Abstracting Faceted Execution

Static Analysis of Dynamic Information-Flow Control for Higher-Order Languages

KRISTOPHER MICINSKI, Haverford College
DAVID DARAIS, University of Vermont
THOMAS GILRAY, University of Maryland, College Park

Faceted execution is a linguistic paradigm for dynamic information-flow control with the distinguishing feature that program values may be faceted. Such values simultaneously simulate multiple executions for different security labels. This enables policy-agnostic programming: a paradigm permitting expressive privacy policies to be stated independently 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 expressive and dynamic privacy policies but imposes significant runtime overhead and delays the discovery of possible violations.

In this paper, we present the first sound and precise abstraction for faceted values in the presence of first-class policies. We develop an abstract interpretation of faceted programs, resolving central issues inherent in static analysis of this language paradigm. The presence of first-class security labels and their policy predicates represents a significant challenge to analysis as *abstract labels* may conflate more than one runtime label under some circumstances—an imprecision which prevents the analysis from being sensitive to any facets on such labels. We provide a remediating technique that involves instrumenting the analysis to track its singleton abstractions. We implement our analysis in Racket, discussing its tunability and practical optimizations.

Additional Key Words and Phrases: faceted execution, information flow, abstract interpretation

1 INTRODUCTION

Digital systems handle more sensitive data than ever before. As they continue to grow in complexity, so do their privacy policies. In the wild, these policies are dynamic, imperfect and change 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 is typically scattered throughout a codebase, making it hard to gain confidence in their implementation. Unlike crashes—which often simply cause downtime—bugs in this code can 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 for Java), analysis techniques (e.g., self-composition and product programs), and core formalisms (e.g., the dependency core-calculus and decentralized label model). However, all of these solutions assume a single static privacy policy will hold forever. As we are reminded daily, this is never the case.

Policy-agnostic programming 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 represents

Authors' addresses: Kristopher Micinski, Haverford College, Philadelphia, PA, micinski@cs.haverford.edu; David Darais, University of Vermont, Burlington, VT, david.darais@uvm.edu; Thomas Gilray, University of Maryland, College Park, College Park, MD, tgilray@cs.umd.edu.

privileged data via decision trees, where each node is a policy (or *label*), and branches represent views of the data from two perspectives: one where the policy is satisfied, and one where it isn't.

As computation progresses, faceted execution propagates data from each view, 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 to design a static analysis in the style of abstract interpretation [Cousot and Cousot 1977] and abstracting abstract machines [Van Horn and Might 2010a]. 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, a simple 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 the join 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 singletons.

We implemented our abstract interpreter in Racket, scaling our core formalism to handle k-ary lambdas, builtins, `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 is known to be imprecise using global store-widening, so we present a frontier semantics for retaining high precision abstract counting while maintaining an efficient analysis. Last, our analysis uses lazy count-based facet collapse, so that we soundly move from a high-precision analysis of facets when possible to a sound but less precise abstraction when necessary.

Specifically, this paper makes the following contributions:

- A formulation of faceted execution as a small-step semantics.
- 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.
- A precise abstract domain for faceted values that retains its sensitivity to distinct branches but is only sound for singleton abstract labels.
- An implementation strategy of our analysis via store-widening and lazy facet merging.
- A prototype implementation of our analysis in Racket.

2 BACKGROUND

To introduce our setting we present the implementation of Battleship, a small guessing-based board game, using a policy-agnostic programming model. In this game each player has a grid-based 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 tile on the other player's 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 is simply the empty list, and adding a piece is done via `cons`:

```
1 (define (makeboard) '())
2 (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)))))))
```

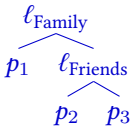
Although `mark-hit` will operate on sensitive data (the game boards), it is written without any special machinery to maintain the secrecy of `board`. Protecting data w.r.t. policies is instead handled automatically and implicitly by a runtime monitor. When Alice and Bob want to play a game, they both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

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

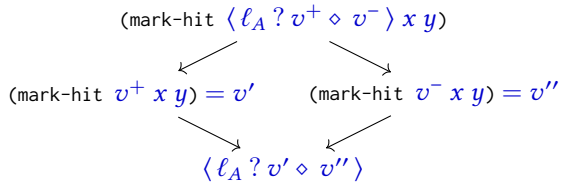
Bob would use a similar label (but with 2 instead of 1). At runtime, the `label` form creates a label ℓ_A and returns it to the binding for `alice-label`. When Alice wants to protect a value, she creates a *facet*, annotated with her label and two *branches*. The positive (left) branch represents the 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) ◇ ★ ⟩)
```

In the above example, Alice uses $\star$ (lazy failure) to represent that others should see nothing if they observe the data. In other applications, she might choose a more sensible 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 for ℓ_{Friends} and ℓ_{Family}. She would present three views of her social-media profile: p_1 to her family, containing her phone number and other contact information, p_2 to her friends showing her interests, and p_3 to everyone else, showing only her name and email.



As play of the game 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 of each branch are merged to produce a facet:



Because the applied function could be stateful, faceted execution also records the current privilege level in a *program counter*. The program counter is used to build facets when writes are made to the store inside of a privileged context. We expand upon these subtleties in Section 3, where we present a full semantics for faceted execution.

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:

$$\text{env} = \{\ell_A \mapsto \langle \lambda \texttt{x. (= x 1)} \rangle\}$$

$$\texttt{(obs alice-label 1 } \langle \ell_A \texttt{ ? } v^+ \texttt{ ⋄ } v^- \rangle\texttt{)} \rightsquigarrow^* \texttt{(if (= 1 1) } v^+ \texttt{ } v^-\texttt{)} \rightsquigarrow^* v^+$$

`obs l` $\texttt{v}^1$ $\texttt{v}^2$ 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, $\text{env}(\ell_A)(1) \rightsquigarrow^* \texttt{true}$, so `obs` select the right side of the Alice's facet.

Analyzing Faceted Execution. Faceted execution will ensure at runtime that no 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, but doesn't tell us anything statically about the policy. Additionally, faceted execution is a fairly heavyweight dynamic monitor, imposing significant performance overhead on code that uses facets unnecessarily. We want the best of both worlds: the flexibility of dynamic policies with the ability to verify them statically. In the case that we cannot statically verify a policy—as all analyses must concede some degree of imprecision—we can still fall back to faceted execution.

In our above example, our static analysis (developed in Section 4) 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 presence of dynamic policies. It also enables an optimization: as the program never attempts to violate 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 langauges) this result could be used to eliminating the machinery needed to manage faceted applications.

Roadmap. Throughout the rest of this paper, we develop an abstract interpretation for faceted execution. The analysis will determine—for any given point in the program—whether faceted values flow to that point, and if so what their branches contain. In Section 3, we present a concrete semantics for faceted execution, and (anticipating abstraction) develop a small-step abstract machine for faceted execution. In Section 4 we observe several challenges in designing an abstract domain to represent faceted values, showing that if our analysis is to analyze facets at all, it must differentiate between the presence of a base value and a faceted value. This leads us to a coarse abstraction for facets in Section 4.1, which we scale up to an abstract interpretation in Section 4.3. In Section 4.4 we present a more precise abstraction for faceted values under certain circumstances,

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

Fig. 1. Syntax of λ_{FE}

that gracefully degrades to our coarse abstraction. Finally, Section 5 presents our prototype implementation in Racket, and comments on several challenges in scaling up our formalism to work for larger programs.

3 SEMANTICS OF FACETED EXECUTION

Figure 1 gives the syntax for our source language. λ_{FE} extends the lambda calculus with references and three new forms unique to faceted execution: label creation, facet creation, and facet observation. Our implementation (described in Section 5) 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 uniquly 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.

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. [Austin and Flanagan 2012]. 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 their execution on 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.

The Facet rule creates a faceted value by calling out to the $\langle\!\langle \cdot\ ?\ \cdot \diamond \cdot \rangle\!\rangle$ meta-operator, which performs the work of *canonicalizing* 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 $\langle \ell\ ?\ a \diamond d \rangle$. This technique was first used by Austin et al. [Austin and Flanagan 2012] 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 [Austin and Flanagan 2012].

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 nonempty program counter, the label address is faceted under the current pc, with a default value of $\star$ (lazy failure). 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 following example, which rebinds the label ℓ:

$$
\begin{array}{l}
\texttt{let}\ l = \texttt{label}[x](\texttt{false})\ \texttt{in} \\
\texttt{let}\ fv = \langle l\ ?\ 100 \diamond 200 \rangle\ \texttt{in} \\
\texttt{let}\ l = \texttt{label}[x](x)\ \texttt{in} \\
\texttt{obs}[l@\texttt{true}](fv)
\end{array}
$$

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 rebind 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., the identity function).

The Obs rule evaluates syntax $\texttt{obs}[e_1@e_2](e_3)$, that introspects on a facted 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 $\texttt{obs}(\ell, b, v_3)$. 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. Consider

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

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

Lam
$$\frac{}{\rho, \sigma, \lambda x.\ e \Downarrow^E_{pc} \sigma, \langle \lambda x.\ e, \rho \rangle}$$

Read
$$\frac{\rho, \sigma, e \Downarrow^E_{pc} \sigma', v \qquad v' = \text{read}(pc, \sigma', v)}{\rho, \sigma, !\, e \Downarrow^E_{pc} \sigma', v'}$$

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

App
$$\frac{\rho, \sigma, e_1 \Downarrow^E_{pc} \sigma', v_1 \qquad \rho, \sigma', e_2 \Downarrow^E_{pc} \sigma'', v_2 \qquad \sigma'', v_1(v_2) \Downarrow^A_{pc} \sigma''', v}{\rho, \sigma, e_1(e_2) \Downarrow^E_{pc} \sigma''', v}$$

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

Obs
$$\frac{\rho, \sigma, e_1 \Downarrow^E_{pc} \sigma', \ell \qquad \rho, \sigma', e_2 \Downarrow^E_{pc} \sigma'', v_2 \qquad \rho, \sigma'', e_3 \Downarrow^E_{pc} \sigma''', v_3 \qquad \sigma''', \sigma'''(\ell)(v_2) \Downarrow^A_{pc} \sigma'''', b \qquad v = \text{obs}(\ell, b, v_3)}{\rho, \sigma, \texttt{obs}[e_1@e_2](e_3) \Downarrow^E_{pc} \sigma'''', v}$$

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

AppBase
$$\frac{\rho[x \mapsto v], \sigma, e \Downarrow^E_{pc} \sigma', v'}{\sigma, \langle \lambda x.\ e, \rho \rangle(v) \Downarrow^A_{pc} \sigma', v'}$$

AppStar
$$\frac{}{\sigma, \star(v) \Downarrow^A_{pc} \sigma, \star}$$

AppFacetSplit
$$\frac{\{+\ell, -\ell\} \cap pc = \varnothing \qquad \sigma, v_1(v_3) \Downarrow^A_{pc \cup \{+\ell\}} \sigma', v_1' \qquad \sigma', v_2(v_3) \Downarrow^A_{pc \cup \{-\ell\}} \sigma'', v_2'}{\sigma, \langle \ell\ ?\ v_1 \diamond v_2 \rangle(v_3) \Downarrow^A_{pc} \sigma'', \langle\!\langle \ell\ ?\ v_1' \diamond v_2' \rangle\!\rangle}$$

$$\begin{array}{rcl} \text{obs} \in \text{label} \times \text{bool} \times \text{val} & \rightharpoonup & \text{val} \\ \text{obs}(\ell, b, bv) & \triangleq & bv \\ \text{obs}(\ell, \texttt{true}, \langle \ell\ ?\ v_1 \diamond v_2 \rangle) & \triangleq & v_1 \\ \text{obs}(\ell, \texttt{false}, \langle \ell\ ?\ v_1 \diamond v_2 \rangle) & \triangleq & v_2 \\ \text{obs}(\ell, b, \langle \ell'\ ?\ v_1 \diamond v_2 \rangle) & \triangleq & \langle \ell'\ ?\ \text{obs}(\ell, b, v_1) \diamond \text{obs}(\ell, b, v_2) \rangle \quad \textit{where} \quad \ell \neq \ell' \end{array}$$

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

the following example:

$$(\lambda x.\ \texttt{ref}(x))(\langle \ell\ ?\ 1 \diamond 0 \rangle)$$

During evaluation of the positive branch of $\langle \ell\ ?\ 1 \diamond 0 \rangle$, $pc = \{+\ell\}$, 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, $\text{read}(\{+\ell\}, \sigma, \langle \ell\ ?\ 1 \diamond$

$$a \in \text{atom} ::= c \mid x \mid \lambda x.\ e$$
$$e \in \text{exp} ::= a \mid \texttt{ref}(a) \mid !\,a \mid a \leftarrow a \mid a(a) \mid \texttt{label}[x](e) \mid \langle a\ ?\ e \diamond e\rangle \mid \texttt{obs}[a@a](a)$$

$$\kappa \in \text{context} \triangleq \text{frame}^*$$
$$\varsigma \in \text{config} ::= E\langle e, \rho, pc, \sigma, \kappa\rangle \quad \textit{eval}$$
$$\mid A\langle v, v, pc, \sigma, \kappa\rangle \quad \textit{apply}$$
$$\mid T\langle v, \sigma, \kappa\rangle \quad \textit{return}$$

$$fr \in \text{frame} ::= \langle \ell\ ?\ \square \diamond E\langle e, \rho, pc\rangle\rangle$$
$$\mid \langle \ell\ ?\ \square \diamond A\langle v, v, pc\rangle\rangle$$
$$\mid \langle \ell\ ?\ v \diamond \square\rangle$$
$$\mid O\langle \ell, \square, v\rangle$$
$$\mid \text{HALT}$$

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

Fig. 3. Concrete Small-step Syntax

$\star\rangle)$ returns 1 and $\text{read}(\{-l\}, \sigma, \langle \ell\ ?\ 1 \diamond \star\rangle)$ returns $\star$. The Write rule works similarly, using the value presently in the reference cell as the default value in the case of facet construction.

The Projection Property and Noninterference. Austin et al. [Austin and Flanagan 2012; Austin et al. 2013] 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 $q \in \wp(branch)$ 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 3.1 (Projection Theorem). *Suppose* $\rho, \rho, e \Downarrow^E_{pc} \sigma', v$. *Then for any* $q \in \wp(branch)$ *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. [Austin et al. 2013] (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 [Austin and Flanagan 2012; Austin et al. 2013].

3.1 A Small-Step Machine

As a first step towards abstraction, we reformulate our big-step semantics in the small-step style using an abstract machine. To simplify the presentation, we assume that expressions in our language have been converted to A-Normal Form [Flanagan et al. 1993a], shown in the top of Figure 3. Our atomic expressions include constants, variables, and lambdas, which are evaluated using $\mathcal{A}[\![\cdot]\!]$.

The bottom of Figure 3 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 (shown in the bottom right of Figure 3).

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 finally terminates with a value in the T configuration, which inspects the continuation and handles the value appropriately.

Stack frames keep track of the work left to be done once reaching a value. For example, in the AppFacetSplit rule of the big-step semantics, the reduction $\Downarrow^A_{pc}$ first evaluates 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 $A\langle\langle l\ ?\ v_1 \diamond v_2\rangle \ldots\rangle$ in the small-step semantics, which first performs the application

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

$$E\langle a,\rho,pc,\sigma,\kappa\rangle \rightsquigarrow T\langle v,\sigma,\kappa\rangle \quad \text{where } v = \mathcal{A}[\![a]\!](\rho)$$

$$E\langle \texttt{ref}(a),\rho,pc,\sigma,\kappa\rangle \rightsquigarrow T\langle \alpha,pc,\sigma[\alpha \mapsto v'],\kappa\rangle \quad \text{where } \begin{cases} v = \mathcal{A}[\![a]\!](\rho) \\ v' = \langle\langle pc\ ?\ v \diamond \star\rangle\rangle \\ \alpha \notin \mathrm{dom}(\sigma) \end{cases}$$

$$E\langle !\, a,\rho,pc,\sigma,\kappa\rangle \rightsquigarrow T\langle \mathrm{read}(pc,\sigma,v),\sigma,\kappa\rangle \quad \text{where } v = \mathcal{A}[\![a]\!](\rho)$$

$$E\langle a_1 \leftarrow a_2,\rho,pc,\sigma,\kappa\rangle \rightsquigarrow T\langle (),\mathrm{write}(pc,\sigma,v_1,v_2),\kappa\rangle \quad \text{where } \begin{cases} v_1 = \mathcal{A}[\![a_1]\!](\rho) \\ v_2 = \mathcal{A}[\![a_2]\!](\rho) \end{cases}$$

$$E\langle a(a),\rho,pc,\sigma,\kappa\rangle \rightsquigarrow A\langle v_1,v_2,pc,\sigma,\kappa\rangle \quad \text{where } \begin{cases} v_1 = \mathcal{A}[\![a_1]\!](\rho) \\ v_2 = \mathcal{A}[\![a_2]\!](\rho) \end{cases}$$

$$E\langle \texttt{label}[x](e),\rho,pc,\sigma,\kappa\rangle \rightsquigarrow T\langle \ell,\sigma[\ell \mapsto v],\kappa\rangle \quad \text{where } \begin{cases} v = \langle\langle pc\ ?\ \langle \lambda x.\ e,\rho\rangle \diamond \star\rangle\rangle \\ \ell \notin \mathrm{dom}(\sigma) \end{cases}$$

$$E\langle a\ ?\ e_1 \diamond e_2,\rho,pc,\sigma,\kappa\rangle \rightsquigarrow E\langle e_1,\rho,pc \cup \{+\ell\},\sigma,\kappa'\rangle \quad \text{where } \begin{cases} \ell = \mathcal{A}[\![a]\!](\rho) \\ \kappa' = \langle \ell\ ?\ \square \diamond E\langle e_2,\rho,pc\rangle\rangle :: \kappa \end{cases}$$

$$E\langle \texttt{obs}[a_1@a_2](a_3),\rho,pc,\sigma,\kappa\rangle \rightsquigarrow A\langle \sigma(\ell),v_2,\rho,pc,\sigma,\kappa'\rangle \quad \text{where } \begin{cases} \ell = \mathcal{A}[\![a_1]\!](\rho) \\ v_2 = \mathcal{A}[\![a_2]\!](\rho) \\ v_3 = \mathcal{A}[\![a_3]\!](\rho) \\ \kappa' = O\langle \ell,\square,v_3\rangle :: \kappa \end{cases}$$

$$A\langle\langle \lambda x.\ e,\rho\rangle,v,pc,\sigma,\kappa\rangle \rightsquigarrow E\langle e,\rho[x \mapsto v],pc,\sigma,\kappa\rangle$$

$$A\langle \star,v,pc,\sigma,\kappa\rangle \rightsquigarrow T\langle \star,\sigma,\kappa\rangle$$

$$A\langle\langle \ell\ ?\ v_1 \diamond v_2\rangle,v,pc,\sigma,\kappa\rangle \rightsquigarrow A\langle v_1,v,pc,\sigma,\kappa\rangle \quad \text{where } +\ell \in pc$$

$$A\langle\langle \ell\ ?\ v_1 \diamond v_2\rangle,v,pc,\sigma,\kappa\rangle \rightsquigarrow A\langle v_2,v,pc,\sigma,\kappa\rangle \quad \text{where } -\ell \in pc$$

$$A\langle\langle \ell\ ?\ v_1 \diamond v_2\rangle,v,pc,\sigma,\kappa\rangle \rightsquigarrow A\langle v_1,v,pc,\sigma,\kappa'\rangle \quad \text{where } \begin{cases} \{+\ell,-\ell\} \cap pc = \varnothing \\ pc' = pc \cup \{+\ell\} \\ \kappa' = \langle \ell\ ?\ \square \diamond A\langle v_2,v,pc\rangle\rangle :: \kappa \end{cases}$$

$$T\langle v,\sigma,\langle \ell\ ?\ \square \diamond E\langle e,\rho,pc\rangle\rangle :: \kappa\rangle \rightsquigarrow E\langle e,\rho,pc \cup \{-\ell\},\sigma,\langle \ell\ ?\ v \diamond \square\rangle :: \kappa\rangle$$

$$T\langle v,\sigma,\langle \ell\ ?\ \square \diamond A\langle v_1,v_2,pc\rangle\rangle :: \kappa\rangle \rightsquigarrow A\langle v_1,v_2,pc \cup \{-\ell\},\sigma,\langle \ell\ ?\ v \diamond \square\rangle :: \kappa\rangle$$

$$T\langle v_1,\sigma,\langle \ell\ ?\ v_2 \diamond \square\rangle :: \kappa\rangle \rightsquigarrow T\langle\langle\langle \ell\ ?\ v_1 \diamond v_2\rangle\rangle,\sigma,\kappa\rangle$$

$$T\langle b,\sigma,O\langle \ell,\square,v\rangle :: \kappa\rangle \rightsquigarrow T\langle \mathrm{obs}(\ell,b,v),\sigma,\kappa\rangle$$

Fig. 4. Concrete Small-step Semantics

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 go back and 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 $\{-\ell\} \cup pc$). Facet 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 appliation 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 (as in Section 3).

4 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 [Van Horn and Might 2010b]. 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 [Cousot and Cousot 1977, 1992]. The heart of the AAM approach is the use of small-step transitions, preparatory store-allocating transformations shown in section 4.3 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 directly 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 degrees of precision in order to achieve guarantees of computability (and a bounded complexity of analysis).

AAM's solution 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(…)`).

This 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) [Gilray et al. 2016a]. Once a finite domain for abstract addresses and abstract base values has been

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

$$
\begin{array}{llll}
\text{(C1)} & e_1 & \rightsquigarrow^* \ \mathtt{obs}[\ell_B@\textsc{bob}](\langle \ell_A \ ?\ \textsc{tails} \diamond \star \rangle) & \rightsquigarrow^* \ \langle \ell_A \ ?\ \textsc{tails} \diamond \star \rangle \\
\text{(U1)} & \widehat{e} = e_1 & \widehat{\rightsquigarrow}^* \ \widehat{e} = \mathtt{obs}[\widehat{\ell}@\textsc{bob}](\langle \widehat{\ell} \ ?\ \textsc{tails} \diamond \star \rangle) & \widehat{\rightsquigarrow}^* \ \widehat{e} = \{\textsc{tails}, \star\} \\
 & \widehat{\sigma} = \varnothing & \widehat{\sigma} = \{\widehat{\ell} \mapsto \{\langle \lambda x.\ x \stackrel{?}{=} a, \{a \mapsto \textsc{alice}\}\rangle & \widehat{\sigma} = \mathit{...unchanged...} \\
 & & \quad , \langle \lambda x.\ x \stackrel{?}{=} a, \{a \mapsto \textsc{bob}\}\rangle\}\} & \\
\\
\text{(C2)} & e_2 & \rightsquigarrow^* \ \langle\!\langle \ell_B \ ?\ \langle \ell_A \ ?\ \textsc{tails} \diamond \star \rangle \diamond \star \rangle\!\rangle & \rightsquigarrow^* \ \langle \ell_A \ ?\ \langle \ell_B \ ?\ \textsc{tails} \diamond \star \rangle \\
 & & & \quad \diamond \langle \ell_B \ ?\ \star \diamond \star \rangle\rangle \\
\text{(U2)} & \widehat{e} = e_2 & \widehat{\rightsquigarrow}^* \ \widehat{e} = \langle\!\langle \widehat{\ell} \ ?\ \langle \widehat{\ell} \ ?\ \textsc{tails} \diamond \star \rangle \diamond \star \rangle\!\rangle & \widehat{\rightsquigarrow}^* \ \widehat{e} = \langle \widehat{\ell} \ ?\ \textsc{tails} \diamond \star \rangle \\
 & \widehat{\sigma} = \varnothing & \widehat{\sigma} = \{\widehat{\ell} \mapsto \{\langle \lambda x.\ x \stackrel{?}{=} a, \{a \mapsto \textsc{alice}\}\rangle & \widehat{\sigma} = \mathit{...unchanged...} \\
 & & \quad , \langle \lambda x.\ x \stackrel{?}{=} a, \{a \mapsto \textsc{bob}\}\rangle\}\} &
\end{array}
$$

Fig. 5. An example of concrete and abstract faceted execution. (C1) and (C2) are concrete executions, and (U1) and (U2) are unsound candidate abstract executions.

selected, these abstractions (formally Galois connections) may be systematically lifted to abstractions for stores, environments, and machine states [Might 2010].

4.1 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:

$$
\widehat{\mathrm{v}} ::= \widehat{bv} \mid \langle \widehat{\mathrm{l}} \ ?\ \widehat{\mathrm{v}} \diamond \widehat{\mathrm{v}} \rangle
\qquad\qquad
\widehat{\mathrm{v}} ::= \ \widehat{\mathrm{l}} \ \big\langle \ \widehat{\mathrm{v}}^+ \quad \widehat{\mathrm{v}}^- \ \big\rangle
$$

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 5. The top of the figure shows two example programs, e_1 and e_2, sharing a common prologue setup. The setting for our examples is a guessing game in which the players, 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-facet 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 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 4. 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 (TAILS and $\star$) are base values—which are simply returned when observed. The final result is the same facet that was originally being observed: $\langle\, \ell_A \;?\; \text{TAILS} \diamond \star \,\rangle$. 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 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 above. To ensure termination, the abstract semantics conflates certain program 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-facet 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 $\{a \mapsto \text{ALICE}\}$. Upon executing the third line of the example, mk-pol is called a second time, extending the label $\widehat{\ell}$ so that its policy also closes over $\{a \mapsto \text{BOB}\}$.

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 attached to 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 $\{\widehat{\text{TAILS}}, \widehat{\star}\}$.

Two Broken Interpretations for Abstract Facets. There are two naïve ways we may interpret $\{\widehat{\text{TAILS}}, \widehat{\star}\}$. First, we might interpret the abstract value $\{\widehat{\text{TAILS}}, \widehat{\star}\}$ as simply the set of concrete values $\{\text{TAILS}, \star\}$. 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 $\{\widehat{\text{TAILS}}, \widehat{\star}\}$ as $\langle\, \ell \,?\, \{\text{TAILS}, \star\} \diamond \{\text{TAILS}, \star\} \,\rangle \sqcup \{\text{TAILS}, \star\}$, (really, under *all* possible concrete ℓ) which includes the result observed 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 $\langle\, \ell \,?\, \widehat{\text{v}}^{+} \diamond \widehat{\text{v}}^{-} \,\rangle$ as $\widehat{\text{v}}^{+} \sqcup \widehat{\text{v}}^{-}$ and vice-versa.

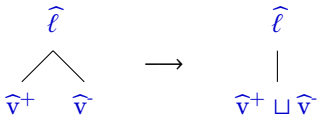
An abstract execution of e_2 using the proposed semantics is also an issue. In this case, as shown in (U2), the facet creation form now acts as a no-op since alice-pol and bob-pol share the abstract label $\widehat{\ell}$. Instead of creating a decision tree with two levels, the comparison proceeds to 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., $\{\text{TAILS}, \star\}$), 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 of abstract labels 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 4 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 $\langle\, \widehat{\ell} \,?\, \widehat{\text{v}}^{+} \diamond \widehat{\text{v}}^{-} \,\rangle$ is sound as long as $\widehat{\ell}$ is not an abstraction of *multiple concrete labels*; once this happens however, the branches $\widehat{\text{v}}^{+}$ and $\widehat{\text{v}}^{-}$ 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 4.4.

4.2 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:



We represent faceted values as a label over the join of both branches, $\langle \widehat{\ell} \,?\, \widehat{\text{v}}^{+} \sqcup \widehat{\text{v}}^{-} \rangle$. 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 a 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 $\widehat{bv} \sqcup \langle \widehat{\ell} \,?\, \widehat{\mathrm{v}} \rangle$? It is natural to simply define $\sqcup$ such that $\widehat{bv} \sqcup \langle \widehat{\ell} \,?\, \widehat{\mathrm{v}} \rangle$ is equal to $\langle \widehat{\ell} \,?\, \widehat{\mathrm{v}} \sqcup \widehat{bv} \rangle$ or equal to $\widehat{\mathrm{v}} \sqcup \widehat{bv}$, but as we've discussed, unsound if we want precision for facet structure. Consider the following example (using monovariant allocation) which leads to such a conflation:

```
1 let id(x) = x
2 let v1   = id(⟨ ℓ ? 1 ◇ 2 ⟩)
3 let v2   = id(3)
4 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 $\widehat{val}$ in figure 7). This allows us to directly represent values which must be a base value ($\widehat{bv} \times \bot$), must be faceted ($\bot \times \widehat{\mathrm{m}}$), or could be either ($\widehat{bv} \times \widehat{\mathrm{m}}$). 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:

```
1 let f(x) = obs[bob-label@bob](x)
2 f(⟨ bob-label ? X ◇ Y ⟩)
3 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 that the first call to `f` returns a facet:

$$\begin{array}{c} \widehat{\ell}_A \\ | \\ \widehat{v_1} \end{array} \sqcup \begin{array}{c} \widehat{\ell}_B \\ | \\ \widehat{v_2} \end{array} \overset{\text{Proposed}}{=} \begin{array}{c} \widehat{\ell}_A \\ | \\ \widehat{\ell}_B \\ | \\ \widehat{v_1} \sqcup \widehat{v_2} \end{array} \qquad l_1 \in \gamma(\widehat{l_A}) \qquad obs(l_1)\left(\gamma\left(\begin{array}{c} \widehat{\ell}_A \\ | \\ \widehat{\ell}_B \\ | \\ \widehat{v_1} \sqcup \widehat{v_2} \end{array}\right)\right) = \begin{array}{c} \gamma(\widehat{\ell}_B) \\ | \\ \gamma(\widehat{v_1} \sqcup \widehat{v_2}) \end{array}$$

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 7. For base values, we assume the presence of a standard join, which can be tuned alongside the abstraction for base values to recover the desired

precision. We also assume an injector $\lfloor bv \rfloor$ for base values, that takes concrete base values and injects them into an abstract representation, $\widehat{bv}$.

Abstract faceted values are represented by $\widehat{v}$, and comprise a pair of a $\widehat{bv} \in \widehat{\text{base-val}}$ and a $\widehat{m} \in \widehat{\text{facet-map}}$—necessary to avoid the unsoundness of conflating facets and base values. We represent abstract facets by a partial map ($\widehat{\text{facet-map}}$), 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, $\langle \widehat{\ell} \ ? \ \widehat{v} \rangle$ would be represented by $\{\widehat{\ell} \mapsto \widehat{v}\}$ and $\langle \widehat{\ell_1} \ ? \ \widehat{v}_1 \rangle \sqcup \langle \widehat{\ell_2} \ ? \ \widehat{v}_2 \rangle$ by $\{(\widehat{\ell_1} \mapsto \widehat{v}_1), (\widehat{\ell_2} \mapsto \widehat{v}_2)\}$.

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 considers separately 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 appear 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.

Last, 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., $\langle \widehat{bv}_1, \{\widehat{\ell}_A \mapsto \widehat{v}_1\} \rangle \sqcup \langle \widehat{bv}_2, \{\widehat{\ell}_B \mapsto \widehat{v}_2\} \rangle = \langle \widehat{bv}_1 \sqcup \widehat{bv}_2, \{\widehat{\ell}_A \mapsto \widehat{v}_1, \widehat{\ell}_B \mapsto \widehat{v}_2\} \rangle$ but $\langle \widehat{bv}_1, \{\widehat{\ell}_A \mapsto \widehat{v}_1\} \rangle \sqcup \langle \widehat{bv}_2, \{\widehat{\ell}_A \mapsto \widehat{v}_2\} \rangle = \langle \widehat{bv}_1 \sqcup \widehat{bv}_2, \{\widehat{\ell}_A \mapsto \widehat{v}_1 \sqcup \widehat{v}_2\} \rangle$. We present a Galois Connection for our abstract domain in Section 4.5.

4.3 An Abstract Interpretation for Faceted Execution

We now present an imprecise but sound abstract interpretation for faceted execution. Our abstract semantics extends the concrete small-step semantics presented in Section 3.1, 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 7. Our abstract machine is parameterized on the choice of an allocator, determined by the function *alloc*, and three address spaces. The first is the abstract address space for values (used for variable bindings and `ref` cells), represented by $\widehat{vaddr}$. Labels have their own address space, $\widehat{label}$, 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, $\widehat{\kappa addr}$. We achieve perfect call-return matching (pushdown precision [Earl et al. 2012; Vardoulakis and Shivers 2010]) in our semantics by leveraging the "pushdown for free" (P4F) technique for allocating abstract contiuations precisely [Gilray et al. 2016b]. In the P4F approach, continuation addresses are a source expression (e) paired with an abstract binding environment ($\widehat{\rho}$).

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

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

$$\begin{array}{rrcll}
fr \in & \text{frame} & ::= & \langle \widehat{pc}\ ?\ \square \diamond E\langle \widehat{e}, \widehat{\rho}, \widehat{pc} \rangle \rangle & \\
& & \mid & \langle \widehat{pc}\ ?\ \widehat{v} \diamond \square \rangle & \\
& & \mid & \langle \widehat{pc} : \square \rangle & \\
& & \mid & O\langle \widehat{pc}, \square, \widehat{v} \rangle & \\
& & \mid & \text{HALT} &
\end{array}$$

$$\begin{array}{rrcll}
\widehat{\varsigma} \in & \widehat{\text{config}} & ::= & E\langle \widehat{e}, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle & \textit{eval} \\
& & \mid & A\langle \widehat{v}, \widehat{v}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle & \textit{apply} \\
& & \mid & T\langle \widehat{v}, \widehat{\sigma}, \widehat{\kappa\alpha} \rangle & \textit{return}
\end{array}$$

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

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

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

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

$$\langle\!\langle \widehat{\ell} : \langle \widehat{bv}, \widehat{m} \rangle \rangle\!\rangle \triangleq \langle \bot, \widehat{m}_1 \sqcup \widehat{m}_2 \sqcup \widehat{m}_3 \rangle$$
$$\textit{where} \quad \begin{cases} \widehat{m}_1 = \{\widehat{\ell} \mapsto \langle \widehat{bv}, \varnothing \rangle \sqcup \widehat{m}(\widehat{\ell})\} \\ \widehat{m}_2 = \{\widehat{\ell} \mapsto \bigcup_{\widehat{\ell} < \widehat{\ell}' \in \text{dom}(\widehat{m})} \{\widehat{\ell}' \mapsto \widehat{m}(\ell')\}\} \\ \widehat{m}_3 = \bigcup_{\widehat{\ell} > \widehat{\ell}' \in \text{dom}(\widehat{m})} \{\widehat{\ell}' \mapsto \langle\!\langle \widehat{\ell} : \widehat{m}(\widehat{\ell}') \rangle\!\rangle\} \end{cases}$$

$$\widehat{\text{read}}(\widehat{pc}, \widehat{\sigma}, \langle \widehat{bv}, \widehat{m} \rangle) \triangleq \widehat{v}_1 \sqcup \widehat{v}_2 \sqcup \langle \widehat{bv}', \widehat{m}' \rangle$$
$$\textit{where} \quad \begin{cases} \widehat{v}_1 = \bigsqcup_{\lfloor \widehat{\alpha} \rfloor \sqsubseteq \widehat{bv}} \widehat{\sigma}(\widehat{\alpha}) \\ \widehat{v}_2 = \bigsqcup_{\widehat{\ell} \in \widehat{pc}} \widehat{\text{read}}(\widehat{pc}, \widehat{\sigma}, \widehat{m}(\widehat{\ell})) \end{cases} \qquad \widehat{m}' = \bigcup_{\widehat{\ell} \in \text{dom}(\widehat{\ell})} \{\widehat{\ell} \mapsto \widehat{\text{read}}(\widehat{pc}, \widehat{\sigma}, \widehat{m}(\widehat{\ell}))\}$$
$$\widehat{bv}' = \begin{cases} \lfloor \star \rfloor \ \textit{if} \ \lfloor \star \rfloor \sqsubseteq \widehat{bv} \\ \bot \ \textit{otherwise} \end{cases}$$

$$\widehat{\text{write}}(\widehat{pc}, \langle \widehat{bv}, \widehat{m} \rangle, \widehat{v}) \triangleq \sigma_1 \sqcup \sigma_2$$
$$\textit{where} \quad \widehat{\sigma}_1 = \bigsqcup_{\lfloor \widehat{\alpha} \rfloor \sqsubseteq \widehat{bv}} \{\widehat{\alpha} \mapsto \langle\!\langle \widehat{pc}\ ?\ \widehat{v} \diamond \widehat{\sigma}(\widehat{\alpha}) \rangle\!\rangle\} \qquad \widehat{\sigma}_2 = \bigsqcup_{\widehat{\ell} \in \text{dom}(\widehat{m})} \widehat{\text{write}}(\widehat{pc} \cup \{\widehat{\ell}\}, \widehat{m}(\widehat{\ell}), \widehat{v})$$

$$\widehat{\text{obs}}(\widehat{\ell}, \widehat{v}, \langle \widehat{bv}, \widehat{m} \rangle) \triangleq \widehat{m}(\widehat{\ell}) \sqcup \langle \widehat{bv}, \widehat{m}_1 \sqcup \widehat{m}_2 \rangle$$
$$\textit{where} \quad \widehat{m}_1 = \{\widehat{\ell} \mapsto \widehat{m}(\ell)\} \qquad \widehat{m}_2 = \bigcup_{\widehat{\ell}' \in \text{dom}(\widehat{m}')} \{\widehat{\ell}' \mapsto \widehat{\text{obs}}(\widehat{\ell}, \widehat{v}, \widehat{m}(\widehat{\ell}'))\}$$

Fig. 6. Coarse Abstract Small-step Syntax and Metafunctions

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

$$E\langle a, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \rightsquigarrow T\langle \widehat{v}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \quad \textit{where} \quad v = \widehat{\mathcal{A}}[\![a]\!](\widehat{\rho}, \widehat{\sigma})$$

$$E\langle !\, a, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \rightsquigarrow T\langle \widehat{\text{read}}(\widehat{pc}, \widehat{\sigma}, \widehat{v}), \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \quad \textit{where} \quad \widehat{v} = \widehat{\mathcal{A}}[\![a]\!](\widehat{\rho}, \widehat{\sigma})$$

$$E\langle a_1 \leftarrow a_2, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \rightsquigarrow T\langle (), \widehat{\sigma} \sqcup \widehat{\text{write}}(\widehat{pc}, \widehat{v}_1, \widehat{v}_2), \widehat{\kappa\alpha}\rangle \quad \textit{where} \quad \widehat{v}_i = \widehat{\mathcal{A}}[\![a_i]\!](\widehat{\rho}, \widehat{\sigma})$$

$$E\langle a(a), \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \rightsquigarrow A\langle \widehat{v}_1, \widehat{v}_2, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \quad \textit{where} \quad \widehat{v}_i = \widehat{\mathcal{A}}[\![a_i]\!](\widehat{\rho}, \widehat{\sigma})$$

$$\underbrace{E\langle \texttt{ref}(a), \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle}_{\widehat{\varsigma}} \rightsquigarrow T\langle \widehat{\alpha}, \widehat{pc}, \widehat{\sigma} \sqcup \{\widehat{\alpha} \mapsto \widehat{v}'\}, \widehat{\kappa\alpha}\rangle$$

$$\textit{where} \quad \widehat{v} = \widehat{\mathcal{A}}[\![a]\!](\widehat{\rho}, \widehat{\sigma}) \qquad \widehat{v}' = \langle\!\langle \widehat{pc} : \widehat{v} \sqcup \lfloor \star \rfloor \rangle\!\rangle \qquad \widehat{\alpha} = \widehat{\text{alloc}}(\widehat{\varsigma})$$

$$\underbrace{E\langle \texttt{label}[x](e), \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle}_{\widehat{\varsigma}} \rightsquigarrow T\langle \widehat{\ell}, \widehat{\sigma} \sqcup \{\widehat{\ell} \mapsto \widehat{v}\}, \widehat{\kappa\alpha}\rangle$$

$$\textit{where} \quad \widehat{bv} = \lfloor \langle \lambda x.\ e, \widehat{\rho}\rangle \rfloor \qquad \widehat{v} = \langle\!\langle \widehat{pc} : \widehat{bv} \sqcup \lfloor \star \rfloor \rangle\!\rangle \qquad \widehat{\ell} = \widehat{\text{alloc}}(\widehat{\varsigma})$$

$$\underbrace{E\langle \langle a\ ?\ e_1 \diamond e_2\rangle, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle}_{\widehat{\varsigma}} \rightsquigarrow E\langle e_1, \widehat{\rho}, \widehat{pc}'', \widehat{\sigma}', \widehat{\kappa\alpha}'\rangle$$

$$\textit{where} \quad \begin{cases} \widehat{v} = \widehat{\mathcal{A}}[\![a]\!](\widehat{\rho}, \widehat{\sigma}) & \widehat{pc}'' = \widehat{pc} \cup \widehat{pc}' \quad \widehat{\kappa} = \langle \widehat{pc}'\ ?\ \square \diamond E\langle \widehat{e}_2, \widehat{\rho}, \widehat{pc}\rangle\rangle :: \widehat{\kappa\alpha} \\ \widehat{pc}' = \{\widehat{\ell} \mid \lfloor \widehat{\ell} \rfloor \sqsubseteq \widehat{v}\} & \widehat{\kappa\alpha}' = \widehat{\text{alloc}}(\widehat{\varsigma}) \quad \widehat{\sigma}' = \widehat{\sigma} \sqcup \{\widehat{\kappa\alpha}' \mapsto \lfloor \widehat{\kappa} \rfloor\} \end{cases}$$

$$\underbrace{E\langle \texttt{obs}[a_1@a_2](a_3), \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle}_{\widehat{\varsigma}} \rightsquigarrow A\langle \widehat{v}, \widehat{v}_2, \widehat{pc}, \widehat{\sigma}', \widehat{\kappa\alpha}'\rangle$$

$$\textit{where} \quad \begin{cases} \widehat{v}_i = \widehat{\mathcal{A}}[\![a_i]\!](\widehat{\rho}, \widehat{\sigma}) & \widehat{v} = \widehat{\text{read}}(\widehat{\sigma}, \widehat{pc}') \quad \widehat{\kappa} = O\langle \widehat{pc}', \square, \widehat{v}_3\rangle :: \widehat{\kappa\alpha} \\ \widehat{pc}' = \{\widehat{\ell} \mid \lfloor \widehat{\ell} \rfloor \sqsubseteq \widehat{v}_1\} & \widehat{\kappa\alpha}' = \widehat{\text{alloc}}(\widehat{\varsigma}) \quad \widehat{\sigma}' = \widehat{\sigma} \sqcup \{\widehat{\kappa\alpha}' \mapsto \lfloor \widehat{\kappa} \rfloor\} \end{cases}$$

$$A\langle \langle \widehat{bv}, \widehat{m}\rangle, \widehat{v}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \rightsquigarrow T\langle \lfloor \star \rfloor, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \quad \textit{where} \quad \lfloor \star \rfloor \sqsubseteq \widehat{bv}$$

$$\underbrace{A\langle \langle \widehat{bv}, \widehat{m}\rangle, \widehat{v}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle}_{\widehat{\varsigma}} \rightsquigarrow E\langle e, \widehat{\rho}[x \mapsto \widehat{\alpha}], \widehat{pc}, \widehat{\sigma}', \widehat{\kappa\alpha}\rangle$$

$$\textit{where} \quad \lfloor \langle \lambda x.\ e, \widehat{\rho}\rangle \rfloor \sqsubseteq \widehat{bv} \qquad \widehat{\sigma}' = \widehat{\sigma} \sqcup \{\widehat{\alpha} \mapsto \widehat{v}\} \qquad \widehat{\alpha} = \widehat{\text{alloc}}(\widehat{\varsigma})$$

$$\underbrace{A\langle \langle \widehat{bv}, \{\widehat{\ell} \mapsto v_1\} \uplus \widehat{m}\rangle, \widehat{v}_2, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle}_{\widehat{\varsigma}} \rightsquigarrow A\langle \widehat{v}_1, \widehat{v}_2, \widehat{pc}', \widehat{\sigma}', \widehat{\kappa\alpha}'\rangle$$

$$\textit{where} \quad \widehat{pc}' = \widehat{pc} \cup \{\widehat{\ell}\} \qquad \widehat{\kappa\alpha}' = \widehat{\text{alloc}}(\widehat{\varsigma}) \qquad \widehat{\kappa} = \langle \widehat{\ell} : \square\rangle :: \widehat{\kappa\alpha} \qquad \widehat{\sigma}' = \widehat{\sigma} \sqcup \{\widehat{\kappa\alpha}' \mapsto \lfloor \widehat{\kappa} \rfloor\}$$

$$\underbrace{T\langle \widehat{v}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle}_{\widehat{\varsigma}} \rightsquigarrow E\langle e, \widehat{\rho}, \widehat{pc} \cup \widehat{pc}', \widehat{\sigma}', \widehat{\kappa\alpha}''\rangle$$

$$\textit{where} \quad \begin{cases} \langle \widehat{pc}'\ ?\ \square \diamond E\langle e, \widehat{\rho}, \widehat{pc}\rangle\rangle :: \widehat{\kappa\alpha}' \in \widehat{\sigma}(\widehat{\kappa\alpha}) & \widehat{\kappa} = \langle pc'\ ?\ \widehat{v} \diamond \square\rangle :: \widehat{\kappa\alpha}' \\ & \widehat{\kappa\alpha}'' = \widehat{\text{alloc}}(\widehat{\varsigma}) \quad \widehat{\sigma}' = \widehat{\sigma} \sqcup \{\widehat{\kappa\alpha}'' \mapsto \lfloor \widehat{\kappa} \rfloor\} \end{cases}$$

$$T\langle \widehat{v}_1, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \rightsquigarrow T\langle \langle\!\langle \widehat{pc}' : \widehat{v}_1 \sqcup \widehat{v}_2 \rangle\!\rangle, \widehat{\sigma}, \widehat{\kappa\alpha}'\rangle \quad \textit{where} \quad \langle \widehat{pc}'\ ?\ \widehat{v}_2 \diamond \square\rangle :: \widehat{\kappa\alpha}' \in \widehat{\sigma}(\widehat{\kappa\alpha})$$

$$T\langle \widehat{v}, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \rightsquigarrow T\langle \langle\!\langle \widehat{pc}' : \widehat{v} \rangle\!\rangle, \widehat{\sigma}, \widehat{\kappa\alpha}'\rangle \quad \textit{where} \quad \langle \widehat{pc}' : \square\rangle :: \widehat{\kappa\alpha}' \in \widehat{\sigma}(\widehat{\kappa\alpha})$$

$$T\langle \widehat{v}_1, \widehat{\sigma}, \widehat{\kappa\alpha}\rangle \rightsquigarrow T\langle \widehat{\text{obs}}(\widehat{pc}', \widehat{v}_1, \widehat{v}_2), \widehat{\sigma}, \widehat{\kappa\alpha}'\rangle \quad \textit{where} \quad O\langle \widehat{pc}', \square, \widehat{v}_2\rangle :: \widehat{\kappa\alpha}' \in \widehat{\sigma}(\widehat{\kappa\alpha})$$

Fig. 7. Coarse Abstract Small-step Semantics

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 ($\widehat{val}$), and disjointly, maps continuation addresses ($\widehat{\kappa\alpha}$) to sets of stacks (i.e., continuations, contexts). 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 4.2.

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.

Atom expression evaluation $\mathcal{A}[\![\cdot]\!]$ 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 7 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 the application for $\widehat{\star}$. The second *A* rule 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 corresonding 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.

4.4 A Branch-sensitive Abstraction via Singleton-analysis

In section 4.1 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 to preserving facet structure in a sound manner, but 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, then a branch-sensitive abstract faceted value approximating $\langle \ell_1 \mathbin{?} \mathrm{v}_1^+ \diamond \mathrm{v}_1^- \rangle \sqcup \langle \ell_2 \mathbin{?} \mathrm{v}_2^+ \diamond \mathrm{v}_2^- \rangle$ would necessarily be

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

conflating both branches regardless. This is required for soundness because a successful observation of ℓ_1 is 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—then we can keep its positive and negative branches distinct, sure that interactions with those branches remain precise, wherever $\widehat{\ell}$ guards a faceted value.

Abstract counting is a technique from Might and Shivers [2006] 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: $\mathtt{0}$, 1, or $\mathtt{>1}$.

The crucial observation then, as it applies to faceted values, is that it is sound to represent an abstract facet without merging 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 8 shows how we expand our abstract domain to account for precise facets. The idea is to represent abstract facets as branch-insensitive facets $\langle \widehat{\ell} \mathbin{?} \widehat{\mathrm{v}} \rangle$ when $\widehat{\ell}$'s count is $\mathtt{>1}$ and branch-sensitive facets $\langle \widehat{\ell} \mathbin{?} \widehat{\mathrm{v}^+} \diamond \widehat{\mathrm{v}^-} \rangle$ otherwise. Instead of placing labels inside of $\widehat{\mathrm{pc}}$ (as in Sections 4.1 and 4.3), we add branches $+\widehat{\ell}$ and $\text{-}\widehat{\ell}$ 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 4.2). 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 $\widehat{\mathrm{val}}$ and $\widehat{\mathrm{val}} \times \widehat{\mathrm{val}}$, with the first representing branch-insensitive facets and the second representing branch-sensitive facets. We must maintain the invariant that—whenever $\widehat{\mathrm{m}}(\widehat{\ell})$ 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 $\mathtt{>1}$, the store is traversed to collapse facets whose labels are no longer singleton. As discussed in section 5, 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 corresponding metafunctions in the Figure 8. 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.

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

$$
\widehat{\text{read}}(\widehat{pc}, \widehat{\sigma}, \langle \widehat{bv}, \widehat{m} \rangle) \triangleq \widehat{v} \sqcup \langle \widehat{bv}', \widehat{m}_1 \sqcup \widehat{m}_2 \rangle
$$

$$
\textit{where} \begin{cases}
\widehat{v} = \bigsqcup_{\lfloor \widehat{\alpha} \rfloor \sqsubseteq \widehat{bv}} \widehat{\sigma}(\widehat{\alpha}) & \widehat{m}_1 = \bigcup_{\widehat{\ell} \in \text{dom}(\widehat{\ell})}^{>1} \{\widehat{\ell} \mapsto \widehat{\text{read}}(\widehat{\sigma}, \widehat{m}(\widehat{\ell}))\} \\
\widehat{bv}' = \begin{cases} \lfloor \star \rfloor \textit{ if } \lfloor \star \rfloor \sqsubseteq \widehat{bv} \\ \bot \textit{ otherwise} \end{cases} & \widehat{m}_2 = \bigcup_{\widehat{\ell} \in \text{dom}(\widehat{\ell})}^{=1} \{\widehat{\ell} \mapsto \langle \widehat{\text{read}}(\widehat{pc}, \widehat{\sigma}, \pi_1(\widehat{m}(\widehat{\ell}))) \\ & \quad , \widehat{\text{read}}(\widehat{pc}, \widehat{\sigma}, \pi_2(\widehat{m}(\widehat{\ell}))) \rangle\}
\end{cases}
$$

$$
\widehat{\text{write}}(\widehat{pc}, \langle \widehat{bv}, \widehat{m} \rangle, \widehat{v}) \triangleq \sigma_1 \sqcup \sigma_2 \sqcup \sigma_3 \sqcup \sigma_4
$$

$$
\textit{where} \begin{cases}
\widehat{\sigma}_1 = \bigsqcup_{\lfloor \widehat{\alpha} \rfloor \sqsubseteq \widehat{bv}} \{\widehat{\alpha} \mapsto \langle\!\langle \widehat{pc} \mathbin{?} \widehat{v} \diamond \widehat{\sigma}(\widehat{\alpha}) \rangle\!\rangle\} & \widehat{\sigma}_3 = \bigsqcup_{\widehat{\ell} \in \text{dom}(\widehat{m})}^{=1} \widehat{\text{write}}(\widehat{pc} \cup \{+\widehat{\ell}\}, \pi_1(\widehat{m}(\widehat{\ell})), \widehat{v}) \\
\widehat{\sigma}_2 = \bigsqcup_{\widehat{\ell} \in \text{dom}(\widehat{m})}^{>1} \widehat{\text{write}}(\widehat{pc} \cup \{\pm\widehat{\ell}\}, \widehat{m}(\widehat{\ell}), \widehat{v}) & \widehat{\sigma}_4 = \bigsqcup_{\widehat{\ell} \in \text{dom}(\widehat{m})}^{=1} \widehat{\text{write}}(\widehat{pc} \cup \{-\widehat{\ell}\}, \pi_2(\widehat{m}(\widehat{\ell})), \widehat{v})
\end{cases}
$$

$$
\widehat{\text{obs}}(\widehat{\ell}, \widehat{v}, \langle \widehat{bv}, \widehat{m} \rangle) \triangleq \widehat{v} \sqcup \langle \widehat{bv}, \widehat{m}_1 \sqcup \widehat{m}_2 \sqcup \widehat{m}_3 \rangle
$$

$$
\textit{where} \begin{cases}
\widehat{v} = \begin{cases} \widehat{m}(\widehat{\ell}) \textit{ where } |\gamma(\widehat{\ell})| > 1 \\ \pi_1(\widehat{m}(\widehat{\ell})) \textit{ where } |\gamma(\widehat{\ell})| = 1 \\ \qquad\qquad \wedge \lfloor \texttt{true} \rfloor \sqsubseteq \widehat{v} \\ \pi_2(\widehat{m}(\widehat{\ell})) \textit{ where } |\gamma(\widehat{\ell})| = 1 \\ \qquad\qquad \wedge \lfloor \texttt{false} \rfloor \sqsubseteq \widehat{v} \end{cases} & \widehat{m}_1 = \begin{cases} \widehat{\ell} \mapsto \widehat{m}(\ell) \textit{ where } |\gamma(\widehat{\ell})| > 1 \\ \varnothing \textit{ where } |\gamma(\widehat{\ell})| = 1 \end{cases} \\
& \widehat{m}_2 = \bigcup_{\widehat{\ell}' \in \text{dom}(\widehat{m})}^{>1} \{\widehat{\ell}' \mapsto \widehat{\text{obs}}(\widehat{\ell}, \widehat{v}, \widehat{m}(\widehat{\ell}'))\} \\
& \widehat{m}_3 = \bigcup_{\widehat{\ell}' \in \text{dom}(\widehat{m})}^{=1} \{\widehat{\ell}' \mapsto \langle \widehat{\text{obs}}(\widehat{\ell}, \widehat{v}, \pi_1(\widehat{m}(\widehat{\ell}'))) \\ & \quad , \widehat{\text{obs}}(\widehat{\ell}, \widehat{v}, \pi_2(\widehat{m}(\widehat{\ell}'))) \rangle\}
\end{cases}
$$

$$
\langle \widehat{bv}_1, \widehat{m}_1 \rangle \sqcup \langle \widehat{bv}_2, \widehat{m}_2 \rangle \triangleq \langle \widehat{bv}_1 \sqcup \widehat{bv}_2, \widehat{m}_1' \sqcup \widehat{m}_2' \rangle
$$

$$
\textit{where} \begin{cases}
\widehat{m}_1' = \bigcup_{\widehat{\ell} \in \text{dom}(\widehat{m}_i)}^{>1} \{\widehat{\ell} \mapsto \widehat{m}_1(\widehat{\ell}) \sqcup \widehat{m}_2(\widehat{\ell})\} \\
\widehat{m}_2' = \bigcup_{\widehat{\ell} \in \text{dom}(\widehat{m}_i)}^{=1} \{\widehat{\ell} \mapsto \langle \pi_1(\widehat{m}_1(\widehat{\ell})) \sqcup \pi_1(\widehat{m}_2(\widehat{\ell})) \\ \qquad , \pi_2(\widehat{m}_1(\widehat{\ell})) \sqcup \pi_2(\widehat{m}_2(\widehat{\ell})) \rangle\}
\end{cases}
$$

$$
\langle\!\langle \widehat{\ell} : \langle \widehat{bv}, \widehat{m} \rangle \rangle\!\rangle \triangleq \langle \bot, \{\widehat{\ell} \mapsto \langle \widehat{bv}, \varnothing \rangle \sqcup \widehat{m}(\widehat{\ell})\} \sqcup \widehat{m}_1 \sqcup \widehat{m}_2 \sqcup \widehat{m}_3 \sqcup \widehat{m}_4 \rangle
$$

$$
\textit{where} \begin{cases}
\widehat{m}_1 = \{\widehat{\ell} \mapsto \bigcup_{\widehat{\ell} < \widehat{\ell}' \in \text{dom}(\widehat{m})}^{>1} \{\widehat{\ell}' \mapsto \widehat{m}(\ell')\}\} & \widehat{m}_3 = \{\widehat{\ell} \mapsto \bigcup_{\widehat{\ell} < \widehat{\ell}' \in \text{dom}(\widehat{m})}^{=1} \{\widehat{\ell}' \mapsto \langle \pi_1(\widehat{m}(\ell')) \\ & \qquad\qquad , \pi_2(\widehat{m}(\ell')) \rangle\}\} \\
\widehat{m}_2 = \bigcup_{\widehat{\ell} > \widehat{\ell}' \in \text{dom}(\widehat{m})}^{>1} \{\widehat{\ell}' \mapsto \langle\!\langle \widehat{\ell} : \widehat{m}(\widehat{\ell}') \rangle\!\rangle\} & \widehat{m}_4 = \bigcup_{\widehat{\ell} > \widehat{\ell}' \in \text{dom}(\widehat{m})}^{=1} \{\widehat{\ell}' \mapsto \langle\!\langle \widehat{\ell} \mathbin{?} \pi_1(\widehat{m}(\widehat{\ell}')) \\ & \qquad\qquad \diamond \pi_2(\widehat{m}(\widehat{\ell}')) \rangle\!\rangle\}
\end{cases}
$$

$$
\langle\!\langle \widehat{\ell} \mathbin{?} \langle \widehat{bv}_1, \widehat{m}_1 \rangle \diamond \langle \widehat{bv}_2, \widehat{m}_2 \rangle \rangle\!\rangle \triangleq \langle \bot, \{\widehat{\ell} \mapsto \langle \widehat{v}_1, \widehat{v}_2 \rangle\} \sqcup \widehat{m}_1 \sqcup \widehat{m}_2 \sqcup \widehat{m}_3 \sqcup \widehat{m}_4 \rangle
$$

$$
\textit{where} \begin{cases}
\widehat{v}_1 = \langle \widehat{bv}_1, \varnothing \rangle \sqcup \pi_1(\widehat{m}(\widehat{\ell})) & \widehat{v}_2 = \langle \widehat{bv}_2, \varnothing \rangle \sqcup \pi_2(\widehat{m}(\widehat{\ell})) \\
\widehat{m}_1 = \{\widehat{\ell} \mapsto \bigcup_{\widehat{\ell} < \widehat{\ell}' \in \text{dom}(\widehat{m})}^{>1} \{\widehat{\ell}' \mapsto \widehat{m}(\ell')\}\} & \widehat{m}_3 = \{\widehat{\ell} \mapsto \bigcup_{\widehat{\ell} < \widehat{\ell}' \in \text{dom}(\widehat{m})}^{=1} \{\widehat{\ell}' \mapsto \langle \pi_1(\widehat{m}(\ell')) \\ & \qquad\qquad , \pi_2(\widehat{m}(\ell')) \rangle\}\} \\
\widehat{m}_2 = \bigcup_{\widehat{\ell} > \widehat{\ell}' \in \text{dom}(\widehat{m})}^{>1} \{\widehat{\ell}' \mapsto \langle\!\langle \widehat{\ell} : \widehat{m}(\widehat{\ell}') \rangle\!\rangle\} & \widehat{m}_4 = \bigcup_{\widehat{\ell} > \widehat{\ell}' \in \text{dom}(\widehat{m})}^{=1} \{\widehat{\ell}' \mapsto \langle\!\langle \widehat{\ell} \mathbin{?} \pi_1(\widehat{m}(\widehat{\ell}')) \\ & \qquad\qquad \diamond \pi_2(\widehat{m}(\widehat{\ell}')) \rangle\!\rangle\}
\end{cases}
$$

Fig. 8. A More Precise Abstraction via Counting

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

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

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

Fig. 9. Galois Connection for Abstract Facet Domain

4.5 Correctness via Galois Connections

We now establish the correctness of our precise abstract domain by constructing a Galois connection between precise faceted values and abstract faceted values. We use this Galois connection to state and prove soundness lemmas for abstract metafunctions used in faceted semantics. Finally, we use these lemmas to prove soundness of the abstract interpreter for faceted execution. Our precise domain (Section 4.4) degrades to gracefully to the coarse abstract domain (Section 4.2) in the absence of abstract counting, so we only formalize the precise domain here.

We show the Galois connection for abstract facets in Figure 9. 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 [Nielson et al. 1999].

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.

Lemma 4.1 (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 4.2 (Abstract Meta-operators Soundness). *(1)* $\mathrm{read}(pc, \sigma, v) \in \gamma(\widehat{\mathrm{read}}(\eta(pc), \eta(\sigma), \eta(v)))$, *(2)* $\mathrm{write}(pc, \sigma, v_1, v_2) \in \gamma(\eta(\sigma) \sqcup \widehat{\mathrm{write}}(\eta(pc), \eta(v_1), \eta(v_2)))$ *and (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:

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

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

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

□

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 4.3 (Abstract Semantics Soundness). $\varsigma \rightsquigarrow^{\sigma} \varsigma' \implies \exists \widehat{\varsigma}' \sqsupseteq \eta(\varsigma'). \ \eta(\varsigma) \rightsquigarrow \widehat{\varsigma}'$.

5 IMPLEMENTATION

We have implemented the precise version of our abstract semantics from section 4.4 in Racket. Our implementation expands the language in Figure 1 with `let` binding, k-ary lambdas, several builtin operators, and conditionals, plus various niceties that are straightforward to desugar to this language with a frontend. To efficiently scale our analysis from Section 4.4, we use a combination of optimizations known in the abstract interpretation community.

Our implementation takes as input an S-expression via Racket's `read`, and converts it to an administrative normal form (ANF) [Flanagan et al. 1993b] desugared to the forms in our core language. We then perform an α-conversion pass over this intermediate form and annotate each syntactic form with its syntactic point, so that (for example) x is transformed to x^l. This ensures the analysis does not lose precision when two syntactically distinct terms are equivalent by assigning them a unique identity via labeling. Our alphatized intermediate representation is then injected into an E configuration and passed to a function that runs the analysis to a fixed point.

Unlike our formalism in Section 4.4, our implementation uses global store-widening (developed for CFA by Shivers [1991]) as to avoid the exponential blowup incurred using a naïve store-passing implementation with per-state stores. Global store-widening moves the store from being a component of configurations to instead being a top-level component of the fixpoint. For example, the configuration $E\langle \widehat{e}, \widehat{\rho}, \widehat{pc}, \widehat{\sigma}, \widehat{\kappa} \rangle$ changes to $E\langle \widehat{e}, \widehat{\rho}, \widehat{pc}, \widehat{\kappa} \rangle$, and the transition function takes $\widehat{\sigma}$ as an argument that is threaded through the fixpoint.

Store Widening in the Presence of Abstract Counting. Unfortunately, abstract counting loses all precision in the presence of global store-widening done in the naïve way. The problem is that configurations such as $E\langle \widehat{e}, \widehat{\rho}, \widehat{pc}, \widehat{\kappa} \rangle$ are incomplete in the sense that we cannot be sure, without knowing the store hasn't been updated, that this combination of $\widehat{e}$, $\widehat{\rho}$, $\widehat{pc}$, and $\widehat{\kappa}$ components does not need to be reexamined in the context new store updates. When the semantics is written in store-passing style, the fixed-point operation may simply maintain a worklist on a set of configurations (that include stores). However, when store-widening is incorporated, this worklist-based implementation is no longer sound: the store *also* changes independent of reachable configurations, thus previously-seen configurations must be reexamined under the updated store. This means that any state creating an abstract label $\widehat{\ell}$ in the store will be rerun each time the store is updated—reallocating $\widehat{\ell}$ and, consequently, increasing its count. The second time $\widehat{\ell}$ is allocated in the global store, its count will bump from 1 to >1, and any facets that depend on the count being 1 must be collapsed. Thus, a naïve fixpoint computation entirely undermines the usefulness of our precise abstract domain.

We need the efficiency of a global store while retaining the benefits of our precise abstract domain, and so we switch to a standard *frontier*-style semantics. Instead of computing a fixed point over sets of configurations and a single global store, we calculate a trace of frontiers of worklists and previously-seen states, paired with previously-seen configurations and a per-frontier-step store:

$$\underbrace{\langle \overbrace{\{e_0\}}^{\text{Worklist}}, \overbrace{\{\}}^{\text{Previously-seen states}}, \overbrace{\{\}}^{\text{Per-step store}} \rangle}_{\text{Initial frontier}} \to \cdots \to \underbrace{\langle \{\}, \overbrace{C}^{\text{Final configurations}}, \overbrace{\sigma}^{\text{Final store}} \rangle}$$

Instead of the traditional worklist-style fixed point in the store-passing semantics, our frontier semantics terminates when there are both no new configurations generated and *also* no new changes to the store.

Lazy Count-Based Facet Collapse. In Section 4.4, we show how eager count-based facet collapse is necessary to retain soundness: after the abstract count for an abstract label $\widehat{\ell}$ goes above 1, we must traverse the store and collapse any facets from $\langle\, \widehat{\ell}\, ?\, v^+ \diamond v^- \,\rangle$ to $\langle \widehat{\ell}\, ?\, v^+ \sqcup v^- \rangle$. This is a fairly burdensome operation, as it must be done every time a label is allocated, and much of the store is likely not be impacted by the change in a single label. To improve the performance of our implementation, we instead perform *lazy* count-based facet collapse.

Lazy count-based facet collapse leverages the following insight: although the count of an abstract label may grow to >1, we do not need to change any facets guarded by that label until those facets are actually used in the semantics. Therefore, we change each place the semantics inspects facets to first call out to `lazy-collapse`, a function that takes a facet and the current store and performs facet collapse if necessary. Similarly, because a label in $\widehat{pc}$ may have its count grow to >1 during execution of a branch (thus invalidating the state of the continuation $\widehat{\ell}\, ?\, \widehat{v} \diamond \square$, we are careful to change $\langle\langle \cdot ?\cdot \diamond \cdot \rangle\rangle$ so that it first checks the abstract count and collapses facets as necessary.

6 RELATED WORK AND FUTURE DIRECTIONS

To the best of 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 of information flow, and programming paradigms for information flow.

Information-flow was first formalized by Denning [1976]. In her seminal work on a lattice model for information flow, she outlined challenges and potential solutions to static information-flow checking. Subsequently, Goguen and Meseguer [1982] defined noninterference, which formalized the idea that privileged data should not influence publicly observable outputs. Clarkson and Schneider [2008] later recognized that information-flow properties fit into a class of program properties that 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 [Barthe et al. 2004]. This idea was later extended to what the authors call product programs, and certified using a relational program logic [Barthe et al. 2011]. Other work has used model checking to check noninterference [van der Meyden and Zhang 2007] along with more general hyperproperties [Clarkson et al. 2014].

Of the mechanisms for static information flow, security type systems have gained the most use. First introduced by Volpano and Smith [1997], these type systems augment the binding environment to track the privilege of variables and prevent writes to variables that would violate noninterference. Myers leveraged this idea to produce Jif, a variant of Java with an information-flow type system [Myers 1999]. Security type systems have been subsequently extended to accommodate concurrent programs [Zdancewic and Myers 2003] and flow sensitivity [Hunt and Sands 2006]. Faceted execution does not require annotating the program with security types, but at the expense of losing a static characterization of the program's security in its type system.

Devriese and Piessens [2010] 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 in the program are Alice and Bob, secure 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 privilaged information. When external effects are made (e.g., writing to disc), the runtime can select which variant to use based on a policy. 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 [Bielova et al. 2011], adding declassification in a granular way [Rafnsson and Sabelfeld 2013], and even preventing side-channel attacks [Kashyap et al. 2011].

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 [Austin and Flanagan 2012]. 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.

At the same time, Yang et al. first implemented Jeeves, a language allowing policy-agnostic programming [Yang et al. 2012]. 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. [Austin et al. 2013]. This formulation includes first-class labels, and is the basis for our concrete semantics.

Several other efforts into dynamic analysis for information flow are worth noting. Stefan et al. [2011] 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 [Schmitz et al. 2016] and variants of `LIO` that extend its power to arbitrary monad transformers [Parker 2014]. 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 [Hudak 1986]. This reference count abstraction is useful for understanding when destructive updates can be performed statically. This cardinality analysis was a direct inspiration for Might and Shivers [2006] to produce the abstract counting approach we build on. Independently, Jagannathan et al. [1998] 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. Sergey et al. [2014] detail a modular presentation of cardinality analysis, which leads to a straightforward implementation for higher-order languages on top of Haskell's strictness analysis.

Last, there has been recent work in verifying noninterference through abstract interpretation. Assaf et al. [2017] introduce hypercollecting semantics, which extends the abstract interpretation framework to a "set of sets" interpretation, matching the hyperproperty of interest.

7 CONCLUSION

In this paper, we have presented the first sound and precise abstraction for faceted execution with first-class labels and security policies. We observed several central challenges that emerge when designing abstract faceted values and proposed an approach for handling each.

An analysis of faceted values is useful when it can recover precise facet structure statically. As we observed in Section 4.1, this means that each level in the faceted-value tree must be able to express that it must be a base value, that it must be a faceted value, or that it could be either. It also means that facets on distinct abstract labels must be represented disjunctively when conflated instead of being nested as would seem natural using a more naïve abstraction. Further, an analysis of programs which may generate arbitrarily many dynamic security labels necessarily must permit such labels to be conflated in its approximation. This allowance gives rise to an issue that abstract labels seemingly cannot be both sound and precisely distinguish their positive and negative branches. We resolve this issue by applying abstract counting, an instrumentation of abstract interpretation which tracks the singleton abstractions (most crucially, singleton abstract labels) in the analysis and permits facets of these labels alone to retain their branch sensitivity.

After discussing two abstract domains for faceted values, we detailed an implementation of our approach in Racket. To provide a scalable implementation, we discuss our approach to lazily collapsing facets whose labels are no longer singletons as we encounter them instead of reestablishing the necessary invariant eagerly. We envision applications for our analysis both in the verification

of policy-agnostic programs, but also in optimizing those programs (e.g., in an interpreter or compiler for a policy-agnostic language).

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Mounir Assaf, David A. Naumann, Julien Signoles, Éric Totel, and Frédéric Tronel. 2017. Hypercollecting Semantics and Its Application to Static Analysis of Information Flow. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 874–887. DOI: http://dx.doi.org/10.1145/3009837.3009889

Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 165–178. DOI: http://dx.doi.org/10.1145/2103656.2103677

Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-agnostic Programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '13)*. ACM, New York, NY, USA, 15–26. DOI: http://dx.doi.org/10.1145/2465106.2465121

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *Proceedings of the 17th International Conference on Formal Methods (FM'11)*. Springer-Verlag, Berlin, Heidelberg, 200–214. http://dl.acm.org/citation.cfm?id=2021296.2021319

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations (CSFW '04)*. IEEE Computer Society, Washington, DC, USA, 100–114. DOI: http://dx.doi.org/10.1109/CSFW.2004.17

N. Bielova, D. Devriese, F. Massacci, and F. Piessens. 2011. Reactive non-interference for a browser model. In *2011 5th International Conference on Network and System Security*. IEEE, 97–104. DOI: http://dx.doi.org/10.1109/ICNSS.2011.6059965

Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–284.

M. R. Clarkson and F. B. Schneider. 2008. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*. IEEE, 51–65. DOI: http://dx.doi.org/10.1109/CSF.2008.7

Patrick Cousot and Radhia Cousot. 1977. 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 (POPL '77)*. ACM, New York, NY, USA, 238–252. DOI: http://dx.doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *Journal of logic and computation* 2, 4 (1992), 511–547.

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243.

D. Devriese and F. Piessens. 2010. Noninterference through Secure Multi-execution. In *2010 IEEE Symposium on Security and Privacy (Oakland '10)*. 109–124. DOI: http://dx.doi.org/10.1109/SP.2010.15

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective Pushdown Analysis of Higher-Order Programs. In *International Conference on Functional Programming*. 177–188.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993a. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 237–247. DOI: http://dx.doi.org/10.1145/155090.155113

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993b. The essence of compiling with continuations. In *ACM Sigplan Notices*, Vol. 28. ACM, 237–247.

Thomas Gilray, Michael D. Adams, and Matthew Might. 2016a. Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-flow Analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16)*. ACM, New York, NY, USA, 407–420. DOI: http://dx.doi.org/10.1145/2951913.2951936

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016b. Pushdown Control-flow Analysis for Free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 691–704. DOI: http://dx.doi.org/10.1145/2837614.2837631

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. DOI: http://dx.doi.org/10.1109/SP.1982.10014

Paul Hudak. 1986. A Semantic Model of Reference Counting and Its Abstraction (Detailed Summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*. ACM, New York, NY, USA, 351–363. DOI: http://dx.doi.org/10.1145/319838.319876

Sebastian Hunt and David Sands. 2006. On Flow-sensitive Security Types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 79–90. DOI: http://dx.doi.org/10.1145/1111037.1111045

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. 1998. 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 (POPL '98)*. ACM, New York, NY, USA, 329–341. DOI: http://dx.doi.org/10.1145/268946.268973

V. Kashyap, B. Wiedermann, and B. Hardekopf. 2011. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *2011 IEEE Symposium on Security and Privacy (Oakland '11)*. 413–428. DOI: http://dx.doi.org/10.1109/SP.2011.19

Matthew Might. 2010. Abstract interpreters for free. In *International Static Analysis Symposium (SAS '10)*. Springer, 407–421.

Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 13–25.

Andrew C. Myers. 1999. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 228–241. DOI: http://dx.doi.org/10.1145/292540.292561

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg.

James Parker. 2014. *LMonad: Information Flow Control for Haskell Web Applications*. Master's thesis. University of Maryland, College Park, Maryland.

W. Rafnsson and A. Sabelfeld. 2013. Secure Multi-execution: Fine-Grained, Declassification-Aware, and Transparent. In *2013 IEEE 26th Computer Security Foundations Symposium*. 33–48. DOI: http://dx.doi.org/10.1109/CSF.2013.10

Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. 2016. Faceted Dynamic Information Flow via Control and Data Monads. In *Proceedings of the 5th International Conference on Principles of Security and Trust - Volume 9635 (POST '16)*. Springer-Verlag New York, Inc., New York, NY, USA, 3–23. DOI: http://dx.doi.org/10.1007/978-3-662-49635-0_1

Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Modular, Higher-order Cardinality Analysis in Theory and Practice. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 335–347. DOI: http://dx.doi.org/10.1145/2535838.2535861

Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, NY, USA, 95–106. DOI: http://dx.doi.org/10.1145/2034675.2034688

Ron van der Meyden and Chenyi Zhang. 2007. Algorithmic Verification of Noninterference Properties. *Electronic Notes in Theoretical Computer Science* 168 (2007), 61 – 75. Proceedings of the Second International Workshop on Views on Designing Complex Architectures.

David Van Horn and Matthew Might. 2010a. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 51–62. DOI: http://dx.doi.org/10.1145/1863543.1863553

David Van Horn and Matthew Might. 2010b. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 51–62. DOI: http://dx.doi.org/10.1145/1863543.1863553

Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: a context-free approach to control-flow analysis. In *Proceedings of the European Symposium on Programming (ESOP '10)*, Vol. 6012, LNCS. 570–589.

Dennis M. Volpano and Geoffrey Smith. 1997. A Type-Based Approach to Program Security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '97)*. Springer-Verlag, London, UK, UK, 607–621. http://dl.acm.org/citation.cfm?id=646620.697712

Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 85–96. DOI: http://dx.doi.org/10.1145/2103656.2103669

S. Zdancewic and A. C. Myers. 2003. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. (CSF '13)*. 29–43. DOI: http://dx.doi.org/10.1109/CSFW.2003.1212703