Logic-Flow Analysis of Higher-Order Programs

Matthew Might

Georgia Institute of Technology
mattm@cc.gatech.edu

Abstract

This work presents a framework for fusing flow analysis and theorem proving called *logic-flow analysis* (LFA). The framework itself is the reduced product of two abstract interpretations: (1) an abstract state machine and (2) a set of propositions in a restricted first-order logic. The motivating application for LFA is the safe removal of implicit array-bounds checks without type information, user interaction or program annotation. LFA achieves this by delegating a given task to either the prover or the flow analysis depending on which is best suited to discharge it. Described within are a concrete semantics for continuation-passing style; a restricted, first-order logic; a woven product of two abstract interpretations; proofs of correctness; and a worked example.

Categories and Subject Descriptors   D.3.4 [*Programming Languages*]: Processors—Optimization

General Terms   Languages

Keywords   logic-flow analysis, LFA, static analysis, environment analysis, lambda calculus, CPS, abstract garbage collection, abstract counting, Gamma-CFA first-order logic, theorem proving

1. The idea

The main idea is really the product of two ideas:

1. Theorem prover as oracle to higher-order flow analysis.

2. Higher-order flow analysis as oracle to theorem prover.

The objective of this fusion is to continue pushing beyond the limitations of the k-CFA framework [17].

The key to this weaving is delegation: the tool best suited for an obligation discharges it. For instance, the theorem prover avoids obligations where it might have to induct, *e.g.*, the introduction of a universal quantifier, since that may require user interaction. To accomplish these tasks, the flow analysis is outfitted with specialized abstract counting machinery [11]. Meanwhile, obligations that exceed the capabilities of the flow analysis, such as reasoning about abstract constraints or canonicalization, go to the prover. We call the threaded framework *logic-flow analysis* (LFA).

For robustness, LFA is engineered so that theorem prover failure is not catastrophic. As the power of the theorem prover decreases, LFA's result gracefully degrades toward a ΓCFA-level flow analysis [11].

Motivating application: Array access safety   Proving the safety of indexing into an array serves as a motivating application. Sometimes, safety is syntactically obvious, as in:

```
for (int i = 0; i < a.length; i++)
    println( a[i] );
```

But, in other cases, the access doesn't occur within the scope of an explicit check:

```
for (int i = 0; i < a.length; i++)
    foo( i );  // foo touches a[i].
```

Complicating matters, the function `foo` might even touch the array a through an alias. Sorting out such issues with sufficient precision can quickly overwhelm existing analyses.

At POPL 2006, Tim Sweeney pointed this out when he issued a challenge to develop a robust analysis for the safety of vertex arrays. Vertex arrays are a technique frequently used in graphics programming and demonstrated by the following fragment:
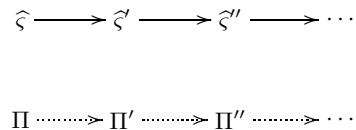
```
float[][3] vertices = ⟨vector of points⟩;
int[]      mesh     = ⟨indices into vertices⟩;

for (int i = 0; i < mesh.length; i++)
    // Safe if 0 ≤ mesh[i] < vertices.length.
    emitv( vertices[mesh[i]] );
```
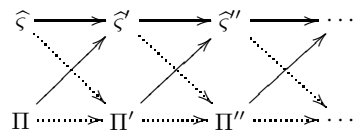
The safety of the access to `vertices` depends upon the manner in which `mesh` is built and modified. Furthermore, at certain points during its lifetime, `mesh` will only partially satisfy the invariants required to prove the safety of the subsequent accesses to `vertices`. Section 6 steps through an example of vertex arrays to show how LFA proves safety under these circumstances.

High-level mechanics   In Section 2, an operational semantics defines a concrete state machine. The analysis then performs *two* abstract interpretations of this machine: one where a concrete machine state (ς) abstracts component-wise into an abstract machine state ($\widehat{\varsigma}$), and one where it abstracts into a set of propositions (Π).

However, rather than run each interpretation independently, as in the following diagram:

$$\widehat{\varsigma} \longrightarrow \widehat{\varsigma}' \longrightarrow \widehat{\varsigma}'' \longrightarrow \cdots$$

$$\Pi \dashrightarrow \Pi' \dashrightarrow \Pi'' \dashrightarrow \cdots$$

the analysis will weave them together, so that the next step of each interpretation is a joint product of the current steps for both:

$$\widehat{\varsigma} \longrightarrow \widehat{\varsigma}' \longrightarrow \widehat{\varsigma}'' \longrightarrow \cdots$$

$$\Pi \dashrightarrow \Pi' \dashrightarrow \Pi'' \dashrightarrow \cdots$$

Through this weaving, the combined interpretation is more precise than running either alone.

$$const \in CONST = \mathbb{Z} + \{\texttt{\#f}, \texttt{\#len}\}$$

$$v \in VAR = \text{an infinite set of identifiers}$$

$$lam \in LAM ::= (\lambda\ (v_1 \cdots v_n)\ call)$$

$$
\begin{aligned}
e, f \in EXP ::=\ & v \\
| \ & const \\
| \ & lam \\
| \ & (prim\ e_1 \cdots e_n)
\end{aligned}
$$

$$
\begin{aligned}
call \in CALL ::=\ & (f\ e_1 \cdots e_n) \\
| \ & (sprim\ e_1 \cdots e_n) \\
| \ & (\texttt{letrec}\ ((v\ lam)^*)\ call)
\end{aligned}
$$

$$
\begin{aligned}
prim &\in PRIM &&= APRIM + HPRIM + REL \\
aprim &\in APRIM &&= \{\texttt{+,-,*,/,}\ldots\} \\
hprim &\in HPRIM &&= \{\texttt{aget}\} \\
rel &\in REL &&= \{\texttt{equal?, <, <=:<,}\ \ldots\} \\
sprim &\in SPRIM &&= \{\texttt{anew, aset!, if, halt}\}
\end{aligned}
$$

Figure 1. A grammar for CPS.

Contributions The chief contributions of this work are:

1. A framework for weaving higher-order flow analyses, first-order logic and theorem proving.

2. Soundness with respect to this weaving.

3. An instantiation of this framework for proving the safety of array-bounds check removal, even in the presence of higher-orderness and continuations.

2. Continuation-passing style (CPS)

Logic-flow analysis operates over a variant of continuation-passing style (Figure 1) extended with basic values, primitive operations, `letrec`, conditionals and a store with arrays.

2.1 Concrete semantics

The concrete semantics is a hybrid call-by-value/call-by-reference state machine for CPS: a *call-by-identity* machine. The *identity* of a value can be either (1) a reference to it, or (2) the value itself if no reference to it yet exists. A *reference* is a globally unique name for a value, such as store location plus an offset, or a variable name plus a *time* at which it was bound.

The shift in perspective pays off because both references and values have a "machine-level" scope; that is, references and values retain their meaning across lexical scopes and even across machine transitions. This, in turn, makes propositions involving these identities meaningful across environments and machine states. For example, informally, we might say, "The value bound to x at time 3 is equal to the value at store location 16 offset 3," or "The value bound to y at time 3 is equal to the value of y at time 4." Note that any machine state and any scope can judge the truth of these propositions.

Later on, to make the analysis computable, propositions will take forms such as: "Any value bound to y at call site 10 is equal to 0," and "Any value ever bound to x is equal to any value ever bound to y." (Yes, this last one implies that x and y have only been bound to one value, but they could've been bound to that value repeatedly, *e.g.*, in a loop.)

Figure 2 gives the domains for a concrete operational semantics. The semantics are a two-stage argument-evaluation/procedure-

$$
\begin{aligned}
\varsigma &\in State &&= Eval + Apply \\
&\ \ \ Eval &&= CALL \times BEnv \times Heap \\
&\ \ \ Apply &&= Proc \times Id^* \times Heap \\
&\ \ \ Heap &&= VEnv \times Store \times Time \\[4pt]
b &\in Bind &&= VAR \times Time \\
\beta &\in BEnv &&= VAR \to Time \\
ve &\in VEnv &&= Bind \to D \\[4pt]
i &\in Index &&= Val \\
\ell &\in Loc &&= \text{an infinite set of locations} \\
arr &\in Array &&= Index \to D \\
\sigma &\in Store &&= Loc \to Array \\[4pt]
bas &\in Bas &&= CONST \\
clo &\in Clo &&= LAM \times BEnv \\
proc &\in Proc &&= Clo + SPRIM \\
val &\in Val &&= Proc + Bas + Loc \\
d &\in D &&= Val \\[4pt]
t &\in Time &&= \text{an infinite set of times (contours)} \\[4pt]
x &\in LogVar &&= \text{a set of logical variables} \\
\iota &\in Id &&::= b \mid d \mid (prim\ \iota_1 \cdots \iota_n) \mid x
\end{aligned}
$$

Figure 2. Concrete domains.

application transition relation $\Rightarrow$ in the set $State \times State$.[1] This machine factors the environment into a lexical binding environment (β) and a $State$-level, "global," binding-to-value environment (ve), as in Shivers' work [16]. Given a variable v, the time $\beta(v)$ is the time that v was bound for the environment β. To retrieve the *value* associated with this binding, we can index into the global environment ve with $ve(v, \beta(v))$. Because of this factoring, the binding $(v, \beta(v))$ acts as a reference for the value $ve(v, \beta(v))$.

The set of identities Id also supplies the terms in the upcoming logic. In the semantics, an identity can be a binding (as explained earlier), a denotable value (if no reference to it is yet available) or a compound identity. A compound identity allows us to describe a value as a function of other identities. Anticipating fusion with the logic, the set Id already includes logical variables; the concrete semantics does not make use of these.

The choice of the set $Time$—the contour set—is left open; for defining the meaning of a program, the naturals suffice. Alternate choices for the abstract set $\widehat{Time}$ later on may require a different choice for the set $Time$ in order to show correctness. For instance, for a k-CFA-level flow analysis, the set $Time$ should be the set of call strings. For Agesen's CPA [1], the set $Time$ should be a sequence of Cartesian products of arguments.

The initial state of a program represented by a call term $call$ is:

$$(call, \bot, \bot, \bot, t_0).$$

Execution proceeds until either a stuck state, or application of the `halt` primitive.

In their definition, the semantics make use of a few auxiliary functions; the first turns an identity into the value it represents:

---

[1] As a shorthand, we decompose states as $(\ldots, ve, \sigma, t)$ instead of $(\ldots, (ve, \sigma, t))$. The domain $Heap$ merely factors out components common to both $Eval$ and $Apply$ states.

Definition 2.1. The function $\mathcal{V}_\varsigma : Id \to D$ obtains the **value of an identity**:

$$\mathcal{V}_\varsigma(b) = ve_\varsigma(b)$$
$$\mathcal{V}_\varsigma(d) = d$$
$$\mathcal{V}_\varsigma[\![(prim\ \iota_1 \cdots \iota_n)]\!] = \mathcal{O}_\varsigma(prim)(\mathcal{V}_\varsigma(\iota_1), \ldots, \mathcal{V}_\varsigma(\iota_n)),$$

where

$$\mathcal{O}_\varsigma[\![\texttt{aget}]\!] = \lambda(\ell, i).\sigma_\varsigma\ \ell\ i$$
$$\mathcal{O}_\varsigma(aprim)\ \text{is the appropriate arithmetic operation}$$
$$\mathcal{O}_\varsigma(rel)\ \text{is the appropriate relation.}$$

The next function converts an expression into an identity:

Definition 2.2. The function $\mathcal{I} : EXP \times BEnv \to Id$ obtains the **identity of an expression**, a $State$-level reference to a value when such a reference is available, and the value itself otherwise:

$$\mathcal{I}(v, \beta) = (v, \beta(v))$$
$$\mathcal{I}(const, \beta) = const$$
$$\mathcal{I}(lam, \beta) = (lam, \beta)$$
$$\mathcal{I}([\![(prim\ e_1\ \cdots e_n)]\!], \beta) = [\![(prim\ \iota_1\ \cdots\ \iota_n)]\!]$$
$$\text{where } \iota_k = \mathcal{I}(e_k, \beta).$$

Loosely, the $\mathcal{V}$ function is to C's pointer-dereference operator '`*`' as the $\mathcal{I}$ function is to the address-of operator, '`&`'.

Definition 2.3. The cases below define the **concrete transition relation**, $\Rightarrow \subseteq State \times State$.

Argument evaluation In an argument-evaluation state, execution has reached the application of a function expression f to arguments $e_1, \ldots, e_n$. The purpose of this transition is to look up the procedure, create a vector of argument identities, and increment the global time:

$$([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, \sigma, t) \Rightarrow (proc, \langle \iota_1, \ldots, \iota_n \rangle, ve, \sigma, t')$$
$$\text{where } \begin{cases} proc = \mathcal{V}_\varsigma(\mathcal{I}(f, \beta)) \\ \iota_k = \mathcal{I}(e_k, \beta) \textbf{ if } \mathcal{V}_\varsigma(\mathcal{I}(e_k, \beta)) \neq \bot \\ t' = t + 1. \end{cases}$$

Procedure application In procedure application, a closure is being applied to a vector of argument identities. Execution proceeds by moving to the call site within the closure, evaluating the identities to values, and updating the environment within the closure for these values:

$$(([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \beta), \iota, ve, \sigma, t) \Rightarrow (call, \beta', ve', \sigma, t)$$
$$\text{where } \begin{cases} \beta' = \beta[v_k \mapsto t] \\ ve' = ve[(v_k, t) \mapsto \mathcal{V}_\varsigma(\iota_k)]. \end{cases}$$

Recursive procedure evaluation In transitioning through `letrec`, the λ terms are closed over the extended environment β' *before* transitioning to the interior call site:

$$([\![(\texttt{letrec}\ ((v\ lam)^*)\ call)]\!], \beta, ve, \sigma, t) \Rightarrow (call, \beta', ve', \sigma, t')$$
$$\text{where } \begin{cases} t' = t + 1 \\ \beta' = \beta[v_k \mapsto t'] \\ ve' = ve[(v_k, t') \mapsto \mathcal{V}_\varsigma(\mathcal{I}(lam_k, \beta'))]. \end{cases}$$

Side-effecting primitive call Calls to side-effecting primitives behave much like argument evaluation, except that there is no need to evaluate the procedure. A side effect can be either a modification to the store, *or* a control-flow effect:

$$([\![(sprim\ e_1 \cdots e_n)]\!], \beta, ve, \sigma, t) \Rightarrow (sprim, \langle \iota_1, \ldots, \iota_n \rangle, ve, \sigma, t')$$
$$\text{where } \begin{cases} \iota_k = \mathcal{I}(e_k, \beta) \textbf{ if } \mathcal{V}_\varsigma(\mathcal{I}(e_k, \beta)) \neq \bot \\ t' = t + 1. \end{cases}$$

Conditional In transitioning through conditionals, the condition is tested against the *false* constant `#f` and the appropriate branch is taken:

$$([\![\texttt{if}]\!], \langle \iota_c, \iota_t, \iota_f \rangle, ve, \sigma, t) \Rightarrow (proc, \langle \rangle, ve, \sigma, t)$$
$$\text{where } proc = \begin{cases} \mathcal{V}_\varsigma(\iota_t) & \mathcal{V}_\varsigma(\iota_c) \neq \texttt{\#f} \\ \mathcal{V}_\varsigma(\iota_f) & \text{otherwise.} \end{cases}$$

Array creation The array-creation primitive allocates a fresh location in the store, inserts the array and applies the continuation to the new location:

$$([\![\texttt{anew}]\!], \langle \iota_{length}, \iota_c \rangle, ve, \sigma, t) \Rightarrow (\mathcal{V}_\varsigma(\iota_c), \langle \ell \rangle, ve, \sigma', t)$$
$$\text{where } \begin{cases} \ell = alloc(\sigma) \\ len = \mathcal{V}_\varsigma(\iota_{length}) \textbf{ if } \mathcal{V}_\varsigma(\iota_{length}) \in \mathbb{N} \\ \sigma' = \sigma[\ell \mapsto [\texttt{\#len} \mapsto len]]. \end{cases}$$

The function $alloc$, of course, returns a fresh location outside the domain of the current store.

Array modification The array-modification primitive inserts an element into the supplied array, if the index is in bounds:

$$\frac{0 \leq i < \sigma\ \ell\ \texttt{\#len}}{([\![\texttt{aset!}]\!], \langle \iota_{loc}, \iota_{ind}, \iota_{val}, \iota_c \rangle, ve, \sigma, t) \Rightarrow (\mathcal{V}_\varsigma(\iota_c), \langle \rangle, ve, \sigma', t)}$$
$$\text{where } \begin{cases} \ell = \mathcal{V}_\varsigma(\iota_{loc}) \neq \bot \\ i = \mathcal{V}_\varsigma(\iota_{ind}) \neq \bot \\ d = \mathcal{V}_\varsigma(\iota_{val}) \neq \bot \\ \sigma' = \sigma[\ell \mapsto (\sigma(\ell))[i \mapsto d]]. \end{cases}$$

3. An abstract space for CPS

This section defines the abstract domains for LFA (Figure 3) and operations upon them. With the exception of the domain $\widehat{Count}$, the structure of these domains is straightforward for a flow analysis by abstract interpretation. The $\widehat{\mu}$ component of each state approximates the number of concrete identities to which each abstract identity corresponds: zero, one or more than one counterparts.

The ability to count (at least to one) becomes important when generating propositions that hold on a state: it is much simpler to make a claim about *all* of the concrete counterparts to an abstract identity if, at the moment, only one such counterpart exists. For example, if two sets A and B are equal, *and* each set is a singleton set, then we can infer that any member of A is equal to any member of B. Note that the approximation $\widehat{\mathbb{N}}$ only counts precisely up to one concrete counterpart. Certainly, we could generalize this to an arbitrary number, but the previous exercise demonstrates that diminishing returns sets in at one. (We can't infer much from two equal sets of size two.)

The choice of the abstract domain $\widehat{Time}$ is left open. For a 0CFA-level analysis, the set $\widehat{Time}$ is a singleton. For a 1CFA-level analysis, the set $\widehat{Time}$ is equal to the set of call sites, and the "next" time is the current call site. For a CPA-level analysis, the set $\widehat{Time}$ is the powerset of sequences of types, and the "next" time is the Cartesian product of the types of the arguments.

Note that abstract identities include both $\widehat{Val}$ and $\widehat{D}$, whereas concrete identities included only D, because $D = Val$.

The concrete-to-abstract mapping The absolute-value notation, $|\cdot|$, denotes "abstraction of," and the symbol $|\cdot|_\alpha$ represents a function in the space $\alpha \to \hat{\alpha}$. These functions define the correspondence between the concrete and the abstract.

For basic values bas, we have $|\texttt{\#len}| = \texttt{\#len}$, $|\texttt{\#f}| = \texttt{\#f}$, $|0| = 0$, $|1| = 1$, or, for $bas > 1$, $|bas| = pos$, and for $bas < 0$, $|bas| = neg$. One could choose a much richer, even infinite, set of basic values should one desire. However, thanks to abstract garbage

$$\widehat{\varsigma} \in \widehat{State} = \widehat{Eval} + \widehat{Apply}$$
$$\widehat{Eval} = CALL \times \widehat{BEnv} \times \widehat{Heap}$$
$$\widehat{Apply} = \widehat{Proc} \times \widehat{Id}^* \times \widehat{Heap}$$
$$\widehat{Heap} = \widehat{VEnv} \times \widehat{Store} \times \widehat{Count} \times \widehat{Time}$$

$$\widehat{\beta} \in \widehat{BEnv} = VAR \to \widehat{Time}$$
$$\widehat{b} \in \widehat{Bind} = VAR \times \widehat{Time}$$
$$\widehat{ve} \in \widehat{VEnv} = \widehat{Bind} \to \widehat{D}$$

$$\widehat{i} \in \widehat{Index} = \widehat{Val}$$
$$\widehat{\ell} \in \widehat{Loc} = \text{a finite set of locations}$$
$$\widehat{arr} \in \widehat{Array} = \widehat{Index} \to \widehat{D}$$
$$\widehat{\sigma} \in \widehat{Store} = \widehat{Loc} \to \widehat{Array}$$

$$\widehat{bas} \in \widehat{Bas} = \{neg, 0, 1, pos, \#\texttt{f}, \#\texttt{len}, \dots\}$$
$$\widehat{clo} \in \widehat{Clo} = LAM \times \widehat{BEnv}$$
$$\widehat{proc} \in \widehat{Proc} = \widehat{Clo} + SPRIM$$
$$\widehat{val} \in \widehat{Val} = \widehat{Proc} + \widehat{Bas} + \widehat{Loc}$$
$$\widehat{d} \in \widehat{D} = \mathcal{P}(\widehat{Val})$$

$$\widehat{\mu} \in \widehat{Count} = (\widehat{Bind} + \widehat{Loc}) \to \widehat{\mathbb{N}}$$
$$\widehat{\mathbb{N}} = \{0, 1, \infty\}$$

$$\widehat{t} \in \widehat{Time} = \text{a finite set of times (contours)}$$

$$\widehat{\iota} \in \widehat{Id} ::= \widehat{b} \mid \widehat{val} \mid \widehat{d} \mid (prim\ \widehat{\iota_1} \cdots \widehat{\iota_n})$$

Figure 3. Abstract domains.

collection (introduced shortly), the finite domains suffice for our purposes. Note that with an infinite set of abstract basic values, widening and narrowing may be required to ensure termination [4].

For the remainder of the concrete domains, the abstraction operation is:

$$|(call, \beta, ve, \sigma, t)|_{Eval} = (call, |\beta|, |ve|, |\sigma|, \mathcal{M}(ve, \sigma), |t|)$$
$$|(proc, \boldsymbol{\iota}, ve, \sigma, t)|_{Apply} = (|proc|, |\boldsymbol{\iota}|, |ve|, |\sigma|, \mathcal{M}(ve, \sigma), |t|)$$
$$|\langle \iota_1, \dots, \iota_n \rangle|_{Id^*} = \langle |\iota_1|_{Id}, \dots, |\iota_n|_{Id} \rangle$$
$$|ve|_{VEnv} = \lambda(v, \widehat{t}). \bigsqcup_{|t| = \widehat{t}} |ve(v, t)|_D$$
$$|\sigma|_{Store} = \lambda\widehat{\ell}. \bigsqcup_{|\ell| = \widehat{\ell}} |\sigma(\ell)|_{Array}$$
$$|arr|_{Array} = \lambda\widehat{i}. \bigsqcup_{|i| = \widehat{i}} |arr(i)|_D$$
$$|(prim\ \iota_1 \cdots \iota_n)|_{Id} = (prim\ |\iota_1| \cdots |\iota_n|)$$

$$|b|_{Id} = |b|_{Bind} \qquad\qquad |d|_{Id} = |d|_D$$
$$|d|_D = \{|d|_{Val}\} \qquad\qquad |sprim|_{Proc} = sprim$$
$$|(lam, \beta)|_{Val} = (lam, |\beta|) \qquad |(v, t)|_{Bind} = (v, |t|)$$
$$|\beta|_{BEnv} = \lambda v.|\beta(v)| \qquad\qquad |clo|_{Proc} = |clo|_D$$

As defined, there is not a straightforward Galois connection with these domains; to see why, consider what the least imprecise concrete counterpart to *pos* is within *Val*. (It doesn't exist.) However,

it is not difficult to generalize the concrete semantics (mostly by making D a powerset of *Val*) to obtain one if desired.

The $\widehat{Count}$-abstractor, $\mathcal{M} : (VEnv \times Store) \to \widehat{Count}$, is

$$\mathcal{M}(ve, \sigma)\ \widehat{b} = \widehat{size}\{b \in dom(ve) : |b| = \widehat{b}\}$$
$$\mathcal{M}(ve, \sigma)\ \widehat{\ell} = \widehat{size}\{\ell \in dom(\sigma) : |\ell| = \widehat{\ell}\}, \text{ where}$$
$$\widehat{size}(S) = \textbf{if } size(S) \in \{0, 1\} \textbf{ then } size(S) \textbf{ else } \infty.$$

For the domain $\widehat{\mathbb{N}}$, the elements 0, 1 and ∞ are incomparable under the order $\sqsubseteq$, and the function $\oplus : \widehat{\mathbb{N}} \times \widehat{\mathbb{N}} \to \widehat{\mathbb{N}}$ is the natural abstraction of addition.

The abstract semantics need the following definitions:

Definition 3.1. The function $\widehat{\mathcal{V}_{\widehat{\varsigma}}} : \widehat{Id} \to \widehat{D}$ obtains the **value of an abstract identity**:

$$\widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{b}) = \widehat{ve}_{\widehat{\varsigma}}(\widehat{b})$$
$$\widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{d}) = \widehat{d}$$
$$\widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{val}) = \{\widehat{val}\}$$
$$\widehat{\mathcal{V}}_{\widehat{\varsigma}}[\![(prim\ \widehat{\iota_1} \cdots \widehat{\iota_n})]\!] = \widehat{\mathcal{O}}_{\widehat{\varsigma}}(prim)(\widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota_1}), \dots, \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota_n})),$$

where:

$$\widehat{\mathcal{O}}_{\widehat{\varsigma}}[\![\texttt{aget}]\!] = \lambda(\widehat{d_1}, \widehat{d_2}). \bigsqcup_{\widehat{\ell} \in \widehat{d_1}} \bigsqcup_{\widehat{i} \in \widehat{d_2}} \widehat{\sigma}_{\widehat{\varsigma}}\ \widehat{\ell}\ \widehat{i}$$
$$\widehat{\mathcal{O}}_{\widehat{\varsigma}}(aprim) \text{ is a sound abstraction of } aprim$$
$$\widehat{\mathcal{O}}_{\widehat{\varsigma}}(rel) \text{ is a sound abstraction of } rel.$$

Definition 3.2. The function $\widehat{\mathcal{I}} : EXP \times \widehat{BEnv} \to \widehat{Id}$ obtains the **abstract identity of an expression**, a $\widehat{State}$-level reference to a value when such a reference is available, and the abstract value itself otherwise:

$$\widehat{\mathcal{I}}(v, \widehat{\beta}) = (v, \widehat{\beta}(v))$$
$$\widehat{\mathcal{I}}(const, \widehat{\beta}) = |const|_{Val}$$
$$\widehat{\mathcal{I}}(lam, \widehat{\beta}) = (lam, \widehat{\beta})$$
$$\widehat{\mathcal{I}}([\![(prim\ e_1 \cdots e_n)]\!], \widehat{\beta}) = [\![(prim\ \widehat{\iota_1} \cdots \widehat{\iota_n})]\!]$$
$$\text{where } \widehat{\iota_k} = \widehat{\mathcal{I}}(e_k, \widehat{\beta}).$$

The set of concrete identities to which an abstract identity corresponds is useful in the upcoming logic and in proofs:

Definition 3.3. The **concretization** of an abstract identity $\widehat{\iota}$ with respect to a state ς is the set $Conc\ \varsigma\ \widehat{\iota}$, where:

$$Conc\ \varsigma\ \widehat{b} = \{b : b \in dom(ve_\varsigma) \text{ and } |b| \sqsubseteq \widehat{b}\}$$
$$Conc\ \varsigma\ \widehat{\ell} = \{\ell : \ell \in dom(\sigma_\varsigma) \text{ and } |\ell| \sqsubseteq \widehat{\ell}\}$$
$$Conc\ \varsigma\ \widehat{bas} = \{bas : |bas| \sqsubseteq \widehat{bas}\}$$
$$Conc\ \varsigma\ \widehat{proc} = \{proc : |proc| \sqsubseteq \widehat{proc}\}$$
$$Conc\ \varsigma\ \widehat{d} = \bigcup_{\widehat{val} \in \widehat{d}} Conc\ \varsigma\ \widehat{val}$$
$$Conc\ \varsigma\ [\![(prim\ \widehat{\iota})]\!] = \{[\![(prim\ \boldsymbol{\iota})]\!] : \iota_k \in Conc\ \varsigma\ \widehat{\iota_k}\}.$$

Any counter $\widehat{\mu}$ naturally extends to abstract identities of all kinds:

$$\widehat{\mu}(\widehat{bas}) = \widehat{size}(Conc \top \widehat{bas})$$

$$\widehat{\mu}(lam, \widehat{\beta}) = \max\big(\{\widehat{\mu}(v, \widehat{\beta}(v)) : v \in free(lam)\} \cup \{1\}\big)$$

$$\widehat{\mu}[\![(prim \ \widehat{\iota})]\!] = \max\{1, \widehat{\mu}(\widehat{\iota_1}), \ldots, \widehat{\mu}(\widehat{\iota_n})\}$$

$$\widehat{\mu}\{\widehat{val_1}, \ldots, \widehat{val_n}\} = \begin{cases} 0 & n = 0 \\ \widehat{\mu}(\widehat{val_1}) & n = 1 \\ \infty & n \geq 2. \end{cases}$$

With the ability to count abstractly, a tighter connection between abstract and concrete knowledge becomes possible:

Lemma 3.1 (Counting). *If $|\varsigma| \sqsubseteq \widehat{\varsigma}$, then $\widehat{size}(Conc \ \varsigma \ \widehat{\iota}) = \widehat{\mu}_{\widehat{\varsigma}}(\widehat{\iota})$.*

3.1 Abstract garbage collection

With abstract garbage collection, unreachable bindings and store locations are re-allocated as fresh. This prevents merging in the abstract, and it boosts both the precision *and* the speed of the analysis simultaneously. The correctness of this technique is addressed elsewhere [11].

The abstract semantics for LFA feature a built-in lazy collector, which waits until precision loss is otherwise imminent before trying to garbage collect a resource. Naturally, this collector requires a definition of what it means for an abstract identity to be reachable from some abstract state; reachability, in turn, requires the concept of *touching*:

Definition 3.4. An abstract identity $\widehat{\iota_1}$ **touches** another abstract identity $\widehat{\iota_2}$ in an abstract state $\widehat{\varsigma}$ iff $\widehat{\iota_2} \in \widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{\iota_1})$, where:

$$\widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{bas}) = \emptyset$$

$$\widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{b}) = \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{b})$$

$$\widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{\ell}) = \bigcup_{i=0}^{\infty} \widehat{\mathcal{V}}_{\widehat{\varsigma}}[\![(\texttt{aget} \ \widehat{\ell} \ |i|)]\!]$$

$$\widehat{\mathcal{T}}_{\widehat{\varsigma}}(lam, \widehat{\beta}) = \{(v, \widehat{\beta}(v)) : v \in free(lam)\}$$

$$\widehat{\mathcal{T}}_{\widehat{\varsigma}}[\![(prim \ \widehat{\iota_1} \cdots \widehat{\iota_n})]\!] = \{\widehat{\iota_1}, \ldots, \widehat{\iota_n}\} \cup \widehat{\mathcal{V}}_{\widehat{\varsigma}}[\![(prim \ \widehat{\iota_1} \cdots \widehat{\iota_n})]\!]$$

$$\widehat{\mathcal{T}}_{\widehat{\varsigma}}\{\widehat{val_1}, \ldots, \widehat{val_n}\} = \widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{val_1}) \cup \cdots \cup \widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{val_n}).$$

Touching extends naturally to an abstract state $\widehat{\varsigma}$:

$$\widehat{\mathcal{T}}_{\widehat{\varsigma}}(call, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t}) = \{(v, \widehat{\beta}(v)) : v \in free(call)\}$$

$$\widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{proc}, \widehat{\iota}, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t}) = \widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{proc}) \cup \bigcup_{k} \widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota_k})).$$

An abstract identity is reachable from a state if there is a chain of touching from the state to the identity:

Definition 3.5. The identities **reachable** from an abstract state $\widehat{\varsigma}$, written $\widehat{\mathcal{R}}(\widehat{\varsigma})$, is the set $\{\widehat{\iota} : \widehat{\iota}_{root} \in \widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{\varsigma}) \text{ and } \widehat{\iota}_{root} \leadsto_{\widehat{\varsigma}}^{*} \widehat{\iota}\}$, where $\widehat{\iota_1} \leadsto_{\widehat{\varsigma}} \widehat{\iota_2}$ iff $\widehat{\iota_2} \in \widehat{\mathcal{T}}_{\widehat{\varsigma}}(\widehat{\iota_1})$.

4. A logic for concrete states

This section builds a restricted, first-order logic for propositions that describe a *concrete* state. Apart from the lack of an existential quantifier, the inclusion of a ranged universal quantifier, and the requirement that all propositions be in prenex normal form, this logic is a standard first-order logic [5]. Three factors led to these restrictions: (1) the safety proofs of concern require only universal quantification; (2) the theorem prover's behavior becomes more predictable when restricted; and (3) abstract garbage collection might destroy the witness for an existential quantification, which forces existentially quantified propositions to be discarded.

Controlling the state-space explosion Within CPS, abstract garbage collection plays a role in controlling statespace explosion. Continuations, or rather, the abstract bindings and locations which become associated with them, are candidates for garbage collection. Consequently, before invoking a function, it is frequently possible to sharpen its continuation argument via a GC step. Once collected, continuations do not merge in the abstract. Hence, the abstract interpretation returns directly to its proper return point, *instead of* forking to the return points of all previous callers of that function. More precisely, abstract garbage collection of continuations leads to polyvariant control-flow behavior.

Consider a call site $(f \ \ldots \ q)$ with continuation q. Suppose that when the abstract interpretation reaches this point, f is bound to a closure with λ term $(\lambda \ (\ldots \ k) \ \ldots)$. Under what circumstances can the analysis collect the binding for k?

If this call is a self tail-call, so that $q = k$, then there is no need to collect as the "merging" will not harm precision. If this call is recursive but not tail recursive, then k will merge with the internal continuation, q. Consequently, it will look as though an internal, recursive call to f could return to an external call to f and *vice versa*. Fortunately, this is only a minor detriment to precision. Moreover, even this internal/external merging disappears when moving from a 0CFA contour set to a 1CFA contour set. Lastly, if this call is an external (non-recursive) call to the function f, then unless the binding to k was previously captured by a `call/cc`-level continuation, the binding to k will be eligible for garbage collection, and this holds even in a 0CFA-level flow analysis.

As a result, in all but the pathological case of *unrestricted* usage of `call/cc`, control-flow polyvariance is achieved. (Several constrained usage patterns for `call/cc` still achieve polyvariance.) It is this polyvariance that is responsible for chopping off the spurious branches of the interpretation that lead to statespace explosion and blurred precision.

4.1 Syntax for propositions

The state logic includes basic propositions (ϕ), quantified propositions (ψ) and assumption bases (Π):

$$
\begin{array}{lcll}
\phi & \in & \Phi & ::= & (= \iota_1 \ \iota_2) \\
& & & | & (\texttt{not} \ \phi) \\
& & & | & (\texttt{or} \ \phi_1 \ \phi_2) \\
\\
\psi & \in & \Psi & ::= & \phi \\
& & & | & (\texttt{forall} \ x : \widehat{\iota} \ \psi) \\
\\
\Pi & \in & Assms & \subseteq & \Psi.
\end{array}
$$

Universally quantified logical variables are restricted to the concrete values of some abstract identity. At first glance, it seems that there is no relation in this logic other than equality; this is because additional relations are encoded as functions mapping to truth values. A shorthand (desugared below) lets us use the more familiar notation for relations in logic.

For later convenience, the expression $\widehat{ids}(\psi)$ represents the set of abstract identities used within a proposition.

4.2 Structure and interpretation

The *terms* in this logic are identities Id from the concrete semantics. Now we'll finally make use of the logical variables included earlier. States themselves define the *structure* of the logic. The interpretation of a term ι in structure ς is the value $\mathcal{V}_{\varsigma}(\iota)$. Consequently, the interpretation of a primitive operator $\mathcal{O}_{\varsigma}(prim)$ is its

conventional meaning; *e.g.*, $\mathcal{O}_\varsigma[\![+]\!] = \lambda(a,b).a+b$. Note that interpretations of a term are denotable values, which makes the domain D the *universe of discourse*.

Given a state ς, an *interpretation*, $\mathfrak{I}$, is a pair (ς, ρ) where $\rho : LogVar \to D$ maps from free logic variables to values. The notation $\mathfrak{I}_\varsigma$ is shorthand for the interpretation $(\varsigma, \bot)$; and the notation $\mathfrak{I}[x \mapsto d]$ is shorthand for $(\varsigma, \rho[x \mapsto d])$ where $\mathfrak{I} = (\varsigma, \rho)$. Lastly, when $\mathfrak{I} = (\varsigma, \rho)$:

$$\mathfrak{I}(\iota) = \begin{cases} \rho(\iota) & \iota \in LogVar \\ \mathcal{V}_\varsigma(\iota) & \text{otherwise.} \end{cases}$$

Definition 4.1. An interpretation $\mathfrak{I}$ **justifies** a proposition ψ iff $\mathfrak{I} \models \psi$, where:

- $\mathfrak{I} \models (=\iota_1\ \iota_2)$ iff $\mathfrak{I}(\iota_1) = \mathfrak{I}(\iota_2)$.
- $\mathfrak{I} \models (\texttt{not}\ \phi)$ iff it is not the case that $\mathfrak{I} \models \phi$.
- $\mathfrak{I} \models (\texttt{or}\ \phi_1\ \phi_2)$ iff $\mathfrak{I} \models \phi_1$ or $\mathfrak{I} \models \phi_2$.
- $\mathfrak{I} \models (\texttt{forall}\ x : \widehat{\iota}\ \psi)$ iff
  for each $\iota \in Conc\ \varsigma\ \widehat{\iota}$, it is the case that $\mathfrak{I}[x \mapsto \mathfrak{I}(\iota)] \models \psi$.

Justification then extends naturally across sets of states and sets of propositions:

- For a set of states Σ, $\Sigma \models \psi$ iff for each state $\varsigma \in \Sigma$, $\mathfrak{I}_\varsigma \models \psi$.
- For an assumption base Π, $\mathfrak{I} \models \Pi$ iff for each proposition $\psi \in \Pi$, $\mathfrak{I} \models \psi$.

For proving the correctness of interacting with a theorem prover, we'll need the notion of *entailment*:

Definition 4.2. An assumption base Π **entails** a proposition ψ, written $\Pi \models \psi$, iff $\mathfrak{I} \models \Pi$ implies $\mathfrak{I} \models \psi$.

In other words, an assumption base entails a proposition if all valid interpretations of the assumption base justify the proposition. The correctness of the analysis also needs the notion of *correspondence*—a relationship between a concrete state, an abstract state and an assumption base:

Definition 4.3. A triple $(\varsigma, \widehat{\varsigma}, \Pi)$ constitutes a **correspondence**, denoted $Cor(\varsigma, \widehat{\varsigma}, \Pi)$, iff $|\varsigma| \sqsubseteq \widehat{\varsigma}$ and $\mathfrak{I}_\varsigma \models \Pi$.

Using correspondence, we can select the set of concrete states that map to a given abstract state *and* satisfy some assumptions:

Definition 4.4. The **filtered concretization** of an abstract state $\widehat{\varsigma}$ under assumption base Π, written $\widehat{\varsigma}/\Pi$, is the set $\{\varsigma : Cor(\varsigma, \widehat{\varsigma}, \Pi)\}$.

This leads to another convenient extension of justification: $(\widehat{\varsigma}, \Pi) \models \psi$ iff $\widehat{\varsigma}/\Pi \models \psi$.

4.3 Syntactic sugar

When used where a proposition is expected, the following desugar:

$$(\neq \iota_1\ \iota_2) \to (\texttt{not}\ (=\iota_1\ \iota_2))$$
$$(rel\ \boldsymbol{\iota}) \to (\neq \texttt{\#f}\ (rel\ \boldsymbol{\iota}))$$
$$(\texttt{implies}\ \phi_1\ \phi_2) \to (\texttt{or}\ (\texttt{not}\ \phi_1)\ \phi_2)$$
$$(\texttt{and}\ \phi_1\ \phi_2) \to (\texttt{not}\ (\texttt{or}\ (\texttt{not}\ \phi_1)\ (\texttt{not}\ \phi_2))).$$

Vector notation quantifies over multiple variables and identities:

$$(\texttt{forall}\ \langle x_1, \ldots, x_n \rangle : \langle \widehat{\iota_1}, \ldots, \widehat{\iota_n} \rangle\ \psi)$$
$$\to (\texttt{forall}\ x_1 : \widehat{\iota_1}$$
$$\quad (\texttt{forall}\ x_2 : \widehat{\iota_2}\ \ldots$$
$$\qquad (\texttt{forall}\ x_n : \widehat{\iota_n}\ \psi)\ldots)).$$

It is often convenient to use an abstract identity where only a concrete identity is syntactically allowed. The convention is that if $\mathcal{C}[\widehat{\iota}]$ is a proposition, where $\mathcal{C}$ is a Felleisen-style [6] one-hole

context of the identity, then this desugars to:

$$(\texttt{forall}\ x : \widehat{\iota}\ \mathcal{C}[x]),$$

where x is fresh. When multiple instances of the same abstract identity occur within a proposition, *each* has its own outer-level universal quantification.

4.4 Syntactic inference rules: Theorem prover as oracle

Rules for syntax-directed reasoning enable interaction with a theorem prover through the concept of a *derivation*:

Definition 4.5. An assumption base Π **derives** a proposition ψ iff there exists a proof of $\Pi \vdash \psi$.

Table 1 gives the core syntactic inference rules for this logic. These rules are complete for combinatorial logic, but due to the restrictions on the logic, they are incomplete in general. Of course, the soundness of the analysis requires the soundness of these rules:

Theorem 4.1 (Syntactic soundness). *If $\Pi \vdash \psi$, then $\Pi \models \psi$.*

Proof. Proofs for rules other than (Int) are standard, following the development found in basic texts [5]. For (Int), assume $\Pi \vdash (\texttt{forall}\ \boldsymbol{x} : \widehat{\boldsymbol{\iota}}\ \phi)$ and $\{\phi\} \vdash \phi'$. Choose any $\mathfrak{I} = (\varsigma, M)$ such that $\mathfrak{I} \models \Pi$. Then we know $\mathfrak{I} \models (\texttt{forall}\ \boldsymbol{x} : \widehat{\boldsymbol{\iota}}\ \phi)$. Now choose any vector $\boldsymbol{\iota}$ such that $\iota_k \in Conc\ \varsigma\ \widehat{\iota_k}$. Let $\mathfrak{I}' = \mathfrak{I}[x_k \mapsto \iota_k]$. We know $\mathfrak{I}' \models \phi$. Thus, $\mathfrak{I}' \models \phi'$. Hence, $\mathfrak{I}' \models (\texttt{and}\ \phi\ \phi')$ and therefore, we have that $\mathfrak{I} \models (\texttt{forall}\ \boldsymbol{x} : \widehat{\boldsymbol{\iota}}\ (\texttt{and}\ \phi\ \phi'))$. $\quad\square$

Having established soundness, a prover can (if desired) emit a verifiable proof tree when it claims that $\Pi \vdash \psi$ holds.

4.5 Semantic inference rules: Flow analysis as oracle

Semantic derivation rules, of the form $(\widehat{\varsigma}, \Pi) \vdash \psi$, obey a tighter soundness theorem:

Theorem 4.2 (Semantic soundness). *If $(\widehat{\varsigma}, \Pi) \vdash \psi$, then $\widehat{\varsigma}/\Pi \models \psi$.*

The proof of this theorem is provided with each nontrivial rule. *Semantic* derivation rules have access to knowledge gathered from the flow analysis, as codified within a state $\widehat{\varsigma}$. As a result, they are strictly more powerful than syntactic rules.

With these rules, the flow analysis acts as an oracle to the prover:

Rule 4.1 (Absence).

$$\frac{|d|_D \not\sqsubseteq \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota})}{(\widehat{\varsigma}, \Pi) \vdash (\texttt{forall}\ x : \widehat{\iota}\ (\neq d\ x))}$$

Proof. By the definition of $\sqsubseteq$ and $|\cdot|$. $\quad\square$

Rule 4.2 (Universal introduction).

$$\frac{\widehat{\mu}_{\widehat{\varsigma}}(\widehat{\iota}) = 1}{(\widehat{\varsigma}, \Pi) \vdash (\texttt{forall}\ \langle x_1, x_2 \rangle : \langle \widehat{\iota}, \widehat{\iota} \rangle\ (= x_1\ x_2))}$$

Proof. By the Counting Lemma. $\quad\square$

Rule 4.3 (Range swap).

$$\frac{(\widehat{\varsigma}, \Pi) \vdash (= \widehat{\iota_1}\ \widehat{\iota_2}) \qquad (\widehat{\varsigma}, \Pi) \vdash (\texttt{forall}\ x : \widehat{\iota_1}\ \psi)}{(\widehat{\varsigma}, \Pi) \vdash (\texttt{forall}\ x : \widehat{\iota_2}\ \psi)}$$

With the Oracle Rule (below), the flow analysis may consult the prover as an oracle, and *vice versa*. By including this rule, LFA can alternate between the flow analysis and the prover in justifying goals:

Rule 4.4 (Oracle).

$$\frac{(\widehat{\varsigma}, \Pi) \vdash \psi_1\ \cdots\ (\widehat{\varsigma}, \Pi) \vdash \psi_n \qquad \Pi \cup \{\psi_1, \ldots, \psi_n\} \vdash \psi'}{(\widehat{\varsigma}, \Pi) \vdash \psi'}$$

Proof. By syntactic soundness. $\quad\square$

$$
\text{(Assm)} \quad \dfrac{\psi \in \Pi}{\Pi \vdash \psi} \qquad
\text{(}\vee\text{Ant)} \quad \dfrac{\Pi \cup \{\phi_1\} \vdash \phi_3 \quad \Pi \cup \{\phi_2\} \vdash \phi_3}{\Pi \cup \{(\text{or } \phi_1 \ \phi_2)\} \vdash \phi_3} \qquad
\text{(Subst)} \quad \dfrac{\Pi \vdash (= \iota \ \iota') \quad \Pi \vdash \psi[\iota/x]}{\Pi \vdash \psi[\iota'/x]}
$$

$$
\text{(Ant)} \quad \dfrac{\begin{array}{c}\Pi \vdash \phi \\ \Pi \subseteq \Pi'\end{array}}{\Pi' \vdash \phi} \qquad
\text{(Cases)} \quad \dfrac{\begin{array}{c}\Pi \cup \{\phi_1\} \vdash \phi_2 \\ \Pi \cup \{(\text{not } \phi_1)\} \vdash \phi_2\end{array}}{\Pi \vdash \phi_2} \qquad
\text{(Contr)} \quad \dfrac{\begin{array}{c}\Pi \cup \{(\text{not } \phi_1)\} \vdash \phi_2 \\ \Pi \cup \{(\text{not } \phi_1)\} \vdash (\text{not } \phi_2)\end{array}}{\Pi \vdash \phi_1}
$$

$$
\text{(Eq)} \quad \dfrac{}{\Pi \vdash (= \iota \ \iota)} \qquad
\text{(}\vee\text{Cons)} \quad \dfrac{\Pi \vdash \phi_1}{\Pi \vdash (\text{or } \phi_1 \ \phi_2), (\text{or } \phi_2 \ \phi_1)} \qquad
\text{(Int)} \quad \dfrac{\Pi \vdash (\text{forall } x : \widehat{\iota} \ \phi) \quad \{\phi\} \vdash \phi'}{\Pi \vdash (\text{forall } x : \widehat{\iota} \ (\text{and } \phi \ \phi'))}
$$

$$
\text{(}\forall\text{Intro)} \quad \dfrac{\Pi \vdash \psi \quad x \notin \mathit{free}(\psi)}{\Pi \vdash (\text{forall } x : \widehat{\iota} \ \psi)} \qquad
\text{(}\forall\text{Swap)} \quad \dfrac{\Pi \vdash (\text{forall } \langle x_1, x_2 \rangle : \langle \widehat{\iota_1}, \widehat{\iota_2} \rangle \ \psi)}{\Pi \vdash (\text{forall } \langle x_2, x_1 \rangle : \langle \widehat{\iota_2}, \widehat{\iota_1} \rangle \ \psi)}
$$

Table 1. Syntactic inference rules.

5. Abstract semantics: LFA

This section defines the analysis LFA as the reduced product of two abstract interpretations. While either interpretation is sound by itself, each serves to enhance the precision of the other when combined. The first interpretation is a straightforward state-machine-based flow analysis plus abstract counting. The second interpretation abstracts each state to a set of propositions holding on that state.

The transition relation $\Mapsto$ in $(\widehat{State} \times Assms) \times (\widehat{State} \times Assms)$ defines the combined interpretation. Running the analysis on a program $call$ consists of exploring this relation when starting from the initial abstract state:

$$((call, \bot, \bot, \bot, \bot, \widehat{t_0}), \{\}).$$

The correctness of LFA is a matter of proving that a correspondence is maintained under transition. The key inductive step for this proof is the following:

Theorem 5.1 ($\Mapsto$ simulates $\Rightarrow$)**.** *If* $Cor(\varsigma, \widehat{\varsigma}, \Pi)$ *and* $\varsigma \Rightarrow \varsigma'$, *there exists a* $(\widehat{\varsigma}', \Pi')$ *such that:* $(\widehat{\varsigma}, \Pi) \Mapsto (\widehat{\varsigma}', \Pi')$ *and* $Cor(\varsigma', \widehat{\varsigma}', \Pi')$.

Proof Outline. The proof for the flow-analysis half is largely straightforward [11]. Except for the few places where this half differs from a straightforward proof, we'll skip discussion of correctness. The other half of the proof, a proof of correctness for the Π'-update rules, is novel and supplied for each nontrivial rule. $\quad \square$

Defining the relation $\Mapsto$ There are many correct ways to define the relation $\Mapsto$. The shortest sound definition is, for instance, $(\widehat{\varsigma}, \Pi) \Mapsto (\top, \emptyset)$.

The concern in this work will be engineering the transition relation so that (1) it fully exploits the information available in the state $\widehat{\varsigma}$ and the assumption base Π; and (2) it explicitly accounts for common programming idioms.

The subsections ahead constitute a case-by-case definition and discussion of the abstract transition, $(\widehat{\varsigma}, \Pi) \Mapsto (\widehat{\varsigma}', \Pi')$. Each subsection contains a pattern describing an abstract state, $\widehat{\varsigma}$, and a form for subsequent states, $\widehat{\varsigma}'$, matching that pattern, like so:

$$\widehat{\varsigma} = \cdots$$
$$\widehat{\varsigma}' = \cdots$$

Each subsection may contain multiple cases, and each case contains rules for computing the new assumption base Π' from the old assumption base Π and the old state $\widehat{\varsigma}$. A guard on the state $\widehat{\varsigma}$ and the assumption base Π for each case determines when that case applies. When guards on cases overlap, the first case has priority.

5.1 Argument evaluation

A state $\widehat{\varsigma}$ is an argument-evaluation state if it is preparing to apply a function expression f to some arguments $e_1, \ldots, e_n$. The purpose of this transition is to look up the set of abstract procedures for the expression f, and to fork the analysis to each one. In the process, each argument e_i is evaluated into an abstract identity:

$$
\begin{aligned}
\widehat{\varsigma} &= ([\![(f \ e_1 \cdots e_n)]\!], \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t}) \\
\widehat{\varsigma}' &= (\widehat{proc}, \langle \widehat{\iota_1}, \ldots, \widehat{\iota_n} \rangle, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t}')
\end{aligned}
$$

$$
\text{where} \begin{cases}
\widehat{proc} \in \widehat{\mathcal{V}_{\widehat{\varsigma}}}(\widehat{\mathcal{I}}(f, \widehat{\beta})) \\
\widehat{\iota_k} = \widehat{\mathcal{I}}(e_k, \widehat{\beta}) \\
\widehat{t}' = \widehat{succ}(\widehat{t})
\end{cases}
$$

The function $\widehat{succ}$ returns an abstract time, and it satisfies following correctness constraint:

$$|t| \sqsubseteq \widehat{t} \implies |t+1| \sqsubseteq \widehat{succ}(\widehat{t}).$$

For instance, for 0CFA precision, only one abstract time exists, so the function $\widehat{succ}$ always returns the same time; for 1CFA, the function $\widehat{succ}$ returns some label for the current call site $[\![(f \ e_1 \cdots e_n)]\!]$ itself.[2]

The subsequent assumption base Π' loses nothing, due to the following rule:

Rule 5.1 (Complete preservation)**.**

$$
\dfrac{\forall \varsigma \in \widehat{\varsigma}/\Pi : ((\varsigma \Rightarrow \varsigma') \text{ implies } \sigma_\varsigma = \sigma_{\varsigma'} \text{ and } ve_\varsigma = ve_{\varsigma'})}{\Pi' \supseteq \Pi}
$$

Proof. Choose any state ς such that $Cor(\varsigma, \widehat{\varsigma}, \Pi)$. Suppose $\varsigma \Rightarrow \varsigma'$. Choose any proposition ψ in Π'. We know that $\mathfrak{I}_\varsigma \models \psi$. Because the relation $\models$ depends only upon the variable environment ve and the store σ, which are identical between states ς and ς', we have that $\mathfrak{I}_{\varsigma'} \models \psi$. $\quad \square$

Several cases below will also achieve a complete preservation of knowledge by avoiding modifications to the variable environment and the store.

5.2 Procedure application: More than zero arguments

The apply transition is the heart of LFA. This is where much of the weaving with the prover happens. It is in this stage that LFA can garbage collect, fork the analysis and expand or contract the assumption base.

The apply transition proceeds through the composition of several smaller transitions—one for each argument passed.[3] Each sub-

---

[2] Since the choice of contour set is not our focus, we use a simplified $\widehat{succ} : \widehat{Time} \to \widehat{Time}$ function. For contour sets beyond 0CFA, the $\widehat{succ}$ operation takes the current state $\widehat{\varsigma}$ in addition to the current time.

[3] In the full proof of correctness for the flow analysis, we need to factor the concrete apply transition similarly and then prove this equivalent to the original definition by induction on the length of the argument vector $\boldsymbol{\iota}$.

transition examines the first identity passed; updates the state and assumption base; and moves to the remaining arguments:

$$\widehat{\varsigma} = (([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \widehat{\beta}), \langle \widehat{\iota}_1, \ldots, \widehat{\iota}_n \rangle, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t})$$
$$\widehat{\varsigma}' = (([\![(\lambda\ (v_2 \cdots v_n)\ call)]\!], \widehat{\beta}'), \langle \widehat{\iota}_2 \ldots, \widehat{\iota}_n \rangle, \widehat{ve}', \widehat{\sigma}, \widehat{\mu}', \widehat{t})$$

$$\text{where} \begin{cases} \widehat{b}_1 = (v_1, \widehat{t}) \\ \widehat{\beta}' = \widehat{\beta}[v_1 \mapsto \widehat{t}] \\ \widehat{ve}' = \cdots \\ \widehat{\mu}' = \cdots \end{cases}$$

Case 5.2.1 ($\widehat{\iota}_1 = \widehat{b}_1$)**.** In this case, the interpretation is rebinding a variable to itself. Consider this case in the concrete. This situation corresponds to having the argument $\iota_1 = (v, t_1)$ and the binding $b_1 = (v, t_2)$, such that $|t_1| = |t_2| = \widehat{t}$. Instead of setting $ve' = ve[(v, t_2) \mapsto ve(v, t_1)]$, the concrete execution could extend only the lexical environment $\beta' = \beta[v \mapsto t_1]$ by mapping this variable v to the *older* time.

The abstraction of *this* concrete transition avoids bumping the allocation counter in the abstract, *i.e.*:

$$\widehat{ve}' = \widehat{ve}$$
$$\widehat{\mu}' = \widehat{\mu}.$$

Note that the abstract lexical environment $\widehat{\beta}'$ remains the same, because $\widehat{\beta}[v \mapsto \widehat{t}] = \widehat{\beta}[v \mapsto |t_1|] = \widehat{\beta}[v \mapsto |t_2|]$. Since $ve = ve'$ and $\sigma = \sigma'$ in the concrete, the Complete Preservation Rule for Π' applies.

This case catches a common higher-order recursion idiom, where a variable is explicitly rebound to itself while recurring, such as the variable f in:

```
(define (map f lst)
  (if (pair? lst)
      (cons (f (car lst)) (map f (cdr lst)))
      '()))
```

In fact, by detecting f's invariance, we can turn this into:

```
(define (map f lst)
  (letrec ((mp (λ (lst)
                 (if (pair? lst)
                     (cons (f (car lst))
                           (mp (cdr lst)))
                     '()))))
    (mp lst)))
```

which allows the argument f to be inlined when `map` is inlined.

Case 5.2.2 $((\widehat{\varsigma}, \Pi) \vdash (= \widehat{\iota}_1\ \widehat{b}_1))$**.** Even if the argument $\widehat{\iota}_1$ is not identical to the binding $\widehat{b}_1$, it may still be the case that the values they represent are equal. Unlike the previous case, there is no clear analog to this in the concrete. It's nonsensical to have a *fresh* binding be equal to the value it's going to be assigned: a *fresh* binding cannot possibly already have a value. In the abstract, however, bindings are a finite resource, and the analysis may be forced into allocating a *stale* binding—one which is already in use. Hence, it's conceivable (and not uncommon) that the abstract value $\widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_1)$ may already be sitting at index $\widehat{b}_1$ within the global environment $\widehat{ve}$. In this case, the analysis can still update the state components as before:

$$\widehat{ve}' = \widehat{ve}$$
$$\widehat{\mu}' = \widehat{\mu}.$$

To prove this behavior correct, we have to modify the concrete semantics so that before binding (v, t), the concrete execution first searches through the domain of the global environment ve for a

binding (v, t') such that $|t'| = |t|$ and $\mathcal{V}_{\varsigma}(v, t) = \mathcal{V}_{\varsigma}(v, t')$. If such a time t' exists, the concrete would instead modify the lexical environment β' so that $\beta' = \beta[v \mapsto t']$ instead of having the variable v map to the current time. Once more, the Complete Preservation Rule applies.

Lastly, note that this case doesn't drive a specific optimization or account for a specific programming idiom so much as it corrects a common source of precision loss for a flow analysis.

Case 5.2.3 ($\widehat{\iota}_1 = \mathcal{C}[\widehat{b}_1]$, $\mathcal{C}$ is invertible, $\widehat{\mu}(\widehat{\iota}_1) = 1$ and $\widehat{b}_1 \notin \widehat{\mathcal{R}}(\widehat{\varsigma})$)**.** In this case, the interpretation is rebinding a variable to an invertible context of itself.[4] Before proceeding, we need to define what an *invertible* context is.

Definition 5.1. A context $\mathcal{C}$ is **invertible** with respect to some term equivalence relation $\equiv$ if for all terms t, there exists a context $\mathcal{C}^{-1}$ such that $\mathcal{C}^{-1}[\mathcal{C}[t]] \equiv t$.

In this context, the equivalence relationship $\widehat{\iota} \equiv \widehat{\iota}'$ is:

$$(\widehat{\varsigma}, \Pi) \vdash (\text{forall}\ \langle x, y \rangle : \langle \widehat{\iota}, \widehat{\iota}' \rangle\ (= x\ y)).$$

In general, an inverse context may not exist, but for most looping idioms, hard-coding rules like the following is sufficient:

$$\widehat{\mu}(\widehat{\iota}) = 1 \text{ and } \mathcal{C} = [\![(\text{+}\ [\ ]\ \widehat{\iota})]\!] \implies \mathcal{C}^{-1} = [\![(\text{-}\ [\ ]\ \widehat{\iota})]\!]$$
$$\mathcal{C} = [\![(\text{cons}\ x\ [\ ])]\!] \implies \mathcal{C}^{-1} = [\![(\text{cdr}\ [\ ])]\!]$$

The first rule covers the `i++` idiom. After the `i++` happens, what the old assumption base Π knew about the binding to `i` has become knowledge of the value `i-1` in the new assumption base Π'. If desired, an algebraic solver can find inverses for other contexts.

To handle invertible rebinding, instances of the binding $\widehat{b}_1$ in the old assumption base Π become the identity $\mathcal{C}^{-1}[\widehat{b}_1]$ in the new assumption base Π':

Rule 5.2.3.1 (Inverse propagation)**.**

$$\frac{\widehat{\iota}_1 = \mathcal{C}[\widehat{b}_1] \qquad \mathcal{C}^{-1} \text{ exists} \qquad \widehat{\mu}(\widehat{\iota}_1) = 1 \qquad \widehat{b}_1 \notin \widehat{\mathcal{R}}(\widehat{\varsigma})}{\Pi' = \Pi[\mathcal{C}^{-1}[\widehat{b}_1]/\widehat{b}_1]}$$

Note that this also acts as the preservation rule for this case. After updating the assumption base Π', the analysis garbage collects the old binding $\widehat{b}_1$ by assigning its new value with a strong update:

$$\widehat{ve}' = \widehat{ve}[\widehat{b}_1 \mapsto \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_1)]$$
$$\widehat{\mu}' = \widehat{\mu}[\widehat{b}_1 \mapsto 1].$$

Case 5.2.4 ($\widehat{\mu}(\widehat{b}_1) \geq 1$ and $\widehat{b}_1 \notin \widehat{\mathcal{R}}(\widehat{\varsigma})$)**.** In this case, the abstract interpretation is about to allocate a stale abstract binding that has become unreachable. As before, the analysis can garbage collect:

$$\widehat{ve}' = \widehat{ve}[\widehat{b}_1 \mapsto \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_1)]$$
$$\widehat{\mu}' = \widehat{\mu}[\widehat{b}_1 \mapsto 1].$$

Again, garbage collection consists of a strong-update overwriting of the abstract value living at index $\widehat{b}_1$ within the variable environment $\widehat{ve}'$. At the same time, the counter $\widehat{\mu}'$ now reflects that the abstract binding $\widehat{b}_1$ corresponds to a single concrete binding. (It is a theorem [11] that if an abstract binding is unreachable, then all of its concrete counterparts are also unreachable.)

Making the collection, however, means that the new assumption base Π' can't preserve propositions that necessarily depend on the binding $\widehat{b}_1$:

Rule 5.2.4.1 (Partial preservation)**.**

$$\frac{(\widehat{\varsigma}, \Pi) \vdash \psi \qquad \widehat{b}_1 \notin \widehat{ids}(\psi)}{\psi \in \Pi'}$$

---

[4] The context $\mathcal{C}$ is Felleisen's [6] one-hole context for the grammar of $\widehat{Id}$.

> **Why would the assumption base contract?** Without a full understanding of the analysis, one might wonder why propositions would ever be discarded. Suppose that the proposition:
>
> $$(\texttt{forall}\ x : (\llbracket \texttt{x} \rrbracket, \widehat{t}_1)\ \psi)$$
>
> is in the current assumption base. This proposition makes a claim about *all* of the concrete counterparts to the abstract binding $(\llbracket \texttt{x} \rrbracket, \widehat{t}_1)$. More specifically, it is making a claim that holds for all values of the variable x when it was bound at times that abstract to $\widehat{t}_1$.
>
>   During the abstract interpretation, it may arrive at a point where it's going to bind x again at time $\widehat{t}_1$. As a result, the set of concrete bindings to which the abstract binding $(\llbracket \texttt{x} \rrbracket, \widehat{t}_1)$ corresponds has expanded. In order to preserve this proposition, the analysis must show that the proposition ψ holds for the new additions to this set. If the assumption base doesn't have enough information to show this, then the analysis cannot preserve the universally quantified proposition.

Case 5.2.5 ($\widehat{\mu}(\widehat{\iota}_1) = 1$ and $\widehat{\mu}(\widehat{b}_1) = 0$). In this case, the abstract binding is fresh, and the identity to which it will be bound has only one concrete counterpart, yielding:

$$\widehat{ve}' = \widehat{ve}[\widehat{b}_1 \mapsto \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_1)]$$
$$\widehat{\mu}' = \widehat{\mu}[\widehat{b}_1 \mapsto 1].$$

After this step, both identities $\widehat{b}_1$ and $\widehat{\iota}_1$ have a single concrete counterpart, so any concrete counterpart of one will be equal to any concrete counterpart of the other in Π':

Rule 5.2.5.1 (Fresh binding).

$$\frac{\widehat{\mu}(\widehat{\iota}_1) = 1 \qquad \widehat{\mu}(\widehat{b}_1) = 0}{(\texttt{forall}\ \langle x_1, x_2 \rangle : \langle \widehat{\iota}_1, \widehat{b}_1 \rangle\ (= x_1\ x_2)) \in \Pi'}$$

In this case, there is also a partial preservation of the assumption base Π, in that the analysis must discard any propositions necessarily involving the binding $\widehat{b}_1$ while constructing the new assumption base Π'. In reality, this costs no precision, as any universally quantified proposition ranging over the empty set would have been both vacuously true and useless. Thus, the Partial Preservation Rule (5.2.4.1) applies.

Case 5.2.6 (Otherwise). If the analysis resorts to this case, it could not handle the binding in a precision-enhancing or -preserving manner. Thus, the analysis must use the weak, merging conservative update:

$$\widehat{ve}' = \widehat{ve} \sqcup [\widehat{b}_1 \mapsto \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_1)]$$
$$\widehat{\mu}' = \widehat{\mu} \oplus (\lambda\_.0)[\widehat{b}_1 \mapsto 1].$$

As before, the analysis can preserve assumptions that don't necessarily involve the binding $\widehat{b}_1$. In this case, the reason is that we would otherwise be expanding the range of a universally quantified variable. And, from $Small \subset Big$ and $\forall x \in Small : \varphi(x)$, we cannot infer $\forall x \in Big : \varphi(x)$. Hence, the Partial Preservation Rule (5.2.4.1) applies.

5.3 Procedure application: Zero arguments

Eventually, the apply transition runs out of arguments, and the analysis transitions with the following:

$$\widehat{\varsigma} = ((\llbracket (\lambda\ ()\ call) \rrbracket, \widehat{\beta}), \langle \rangle, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t})$$
$$\widehat{\varsigma}' = (call, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t})$$

In this case, the Complete Preservation Rule (5.1) applies.

5.4 Recursive procedure evaluation

In LFA, the construct `letrec` behaves much like a specific instance of procedure application. To simplify the presentation, this subsection covers the `letrec` of a single λ term. However, it is not difficult to handle a mutually recursive `letrec` by decomposing the transition as was done in procedure application.

$$\widehat{\varsigma} = (\llbracket (\texttt{letrec}\ ((v\ lam))\ call) \rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t})$$
$$\widehat{\varsigma}' = (call, \widehat{\beta}', \widehat{ve}', \widehat{\sigma}, \widehat{\mu}', \widehat{t}')$$

$$\text{where} \begin{cases} \widehat{t}' = \widehat{succ}(\widehat{t}) \\ \widehat{b} = (v, \widehat{t}') \\ \widehat{\beta}' = \widehat{\beta}[v \mapsto \widehat{t}'] \\ \widehat{\iota} = (lam, \widehat{\beta}') \\ \widehat{\mu}' = \widehat{\mu}[\widehat{b} \mapsto \widehat{n}] \\ strong? = \widehat{b} \notin \widehat{\mathcal{R}}(\widehat{\varsigma}) \text{ or } \widehat{\mu}(\widehat{b}) = 0 \\ \widehat{ve}' = \begin{cases} \widehat{ve}[\widehat{b} \mapsto \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota})] & strong? \\ \widehat{ve} \sqcup [\widehat{b} \mapsto \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota})] & \text{otherwise} \end{cases} \\ \widehat{n} = \begin{cases} 1 & strong? \\ \infty & \text{otherwise.} \end{cases} \end{cases}$$

By replacing the terms $\widehat{\iota}_1$ with $\widehat{\iota}$ and $\widehat{b}_1$ with $\widehat{b}$, this case imports the Fresh Binding Rule (5.2.5.1) and has its own complete preservation rule:

Rule 5.2 (Qualified complete preservation).

$$\frac{(\widehat{\varsigma}, \Pi) \vdash (= \widehat{\iota}\ \widehat{b})}{\Pi' \supseteq \Pi}$$

If the analysis can't preserve all propositions for the new assumption base Π', the Partial Preservation Rule (5.2.4.1) applies for propositions not necessarily involving the binding $\widehat{b}$.

5.5 Side-effecting primitive call

The abstract interpretation handles side-effecting primitive-call states identically to argument-evaluation states, except that there is no need to look up the procedure.

5.6 Conditionals

The handling of conditional transitions depends on how much information is known about the condition:

$$\widehat{\varsigma} = (\llbracket \texttt{if} \rrbracket, \langle \widehat{\iota}_c, \widehat{\iota}_t, \widehat{\iota}_f \rangle, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t})$$
$$\widehat{\varsigma}' = (\widehat{proc}, \langle \rangle, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t})$$

Case 5.6.1 ($(\widehat{\varsigma}, \Pi) \vdash (\neq \widehat{\iota}_c\ \texttt{\#f})$ or $(\widehat{\varsigma}, \Pi) \vdash (= \widehat{\iota}_c\ \texttt{\#f})$). If there is enough information to prove that the condition either must be true or must be false, as in this case, then the abstract interpretation takes only the appropriate branch:

$$\widehat{proc} \in \begin{cases} \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_t) & (\widehat{\varsigma}, \Pi) \vdash (\neq \widehat{\iota}_c\ \texttt{\#f}) \\ \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_f) & (\widehat{\varsigma}, \Pi) \vdash (= \widehat{\iota}_c\ \texttt{\#f}). \end{cases}$$

Clearly, the Complete Preservation Rule applies here.

Case 5.6.2 ($\widehat{\mu}(\widehat{\iota}_c) = 1$). If the analysis can't precisely evaluate the condition, yet its count is 1, then the interpretation forks in both directions. Meanwhile, the true branch asserts the condition in the new assumption base Π', and the false branch asserts its negation.

  Thus, the analysis preserves all knowledge, *and* it adds the following to the true branches' assumption base:

$$(\neq \widehat{\iota}_c\ \texttt{\#f}) \in \Pi',$$

while for the false branches, it adds:

$$(= \widehat{\iota}_c \ \texttt{\#f}) \in \Pi'.$$

The abstract continuation is the join of both continuations:

$$\widehat{proc} \in \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_t) \sqcup \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_f).$$

But, how could the condition have a count of one, *and* have an unknown truth value? In practice, if the analysis were run on a single function, a condition might evaluate to $\top$ if it depends on data outside the scope of the function.

Case 5.6.3 $(\widehat{\mu}(\widehat{\iota}_c) > 1)$**.** In this case, the abstract identity of the condition corresponds to multiple concrete identities. This case is handled identically to the previous one, except that the assumption bases do not expand, *i.e.*, $\Pi' = \Pi$.

Merging forked branches In handling conditionals, the interpretation sometimes had to fork. Left unchecked, this forking could lead to explosion. Once more, abstract garbage collection comes to our rescue. Using the non-lazy abstract garbage collector from previous work on ΓCFA [11], it is possible to merge forked branches. By garbage collecting when each fork hits the joining continuation, it is often the case that their garbage-collected states collapse back into the same state, or into states such that one is more precise than the other. Whenever this is the case, merging happens automatically, and it costs no precision.

5.7 Array creation

For array creation, the analysis attempts to garbage collect the abstract location it's about to allocate. If the abstract location is stale but unreachable, a merging is prevented.

$$\widehat{\varsigma} = (\llbracket \texttt{anew} \rrbracket, \langle \widehat{\iota}_{length}, \widehat{\iota}_c \rangle, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t})$$
$$\widehat{\varsigma}' = (\widehat{proc}, \langle \widehat{\ell} \rangle, \widehat{ve}, \widehat{\sigma}', \widehat{\mu}', \widehat{t})$$

$$\text{where} \begin{cases} \widehat{proc} \in \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_c) \\ \widehat{\ell} = \widehat{alloc}(\widehat{\sigma}) \\ \widehat{\sigma}' = \widehat{\sigma}[\widehat{\ell} \mapsto \widehat{arr} \sqcup [\texttt{\#len} \mapsto \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_{length})]] \\ \widehat{arr} = \begin{cases} \bot & \widehat{\ell} \notin \widehat{\mathcal{R}}(\widehat{\varsigma}) \text{ or } \widehat{\mu}(\widehat{\ell}) = 0 \\ \widehat{\sigma}(\widehat{\ell}) & \text{otherwise} \end{cases} \\ \widehat{\mu}' = \widehat{\mu}[\widehat{\ell} \mapsto \widehat{n}] \\ \widehat{n} = \begin{cases} 1 & \widehat{\ell} \notin \widehat{\mathcal{R}}(\widehat{\varsigma}) \text{ or } \widehat{\mu}(\widehat{\ell}) = 0 \\ \infty & \text{otherwise.} \end{cases} \end{cases}$$

Like the function $\widehat{succ}$, the function $\widehat{alloc}$ is constrained so that:

$$|\sigma| \sqsubseteq \widehat{\sigma} \implies |alloc(\sigma)| \sqsubseteq \widehat{alloc}(\widehat{\sigma}).$$

When allocating an array, a partial preservation rule applies:

Rule 5.3 (Partial preservation, array)**.**

$$\frac{(\widehat{\varsigma}, \Pi) \vdash \psi \qquad \widehat{\ell} \notin \widehat{ids}(\psi)}{\psi \in \Pi'}$$

If the identity $\widehat{\iota}_{length}$ has a single counterpart, *and* the location $\widehat{\ell}$ is fresh, then the new assumption base Π' can chain them together:

Rule 5.4 (Array length chaining)**.**

$$\frac{\widehat{\mu}(\widehat{\iota}_{length}) = 1 \qquad (\widehat{\ell} \notin \widehat{\mathcal{R}}(\widehat{\varsigma}) \text{ or } \widehat{\mu}(\widehat{\ell}) = 0)}{(= \widehat{\iota}_{length} \ (\texttt{aget} \ \widehat{\ell} \ \texttt{\#len})) \in \Pi'}$$

5.8 Array modification

In handling array modification, there are several issues to consider:

1. Arrays are updated one element at a time.

2. Not all elements may satisfy a given property all the time.

3. The properties an array satisfies can change over time.

The analysis can only reason about a finite number of concrete objects with perfect precision at any one moment. An array, however, may contain arbitrarily many elements. This necessitates a mechanism for handling *abstract intervals* of the array, and a way to merge these abstract intervals. For a traditional flow analysis, this task is difficult. For a theorem prover, this task is much simpler once the flow analysis has peeled away the aliasing and the higher-orderness.

The rules in this subsection are concerned primarily with $\texttt{i++}$-style array updates. As a result, the prover will be dealing with closed intervals such as $[i, j]$ and half-open intervals such as $[i, j)$. For steadily expanding or shrinking the interval, the prover can take advantage of lemmas like $[i, j - 1] = [i, j)$. In LFA, the endpoints of these intervals are constrained to be the concrete counterparts to abstract identities.

The transition in this case is:

$$\widehat{\varsigma} = (\llbracket \texttt{aset!} \rrbracket, \langle \widehat{\iota}_{loc}, \widehat{\iota}_{index}, \widehat{\iota}_{val}, \widehat{\iota}_c \rangle, \widehat{ve}, \widehat{\sigma}, \widehat{\mu}, \widehat{t})$$
$$\widehat{\varsigma}' = (\widehat{proc}, \langle \rangle, \widehat{ve}, \widehat{\sigma}', \widehat{\mu}, \widehat{t})$$

$$\text{where} \begin{cases} \widehat{proc} \in \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_c) \\ \widehat{\ell} \in \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_{loc}) \\ \widehat{i} \in \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_{index}) \\ \widehat{d} = \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota}_{val}) \\ \widehat{\sigma}' = \widehat{\sigma}[\widehat{\ell} \mapsto (\widehat{\sigma}(\widehat{\ell})) \sqcup [\widehat{i} \mapsto \widehat{d}]]. \end{cases}$$

The outline for the update of the new assumption base Π' is:

1. Check for a property ϕ_{prop} holding on the value for $\widehat{\iota}_{val}$.

2. Check for intervals adjacent to the index $\widehat{\iota}_{index}$ where the property ϕ_{prop} holds.

3. When found, update the assumption base Π' to reflect the newly expanded abstract interval.

More formally, the prover looks for a property ϕ_{prop} where

$$(\widehat{\varsigma}, \Pi) \vdash \ (\texttt{forall} \ \langle x, y_1, \ldots, y_n \rangle : \langle \widehat{\iota}_{val}, \widehat{\iota}_1, \ldots, \widehat{\iota}_n \rangle \ \phi_{prop})$$

holds.

> **Finding** ϕ_{prop} There are a number of ways to find propositions that qualify for the property ϕ_{prop}. Finding them all is clearly incomputable. A few heuristics, however, focus the search. The easiest approach is to look through the current assumption base Π for occurrences of the identity $\widehat{\iota}_{val}$ in a proposition. Of these propositions, those containing relational primitives (REL) and those generated by a conditional transition are good candidates. If no candidates emerge, the search expands to propositions that use identities equivalent to $\widehat{\iota}_{val}$. If still no candidates emerge, the search is abandoned.

Then, the prover checks to see if the property ϕ_{prop} holds for all elements of an interval:

$$(\widehat{\varsigma}, \Pi) \vdash \begin{array}{l} (\texttt{forall} \ i : \widehat{\mathbb{N}} \\ \quad (\texttt{forall} \ a : \widehat{\iota}_{loc} \\ \quad \quad (\texttt{forall} \ j : \widehat{\iota}_{index} \\ \quad \quad \quad (\texttt{forall} \ \langle y_1, \ldots, y_n \rangle : \langle \widehat{\iota}_1, \ldots, \widehat{\iota}_n \rangle \\ \quad \quad \quad \quad (\texttt{implies} \ \ \phi_{interval} \ \ \phi_{prop}[(\texttt{aget} \ a \ i)/x])))))\end{array}$$

where the guard $\phi_{interval}$ checks whether j is adjacent to a known interval. For catching a start-at-zero-$\texttt{i++}$-style iteration, the interval is:

$$\phi_{interval} = (\texttt{<=} \ 0 \ i \ (\texttt{-} \ j \ 1)).$$

194

Putting this all together yields a rule for handling incremental array update:

Rule 5.5 (Incremental array update)**.**

$$(\widehat{\varsigma}, \Pi) \vdash (\texttt{forall } \langle x, y_1, \ldots, y_n \rangle : \langle \widehat{\iota}_{val}, \widehat{\iota}_1, \ldots, \widehat{\iota}_n \rangle \, \phi_{prop})$$

$$(\widehat{\varsigma}, \Pi) \vdash \begin{array}{l} (\texttt{forall } i : \widehat{\mathbb{N}} \\ \quad (\texttt{forall } a : \widehat{\iota}_{loc} \\ \quad\quad (\texttt{forall } j : \widehat{\iota}_{index} \\ \quad\quad\quad (\texttt{forall } \langle y_1, \ldots, y_n \rangle : \langle \widehat{\iota}_1, \ldots, \widehat{\iota}_n \rangle \\ \quad\quad\quad\quad (\texttt{implies} \quad (\texttt{<= 0 } i \texttt{ (- } j \texttt{ 1))} \\ \quad\quad\quad\quad\quad\quad \phi_{prop}[(\texttt{aget } a \, i)/x]))))) \end{array}$$

$$\begin{array}{l} (\texttt{forall } i : \widehat{\mathbb{N}} \\ \quad (\texttt{forall } a : \widehat{\iota}_{loc} \\ \quad\quad (\texttt{forall } j : \widehat{\iota}_{index} \\ \quad\quad\quad (\texttt{forall } \langle y_1, \ldots, y_n \rangle : \langle \widehat{\iota}_1, \ldots, \widehat{\iota}_n \rangle \qquad \in \Pi' \\ \quad\quad\quad\quad (\texttt{implies} \quad (\texttt{<= 0 } i \, j) \\ \quad\quad\quad\quad\quad\quad \phi_{prop}[(\texttt{aget } a \, i)/x]))))) \end{array}$$

The Partial Preservation Rule for arrays (5.3) also applies.

The incremental rule also serves as a starting point for more general rules. The next rule up the ladder of engineering complexity would be one that looks for abstract intervals not starting at index 0. However, the frequency of the idiom `for (i = 0; i < length; i++)` and its equivalents makes the simple rule widely applicable.

5.9 Termination

A branch of LFA terminates in stuck states; when the `halt` primitive is applied; or when the current state is more precise than (via $\sqsubseteq$) a state already visited while the current assumption base is stronger than (via $\models$) the assumption base associated with the visited state. Formally, a branch terminates if its current state-assumption base pairing is $(\widehat{\varsigma}, \Pi)$ and for some state-assumption base pairing $(\widehat{\varsigma}_v, \Pi_v)$ already visited:

$$\widehat{\varsigma} \sqsubseteq \widehat{\varsigma}_v \text{ and } (\widehat{\varsigma}, \Pi) \models \Pi_v.$$

Of course, the prover $\vdash$ is the approximation for entailment ($\models$).

As defined, termination of the analysis is not guaranteed because (1) the prover, as an external entity, may not halt, and (2) our abstract domains are not finite in one place: the height of the syntax tree for $\widehat{Id}$ is unbounded.

A time limit on the prover removes the termination concern. Even if the prover fails or times out, LFA will still continue, although its precision degrades toward ΓCFA as the assumption bases shrink.

But, what if the prover always fails or times out during the termination check? If this is a concern, the always-terminating approximation to $(\widehat{\varsigma}, \Pi) \models \Pi_v$:

$$\Pi \supseteq \Pi_v \implies (\widehat{\varsigma}, \Pi) \models \Pi_v,$$

eventually leads to termination. Smarter terminating approximations exist, but this is sufficient.

The identity-height concern applies solely to the assumption base, since the identities produced within an abstract state $\widehat{\varsigma}$ are bounded in height by the syntactic expression from which they came. For the assumption base, this concern is removed by bounding the height of an abstract identity's syntax tree at some fixed height h, and pruning identities that break this height with a widening operation [4], $\mathcal{H}_h : \widehat{Id} \to \widehat{Id}$:

$$\mathcal{H}_0 \, \widehat{\iota} = \widehat{\mathcal{V}}_{\widehat{\varsigma}}(\widehat{\iota})$$

$$\mathcal{H}_{h+1} \, \widehat{\iota} = \begin{cases} [\![(prim \ \mathcal{H}_h(\widehat{\iota}_1) \cdots \mathcal{H}_h(\widehat{\iota}_n))]\!] & \widehat{\iota} = [\![(prim \ \widehat{\iota})]\!] \\ \widehat{\iota} & \text{otherwise.} \end{cases}$$

Care must be taken, however, as the pruned identity may be less precise. When the range of a universal quantifier loses precision, the conservative action is to discard the entire proposition.

The alternative to this widening is simplification, wherein the prover symbolically manipulates an identity with the goal of reducing it to a smaller identity. The analysis utilizes this tactic explicitly when dealing with abstract intervals by, for instance, converting $[i, j - 1]$ into $[i, j)$ and $[i + 1, j]$ into $(i, j]$.

5.10 Using imperfect provers

In practice, of course, the analysis uses an imperfect prover, represented by $\vdash^*$, which obeys the following partial completeness properties:

$$(\varsigma, \Pi) \vdash^* \psi \implies (\varsigma, \Pi) \vdash \psi \tag{1}$$

$$\Pi \vdash^* \psi \implies \Pi \vdash \psi. \tag{2}$$

This prover may not be *monotonic*, *i.e.*, the following may not hold:

$$\text{If } \Pi \subset \Pi', \text{ then } \Pi \vdash^* \psi \implies \Pi' \vdash^* \psi.$$

Fortunately, the definition of the relation $|\!\!\Rightarrow$ has soundness in the face of partial completeness and non-monotonicity built-in. The cases in each subsection are ordered by decreasing precision, with latter cases subsuming previous ones. If $(\widehat{\varsigma}', \Pi')$ would have been the subsequent state-assumptions pairing with a perfect prover, then there is at least one subsequent state-assumptions pairing $(\widehat{\varsigma}'_*, \Pi'_*)$ from the prover $\vdash^*$ such that $\widehat{\varsigma}' \sqsubseteq \widehat{\varsigma}'_*$ and $(\widehat{\varsigma}', \Pi') \models \Pi'_*$.

6. Worked example: Vertex arrays

To build a better understanding of how LFA works, we'll trace the analysis for the CPS-translated version of the vertex array code (Figure 4). This example in particular helps to illustrate the inductive interplay between the Inverse Propagation Rule and the Incremental Array Update Rule. Briefly, the code works as follows:

1. Read in a vertex array from disk.

2. Read in the indices (into the vertex array) for a mesh, checking the safety of each index as it is read.

3. Emit the mesh, one vertex at a time.

LFA proves that when the function `read-array` exits, all of the entries within the array `mesh` are valid indices into the array `vertices`. This proves the array access at the call to `emitv` is safe. As we trace, we'll highlight the relevant components of the state $\widehat{\varsigma}$ and the key additions (alen e) to the assumption base Π.

For this example, (alen e) desugars to (aget e #len). In addition, we're using a 1CFA contour set, not because it's required to prove safety, but because it allows us to avoid visiting only special (and perhaps misleading) cases of the rules presented for the analysis. With a 1CFA contour set, the $\widehat{succ}$ function returns the current call site. Note that we've labeled each relevant call site in the example. We jump into the interpretation once LFA has reached the `read-array`$_3$ call site. This puts LFA into the following state:

$$\widehat{\varsigma}_1 = ([\![(\texttt{read-array}_3 \ \texttt{mesh} \ \cdots)]\!], \widehat{\beta}_1, \widehat{ve}_1, \widehat{\sigma}_1, \widehat{\mu}_1, \widehat{t}), \text{ where:}$$

$$\widehat{ve}_1 = [\ldots, ([\![\texttt{verts}]\!], \widehat{t}_0) \mapsto \{\widehat{\ell}_0\}, ([\![\texttt{mesh}]\!], \widehat{t}_1) \mapsto \{\widehat{\ell}_1\}]$$

$$\widehat{\sigma}_1 = [\widehat{\ell}_0 \mapsto [\ldots, \texttt{\#len} \mapsto \{pos\}], \widehat{\ell}_1 \mapsto [\ldots, \texttt{\#len} \mapsto \{pos\}]]$$

$$\widehat{\mu}_1 = [..., \widehat{\ell}_0 \mapsto 1, \widehat{\ell}_1 \mapsto 1, ([\![\texttt{verts}]\!], \widehat{t}_0) \mapsto 1, ([\![\texttt{mesh}]\!], \widehat{t}_1) \mapsto 1]$$

Next, we're in an apply state, applying the closure for `read-array`.

$$\widehat{\varsigma}_2 = (([\![(\lambda \ (\texttt{a i lo hi k}) \ \cdots)]\!], \widehat{\beta}), \widehat{\iota}_2, \ldots)$$

$$\widehat{\iota}_2 = \langle ([\![\texttt{mesh}]\!], \widehat{t}_1), 0, 0, [\![(\texttt{alen } ([\![\texttt{verts}]\!], \widehat{t}_0))]\!], \widehat{clo} \rangle$$

And, now, we're in an eval state, preparing to check whether we've read in the entire mesh:

```
(letrec ((read-verts (λ (a k) ...)) ; Read vertices into A.
       ...
       ; Reads vertex indices into A.
       (read-array (λ (a i lo hi k)
         (if (>=₅ i (alen a))
               k₆
               (λ () (read-int₇ (λ (n)
                 (if (<=:<₈ lo n hi)
                       (λ () (aset!₉ a i n (λ ()
                         (read-array₁₁ a (+ i 1) lo hi k))))))
                     error₁₂)))))

       ; Emit each vertex in a mesh.
       (emit-mesh (λ (vrt msh i k)
         (if (>=₁₃ i (alen msh))
               k₁₄
               (λ () (emitv₁₅ (aget vrt (aget msh i)) (λ ()
                 (emit-mesh₁₇ vrt msh (+ i 1) k)))))))))

 (anew₀ n (λ (verts)
  (anew₁ m (λ (mesh)
   (read-verts₂ verts (λ ()
    (read-array₃ mesh 0 0 (alen verts) (λ ()
     (emit-mesh₄ verts mesh 0 halt)))))))))))))
```

Figure 4. A CPS translation of the vertex array code for rendering a 3D mesh.

$\widehat{\varsigma}_3 = ([\![(\text{if } (>=_5 \text{ i } (\text{alen a})) \ \dots)]\!], \dots)$
After passing through the previous apply state, we picked up information for Π_3:
$(= ([\![\text{mesh}]\!], \widehat{t}_1) ([\![\text{a}]\!], \widehat{t}_3)) \in \Pi_3$
$(= 0 ([\![\text{i}]\!], \widehat{t}_3)) \in \Pi_3$
$(= 0 ([\![\text{lo}]\!], \widehat{t}_3)) \in \Pi_3$
$(= (\text{alen } ([\![\text{verts}]\!], \widehat{t}_0)) ([\![\text{hi}]\!], \widehat{t}_3)) \in \Pi_3$.

Now we're in a conditional apply state:
$\widehat{\varsigma}_4 = ([\![\text{if}]\!], \langle [\![(>= ([\![\text{i}]\!], \widehat{t}_3) (\text{alen } ([\![\text{a}]\!], \widehat{t}_3)))]\!] \rangle, \dots)$
Because $(\widehat{\varsigma}_4, \Pi_4) \vdash (= 0 ([\![\text{i}]\!], \widehat{t}_3))$ and because there is more than one point in the mesh:

$(\widehat{\varsigma}_4, \Pi_4) \vdash (= \text{\#f } (>= ([\![\text{i}]\!], \widehat{t}_3) (\text{alen } ([\![\text{a}]\!], \widehat{t}_3))))$.

Hence, we don't have to fork.

This brings us to the apply state:
$\widehat{\varsigma}_5 = (([\![(\lambda () (\text{read-int}_7 \cdots))]\!], \widehat{\beta}), \dots)$.

Now we're in an eval state:
$\widehat{\varsigma}_6 = ([\![(\text{read-int}_7 (\lambda (n) \dots))]\!], \dots)$.

And, this leads to the following apply state:
$\widehat{\varsigma}_7 = ([\![\text{read-int}_7]\!], \langle \widehat{clo} \rangle, \dots)$.

In the abstract, read-int returns $\{neg, 0, 1, pos\}$, which leads us to the following apply state:
$\widehat{\varsigma}_8 = (([\![(\lambda (n) \dots)]\!], \widehat{\beta}), \langle \{neg, 0, 1, pos\} \rangle, \dots, \widehat{t}_7)$.

Next, we're in in the eval state, preparing to check whether the index we just read in is within the bounds of verts:
$\widehat{\varsigma}_9 = ([\![(\text{if } (<=:<_8 \text{ lo n hi}) \cdots)]\!], \dots)$, where
$\widehat{ve}_9 = [\dots, ([\![\text{n}]\!], \widehat{t}_7) \mapsto \{neg, 0, 1, pos\}]$
$\widehat{\mu}_9 = [\dots, ([\![\text{n}]\!], \widehat{t}_7) \mapsto 1]$.
It helps to recall that:
$(\widehat{\varsigma}_9, \Pi_9) \vdash (= ([\![\text{lo}]\!], \widehat{t}_3) 0)$
$(\widehat{\varsigma}_9, \Pi_9) \vdash (= ([\![\text{hi}]\!], \widehat{t}_3) (\text{alen } ([\![\text{verts}]\!], \widehat{t}_0)))$.

Now, we're in the conditional apply state:
$\widehat{\varsigma}_{10} = ([\![\text{if}]\!], \langle [\![(<=:< ([\![\text{lo}]\!], \widehat{t}_3) ([\![\text{n}]\!], \widehat{t}_7) ([\![\text{hi}]\!], \widehat{t}_3))]\!] \rangle, \dots)$
The prover doesn't have enough information to determine whether or not this condition holds, so the analysis must fork. However, because

$$\mu[\![(<=:< ([\![\text{lo}]\!], \widehat{t}_3) ([\![\text{n}]\!], \widehat{t}_7) ([\![\text{hi}]\!], \widehat{t}_3))]\!] = 1,$$

we can assume the condition holds on the true fork, and that it does not on the false fork.
The false fork terminates quickly without touching verts, so we follow only the true fork.

By this state, we have:
$(<=:< ([\![\text{lo}]\!], \widehat{t}_3) ([\![\text{n}]\!], \widehat{t}_7) ([\![\text{hi}]\!], \widehat{t}_3)) \in \Pi_{11}$.
And, we're applying the true continuation:
$\widehat{\varsigma}_{11} = ([\![(\lambda () (\text{aset!}_9 \cdots))]\!], \langle \rangle, \dots)$.

And, then, we enter the eval state for the array update:
$\widehat{\varsigma}_{12} = ([\![(\text{aset!}_9 \text{ a i n } \cdots)]\!], \dots)$.

Finally, we've reached the point where we update the array:
$\widehat{\varsigma}_{13} = ([\![\text{aset!}]\!], \langle ([\![\text{a}]\!], \widehat{t}_3), ([\![\text{i}]\!], \widehat{t}_3), ([\![\text{n}]\!], \widehat{t}_7), \widehat{clo} \rangle, \dots)$
When we transition from this state, LFA invokes the Incremental Array Update Rule. First, it looks for a fact ψ involving the binding $([\![\text{n}]\!], \widehat{t}_7)$ such that $(\widehat{\varsigma}, \Pi_{13}) \vdash \psi$. Specifically, it looks for a property ϕ such that:
$(\widehat{\varsigma}_{13}, \Pi_{13}) \vdash (\text{forall } \langle x, y_1, \dots, y_n \rangle : \langle ([\![\text{n}]\!], \widehat{t}_7), \widehat{\iota}_1, \dots, \widehat{\iota}_n \rangle \ \phi)$.
Searching through Π_{13} for $([\![\text{n}]\!], \widehat{t}_7)$, the prover finds:
$(<=:< ([\![\text{lo}]\!], \widehat{t}_3) ([\![\text{n}]\!], \widehat{t}_7) ([\![\text{hi}]\!], \widehat{t}_3))$,
from Π_{11}, which, importantly, is equivalent to the safety condition:
$(\text{forall } x : ([\![\text{n}]\!], \widehat{t}_7) (<=:< 0 \ x \ (\text{alen } ([\![\text{verts}]\!], \widehat{t}_0))))$.
Next, the prover tests for an interval starting at index 0 and adjacent to the index $([\![\text{i}]\!], \widehat{t}_3)$ in the array $([\![\text{a}]\!], \widehat{t}_3)$ where the property ϕ holds by testing:

```
(forall k : ℕ̂
 (forall a : ([[a]], t̂₃)
  (forall i : ([[i]], t̂₃)
   (implies (<= 0 k (- i 1))  φ[(aget a k)/x]))))
```

Right now, this interval is [0,-1] (empty), so this property holds trivially. Once found, the prover updates adjacent intervals to *include* this index. In this case, in Π_{13+1}, the interval will become:

$$(<= 0 \ k \ i).$$

Looking forward, we'll eventually hit an application of the Inverse Propagation Rule in a recursive call to read-array, where we increment the index into mesh. When the Inverse Propagation Rule is applied, the interval will be transformed back into:

$$(<= 0 \ k \ (- i 1)).$$

This is *exactly* the precondition required for the next expansion of the range to succeed.
The prover doesn't have to do induction explicitly, and yet induction's trademark signs are apparent in this step. What's *actually* happening is that the flow analysis is gradually breaking the inductive proof into more manageable, flow-specific, context-specific proofs.

Next, we're in an apply state for the continuation to aset!:
$\widehat{\varsigma}_{14} = (([\![(\lambda () (\text{read-array}_{11} \cdots))]\!], \widehat{\beta}), \langle \rangle, \dots)$

Now, we're in an eval state for the recursive call to read-array:
$\widehat{\varsigma}_{15} = ([\![(\text{read-array}_{11} \text{ a (+ i 1) lo hi k})]\!], \dots)$

Next, we're in the corresponding apply state:

$\widehat{\varsigma}_{16} = (([\![(\lambda \ (\texttt{a i lo hi k}) \ \cdots)]\!], \widehat{\beta}), \widehat{\iota}_{16}, \ldots)$, where
$\widehat{\iota}_{16} = \langle ([\![\texttt{a}]\!], \widehat{t}_3), \widehat{\iota}_{\texttt{i++}}, ([\![\texttt{lo}]\!], \widehat{t}_3), ([\![\texttt{hi}]\!], \widehat{t}_3), ([\![\texttt{k}]\!], \widehat{t}_3) \rangle$, and
$\widehat{\iota}_{\texttt{i++}} = [\![(\texttt{+} \ ([\![\texttt{i}]\!], \widehat{t}_3) \ \texttt{1})]\!]$
We have the following new bindings:
$\widehat{\boldsymbol{b}}_{16} = \langle ([\![\texttt{a}]\!], \widehat{t}_{11}), ([\![\texttt{i}]\!], \widehat{t}_{11}), ([\![\texttt{lo}]\!], \widehat{t}_{11}), ([\![\texttt{hi}]\!], \widehat{t}_{11}), ([\![\texttt{k}]\!], \widehat{t}_{11}) \rangle$
Previously, the bindings to these variables were made at time $\widehat{t}_3$, so we suffer no precision loss from merging in $\widehat{ve}$ yet.[5]

At this point, we're again in an eval state, preparing to check whether we've filled the array:

$\widehat{\varsigma}_{17} = ([\![(\texttt{if} \ (\texttt{>=}_5 \ \texttt{i} \ (\texttt{alen a})) \ \ldots)]\!], \ldots)$
As before, we have relevant updates in Π_{17}:

$(= ([\![\texttt{a}]\!], \widehat{t}_3) \ ([\![\texttt{a}]\!], \widehat{t}_{11})) \in \Pi_{17}$
$(= ([\![\texttt{i}]\!], \widehat{t}_{11}) \ (\texttt{+} \ ([\![\texttt{i}]\!], \widehat{t}_3) \ \texttt{1})) \in \Pi_{17}$
$(= ([\![\texttt{lo}]\!], \widehat{t}_3) \ ([\![\texttt{lo}]\!], \widehat{t}_{11})) \in \Pi_{17}$
$(= (\texttt{alen} \ ([\![\texttt{hi}]\!], \widehat{t}_3)) \ ([\![\texttt{hi}]\!], \widehat{t}_{11})) \in \Pi_{17}$
From this, we see that the action of binding serves to chain the equality of identities formed within different environments.

Now we're a conditional apply state:

$\widehat{\varsigma}_{18} = ([\![\texttt{if}]\!], \langle [\![(\texttt{>=} \ ([\![\texttt{i}]\!], \widehat{t}_{11}) \ (\texttt{alen} \ ([\![\texttt{a}]\!], \widehat{t}_{11})))]\!] \rangle, \ldots)$
This time, we cannot determine the truth of the condition, so forking is inevitable. However, because the count of the condition is one, we can assert its truth or falsity on each branch.

Next, we'll delve one state into the branch where the condition holds (that is, where we are done with `read-array`). After that, we'll switch back to the branch where the condition does not hold.

In the case where the condition holds, we added to Π_{19}:

$(\neq \texttt{\#f} \ (\texttt{>=} \ ([\![\texttt{i}]\!], \widehat{t}_{11}) \ (\texttt{alen} \ ([\![\texttt{a}]\!], \widehat{t}_{11})))) \in \Pi_{19}$.
Note that at this point, we can derive the following from $(\widehat{\varsigma}, \Pi_{19})$:

```
(forall k : ℕ̂
 (forall a : ([[mesh]], t̂₁)
  (implies (<=:< 0 k (alen a))
           (<=:< 0 (aget a k) (alen ([[verts]], t̂₀)))))).
```
That is, we have proved that every index in `mesh` contains a valid index within `verts`. As this branch is about to return from `read-array` and enter `emit-mesh`, we'll switch back the false branch from the prior state.

Having switched back to the false branch, we're in an eval state:

$\widehat{\varsigma}_{20} = ([\![(\texttt{read-int}_7 \ (\lambda \ (\texttt{n}) \ \ldots))]\!], \ldots)$.
To avoid tedious repetition, we'll skip straight ahead to the next apply state for `aset!`.

$$\vdots$$

Jumping forward, we're in an apply state:

$\widehat{\varsigma}_{22} = ([\![\texttt{aset!}]\!], \langle ([\![\texttt{a}]\!], \widehat{t}_{11}), ([\![\texttt{i}]\!], \widehat{t}_{11}), ([\![\texttt{n}]\!], \widehat{t}_{11}), \widehat{clo} \rangle, \ldots)$
Just as we did the last time execution reached `aset!`, we'll attempt a generalization over the array. This time, the prerequisite to perform the generalization is *not* vacuously true: the range $[0, (\texttt{-} \ ([\![\texttt{i}]\!], \widehat{t}_{11}) \ \texttt{1})]$ is non-empty. The prerequisite, however, for the array update rule is still derivable from $(\widehat{\varsigma}_{22}, \Pi_{22})$. Conse-

quently, the prover adds the following to the next assumption base:

```
(forall k : ℕ̂
 (forall a : ([[a]], t̂₁₁)
  (forall i : ([[i]], t̂₁₁)
   (implies
    (<= 0 k i)
    (<=:< 0 (aget a k) (alen ([[verts]], t̂₀)))))))
```

Now, we'll jump ahead to the recursive application of `read-array`.

$$\vdots$$

After jumping forward, we're in an apply state:

$\widehat{\varsigma}_{24} = (([\![(\lambda \ (\texttt{a i lo hi k}) \ \cdots)]\!], \widehat{\beta}), \widehat{\iota}_{24}, \ldots)$, where
$\widehat{\iota}_{24} = \langle ([\![\texttt{a}]\!], \widehat{t}_{11}), \widehat{\iota}_{\texttt{i++}}, ([\![\texttt{lo}]\!], \widehat{t}_{11}), ([\![\texttt{hi}]\!], \widehat{t}_{11}), ([\![\texttt{k}]\!], \widehat{t}_{11}) \rangle$, and
$\widehat{\iota}_{\texttt{i++}} = [\![(\texttt{+} \ ([\![\texttt{i}]\!], \widehat{t}_{11}) \ \texttt{1})]\!]$.
We have the following "new" bindings:
$\widehat{\boldsymbol{b}}_{24} = \langle ([\![\texttt{a}]\!], \widehat{t}_{11}), ([\![\texttt{i}]\!], \widehat{t}_{11}), ([\![\texttt{lo}]\!], \widehat{t}_{11}), ([\![\texttt{hi}]\!], \widehat{t}_{11}), ([\![\texttt{k}]\!], \widehat{t}_{11}) \rangle$.
The variables a, lo, hi and k are being rebound to themselves, so their bindings lose no precision. The new binding for i, however, is not to itself—the binding is to itself plus one. If we don't compensate for this, we'll have to throw out assumptions involving the binding $([\![\texttt{i}]\!], \widehat{t}_{11})$—including those about which indices in `mesh` are safe. Fortunately, the binding $([\![\texttt{i}]\!], \widehat{t}_{11})$ is eligible for abstract garbage collection, *i.e.*, $([\![\texttt{i}]\!], \widehat{t}_{11}) \notin \widehat{\mathcal{R}}(\widehat{\varsigma}_{24})$. Hence, the Inverse Propagation Rule applies.

Now, LFA is going to perform the following:

1. Replace $([\![\texttt{i}]\!], \widehat{t}_{11})$ with inverse $[\![(\texttt{-} \ ([\![\texttt{i}]\!], \widehat{t}_{11}) \ \texttt{1})]\!]$ in Π_{24}.

2. Garbage collect $([\![\texttt{i}]\!], \widehat{t}_{11})$ in $\widehat{\varsigma}_{24}$.

3. Add the new binding for $([\![\texttt{i}]\!], \widehat{t}_{11})$ in $\widehat{\varsigma}_{24}$.

What's important to us in this process is that it shifts the interval in `mesh` whose entries are safe from $[0, ([\![\texttt{i}]\!], \widehat{t}_{11})]$ back to $[0, (\texttt{-} \ ([\![\texttt{i}]\!], \widehat{t}_{11}) \ \texttt{1})]$. The causes the analysis to visit a state-assumption base pairing that it's already seen, and hence, it terminate on this branch. Consequently, every path out of `read-array` leaves `mesh` satisfying the requisite safety condition.

7. Related work

This work is embedded in the framework of abstract interpretation laid out by Cousot and Cousot [3]. In addition, the pruning operation $\mathcal{H}$ used to move up the lattice of approximation for convergence is an instance of widening [4]. There are also relationships between LFA's propositions and Miné's work in relational abstract domains [12, 13]. The counting component $\widehat{\mu}$ is similar to the abstract reference counting by Hudak [7] in his static analysis of sharing, except that where his work counts "references to," LFA counts "concrete counterparts to," as in ΓCFA [11]. The propositional abstract interpretation has connections to Ball, *et al.*'s work on predicate abstraction [2].

The flow analysis portion of this work descends directly from Shivers' original work [17] and from earlier work on ΓCFA and ΔCFA [11, 10]. LFA has also been influenced by efforts in improving speed and precision, such as work by Rehof, *et al.* [15] and Agesen [1]. Recently, Meunier, Felleisen and Findler [9] used a theorem prover in their work on a modular set-based analysis with contracts. In that work, the prover is used to determine whether obligations have been met across boundaries. LFA differs in that it requires no contracts or user annotations. Given this, it would be interesting to explore a hybrid modularized, contract-based logic-flow analysis framework.

---

[5] If we were running with a 0CFA contour set, we would garbage collect all of these bindings to prevent merging, since all of them are unreachable.

The logic in this work draws on Ebbinghaus, *et al.*'s introductory text [5]. The method for the correctness of interacting with the theorem prover is inspired by the approach taken in Nanevski, Morrisett and Birkedal's recent work [14], where they used the soundness of an assertion logic to safely employ a theorem prover in type checking.

The notion of a syntactic context descends from Felleisen's work [6]. The notion of an inverse context appears to be novel.

Static analysis for the safety of array-bounds accesses is an old idea, and much progress has been made in the field as of late. Recent work by Jia and Walker [8] has even begun working directly on pointers (as opposed to arrays) through the use of an "Intuitionistic Linear logic with Constraints." However, much of the work in the realm of verifying the safety of array-bounds accesses focuses purely on type systems, logics and theorem proving. LFA's message is that these approaches pick up more power when woven into an abstract interpretation.

8. Future work

Given LFA's connections to relational abstract domains, fusing Miné's work on weakly relational abstract domains [12] and octagon domains [13] with the LFA framework presents a promising avenue for research. Future versions of LFA should also generalize the basic value domain to an infinite lattice, and introduce the appropriate widening and narrowing operations. Given that the choice of the set $\widehat{Time}$ is also left open, it's easy to imagine making it an infinite domain, and once again, applying widening and narrowing to achieve the "appropriate" degree of polyvariance for the input program. Lastly, there is a wealth of work on shape analysis, which could be brought to bear on improving the precision of handling abstract arrays.

Acknowledgments

References

[1] AGESEN, O. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of ECOOP 1995* (1995), pp. 2–26.

[2] BALL, T., MILLSTEIN, T., AND RAJAMANI, S. K. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst. 27*, 2 (2005), 314–343.

[3] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California, Jan. 1977), vol. 4, pp. 238–252.

[4] COUSOT, P., AND COUSOT, R. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,* (1992), M. Bruynooghe and M. Wirsing, Eds., Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, Springer-Verlag, Berlin, Germany, pp. 269–295.

[5] EBBINGHAUS, H.-D., FLUM, J., AND THOMAS, W. *Mathematical Logic*, 2nd ed. Springer-Verlag, New York, 1994.

[6] FELLEISEN, M., AND HIEB, R. A Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science 103*, 2 (1992), 235–271.

[7] HUDAK, P. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, Aug. 1986), pp. 351–363.

[8] JIA, L., AND WALKER, D. ILC: A Foundation for Automated Reasoning About Pointer Programs. In *European Symposium on Programming Languages* (March 2006), pp. 131–145.

[9] MEUNIER, P., FINDLER, R. B., AND FELLEISEN, M. Modular Set-Based Analysis From Contracts. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (Charleston, South Carolina, January 2006), pp. 218–231.

[10] MIGHT, M., AND SHIVERS, O. Environment Analysis via ΔCFA. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (Charleston, South Carolina, January 2006), pp. 127–140.

[11] MIGHT, M., AND SHIVERS, O. Improving Flow Analysis via ΓCFA: Abstract Garbage Collection and Counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)* (Portland, Oregon, September 2006).

[12] MINÉ, A. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04* (2004), vol. 2986 of *LNCS*, Springer, pp. 3–17.

[13] MINÉ, A. The octagon abstract domain. *Higher-Order and Symbolic Computation 19* (2006), 31–100.

[14] NANEVSKI, A., MORRISETT, G., AND BIRKEDAL, L. Polymorphism and Separation in Hoare Type Theory. In *ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, September 2006).

[15] REHOF, J., AND FÄHNDRICH, M. Type-based Flow Analysis: From Polymorphic Subtyping to CFL-reachability. In *Proceedings of the 28th Annual ACM Symposium on the Principles of Programming Languages* (2001).

[16] SHIVERS, O. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, June 1988), pp. 164–174.

[17] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

A. Conventions

For all domains, we assume the "natural" meaning for the lattice operators $\sqcap$ and $\sqcup$ as well as the relation $\sqsubseteq$; that is, a point-wise lifting (for functions), or an index-wise lifting (for vectors and tuples). We assume implicit top $\top$ and bottom $\bot$ element for domains lacking them. When a function is applied to an element outside of its domain, it yields $\bot$; thus, we get $dom(f) = \{x : f(x) \neq \bot\}$. The "absolute value" notation $|x|$ should be read and interpreted as "the abstraction of x."

We use boldface to denote vectors, *i.e.*, $\boldsymbol{d} = \langle d_1, \ldots, d_n \rangle$. The function $f[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ is the function f except that when applied to x_k, it yields y_k. Operators are implicitly lifted point-wise over ranges for functions; that is: if $\oplus : Y \times Y \to Y$ and $f, g : X \to Y$, then $f \oplus g = \lambda x . f(x) \oplus g(x)$. For a tuple $t = (a, b, \ldots)$, we have $t = (t_a, t_b, \ldots)$.

The function *free* returns the set of free variables for a given piece of syntax. For syntactic entities with lexically scoped bindings (such as λ-calculus terms or logical formulae with quantifiers), the notation $s[t'/t]$ denotes the capture-avoiding substitution of t with t' in form s.