}bet} = \langle \mathrm{alice\text{-}pol} \mathbin{?} \textsc{tails} \diamond \star\rangle \\
\mathrm{bob\text{-}pol} \;\; = \mathrm{mk\text{-}pol}(\textsc{bob}) \;\;\; ;\ \mathrm{bob\text{-}bet} \;\; = \langle \mathrm{bob\text{-}pol} \mathbin{?} \textsc{heads} \diamond \star\rangle
\end{array}
\\[1em]
e_1 \triangleq \mathtt{let}\ \mathrm{setup}\ \mathtt{in}\ \mathtt{obs}[\mathrm{bob\text{-}pol}@\textsc{bob}](\mathrm{alice\text{-}bet}) \\
e_2 \triangleq \mathtt{let}\ \mathrm{setup}\ \mathtt{in}\ \langle \mathrm{bob\text{-}pol} \mathbin{?} \mathrm{alice\text{-}bet} \diamond \star\rangle
\end{array}
$$

$$
\begin{array}{lllllll}
\text{(C1)} & e_1 & \rightsquigarrow^{*} & \mathtt{obs}[\ell_{\mathrm{B}}@\textsc{bob}](\langle \ell_{\mathrm{A}} \mathbin{?} \textsc{tails} \diamond \star\rangle) & \rightsquigarrow^{*} & \langle \ell_{\mathrm{A}} \mathbin{?} \textsc{tails} \diamond \star\rangle \\
\text{(U1)} & \widehat{e} = e_1 & \widehat{\rightsquigarrow}^{*} & \widehat{e} = \mathtt{obs}[\widehat{\ell}@\textsc{bob}](\langle \widehat{\ell} \mathbin{?} \textsc{tails} \diamond \star\rangle) & \widehat{\rightsquigarrow}^{*} & \widehat{e} = \{\textsc{tails}, \star\} \\
 & \widehat{\sigma} = \varnothing & & \widehat{\sigma} = \{\widehat{\ell} \mapsto \{\langle \lambda x.\ x \overset{?}{=} a, \{a \mapsto \textsc{alice}\}\rangle & & \widehat{\sigma} = \ldots \textit{unchanged} \\
 & & & \qquad ,\langle \lambda x.\ x \overset{?}{=} a, \{a \mapsto \textsc{bob}\}\rangle\}\} & & \\[1em]
\text{(C2)} & e_2 & \rightsquigarrow^{*} & \langle\!\langle \ell_{\mathrm{B}} \mathbin{?} \langle \ell_{\mathrm{A}} \mathbin{?} \textsc{tails} \diamond \star\rangle \diamond \star\rangle\!\rangle & \rightsquigarrow^{*} & \langle \ell_{\mathrm{A}} \mathbin{?} \langle \ell_{\mathrm{B}} \mathbin{?} \textsc{tails} \diamond \star\rangle \\
 & & & & & \quad \diamond \langle \ell_{\mathrm{B}} \mathbin{?} \star \diamond \star\rangle\rangle \\
\text{(U2)} & \widehat{e} = e_2 & \widehat{\rightsquigarrow}^{*} & \widehat{e} = \langle\!\langle \widehat{\ell} \mathbin{?} \langle \widehat{\ell} \mathbin{?} \textsc{tails} \diamond \star\rangle \diamond \star\rangle\!\rangle & \widehat{\rightsquigarrow}^{*} & \widehat{e} = \langle \widehat{\ell} \mathbin{?} \textsc{tails} \diamond \star\rangle \\
 & \widehat{\sigma} = \varnothing & & \widehat{\sigma} = \{\widehat{\ell} \mapsto \{\langle \lambda x.\ x \overset{?}{=} a, \{a \mapsto \textsc{alice}\}\rangle & & \widehat{\sigma} = \ldots \textit{unchanged} \\
 & & & \qquad ,\langle \lambda x.\ x \overset{?}{=} a, \{a \mapsto \textsc{bob}\}\rangle\}\} & &
\end{array}
$$

Fig. 4. An example of concrete and abstract faceted execution. (C1) and (C2) are concrete executions, and (U1) and (U2) are unsound candidate abstract executions.

Finitizing the store is fundamentally how AAM cedes precision in order to gain a finite, computable, and sound analysis, and the selection of abstract addresses in this process has also been shown to be a exceptionally broad parameter with which to tune analysis polyvariance (e.g., context sensitivity) [19]. Once a finite domain for abstract addresses and abstract base values has been selected, these abstractions (formally Galois connections) may be systematically lifted to abstractions for stores, environments, and machine states [31].

A. Challenges Abstracting Faceted Values

As much of the AAM process for a language like ours is standard, the core issue is our representation for abstract values—especially abstract faceted values. The representation of abstract base values is relatively straightforward: we abstract constants with a flat lattice (e.g., $\bot \sqsubset 1 \sqsubset \top$, but $1 \not\sqsubseteq 2$ and $2 \not\sqsubseteq 1$). We abstract closures to a powerset of abstract closures (that is, a syntactic lambda paired with an abstract environment mapping variables to abstract addresses). We abstract addresses for ref cells to a powerset of abstract addresses. Finally, we abstract concrete addresses according to our desired polyvariance—in the simplest case, this means assigning one abstract address to each syntactic variable (called *monovariance* or *context insensitivity*).

We are now faced with our choice of a domain for abstract faceted values. As a first attempt, we might structurally abstract facets via their components. Unfortunately, this approach is necessarily unsound if we want to retain any precision in our abstraction of facet-structure (i.e., the existence of labels and differentiation of branches).

To understand the issue, consider Figure 4. The top of the figure shows two examples, e_1 and e_2, sharing a common prologue setup. The setting for our examples is a guessing game in which Alice and Bob both place bets as to the result of a coin flip (heads or tails). Both players will use facets to hide the information from each other. The function mk-pol takes an argument a and returns a label (capturing a in the closure encoding the label's policy) that will reveal its positive branch only to a user with key a. The label alice-pol uses mk-pol to generate a label specialized to key ALICE, and Alice's bet is faceted on this label. Similarly, Bob creates a label bob-pol that reveals its positive branch to himself, and facets his bid for $\textsc{heads}$. The example e_1 attempts to observe Alice's facet via Bob's label, which should make no change to the faceted value because ℓ_B is not present in it. Second, e_2 forms a facet of Alice's bet under Bob's label, which our concrete semantics represents as a faceted value encoding a decision tree with two levels: one for Alice's label, and one for Bob's. Only after successively observing on both of these labels will the bet become visible.

Directly under the example, we show a concrete execution (C1) of e_1 via the small-step semantics (abbreviated somewhat for presentation). Starting with the initial state for e_1, control will eventually step to the `obs` form on the last line, in a configuration where a label ℓ_A was dynamically generated for alice-pol by the rule for `label` in Figure 3. In our example, the labels alice-pol and bob-pol are distinct addresses at runtime, and so the observation on the last line is simply a no-op. The rule for facet observation (see obs in figure 2) splits in the case that the label being observed is different than the one guarding the facet, and both branches ($\textsc{tails}$ and $\star$) are base values—which are simply returned when observed. The final result is the same facet that was originally being observed: $\langle\, \ell_A \;?\; \textsc{tails} \diamond \star \,\rangle$. Similarly, (C2) shows a concrete execution of e_2. As control steps to the facet creation form, alice-bet will become bound to a faceted value representing Alice's bet. Facet creation in the final expression will then create a facet with Bob's label, guarding alice-bet, and canonicalization will reorder the labels. The result is a tree of facets placing the result $\textsc{tails}$ under the branch $\{+\ell_B, +\ell_A\}$. Canonicalization ensures that labels appear in some sorted order, so Alice's label will appear higher than Bob's in the tree.

Below each concrete execution we demonstrate a trace showing the behavior using the proposed abstract interpretation of the corresponding example using our naïve structural abstraction. To ensure termination, the abstract semantics conflates certain values, and in particular must necessarily finitize the set of labels. A monovariant allocator will generate an abstract address for each label unique to its program point. In our concrete semantics, the two distinct calls to mk-pol generate two distinct labels ℓ_A and ℓ_B. By contrast, a monovariant semantics generates a single *abstract* label $\widehat{\ell}$, based on the program point in mk-pol at which the label is created.

As we execute the prologue setup in an abstract setting, the first call to mk-pol returns the label $\widehat{\ell}$. In the store, this label maps to a policy closed over the environment $\{a \mapsto \textsc{alice}\}$. Upon executing the third line of the example, mk-pol is called again, extending $\widehat{\ell}$ so that its policy also closes over $\{a \mapsto \textsc{bob}\}$.

At this point, the abstract semantics cannot differentiate between what would have been (in the concrete semantics) ℓ_A and ℓ_B. Therefore, when control reaches the `obs` form in e_1, instead of splitting on ℓ_B (as would have been done in the concrete semantics), the abstract label for Bob's bet is now the same as the label on Alice's bet being observed. This could result in the facet protecting Alice's value to be removed, as shown in (U1), and yielding the abstract value $\{\widehat{\textsc{tails}}, \widehat{\star}\}$.

Two Broken Interpretations for Abstract Facets. There are two naïve ways we may interpret $\{\widehat{\textsc{tails}}, \widehat{\star}\}$. First, we might interpret the abstract value $\{\widehat{\textsc{tails}}, \widehat{\star}\}$ as simply the set of concrete values $\{\textsc{tails}, \star\}$. In other words, we may take the view that an abstract base-value is *definitely not* faceted, and that an abstract facet is definitely not a base value. However, this concretization does not include the faceted layer generated by the concrete semantics, and thus incorrectly "proves" (unsoundly) that a base value results from the observation.

Alternatively, we might interpret all abstract values as possibly faceted or not. If we were to take this approach, we would achieve a sound result, concretizing $\{\widehat{\textsc{tails}}, \widehat{\star}\}$ as $\langle\, \ell \;?\; \{\textsc{tails}, \star\} \diamond \{\textsc{tails}, \star\} \,\rangle \sqcup \{\textsc{tails}, \star\}$, (really, under *all* possible concrete ℓ) which includes the result in the concrete run. However, if we interpret abstract values in this manner, we lose *all* precision for facet structure (retaining only information for base values), as we must interpret any faceted value $\langle\, \ell \;?\; \widehat{v}^{+} \diamond \widehat{v}^{-} \,\rangle$ as $\widehat{v}^{+} \sqcup \widehat{v}^{-}$ and vice-versa.

An abstract execution of e_2 using the proposed semantics is also an issue. In this case, as shown in (U2), the facet creation form now acts as a no-op since alice-pol and bob-pol share the abstract label $\widehat{\ell}$. Instead of creating a decision tree with two levels, the comparison proceeds to create a single facet guarded by an abstract label representing the disjunction of Alice's label *or* Bob's label. If we take the first approach in interpreting this abstract value, we would view it as definitely faceted on $\widehat{\ell}$, over the concretization of each branch (i.e., $\{\textsc{tails}, \star\}$), but would not know if it were actually faceted on ℓ_A, ℓ_B, or both. If we interpret the concretization of abstract labels disjunctively, Bob's bet could appear visible to Alice (or the reverse); if we interpret the concretization conjunctively, Bob's bet could appear opaque to himself. In either case our analysis is unsound. On the other hand, the naïve, sound interpretation treats any abstract value (explicitly faceted or not) as being potentially faceted by arbitrary further labels (or not at all), which is useless for verifying security policies.

In fact, this situation is worse than it first appears: *any* aspect of our concrete semantics that compares labels can no longer be relied on (to be—to any degree—both sound and precise) in the abstract setting. For example, many the rules in Figure 3 decide whether or not to split based on whether a particular label is in pc. In our abstract semantics, this inclusion check can no longer make such determinations for abstract labels. Representing an abstract facet $\langle\, \widehat{\ell} \;?\; \widehat{v}^{+} \diamond \widehat{v}^{-} \,\rangle$ is sound as long as $\widehat{\ell}$ is not an abstraction of *multiple concrete labels*; once this happens however, the branches $\widehat{v}^{+}$ and $\widehat{v}^{-}$ can no longer be soundly kept distinct. This necessitates that the negative facet of one concrete label must be conflated with the positive facet of the other and vice-versa. We leverage this specific insight to develop a sound *and* precise solution to the *branch sensitivity problem* in Section IV-D.

B. Toward a Sound Abstract Domain for Facets

As a first step toward sound and precise abstract facets, we formulate a sound abstraction which is precise for the most essential aspect of facet structure: the abstract labels that are associated with a faceted value. A key is to observe that faceted values are a decision tree, and the problem of representing the abstract labels that *may*—and that *must*—facet the value, exists at each level of this tree. We construe this problem as orthogonal to branch sensitivity (keeping positive facets soundly distinct from negative facets) and present an abstract domain for precisely representing the abstract labels that may and must exist for a value.

For the moment, we give up on branch sensitivity and conflate all positive and negative facets. Instead, we represent faceted values as a label over the join of both branches, $\langle \widehat{\ell} \ ? \ \widehat{\mathrm{v}}^{+} \sqcup \widehat{\mathrm{v}}^{-} \rangle$. Although this abstraction of facets does not allow us to distinguish branches in faceted values, it still provides us with a sensible result: is the value reaching a program point faceted and, if so, what base values are being faceted. For example, we can envision using this analysis in an optimizing compiler for faceted programs that drops the dynamic machinery for performing faceted application in the case that only base values reach a particular application form.

In defining an abstract domain, we must also define a join operator ($\sqcup$) for abstract facets. This leads to an immediate question: how should base values be joined into abstract facets—in other words, if we represent abstract facets as $\langle \widehat{\ell} \ ? \ \widehat{\mathrm{v}} \rangle$, how should we perform the join $\widehat{\mathrm{bv}} \sqcup \langle \widehat{\ell} \ ? \ \widehat{\mathrm{v}} \rangle$? It is natural to simply define $\sqcup$ such that $\widehat{\mathrm{bv}} \sqcup \langle \widehat{\ell} \ ? \ \widehat{\mathrm{v}} \rangle$ is equal to $\langle \widehat{\ell} \ ? \ \widehat{\mathrm{v}} \sqcup \widehat{\mathrm{bv}} \rangle$ or equal to $\widehat{\mathrm{v}} \sqcup \widehat{\mathrm{bv}}$, but, as we've discussed, is unsound if we want precision for facet structure. Consider the following example (using monovariant allocation) which leads to such a conflation:

```
let id(x) = x
let v₁   = id(⟨ ℓ ? 1 ⋄ 2 ⟩)
let v₂   = id(3)
obs[ℓ @ true] v₂
```

A concrete execution of this program will result in an error: the obs form does not work for faceted values. Because id is called twice, the abstract value for x will be the join of abstractions $\langle \ell \ ? \ 1 \sqcup 2 \rangle$ and 3. If we define $\sqcup$ to distribute 3 inside of the facet, the abstraction of x will be a faceted value $\langle \ell \ ? \ \top \rangle$ (as we use a flat lattice to abstract constants, the join of two different constants is $\top$). Instead, we generalize our domain for abstract values to be product of base values and faceted value (see $\widehat{\mathrm{val}}$ in figure 5). This allows us to directly represent values which must be a base value ($\widehat{\mathrm{bv}} \times \bot$), must be faceted ($\bot \times \widehat{\mathrm{m}}$), or could be either ($\widehat{\mathrm{bv}} \times \widehat{\mathrm{m}}$). With this encoding, the abstraction for x in the above example can be represented: $3 \times \langle 1 \ ? \ \top \rangle$.

There is one last subtlety in defining $\sqcup$, which relates how to merge facets with different labels. We might be tempted to merge faceted values using canonicalization. To see why this is incorrect, consider the following example:

```
let f(x) = obs[bob-label@bob](x)
f(⟨ bob-label ? X ⋄ Y ⟩)
f(⟨ alice-label ? X ⋄ Y ⟩)
```

In a concrete execution, the first call to `f` returns a base value, while the second returns a faceted value. However, if we use a canonicalizing join, the abstract execution merges the two facets $\langle \widehat{\ell_B} \, ? \, X \diamond Y \rangle$ and $\langle \widehat{\ell_A} \, ? \, X \diamond Y \rangle$ to produce $\langle \widehat{\ell_A} \, ? \, \langle \widehat{\ell_B} \, ? \, X \diamond X \sqcup Y \rangle \diamond \langle \widehat{\ell_B} \, ? \, Y \sqcup X \diamond Y \rangle \rangle$. As a result, both calls to `f` produce a facet guarded by $\widehat{\ell_B}$, incorrectly "proving" that the first call to `f` returns a facet.

We could avoid both of these issues by identifying facets with base values, essentially interpreting $\langle \widehat{\ell} \ ? \ \widehat{\mathrm{v}}^{+} \diamond \widehat{\mathrm{v}}^{-} \rangle$ as $\widehat{\mathrm{v}}^{+} \sqcup \widehat{\mathrm{v}}^{-}$. However, we believe this would significantly hinder the usefulness of our analysis, as we could no longer use the analysis to tell us whether any given value was a facet or a base value representing the join of its branches.

A Branch-insensitive Abstraction for Facets. Our complete abstraction for branch-insensitive abstract facets is presented in the top right of Figure 5. For base values we assume a standard join, which can be tuned alongside the abstraction for base values to recover the desired precision. We also assume an injector $\lfloor \mathrm{bv} \rfloor$ for base values that takes concrete values and injects them into an abstract representation, $\widehat{\mathrm{bv}}$.

Abstract faceted values are represented by $\widehat{\mathrm{v}}$, and comprise a pair of a $\widehat{\mathrm{bv}} \in \widehat{\mathrm{base\text{-}val}}$ and a $\widehat{\mathrm{m}} \in \widehat{\mathrm{facet\text{-}map}}$—necessary to avoid the unsoundness of conflating facets and base values. We represent abstract facets with a partial map ($\widehat{\mathrm{facet\text{-}map}}$), as opposed to a pair of label and underlying value, to avoid the issues disjoining abstract faceted values with different abstract labels. This map generalizes a single facet (with a single collapsed branch) disjunctively to a set of labels and their associated collapsed branches. That is, $\langle \widehat{\ell} \ ? \ \widehat{\mathrm{v}} \rangle$ would be represented by $\{\widehat{\ell} \mapsto \widehat{\mathrm{v}}\}$ and $\langle \widehat{\ell_1} \ ? \ \widehat{\mathrm{v}_1} \rangle \sqcup \langle \widehat{\ell_2} \ ? \ \widehat{\mathrm{v}_2} \rangle$ by $\{(\widehat{\ell_1} \mapsto \widehat{\mathrm{v}_1}), (\widehat{\ell_2} \mapsto \widehat{\mathrm{v}_2})\}$.

To accommodate our abstract domain for branch-insensitive facets, we must make corresponding updates to the metafunctions that interact with facets. First, facet creation ($\langle\!\langle \cdot \ ? \ \cdot \rangle\!\rangle$)—which only takes one branch in our coarse abstract domain—must be updated to form abstract facets via maps.

Store read separately considers the addresses contained in the base-value component, the labels in $\widehat{pc}$, and the facets in the map. For each $\widehat{\ell} \in pc$, read unfacets $\widehat{\ell}$ from the facet map by projecting it, calling *read* again to push one level down in the facet map, and rebuilds the result as a facet. This may seem surprising at first, as our concrete semantics unfacets labels in pc. However, this is crucial to retain soundness, as we must remember that ℓ could concretize to an facet of arbitrary depth. Separately, read unwraps each label in the facet map to perform a read and rebuilds the results as a facet map. The rule for store write is similar, separately considering the set of base values and faceted values contained the abstract value. Last, obs joins both the projection of ℓ and a facet map containing its projection (along with the rest of the values in the facet map). That obs includes both the projection and a a facet of the projection is crucial to soundness, again because abstract depth does not predict concrete depth.

Finally, we define $\sqcup$ on abstract values. Join for abstract values distributes pointwise both for values and for facet maps (the domain of two facet maps is joined by set union); e.g.: $\langle \widehat{\mathrm{bv}_1}, \{\widehat{\ell_A} \mapsto \widehat{v_1}\} \rangle \sqcup \langle \widehat{\mathrm{bv}_2}, \{\widehat{\ell_B} \mapsto \widehat{v_2}\} \rangle = \langle \widehat{\mathrm{bv}_1} \sqcup \widehat{\mathrm{bv}_2}, \{\widehat{\ell_A} \mapsto \widehat{v_1}, \widehat{\ell_B} \mapsto \widehat{v_2}\} \rangle$ but $\langle \widehat{\mathrm{bv}_1}, \{\widehat{\ell_A} \mapsto \widehat{v_1}\} \rangle \sqcup \langle \widehat{bv_2}, \{\widehat{\ell_A} \mapsto \widehat{v_2}\} \rangle = \langle \widehat{\mathrm{bv}_1} \sqcup \widehat{\mathrm{bv}_2}, \{\widehat{\ell_A} \mapsto \widehat{v_1} \sqcup \widehat{v_2}\} \rangle$. We present a Galois Connection for our abstract domain in Section IV-E.

C. An Abstract Interpretation for Faceted Execution

We now present an imprecise but sound abstract interpretation for faceted execution. Our abstract semantics extends

$$
\begin{aligned}
\widehat{\alpha} &\in \widehat{\mathrm{vaddr}} \\
\widehat{\ell} &\in \widehat{\mathrm{label}} \\
\widehat{\kappa\alpha} &\in \widehat{\kappa\mathrm{addr}} \\
\widehat{r} &\in \widehat{\mathrm{closure}} ::= \langle \lambda x.\ e, \widehat{\rho} \rangle \\
\widehat{z} &\in \widehat{\mathrm{failure}} ::= \bot \mid \star \\
\widehat{l} &\in \widehat{\mathrm{literal}} ::= \bot \mid c \mid \top \\
\widehat{pc} &\in \widehat{\mathrm{PC}} \triangleq \wp(\widehat{\mathrm{label}})
\end{aligned}
$$

$$
\begin{aligned}
\widehat{bv} \in \widehat{\text{base-val}} &\triangleq \widehat{\mathrm{literal}} \times \wp(\widehat{\mathrm{addr}} \uplus \widehat{\mathrm{label}}) \\
&\quad \times \wp(\widehat{\mathrm{closure}}) \times \widehat{\mathrm{failure}} \\
\widehat{m} \in \widehat{\text{facet-map}} &\triangleq \widehat{\mathrm{label}} \rightharpoonup \widehat{\mathrm{val}} \\
\widehat{v} \in \widehat{\mathrm{val}} &\triangleq \widehat{\text{base-val}} \times \widehat{\text{facet-map}} \\
\widehat{\rho} \in \widehat{\mathrm{env}} &\triangleq \mathrm{var} \rightharpoonup \widehat{\mathrm{vaddr}} \uplus \widehat{\mathrm{label}} \\
\widehat{\sigma} \in \widehat{\mathrm{store}} &\triangleq (\widehat{\mathrm{vaddr}} \rightharpoonup \widehat{\mathrm{val}}) \uplus (\widehat{\mathrm{label}} \rightharpoonup \widehat{\mathrm{val}}) \\
&\quad \uplus (\widehat{\kappa\mathrm{addr}} \rightharpoonup \wp(\widehat{\mathrm{context}})) \\
\widehat{\kappa} \in \widehat{\mathrm{context}} &::= \widehat{fr} :: \widehat{\kappa\alpha}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{\varsigma} \in \widehat{\mathrm{config}} &::= E\langle \widehat{e}, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle && \textit{eval} \\
&\mid A\langle \widehat{v}, \widehat{v}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle && \textit{apply} \\
&\mid T\langle \widehat{v}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle && \textit{return} \\
\widehat{fr} \in \mathrm{frame} &::= \langle \widehat{pc} \ ?\ \square \diamond E\langle \widehat{e}, \widehat{\rho}, \widehat{pc} \rangle \rangle \\
&\mid \langle \widehat{pc} \ ?\ \widehat{v} \diamond \square \rangle \\
&\mid \langle \widehat{pc} : \square \rangle \\
&\mid O\langle \widehat{pc}, \square, \widehat{v} \rangle \\
&\mid \mathrm{HALT}
\end{aligned}
$$

$$
\textit{(Parameter)} \quad \widehat{\mathrm{alloc}} \in (\widehat{\mathrm{config}} \to \widehat{\mathrm{vaddr}}) \uplus (\widehat{\mathrm{config}} \to \widehat{\mathrm{label}}) \uplus (\widehat{\mathrm{config}} \to \widehat{\kappa\mathrm{addr}})
$$

$$
\begin{array}{lll}
\widehat{\mathcal{A}}[\![\cdot]\!] \in \mathrm{atom} \times \widehat{\mathrm{env}} \times \widehat{\mathrm{store}} \rightharpoonup \widehat{\mathrm{value}} & \cdot\sqcup\cdot \in \widehat{\mathrm{val}} \times \widehat{\mathrm{val}} \to \widehat{\mathrm{val}} & \langle\!\langle \cdot : \cdot \rangle\!\rangle \in \widehat{\mathrm{label}} \times \widehat{\mathrm{val}} \to \widehat{\mathrm{val}} \\
\widehat{\mathrm{read}} \in \widehat{\mathrm{PC}} \times \widehat{\mathrm{store}} \times \widehat{\mathrm{val}} \to \widehat{\mathrm{val}} & \widehat{\mathrm{write}} \in \widehat{\mathrm{PC}} \times \widehat{\mathrm{val}} \times \widehat{\mathrm{val}} \to \widehat{\mathrm{store}} & \widehat{\mathrm{obs}} \in \widehat{\mathrm{label}} \times \widehat{\mathrm{val}} \times \widehat{\mathrm{val}} \to \widehat{\mathrm{val}}
\end{array}
$$

$$
\widehat{\mathcal{A}}[\![c]\!](\widehat{\rho}, \widehat{\sigma}) \triangleq \lfloor c \rfloor \qquad \widehat{\mathcal{A}}[\![x]\!](\widehat{\rho}, \widehat{\sigma}) \triangleq \widehat{\sigma}(\widehat{\rho}(x)) \qquad \widehat{\mathcal{A}}[\![\lambda x.\ e]\!](\widehat{\rho}, \widehat{\sigma}) \triangleq \lfloor \langle \lambda x.\ e, \widehat{\rho} \rangle \rfloor
$$

$$
\langle \widehat{bv}_1, \widehat{m}_1 \rangle \sqcup \langle \widehat{bv}_2, \widehat{m}_2 \rangle \triangleq \langle \widehat{bv}_1 \sqcup \widehat{bv}_2, \widehat{m} \rangle \quad \textit{where} \quad \widehat{m} = \bigcup_{\widehat{\ell} \in \mathrm{dom}(\widehat{m}_1) \cup \mathrm{dom}(\widehat{m}_2)} \{ \widehat{\ell} \mapsto \widehat{m}_1(\widehat{\ell}) \sqcup \widehat{m}_2(\widehat{\ell}) \}
$$

$$
\langle\!\langle \widehat{\ell} : \langle \widehat{bv}, \widehat{m} \rangle \rangle\!\rangle \triangleq \langle \bot, \widehat{m}_1 \sqcup \widehat{m}_2 \sqcup \widehat{m}_3 \rangle \quad \textit{where} \quad \left\{ \begin{array}{ll} \widehat{m}_1 = \{ \widehat{\ell} \mapsto \langle \widehat{bv}, \varnothing \rangle \sqcup \widehat{m}(\widehat{\ell}) \} & \widehat{m}_2 = \bigcup_{\widehat{\ell} < \widehat{\ell'} \in \mathrm{dom}(\widehat{m})} \{ \widehat{\ell'} \mapsto \widehat{m}(\ell') \} \\ & \widehat{m}_3 = \bigcup_{\widehat{\ell} > \widehat{\ell'} \in \mathrm{dom}(\widehat{m})} \{ \widehat{\ell'} \mapsto \langle\!\langle \widehat{\ell} : \widehat{m}(\widehat{\ell'}) \rangle\!\rangle \} \end{array} \right.
$$

$$
\widehat{\mathrm{read}}(\widehat{pc}, \widehat{\sigma}, \langle \widehat{bv}, \widehat{m} \rangle) \triangleq \widehat{v}_1 \sqcup \widehat{v}'_1 \sqcup \widehat{v}''_1 \sqcup \widehat{v}_2 \sqcup \langle \widehat{bv}', \widehat{m}' \rangle \quad \textit{where} \quad \left\{ \begin{array}{ll} \widehat{v}_1 = \bigsqcup_{\lfloor \widehat{\alpha} \rfloor \sqsubseteq \widehat{bv}} \widehat{\sigma}(\widehat{\alpha}) & \widehat{v}_2 = \bigsqcup_{\widehat{\ell} \in \widehat{pc}} \widehat{\mathrm{read}}(\widehat{pc}, \widehat{\sigma}, \widehat{m}(\widehat{\ell})) \\ \widehat{v}'_1 = \bigsqcup_{\lfloor \widehat{\ell} \rfloor \sqsubseteq \widehat{bvm}} \widehat{\sigma}(\widehat{\ell}) & \widehat{m}' = \bigcup_{\widehat{\ell} \in \mathrm{dom}(\widehat{m})} \{ \widehat{\ell} \mapsto \widehat{\mathrm{read}}(\widehat{pc}, \widehat{\sigma}, \widehat{m}(\widehat{\ell})) \} \\ \widehat{v}''_1 = \bigsqcup_{\lfloor \widehat{\kappa\alpha} \rfloor \sqsubseteq \widehat{bv}} \widehat{\sigma}(\widehat{\kappa\alpha}) & \widehat{bv}' = \lfloor \star \rfloor \ \textit{if}\ \lfloor \star \rfloor \sqsubseteq \widehat{bv}\ ;\ \bot\ \textit{otherwise} \end{array} \right.
$$

$$
\widehat{\mathrm{write}}(\widehat{pc}, \langle \widehat{bv}, \widehat{m} \rangle, \widehat{v}) \triangleq \sigma_1 \sqcup \sigma_2 \quad \textit{where} \quad \left\{ \begin{array}{l} \widehat{\sigma}_1 = \bigsqcup_{\lfloor \widehat{\alpha} \rfloor \sqsubseteq \widehat{bv}} \{ \widehat{\alpha} \mapsto \langle\!\langle \widehat{pc} \ ?\ \widehat{v} \diamond \widehat{\sigma}(\widehat{\alpha}) \rangle\!\rangle \} \\ \widehat{\sigma}_2 = \bigsqcup_{\widehat{\ell} \in \mathrm{dom}(\widehat{m})} \widehat{\mathrm{write}}(\widehat{pc} \cup \{ \widehat{\ell} \}, \widehat{m}(\widehat{\ell}), \widehat{v}) \end{array} \right.
$$

$$
\widehat{\mathrm{obs}}(\widehat{\ell}, \widehat{v}, \langle \widehat{bv}, \widehat{m} \rangle) \triangleq \widehat{m}(\widehat{\ell}) \sqcup \langle \widehat{bv}, \widehat{m}_1 \sqcup \widehat{m}_2 \rangle \quad \textit{where} \quad \left\{ \begin{array}{l} \widehat{m}_1 = \{ \widehat{\ell} \mapsto \widehat{m}(\widehat{\ell}) \} \\ \widehat{m}_2 = \bigcup_{\widehat{\ell'} \in \mathrm{dom}(\widehat{m})} \{ \widehat{\ell'} \mapsto \widehat{\mathrm{obs}}(\widehat{\ell}, \widehat{v}, \widehat{m}(\widehat{\ell'})) \} \end{array} \right.
$$

$$
\boxed{\widehat{\varsigma} \rightsquigarrow \widehat{\varsigma}}
$$

$$
\begin{array}{lll}
E\langle a, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle \rightsquigarrow T\langle \widehat{v}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle & \textit{where} & v = \widehat{\mathcal{A}}[\![a]\!](\widehat{\rho}, \widehat{\sigma}) \\
E\langle !\,a, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle \rightsquigarrow T\langle \widehat{\mathrm{read}}(\widehat{pc}, \widehat{\sigma}, \widehat{v}), \widehat{\sigma}, \widehat{\kappa\alpha} \rangle & \textit{where} & \widehat{v} = \widehat{\mathcal{A}}[\![a]\!](\widehat{\rho}, \widehat{\sigma}) \\
E\langle a_1 \leftarrow a_2, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle \rightsquigarrow T\langle (), \widehat{\sigma} \sqcup \widehat{\mathrm{write}}(\widehat{pc}, \widehat{v}_1, \widehat{v}_2), \widehat{\kappa\alpha} \rangle & \textit{where} & \widehat{v}_i = \widehat{\mathcal{A}}[\![a_i]\!](\widehat{\rho}, \widehat{\sigma}) \\
E\langle a(a), \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle \rightsquigarrow A\langle \widehat{v}_1, \widehat{v}_2, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle & \textit{where} & \widehat{v}_i = \widehat{\mathcal{A}}[\![a_i]\!](\widehat{\rho}, \widehat{\sigma}) \\
\underbrace{E\langle \mathtt{ref}(a), \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle}_{\widehat{\varsigma}} \rightsquigarrow T\langle \widehat{\alpha}, \widehat{pc}, \widehat{\sigma} \sqcup \{ \widehat{\alpha} \mapsto \widehat{v}' \}, \widehat{\kappa\alpha} \rangle & \textit{where} & \widehat{v} = \widehat{\mathcal{A}}[\![a]\!](\widehat{\rho}, \widehat{\sigma}) \quad \widehat{v}' = \langle\!\langle \widehat{pc} : \widehat{v} \sqcup \lfloor \star \rfloor \rangle\!\rangle \quad \widehat{\alpha} = \widehat{\mathrm{alloc}}(\widehat{\varsigma}) \\
\underbrace{E\langle \mathtt{label}[x](e), \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle}_{\widehat{\varsigma}} \rightsquigarrow T\langle \widehat{\ell}, \widehat{\sigma} \sqcup \{ \widehat{\ell} \mapsto \widehat{v} \}, \widehat{\kappa\alpha} \rangle & \textit{where} & \widehat{bv} = \lfloor \langle \lambda x.\ e, \widehat{\rho} \rangle \rfloor \quad \widehat{v} = \langle\!\langle \widehat{pc} : \widehat{bv} \sqcup \lfloor \star \rfloor \rangle\!\rangle \quad \widehat{\ell} = \widehat{\mathrm{alloc}}(\widehat{\varsigma}) \\
\underbrace{E\langle \langle a \ ?\ e_1 \diamond e_2 \rangle, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle}_{\widehat{\varsigma}} \rightsquigarrow E\langle e_1, \widehat{\rho}, \widehat{pc}'', \widehat{\sigma}', \widehat{\kappa\alpha}' \rangle & \textit{where} & \left\{ \begin{array}{lll} \widehat{v} = \widehat{\mathcal{A}}[\![a]\!](\widehat{\rho}, \widehat{\sigma}) & \widehat{pc}'' = \widehat{pc} \cup \widehat{pc}' & \widehat{\kappa} = \langle \widehat{pc}' \ ?\ \square \diamond E\langle e_2, \widehat{\rho}, \widehat{pc} \rangle \rangle :: \widehat{\kappa\alpha} \\ \widehat{pc}' = \{ \widehat{\ell} \mid \lfloor \widehat{\ell} \rfloor \sqsubseteq \widehat{v} \} & \widehat{\kappa\alpha}' = \widehat{\mathrm{alloc}}(\widehat{\varsigma}) & \widehat{\sigma}' = \widehat{\sigma} \sqcup \{ \widehat{\kappa\alpha}' \mapsto \lfloor \widehat{\kappa} \rfloor \} \end{array} \right. \\
\underbrace{E\langle \mathtt{obs}[a_1 @ a_2](a_3), \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle}_{\widehat{\varsigma}} \rightsquigarrow A\langle \widehat{v}, \widehat{v}_2, \widehat{pc}, \widehat{\sigma}', \widehat{\kappa\alpha}' \rangle & \textit{where} & \left\{ \begin{array}{lll} \widehat{v}_i = \widehat{\mathcal{A}}[\![a_i]\!](\widehat{\rho}, \widehat{\sigma}) & \widehat{v} = \bigsqcup \widehat{\mathrm{read}}(\widehat{pc}, \widehat{\sigma}, \lfloor \widehat{pc}' \rfloor) & \widehat{\kappa} = O\langle \widehat{pc}', \square, \widehat{v}_3 \rangle :: \widehat{\kappa\alpha} \\ \widehat{pc}' = \{ \widehat{\ell} \mid \lfloor \widehat{\ell} \rfloor \sqsubseteq \widehat{v}_1 \} & \widehat{\kappa\alpha}' = \widehat{\mathrm{alloc}}(\widehat{\varsigma}) & \widehat{\sigma}' = \widehat{\sigma} \sqcup \{ \widehat{\kappa\alpha}' \mapsto \lfloor \widehat{\kappa} \rfloor \} \end{array} \right. \\
A\langle \langle \widehat{bv}, \widehat{m} \rangle, \widehat{v}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle \rightsquigarrow T\langle \lfloor \star \rfloor, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle & \textit{where} & \lfloor \star \rfloor \sqsubseteq \widehat{bv} \\
\underbrace{A\langle \langle \widehat{bv}, \widehat{m} \rangle, \widehat{v}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle}_{\widehat{\varsigma}} \rightsquigarrow E\langle e, \widehat{\rho}[x \mapsto \widehat{\alpha}], \widehat{pc}, \widehat{\sigma}', \widehat{\kappa\alpha} \rangle & \textit{where} & \lfloor \langle \lambda x.\ e, \widehat{\rho} \rangle \rfloor \sqsubseteq \widehat{bv} \quad \widehat{\sigma}' = \widehat{\sigma} \sqcup \{ \widehat{\alpha} \mapsto \widehat{v} \} \quad \widehat{\alpha} = \widehat{\mathrm{alloc}}(\widehat{\varsigma}) \\
\underbrace{A\langle \langle \widehat{bv}, \{ \widehat{\ell} \mapsto v_1 \} \uplus \widehat{m} \rangle, \widehat{v}_2, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle}_{\widehat{\varsigma}} \rightsquigarrow A\langle \widehat{v}_1, \widehat{v}_2, \widehat{pc}', \widehat{\sigma}', \widehat{\kappa\alpha}' \rangle & \textit{where} & \widehat{pc}' = \widehat{pc} \cup \{ \widehat{\ell} \} \quad \widehat{\kappa\alpha}' = \widehat{\mathrm{alloc}}(\widehat{\varsigma}) \quad \widehat{\kappa} = \langle \widehat{pc}' : \square \rangle :: \widehat{\kappa\alpha} \quad \widehat{\sigma}' = \widehat{\sigma} \sqcup \{ \widehat{\kappa\alpha}' \mapsto \lfloor \widehat{\kappa} \rfloor \} \\
\underbrace{T\langle \widehat{v}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle}_{\widehat{\varsigma}} \rightsquigarrow E\langle e, \widehat{\rho}, \widehat{pc} \cup \widehat{pc}', \widehat{\sigma}', \widehat{\kappa\alpha}'' \rangle & \textit{where} & \left\{ \begin{array}{ll} \langle \widehat{pc}' \ ?\ \square \diamond E\langle e, \widehat{\rho}, \widehat{pc} \rangle \rangle :: \widehat{\kappa\alpha}' \in \widehat{\sigma}(\widehat{\kappa\alpha}) & \widehat{\kappa} = \langle pc' \ ?\ \widehat{v} \diamond \square \rangle :: \widehat{\kappa\alpha}' \\ \widehat{\kappa\alpha}'' = \widehat{\mathrm{alloc}}(\widehat{\varsigma}) & \widehat{\sigma}' = \widehat{\sigma} \sqcup \{ \widehat{\kappa\alpha}'' \mapsto \lfloor \widehat{\kappa} \rfloor \} \end{array} \right. \\
T\langle \widehat{v}_1, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle \rightsquigarrow T\langle \langle\!\langle \widehat{pc}' : \widehat{v}_1 \sqcup \widehat{v}_2 \rangle\!\rangle, \widehat{\sigma}, \widehat{\kappa\alpha}' \rangle & \textit{where} & \langle \widehat{pc}' \ ?\ \widehat{v}_2 \diamond \square \rangle :: \widehat{\kappa\alpha}' \in \widehat{\sigma}(\widehat{\kappa\alpha}) \\
T\langle \widehat{v}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle \rightsquigarrow T\langle \langle\!\langle \widehat{pc}' : \widehat{v} \rangle\!\rangle, \widehat{\sigma}, \widehat{\kappa\alpha}' \rangle & \textit{where} & \langle \widehat{pc}' : \square \rangle :: \widehat{\kappa\alpha}' \in \widehat{\sigma}(\widehat{\kappa\alpha}) \\
T\langle \widehat{v}_1, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle \rightsquigarrow T\langle \widehat{\mathrm{obs}}(\widehat{pc}', \widehat{v}_1, \widehat{v}_2), \widehat{\sigma}, \widehat{\kappa\alpha}' \rangle & \textit{where} & O\langle \widehat{pc}', \square, \widehat{v}_2 \rangle :: \widehat{\kappa\alpha}' \in \widehat{\sigma}(\widehat{\kappa\alpha})
\end{array}
$$

Fig. 5. Coarse Abstract Small-step Syntax, Metafunctions and Semantics

the concrete small-step semantics presented in Section III, but redirects all sources of infinite structure through the store in the standard AAM methodology. Crucially, this means all values are store allocated so that environments are no longer directly recursive, and that stacks/continuations are likewise store allocated as a linked list—this permits stack frames to be conflated at abstract memory locations and for cycles to be directly represented.

The abstract domains for our machine are shown in Figure 5. Our abstract machine is parameterized on the choice of an allocator, determined by the function $\widehat{\mathrm{alloc}}$, and three address spaces. The first is the abstract address space for values (used for variable bindings and `ref` cells), represented by $\widehat{\mathrm{vaddr}}$. Labels have their own address space, $\widehat{\mathrm{label}}$, so that label polyvariance is tunable, independent of the choice for value polyvariance—as we'll see in the next section, this tuning is particularly crucial for obtaining precision in a faceted language. Last, we have a separate address space for continuations, $\widehat{\kappa\mathrm{addr}}$. We achieve perfect call-return matching (pushdown precision [15], [45]) in our semantics by leveraging the "pushdown for free" (P4F) technique for allocating abstract continuations precisely [20]. In the P4F approach, continuation addresses are a source expression (e) paired with an abstract binding environment ($\widehat{\rho}$). This means, each polyvariant (context sensitive) binding environment and procedure entry point, has its own set of abstract continuations. That is: two continuations are only conflated during analysis, if they are the respective continuations of two dynamic function calls that are conflated under the current value-space polyvariance (abstract value-space allocator). P4F achieves optimal precision without any complexity overhead and permits us to vary analysis sensitivity without concern for proper call-return matching.

Abstract closures in our semantics are expressions paired with abstract environments ($\widehat{\rho}$), which map variables to either value addresses ($\widehat{\alpha}$) or labels ($\widehat{\ell}$), both of which are permitted in the domain of the abstract value store $\widehat{\sigma}$. The store maps abstract labels and value addresses to abstract values ($\widehat{\mathrm{val}}$), and disjointly, maps continuation addresses ($\widehat{\kappa\alpha}$) to sets of stacks (i.e., continuations). A continuation is a stack frame paired with an address referencing the tail of the stack (reaching a HALT frame terminates execution along that path). Abstract values use the abstraction developed in Section IV-B.

Specific to faceted execution, our abstract semantics tracks an abstract program counter, $\widehat{pc}$. Our abstract program counter $\widehat{pc}$ is a set of labels, rather than a set of branches. This reflects the fact that while we know we have branched on each abstract label $\widehat{l} \in \widehat{pc}$, we do not know whether we are in the positive or negative branch of any given $\widehat{l}$. As sketched in the previous section, abstract faceted values are either abstractions of base values or they are collapsed facets $\langle \widehat{l} : \widehat{v} \rangle$.

Our abstract store, $\widehat{\sigma}$, maps value addresses ($\widehat{vaddr}$) and labels ($\widehat{label}$) to abstract values, and continuation addresses ($\widehat{\kappa addr}$) to sets of abstract contexts—pairs of stack frames and another continuation address.

Atomic-expression evaluation $\widehat{\mathcal{A}}[\![\cdot]\!]$ proceeds similarly to our concrete semantics, taking an atom, abstract environment $\widehat{\rho}$, and abstract store $\widehat{\sigma}$. Base values (including constants, addresses, labels, and closures) are injected into the abstract domain via $\lfloor \cdot \rfloor$. Variable lookup is redirected through the store. Last, abstract closures are formed in the expected way, pairing an expression with the abstract environment.

Figure 5 shows the small-step rules for our abstract semantics. Similar to our concrete semantics, the E frame handles expression evaluation. All of the rules are largely unchanged from the concrete small-step semantics, the major difference being that we join values to form a collapsed facet. The notation $\{\widehat{\ell} \mid \lfloor \widehat{\ell} \rfloor \sqsubseteq \widehat{v}\}$ selects the set of abstract labels from the base-value component of $\widehat{v}$. Because abstract values now contain sets of labels rather than a single label, the set $\{\widehat{\ell} \mid \lfloor \widehat{\ell} \rfloor \sqsubseteq \widehat{v}\}$ is joined with $\widehat{pc}$.

As in the concrete semantics, the A configuration evaluates the application of possibly-faceted values. The first rule handles application of $\widehat{\star}$. The second considers the application of base values, represented by a set of closures inside $\widehat{bv}$. Application is performed by allocating for the argument and jumping to an E frame. In our abstract semantics, we represent facets by facet-maps, and so A must be nondeterministic over the domain of the facet-map. The last rule for A handles this case, decomposing the facet map into its individual components and jumping to another A frame, storing a continuation to build the result. Unlike the corresponding rule in the concrete semantics, the continuation never needs to evaluate the right side of a facet, as all facets have been collapsed.

Last, the T configuration inspects the continuation and handles it appropriately. As continuations will be conflated in the store, this rule is nondeterministic over the set of elements at $\widehat{\sigma}(\widehat{\kappa\alpha})$. The first T rule begins to explore the right hand side of a facet expression (not to be confused with a faceted value, all of which have been collapsed in our coarse abstraction). The second forms a (collapsed) facet from an already-evaluated left side. The third forms a facet from the value in the atom position. Last, the O frame performs the observation via the obs meta-operation.

D. A Branch-sensitive Abstraction via Singleton-analysis

In section IV-A, we observed several challenges in achieving both soundness and precision in an abstraction for faceted values. First, there is the challenge of conflating base values and facets (i.e., faceted values of differing height). Second, there is the challenge of conflating facets with different abstract labels without nesting one under the other as would occur according to the concrete semantics. Last, there is what we called the branch sensitivity problem: because abstract labels can be approximating two (or any number) dynamically generated concrete labels, we are unable to keep the positive and negative branches soundly apart.

Our branch-insensitive abstraction for abstract faceted values overcomes the first two challenges, preserving facet structure in a sound manner. The third is more difficult, requiring a more fundamental enhancement to the analysis. The problem is that if an abstract label $\widehat{\ell}$ can represent two different dynamic labels ℓ_1 and ℓ_2, a branch-sensitive abstract faceted value approximating $\langle \ell_1 \ ? \ \mathrm{v}_1^+ \ \diamond \ \mathrm{v}_1^- \rangle \sqcup \langle \ell_2 \ ? \ \mathrm{v}_2^+ \ \diamond \ \mathrm{v}_2^- \rangle$ would necessarily be

$$\langle \widehat{\ell} \ ? \ \widehat{\mathrm{v}_1^+} \sqcup \widehat{\mathrm{v}_1^-} \sqcup \widehat{\mathrm{v}_2^+} \sqcup \widehat{\mathrm{v}_2^-} \ \diamond \ \widehat{\mathrm{v}_1^+} \sqcup \widehat{\mathrm{v}_1^-} \sqcup \widehat{\mathrm{v}_2^+} \sqcup \widehat{\mathrm{v}_2^-} \rangle$$

conflating both branches regardless. This is required for soundness because a successful observation of ℓ_1 may be an unsuccessful observation of ℓ_2 and vice-versa. Short of proving all policies for an abstract label extensionally equivalent, interactions with either branch for ℓ_1 may need to pollute the opposite branch of ℓ_2 and the converse. If, however, we know that at a given point in the program, $\widehat{\ell}$ is a *singleton abstraction*—meaning it represents only a single exact concrete label—we can keep its positive and negative branches distinct, retaining the precision of branches.

Abstract counting is a technique from Might and Shivers [32] that augments an abstract interpreter to track a conservative overapproximation of how many concrete objects an abstract object is an abstraction of, at a certain point in the program's execution. The core idea is to extend the abstract store so that—for each address $\widehat{\alpha}$ in the abstract store—there is a corresponding address $\mathrm{count}(\widehat{\alpha})$ representing how many times $\widehat{\alpha}$ has been allocated: `0`, `1`, or `>1`.

The crucial observation then, as it applies to faceted values, is that it is sound to represent an abstract facet without merging

$$
\begin{array}{lll}
\widehat{m} \in \widehat{\text{facet-map}} & \triangleq & \widehat{\text{label}} \rightharpoonup \widehat{\text{val}} \uplus \widehat{\text{val}} \times \widehat{\text{val}} \\
\widehat{b} \in \widehat{\text{branch}} & ::= & +\widehat{\ell} \mid -\widehat{\ell} \mid \pm\widehat{\ell} \\
\widehat{pc} \in \widehat{\text{PC}} & \triangleq & \wp(\widehat{\text{branch}}) \\
\widehat{n} \in \widehat{\mathbb{C}} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{>1} \\
\widehat{\sigma} \in \widehat{\text{store}} & \triangleq & (\widehat{\text{vaddr}} \rightharpoonup \widehat{\text{val}}) \uplus (\widehat{\text{label}} \rightharpoonup \widehat{\text{val}} \times \widehat{\mathbb{C}})
\end{array}
$$

Fig. 6. A More Precise Abstraction via Counting

$$
\eta \in \text{val} \to \widehat{\text{val}} \qquad \alpha \in \wp(\text{val}) \to \widehat{\text{val}} \qquad \gamma \in \widehat{\text{val}} \to \wp(\text{val})
$$

$$
\eta(bv) \triangleq \langle \eta(bv), \varnothing \rangle
$$

$$
\eta(\langle \ell \ ? \ v_1 \diamond v_2 \rangle) \triangleq \begin{cases} \langle\!\langle \eta(\ell) : \eta(v_1) \sqcup \eta(v_2) \rangle\!\rangle & \textit{where} \quad |\gamma(\eta(\ell))| > 1 \\ \langle\!\langle \eta(\ell) \ ? \ \eta(v_1) \diamond \eta(v_2) \rangle\!\rangle & \textit{where} \quad |\gamma(\eta(\ell))| = 1 \end{cases}
$$

$$
\alpha(V) \triangleq \textstyle\bigsqcup_{v \in V} \eta(v) \qquad\qquad \gamma(\widehat{v}) \triangleq \{v \mid \eta(v) \sqsubseteq \widehat{v}\}
$$

Fig. 7. Galois Connection for our Abstract Facet Domain

its branches as long as its label has abstract count of 1. This is because—as long as we know an abstract label is an abstraction of only a single dynamic label—the equality checks on abstract labels in our semantics can be both sound and precise.

Figure 6 shows how we expand our abstract domain to account for precise facets. The idea is to represent abstract facets as branch-insensitive facets $\langle \widehat{\ell} \ ? \ \widehat{\text{v}} \rangle$ when $\widehat{\ell}$'s count is >1 and branch-sensitive facets $\langle \widehat{\ell} \ ? \ \text{v}^+ \diamond \widehat{\text{v}^-} \rangle$ otherwise. Instead of placing labels inside of $\widehat{\text{pc}}$ (as in Sections IV-A and IV-C), we add branches $+\widehat{\ell}$ and $-\widehat{\ell}$ when $\widehat{\ell}$ is singleton and $\pm\widehat{\ell}$ otherwise to represent that we must treat $\widehat{\ell}$ as non-singleton. Our metafunctions for store read, write, and update include all of the same functionality as they did in our coarse domain, but additionally exploit abstract counting for branch-sensitive facets. For example, observation of branch-sensitive facets simply projects the correct branch, rather than losing precision and reforming a facet (as in Section IV-B). Recall that for branch-insensitive facets, we had to reform facets alongside their projections, as abstract depth for branch-insensitive facets does not correspond to concrete depth.

We change the codomain of facet maps to a disjoint union of $\widehat{\text{val}}$ and $\widehat{\text{val}} \times \widehat{\text{val}}$, with the first representing branch-insensitive facets and the second representing branch-sensitive facets. We must maintain the invariant that—whenever $\widehat{\text{m}}(\widehat{\ell})$ is a product, $\widehat{\ell}$'s count is 1. To do this, we assume that the label allocator performs eager *count-based facet collapse*: whenever a label's count grows to >1, the store is traversed to collapse facets whose labels are no longer singleton. As discussed in section V, our implementation artifact uses a more involved *lazy fixing* approach that we eschew here for simplicity of presentation and soundness proofs.

We show updates to the metafunctions in the Figure 6. In particular the join operator $\sqcup$ changes to perform the join of maps at each point $\widehat{\ell}$ such that $\widehat{\ell}$ is non-singleton, and distributes across the pair in the same manner in the case that $\widehat{\ell}$ is singleton.

E. Correctness via Galois Connections

We formalize the correctness of our precise abstract domain by constructing a Galois connection between concrete faceted values and abstract faceted values. We then use this Galois connection to state and prove several soundness lemmas. Our Galois connection for abstract facets is shown in Figure 7. The abstraction side of our Galois connection is given in η-form, which necessarily induces α as the join over η. We induce the concretization function γ from η as well—this construction is standard, and allows the construction of any adjoint maps α and γ from any η [36].

The abstraction function η abstracts concrete facets to a single-branch facet when there may be other concrete facets which abstract to the same label, written $|\gamma(\eta(\ell))|$. When the concrete label is the only one contained in its abstraction, a precise double-branch facet is created. The Galois connection uses the abstract canonicalization metafunctions $\langle\!\langle \cdot : \cdot \rangle\!\rangle$ and $\langle\!\langle \cdot \ ? \ \cdot \diamond \cdot \rangle\!\rangle$ in its definition. Abstract canonicalization is therefore not sound per-se—rather it is part of the specification for soundness.

The join operator $\cdot \sqcup \cdot$ for abstract facets is used explicitly in the definition of α, and implicitly in the definition of γ, in that the partial order for facets is induced by the join. The join is trivially sound, as it is used in the definition of the Galois connection. However, we still must show it is a proper join operator, that is, associative, commutative, and idempotent. This is done in Appendix A.

Finally, we use these lemmas to prove soundness of the abstract interpretation. The formalization follows from the standard recipe for soundness via AAM and is sketched in Appendix A. As our precise domain (Section IV-D) degrades gracefully to the coarse abstract domain (Section IV-B) in the absence of abstract counting, we present only a formalization of the precise abstract domain.

V. Implementation and Evaluation

We have evaluated our ideas in several ways. First, we have implemented both concrete and abstract interpreters for λ_{FE}. Our implementation mirrors (where possible) the coarse abstract domain presented in Section IV-C, however it adds several features useful for writing more realistic programs (such as k-ary lambdas, builtins, `let` binding, and conditionals). Our abstract interpreter also uses global store-widening (developed for CFA by [41]) to avoid the exponential blowup incurred with per-state stores, and is roughly 1,600 lines of Racket source. Global store-widening moves the store from being a component of configurations to instead being a top-level component of the fixpoint. We have evaluated our abstract interpreter on five small example programs that make use of facets. Last, we briefly present an example of how our fine-grained abstract domain may be used to verify noninterference.

Evaluation Programs We wrote five small but emblematic benchmark programs. Each of these example programs uses facets in a slightly different way. The programs are listed on the left hand side of in Table I, with their corresponding line count (LOC) directly to the right. We implemented two benchmarks, `noninterf1` and `noninterf2`, which test basic reasoning about information flow. The next three benchmarks use facets to design a secure auction (securing bids), a grading system (securing submissions), and our Battleship example. Each of our benchmark programs includes one use `obs`.

TABLE I
DETAILS OF OUR BENCHMARK PROGRAMS

Program	LOC	# States	Analysis Time (ms)	Obs Elim?
noninterf1	8	72	12	✓
noninterf2	8	82	18	✗
auction	31	876	747	✗
grades	32	327	447	✓
battleship	56	514	743	✓

Performance of the coarse-grained domain We ran our abstract interpreter on each benchmark program and report the number of states and time (in milliseconds) taken by the analysis in the center columns of Table I. For each of the benchmarks, we verified our abstract interpreter produced sound results via manual inspection of the state space assisted via a visualization tool we built for the task.

To assess the precision of our coarse domain, we looked at the results of the analysis for each benchmark and determined whether we could statically eliminate the `obs` forms included in each. The right side of Table I indicates a check (✓) whenever we can either statically determine that both sides of an observed facet are equal (which we can do in benchmark `noninterf1`) or whenever we know that one a single side of the facet will be observed (as we can do in benchmarks `grades` and `battleship`). We observed that, even though our coarse domain does not reason precisely about the contents of facets, it was still able to eliminate `obs` checks in three of our benchmarks. The other two were not able to be eliminated because our coarse analysis loses precision. For example, `noninterf2` branches on a facet to create another facet, which ends up merging branches and losing the precision necessary to eliminate the `obs` form.

Worked Example: Verifying Information Flow We now demonstrate how our precise abstract domain offers a basis for verification of information flow properties. We illustrate this by showing how we can check noninterference for the following example:

```
(let* ([x (secret)] [y (ref 0)] [z (ref 0)])
  (if (= x 0)
      (begin (y ← 0) (z ← −1))
      (begin (y ← 1) (z ← 1)))
  (begin (y ← (+ y 1)) (z ← (* z z)))
  (output y z))
```

In this example, the call to `output` leaks (the zeroness of) the secret variable `x` through the variable `y`, while the variable `z` is safe. To check this, we treat the call to `secret` as creating a facet over some arbitrary "secret" label ℓ, and returning a facet $\langle \ell \ ? \top \diamond \top \rangle$. This has the effect of using the analysis to explore all possible *pairs* of program paths (as is traditional in checking noninterference) by leveraging the semantics of faceted execution to pull the branching from the state space into the value space. Checking noninterference is then reduced to checking whether both branches of each of the facets for `y` and `z` are equal once they reach the call to `output`. When analyzed using a path-sensitive abstraction and the constant propagation value lattice, our precise domain is able to keep the facet in `y` precise, allowing us to derive a counterexample wherein `y` contains a reference to the facet $\langle \ell \ ? \ 1 \diamond 2 \rangle$, while `z` always contains a reference to the facet $\langle \ell \ ? \ 1 \diamond 1 \rangle$.

VI. Related Work

To our knowledge, we are the first to present an abstract interpretation for faceted execution. There are several threads of related work in dynamic information flow, static analysis thereof, and programming paradigms for information flow.

Information-flow was first formalized by Denning [13]. In her seminal work on a lattice model for information flow, she outlined challenges and solutions to static information-flow checking. Subsequently, Goguen and Meseguer [21] defined noninterference, formalizing the idea that privileged data should not influence public outputs. Clarkson and Schneider [9] later recognized that information-flow properties could not be characterized by a single trace of a program, but rather a set of traces, and called these hyperproperties.

Since their original definitions, there has been much work on statically checking information-flow properties. Barthe's work on self-composition copies the program twice and asserts a relational property to check noninterference [6]. This idea was later extended to what the authors call product programs, and certified using a relational program logic [5]. Other work has used model checking to check noninterference [28] along with more general hyperproperties [10].

Of the mechanisms for static information flow, security type systems have gained the most use. First introduced by Volpano and Smith [46], these type systems augment the binding environment to track privilege of variables and prevent writes that would violate noninterference. Myers leveraged this idea to produce Jif, a variant of Java with an information-flow type system [33]. Security type systems have been subsequently extended to accommodate concurrent programs [49] and flow sensitivity [23]. Faceted execution does not require adding type annotations, but at the expense of losing a static characterization of the program's security in its type system.

Devriese and Piessens [14] first introduced secure multi-execution as a dynamic enforcement technique for information flow. Secure multi-execution runs 2^k copies of a program in parallel, where each run represents a subset of $\mathcal{P}(Prin)$, where $Prin$ is a set of principals. For example, if the principals are Alice and Bob, multi-execution executes four copies of the program: one that replaces all secret inputs by $\perp$, one that replaces Bob's input by $\perp$ but Alice's input by the true input, one for Bob's input, and one with access to all privileged information. When external effects are made (e.g., writing to disc), the runtime can select which variant to use. Secure multi-execution prevents information flow violations at runtime by ensuring that observations which violate the information-flow policy receive a view of the data computed without access to the secret inputs. Secure multi-execution has been extended in a variety of ways, e.g., scaling to its implementation in web browsers [7], adding declassification in a granular way [38], and even preventing side-channel attacks [25].

As the number of principals increases, secure multi-execution's overhead increases exponentially, unnecessarily duplicating work not influenced by secret inputs. Austin et al. introduced faceted execution as an optimization of secure multi-execution in [3]. Instead of treating the whole program as a potentially-secret computation, faceted execution realizes that influence can be tracked and propagated in a granular way using facets. Notably, Austin et al.'s work does not include first-class labels, as it was simulating secure multi-execution, where the principals could not be dynamically generated. More recently, Schmitz et al. [39] have harmonized both of these ideas into a unifying framework (Multef) that allows mixing both faceted and secure multi-execution.

Yang et al. first implemented Jeeves, a language allowing policy-agnostic programming [47]. Policy-agnostic programming takes the view that programs should be written without regard to a particular privacy policy, because as the policy changes, correctly updating program logic is cumbersome and error-prone. Policy-agnostic programming was first implemented in the domain-specific language Jeeves, using an SMT solver to decide which view of secret data to reveal based on a policy. Later, both authors collaborated to implement Jeeves using faceted execution. [4]. This formulation includes first-class labels, and is the basis for our concrete semantics. This semantics was the motivation for Micinski et al.'s recent work on implementing faceted execution via an expressive macro system [29]. We envision that our analysis of faceted execution might enable future languages based on facets by allowing more optimal compilation and static checking of facets.

Several other efforts into dynamic analysis for information flow are worth noting. Stefan et al [42] first presented `LIO`—a monad (with implementation in Haskell) that tracks privilege of the current program counter and forbids effects that would violate the security policy. It might be surprising that `LIO` works well for Haskell programs, given that faceted execution is more precise than `LIO`—allowing values to become faceted rather than halting the program. One key difference is that Haskell programs emphasize purity while languages such as JavaScript (the original target of faceted execution) does not, so much of the machinery for faceted execution's effect on the store is less interesting. Several authors have implemented related systems to `LIO`, including variants of faceted execution [40] and variants of `LIO` that extend its power to arbitrary monad transformers [37]. We believe that it would be possible to implement a variant of our technique that would give similar insights to programs using `LIO`, though much of the interesting machinery for handling state may be unnecessary.

Our precise abstract domain for facets relies upon cardinality analysis. Hudak first proposed an abstract domain for approximating a value's reference count in the presence of sharing [22]. This reference count abstraction is useful for understanding when destructive updates can be performed statically. Cardinality analysis was a direct inspiration for Might and Shivers [32] to produce the abstract counting approach we build on. Independently, Jagannathan [24] presented an analysis for higher-order languages that tracks whether abstract locations are singletons, which enables a number of optimizations such as lightweight closure-conversion and strong updates on reference cells.

Last, there have been several exciting recent efforts in the development of static analyses for hyperproperties and information flow. Assaf et al. [1] disuss how correct-by-construction dynamic security monitors using a technique based on abstract interpretation. In a similar direction, Assaf et al. [2] explored "hypercollecting semantics" and abstract interpretation for hyperproperties via a set of sets transformer which allows using Galois connections for hyperproperties alongside a traditional trace-based semantics. Mastroeni and Pasqua [26], [27] study abstract interpretation for *subset-closed* hyperproperties. Giacobazzi et al. [17], [18] present abstract non-interference, developing a framework which allows extremely flexible specification of information flow properties parameterized on observations, principals, and the observational abilities of external observers. Chudnov et al. [8] study how relational logic properties can be checked by way of interpreting a dynamic monitor's state as an abstract interpretation over sets of program executions.

VII. Conclusion

We have presented the first sound and precise abstraction for faceted execution in the presence of first-class security policies. This required formulating two abstractions: a precise abstract domain that preserves facet structure, and a coarse domain (to which the fine-grained domain gracefully degrades) that collapses the branches of facets but still allows reasoning about facet structure (differentiating between faceted and unfaceted values). We see this as a central challenge in the verification and optimization of policy-agnostic programs.

Faceted execution, along with other dynamic information-flow monitors, present exciting opportunities, but there are many questions about how we may use these techniques to design languages and systems. We believe that one particularly promising use of dynamic information flow is in its piecemeal application to potentially-insecure pieces of programs, with powerful static analyses to verify when heavyweight machinery may be elided. We believe that powerful static analysis for dynamic information-flow monitors will be useful not just for efficiency, but also so that programmers may gain confidence that security checks will not fail in unexpected ways at runtime. We see this work as a foundational step toward that goal in enhancing our understanding of systems utilizing faceted execution.

Acknowledgments

We thank the reviewers for their valuable feedback in improving this paper. This work was suppored in part by ODNI IARPA via 2019-1902070008, and NSF award CCF-1901278. The views, opinions and/or findings expresses are those of the authors and should not be interpreted as representing the official views or policies of any US Government agency. The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation therein.

References

[1] Assaf, M., Naumann, D.A.: Calculational design of information flow monitors. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF). pp. 210–224. IEEE Computer Society, Los Alamitos, CA, USA (jul 2016). https://doi.org/10.1109/CSF.2016.22, `https://doi.ieeecomputersociety.org/10.1109/CSF.2016.22`

[2] Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 874–887. POPL 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009889, `http://doi.acm.org/10.1145/3009837.3009889`

[3] Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 165–178. POPL '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2103656.2103677, `http://doi.acm.org/10.1145/2103656.2103677`

[4] Austin, T.H., Yang, J., Flanagan, C., Solar-Lezama, A.: Faceted execution of policy-agnostic programs. In: Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. pp. 15–26. PLAS '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2465106.2465121, `http://doi.acm.org/10.1145/2465106.2465121`

[5] Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Proceedings of the 17th International Conference on Formal Methods. pp. 200–214. FM'11, Springer-Verlag, Berlin, Heidelberg (2011), `http://dl.acm.org/citation.cfm?id=2021296.2021319`

[6] Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the 17th IEEE Workshop on Computer Security Foundations. pp. 100–114. CSFW '04, IEEE Computer Society, Washington, DC, USA (2004). https://doi.org/10.1109/CSFW.2004.17, `https://doi.org/10.1109/CSFW.2004.17`

[7] Bielova, N., Devriese, D., Massacci, F., Piessens, F.: Reactive non-interference for a browser model. In: 2011 5th International Conference on Network and System Security. pp. 97–104. IEEE (Sept 2011). https://doi.org/10.1109/ICNSS.2011.6059965

[8] Chudnov, A., Kuan, G., Naumann, D.A.: Information flow monitoring as abstract interpretation for relational logic. In: 2014 IEEE 27th Computer Security Foundations Symposium. pp. 48–62 (2014)

[9] Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 2008 21st IEEE Computer Security Foundations Symposium. pp. 51–65. IEEE (June 2008). https://doi.org/10.1109/CSF.2008.7

[10] Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) Principles of Security and Trust. pp. 265–284. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

[11] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973, `http://doi.acm.org/10.1145/512950.512973`

[12] Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of logic and computation **2**(4), 511–547 (1992)

[13] Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (May 1976)

[14] Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: 2010 IEEE Symposium on Security and Privacy. pp. 109–124. Oakland '10 (May 2010). https://doi.org/10.1109/SP.2010.15

[15] Earl, C., Sergey, I., Might, M., Van Horn, D.: Introspective pushdown analysis of higher-order programs. In: International Conference on Functional Programming. pp. 177–188 (September 2012)

[16] Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. pp. 237–247. PLDI '93, ACM, New York, NY, USA (1993). https://doi.org/10.1145/155090.155113, `http://doi.acm.org/10.1145/155090.155113`

[17] Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 186–197. POPL '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/964001.964017, `https://doi.org/10.1145/964001.964017`

[18] Giacobazzi, R., Mastroeni, I.: Abstract non-interference: A unifying framework for weakening information-flow. ACM Trans. Priv. Secur. **21**(2) (Feb 2018). https://doi.org/10.1145/3175660, `https://doi.org/10.1145/3175660`

[19] Gilray, T., Adams, M.D., Might, M.: Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 407–420. ICFP '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2951913.2951936, `http://doi.acm.org/10.1145/2951913.2951936`

[20] Gilray, T., Lyde, S., Adams, M.D., Might, M., Van Horn, D.: Pushdown control-flow analysis for free. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 691–704. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837631, `http://doi.acm.org/10.1145/2837614.2837631`

[21] Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy. pp. 11–11 (April 1982). https://doi.org/10.1109/SP.1982.10014

[22] Hudak, P.: A semantic model of reference counting and its abstraction (detailed summary). In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming. pp. 351–363. LFP '86, ACM, New York, NY, USA (1986). https://doi.org/10.1145/319838.319876, `http://doi.acm.org/10.1145/319838.319876`

[23] Hunt, S., Sands, D.: On flow-sensitive security types. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 79–90. POPL '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1111037.1111045, `http://doi.acm.org/10.1145/1111037.1111045`

[24] Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: Single and loving it: Must-alias analysis for higher-order languages. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 329–341. POPL '98, ACM, New York, NY, USA (1998). https://doi.org/10.1145/268946.268973, `http://doi.acm.org/10.1145/268946.268973`

[25] Kashyap, V., Wiedermann, B., Hardekopf, B.: Timing- and termination-sensitive secure information flow: Exploring a new approach. In: 2011 IEEE Symposium on Security and Privacy. pp. 413–428. Oakland '11 (May 2011). https://doi.org/10.1109/SP.2011.19

[26] Mastroeni, I., Pasqua, M.: Verifying bounded subset-closed hyperproperties. In: Podelski, A. (ed.) Static Analysis. pp. 263–283. Springer International Publishing, Cham (2018)

[27] Mastroeni, I., Pasqua, M.: Statically analyzing information flows: An abstract interpretation-based hyperanalysis for non-interference. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 2215–2223. SAC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297498, `https://doi.org/10.1145/3297280.3297498`

[28] van der Meyden, R., Zhang, C.: Algorithmic verification of noninterference properties. Electronic Notes in Theoretical Computer Science **168**, 61 – 75 (2007), proceedings of the Second International Workshop on Views on Designing Complex Architectures

[29] Micinski, K., Wang, Z., Gilray, T.: Racets: Faceted execution in racket. In: ACM Workshop on Scheme and Functional Programming (Scheme) '18 (2018), `http://kmicinski.com/assets/scheme19.pdf`

[30] Micinski, Kristopher and Darais, David and Gilray, Thomas: Abstracting Faceted Execution (Tech Report). Tech. rep., ArXiV (2020)

[31] Might, M.: Abstract interpreters for free. In: International Static Analysis Symposium. pp. 407–421. SAS '10, Springer (2010)

[32] Might, M., Shivers, O.: Improving flow analyses via ΓCFA: abstract garbage collection and counting. In: ACM SIGPLAN Notices. vol. 41, pp. 13–25. ACM (2006)

[33] Myers, A.C.: Jflow: Practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 228–241. POPL '99, ACM, New York, NY, USA (1999). https://doi.org/10.1145/292540.292561, `http://doi.acm.org/10.1145/292540.292561`

[34] Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology **9**(4), 410–442 (October 2000), `http://www.cs.cornell.edu/andru/papers/iflow-tosem.pdf`

[35] Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif 3.0: Java information flow (July 2006), `http://www.cs.cornell.edu/jif`

[36] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag, Berlin, Heidelberg (1999)

[37] Parker, J.: LMonad: Information Flow Control for Haskell Web Applications. Master's thesis, University of Maryland, College Park, Maryland (2014)

[38] Rafnsson, W., Sabelfeld, A.: Secure multi-execution: Fine-grained, declassification-aware, and transparent. In: 2013 IEEE 26th Computer Security Foundations Symposium. pp. 33–48 (June 2013)

[39] Schmitz, T., Algehed, M., Flanagan, C., Russo, A.: Faceted secure multi execution. In: Proceedings of the 2018 ACM SIGSAC

Conference on Computer and Communications Security. pp. 1617–1634. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243806, `https://doi.org/10.1145/3243734.3243806`

[40] Schmitz, T., Rhodes, D., Austin, T.H., Knowles, K., Flanagan, C.: Faceted dynamic information flow via control and data monads. In: Proceedings of the 5th International Conference on Principles of Security and Trust - Volume 9635. pp. 3–23. POST '16, Springer-Verlag New York, Inc., New York, NY, USA (2016)

[41] Shivers, O.G.: Control-flow Analysis of Higher-order Languages of Taming Lambda. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991), uMI Order No. GAX91-26964

[42] Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in haskell. In: Proceedings of the 4th ACM Symposium on Haskell. pp. 95–106. Haskell '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2034675.2034688, `http://doi.acm.org/10.1145/2034675.2034688`

[43] Van Horn, D., Might, M.: Abstracting abstract machines. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 51–62. ICFP '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863543.1863553, `http://doi.acm.org/10.1145/1863543.1863553`

[44] Van Horn, D., Might, M.: Abstracting abstract machines. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 51–62. ICFP '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863543.1863553, `http://doi.acm.org/10.1145/1863543.1863553`

[45] Vardoulakis, D., Shivers, O.: CFA2: a context-free approach to control-flow analysis. In: Proceedings of the European Symposium on Programming. ESOP '10, vol. 6012, LNCS, pp. 570–589 (2010)

[46] Volpano, D.M., Smith, G.: A type-based approach to program security. In: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development. pp. 607–621. TAPSOFT '97, Springer-Verlag, London, UK, UK (1997), `http://dl.acm.org/citation.cfm?id=646620.697712`

[47] Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 85–96. POPL '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2103656.2103669, `http://doi.acm.org/10.1145/2103656.2103669`

[48] Yang, J., et al.: Preventing information leaks with policy-agnostic programming. Ph.D. thesis, Massachusetts Institute of Technology (2015)

[49] Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: 16th IEEE Computer Security Foundations Workshop, 2003. pp. 29–43. CSF '13 (June 2003). https://doi.org/10.1109/CSFW.2003.1212703

Appendix

Proofs of Correctness via Galois Connections

We now sketch several lemmas necessary to prove the soundness of our abstract interpretation. We first show that our abstract join is a proper join operator. Next, we justify the soundness of our meta-operations. Last, we demonstrate how these can be used to show the soundness of our abstract interpretation from Section IV.

Lemma A.1 (Abstract Facet Join Proper)**.** *The join operation* $\cdot \sqcup \cdot$ *is associative* $(\widehat{v_1} \sqcup (\widehat{v_2} \sqcup \widehat{v_3}) = (\widehat{v_1} \sqcup \widehat{v_2}) \sqcup \widehat{v_3}))$ *commutative* $(\widehat{v_1} \sqcup \widehat{v_2} = \widehat{v_2} \sqcup \widehat{v_1})$ *and idempotent* $\widehat{v} \sqcup \widehat{v} = \widehat{v}$.

Proof. A simple calculation. Most of the functionality of $\cdot \sqcup \cdot$ are operations either on the lattice of underlying base values (for which these properties hold) or by joining finite maps, for which these properties also hold. □

We turn next to the meta-operations read, write and obs, which implement the functionality of slicing some operation through a tree of facets. These operations are sound when the concrete interpretations are contained in the concretization of the abstract interpretations.

Lemma A.2 (Abstract Meta-operators Soundness)**.**

1) $\mathrm{read}(pc, \sigma, v) \in \gamma(\widehat{\mathrm{read}}(\eta(pc), \eta(\sigma), \eta(v)))$,
2) $\mathrm{write}(pc, \sigma, v_1, v_2) \in \gamma(\eta(\sigma) \sqcup \widehat{\mathrm{write}}(\eta(pc), \eta(v_1), \eta(v_2)))$
3) $\mathrm{obs}(\ell, b, v) \in \gamma(\widehat{\mathrm{obs}}(\eta(l), \eta(b), \eta(v)))$.

Proof. The proof for each of (1–3) are similar; we only sketch the proof for (2). It suffices to show $\eta(\mathrm{write}(pc, \sigma, v_1, v_2)) \sqsubseteq \eta(\sigma) \sqcup \widehat{\mathrm{write}}(\eta(pc), \eta(v_1), \eta(v_2))$. By induction on v_1, which is either a base value or a facet. In the case it is a base value, we have:

$$\begin{aligned}
&\eta(\mathrm{write}(pc, \sigma, \alpha, v))\\
&= \eta(\sigma[\alpha \mapsto \langle\!\langle pc \ ?\ v \diamond \sigma(\alpha) \rangle\!\rangle])\\
&\sqsubseteq \eta(\sigma) \sqcup \{\eta(\alpha) \mapsto \langle\!\langle \eta(pc) \ ?\ \eta(v) \diamond \eta(\sigma)(\eta(\alpha)) \rangle\!\rangle\\
&= \eta(\sigma) \sqcup \widehat{\mathrm{write}}(\eta(pc), \eta(\alpha), \eta(v))
\end{aligned}$$

When it is a faceted value and $|\gamma(\eta(\ell))| > 1$:

$$\begin{aligned}
&\eta(\mathrm{write}(pc, \sigma, \langle\!\langle \ell \ ?\ v_1^+ \diamond v_1^- \rangle\!\rangle, v_2)\\
&= \eta(\mathrm{write}(pc \cup \{-\ell\}, \mathrm{write}(pc \cup \{+\ell\}, \sigma, v_1^+, v_2), v_1^-, v_2))\\
&\wr\ \text{Induction Hypothesis}\ \wr\\
&\sqsubseteq \eta(\sigma) \begin{array}{l} \sqcup\, \widehat{\mathrm{write}}(\eta(pc) \cup \{\eta(+\ell)\}, v_1^+, \eta(v_2)) \\ \sqcup\, \widehat{\mathrm{write}}(\eta(pc) \cup \{\eta(-\ell\}, \sqcup v_1^-, \eta(v_2)) \end{array}\\
&\sqsubseteq \eta(\sigma) \sqcup \widehat{\mathrm{write}}(\eta(pc) \cup \{\eta(\ell)\}, v_1^+ \sqcup v_1^-, \eta(v_2))\\
&= \eta(\sigma) \sqcup \widehat{\mathrm{write}}(\eta(pc), \eta(\alpha), \eta(v))
\end{aligned}$$

□

Finally, we turn to the abstract small-step semantics for λ_{FE}. The transition rules are a straightforward structural abstraction, following the abstracting abstract machines methodology. As a consequence, we prove an abstraction theorem, which uses the prior lemma. Before we present the proof, we posit an alternative presentation of the concrete small-step semantics which store-allocates arguments to functions in the obvious way, and simulates the non-allocating concrete semantics. The proof is then a direct application of the AAM proof recipe: composition of step-wise abstraction of the store-allocating concrete semantics with simulation of the natural concrete semantics by the store-allocating one. We notate transitions in the store-allocating semantics $\varsigma \rightsquigarrow^{\sigma} \varsigma$, and the natural semantics $\varsigma \rightsquigarrow^{\sigma} \varsigma$. Because the AAM recipe is straightforward and standard, we omit a detailed proof.

Theorem A.3 (Abstract Semantics Soundness)**.**
$\varsigma \rightsquigarrow^{\sigma} \varsigma' \Rightarrow \exists \widehat{\varsigma'} \sqsupseteq \eta(\varsigma').\ \eta(\varsigma) \rightsquigarrow \widehat{\varsigma'}$.

Simulation of Big-step by Small-step (Definitions)

Our companion tech report includes 9 more figures which show the full syntax and semantics for both the direct-style/big-step model, as well as the A-normal-form/small-step model [30]. Following the figures is a proof of simulation between the two semantics for A-normal-form terms.