Reasoning About Paths in the Interface Graph

Michael Greenberg
Stevens Institute of Technology, Hoboken, NJ, USA

Abstract

Clearly specified interfaces between software components are invaluable: development proceeds in parallel; implementation details are abstracted away; invariants are enforced; code is reused. But this abstraction comes with a cost: well chosen interfaces let related tasks be grouped together, but a running program interleaves tasks of all kinds. Reasoning about which values cross a given interface or which interfaces a value will cross is challenging.

It is particularly hard to know that interfaces apply runtime enforcement mechanisms correctly: as programs run, values cross abstraction boundaries in subtle ways. One particular case of such reasoning – proving that a contract system checks contracts correctly at runtime [2, 3] – uses a dynamic analysis to keep track of which interfaces are responsible for which values. The dynamic analysis works by giving an alternative semantics that "colors" values to match the components responsible for them. No program is ever *run* in this alternative semantics – it's a formal tool to verify that the contract system's enforcement is correct.

In this short paper, we refine Dimoulas et al.'s dynamic analysis to more precisely track colors, phrasing our results graph theoretically: a value's colors are a path in the interface graph of the original program. Our graph theoretic framing makes it easy to see that the dynamic analysis is subsumed by Eelco Visser's scope graphs.

2012 ACM Subject Classification Software and its engineering → Functional languages; Mathematics of computing → Graph theory; Software and its engineering → Abstraction, modeling and modularity

Keywords and phrases interfaces, components, lambda calculus, dynamic analysis

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.11

Acknowledgements I gratefully acknowledge the reviewers for their advice.

1 Introduction

Clearly specified interfaces between parts of a program – modules, libraries, components, etc. – are key to software development. Good interfaces lighten the load of design, development, and maintenance; good abstractions between interfaces make it easier to reason about programs. Clever uses of module boundaries can yield dynamic guarantees: contracts on interfaces help programmers identify modules that violate preconditions or don't live up to postconditions; object capability interfaces help programmers moderate access to critical resources.

But nicely abstracted, compartmentalized code doesn't *stay* nicely abstracted and compartmentalized at runtime: at runtime, our beautiful abstraction barriers collapse into a dynamic morass. Proving that a static enforcement mechanism yields worthwhile guarantees can be hard – but it isn't even clear where to start for dynamic enforcement mechanisms. What property must we even prove to know that our runtime checks mean nothing goes wrong?

Dimoulas et al. [2] use a dynamic analysis to prove that their higher-order contracts check values appropriately. They describe a simple functional language – Contract PCF (CPCF) – that puts *contract monitors* at interfaces, e.g., a math module might export a square-root function with the contract $\{x : \mathsf{Float} \mid x \geq 0\} \to \{y : \mathsf{Float} \mid \mathsf{abs}(y^2 - x) < \epsilon\}$, requiring that its clients give non-negative inputs and promising to return square roots (within some constant ϵ). At first order, these contracts amount to pre- and post-conditions. At higher orders, *blame labels* identify the component responsible for contract failures, following Findler and Felleisen [4]. For Dimoulas et al., the core question is: if a value fails a check, does the contract blame the correct component? Their dynamic analysis tracks which components are "responsible" for each value. Each component has a "color" (a unique label, used also for blame); values are colored with the components that are responsible for them. Using bisimulation techniques, they prove that when a value fails a contract check, the label blamed, the value that failed is colored with the same label – that is, when a value fails a check, we blame the responsible component.

Dimoulas et al.'s approach is clever: the dynamic analysis annotates values with crucial information... but rather than actually *running* the dynamic analysis, they prove a theorem about it. Their approach is also complex: their analysis is specific to contracts; their analysis is doesn't precisely track the order of components responsible for a value. **In this short paper, we extend and generalize Dimoulas et al.'s analysis to track how values dynamically traverse a program's static *interface graph*.** Our extension (Section 2) yields a more precise dynamic analysis for reasoning about a program's components graph theoretically (Section 3). Our more precise analysis yields tools for relating values and program control flow back to a program's interface graph (Section 4). Finally, our notion of interface graph is not dissimilar to Eelco Visser's scope graphs [8] (Section 5).

2 A dynamic analysis for components and responsibility

Behavioral contracts can specify as little as the type of a function or as much as the full behavior of a component through time. In first-order languages, contracts amount to pre- and post-conditions; in higher-order languages, contracts must carefully track *blame* in order to account for the flow of higher-order values. Findler and Felleisen [4] show that it suffices to have just two blame labels to track a higher-order value: a "positive" label for the value itself, and "negative" label for the context.

Dimoulas et al. study higher-order behavioral contracts, with the aim of proving that contract violations blame the correct party. They extend the middleweight PCF calculus [9] to Contract PCF (CPCF), adding a notion of contracts to monitor the interfaces between components. Their key insight is that it's possible to run the program in a semantics that uses a dynamic analysis to track the parties responsible for each value, devising a semantics that: "(1) for each party, keeps track of its contract obligations and (2) for each value, accounts for its origin" [2]§3. That is, a contract system is correct when every time it checks a value for contract conformance at some interface, then that interface is now "responsible for" that value. To fulfill (1), each predicate contract – like `number?` or `nonempty-list?` – keeps track of a set of responsibilities, i.e., the *colors* (unique labels) of those values it has monitored. To fulfill (2), expressions and values keep track of a vector of colors identifying the components responsible for that expression. Conflating blame labels and component colors, Dimoulas et al. use this notion of correctness to great effect: they discover the need for a new *indy* semantics that carries a *third* label in case contracts themselves violate preconditions (writing monitors as $\mathsf{mon}^{l_1,l_2}_{l_3}$, where l_3 refers to the contract); they show that the *picky* semantics for contracts cannot be correct in Racket [2], while the *lax* one can miss violations [3].

We refine Dimoulas et al.'s Contract PCF into the *colored lambda calculus*, the CLAM calculus for short. In doing so, we separate their tool – the dynamic analysis in the semantics – from their use of it – proving correctness for contracts. Along the way, we refine their analysis to make it more precise.

$$\begin{array}{llll} \textbf{Terms} & e & ::= & x \mid k \mid op(e_1, \ldots, e_n) \mid \lambda x.\ e \mid e_1\ e_2 \mid \textbf{if}\ e_1\ e_2\ e_3 \mid \\ & & & \mathsf{ifc}(e)^{l_1,l_2} \mid \|e\|^{l_1,l_2} \\ \textbf{Labels} & l & \in & \mathsf{Label} = \{l_0, l_1, \ldots\} \end{array}$$

Figure 1 Syntax for the CLAM calculus.

2.1 Syntax

The CLAM calculus extends untyped lambda calculus with conventional notions of constant, first-order operation, and conditional control flow – along with two new forms for marking interfaces between components and colored expressions (Figure 1). These two extensions, borrowed from Dimoulas et al., bear further discussion. Interfaces connect *components* (deliberately generically named); we conflate components and their colors/labels, drawn from some infinite set, Label. An interface form $\mathsf{ifc}(e)^{l_1,l_2}$ represents a static interface boundary: l_2 is the label for the "outside" of the interface, and l_1 is the label for the inside of the interface, which is implemented by the term e. These interfaces are *static*, occurring in the original program text. A colored expression $\|e\|^{l_1,l_2}$ represents dynamically determined responsibility: the component colored l_1 was previously responsible for the expression e, but now the component colored l_2 is responsible. These colored expressions are *dynamic*, arising as values flow through interfaces. Our syntax is directly adapted from Dimoulas et al.'s original framing [2]: we've removed the contract checks, renaming their monitors mon into our interfaces ifc. By removing contract checks, we no longer need the third blame label; we track a *list* of labels on interfaces rather than their set of obligations on a predicate contract, allowing us to characterize values using more precise, graph-theoretic notions (Section 3). Finally, a colored expression gives both an inner and an outer label, unlike Dimoulas's single label. To put the last difference another way, our notion of responsibility is always expressed as a *transfer*; when just the component colored l is responsible for the un-transferred expression, we write $\|e\|^{l,l}$.

We use standard encodings of notations like let; given a finite series of colors l_n, we write the left-to-right multi-coloring of an expression as: $\|e\|^{\overrightarrow{l_n}} = \|\|\|\|\|e\|^{l_1,l_2}\|^{l_2,\cdots}\|^{\cdots,l_{n-1}}\|^{l_{n-1},l_n}$ where l_n is the outermost color. When the list of colors is empty, $\|e\|^{\epsilon} = e$.

A *source program* has interfaces but no colored expressions: that is, a source program's components are connected, but no component has yet taken responsibility for the value.

We conflate a component and its label, but a component's constituent expressions may be scattered through a term. For example, consider the following term e_0:

$$\begin{array}{lcl} e_0 & = & \mathsf{ifc}(\mathsf{ifc}(e_1)^{l_2,l_1}\ 1)^{l_1,l_0} \\ e_1 & = & \mathsf{ifc}(\mathsf{ifc}(e_2)^{l_1,l_3})^{l_3,l_2} \\ e_2 & = & \lambda x.\ x + 1 \end{array}$$

By convention, l_0 is the outermost label; the component l_1 applies $\mathsf{ifc}(\ldots)^{l_2,l_1}$ to the value 1 – and deeply nested inside, the same component l_1 contains the function $\lambda x.\ x + 1$. As another example, consider the following term, e_3:

$$\begin{array}{lcl} e_3 & = & \mathsf{ifc}(\mathsf{ifc}(e_4)^{l_m,l_c}\ \ldots)^{l_c,l_0} \\ e_4 & = & \lambda x.\ \mathsf{let}\ db = \mathsf{ifc}(e_5)^{l_d,l_m}\ \mathsf{in} \\ & & \quad \textbf{if}\ (fst\ db = x)\ (snd\ db)\ (-1) \\ e_5 & = & (\mathsf{password}, \mathsf{secret}) \end{array}$$

Here, the component l_c applies the results of component l_m to some unspecified value ($\ldots$ in e_3); the component l_d holds the database (e_5) and is used by component l_m (e_4).

$$
\begin{array}{llcl}
\textbf{Values} & v & ::= & \|u\|^{l_1,l_2} \mid \|v\|^{l_1,l_2} \\
\textbf{Pre-values} & u & ::= & k \mid \lambda x.\ e \\
\textbf{Frames} & \mathit{Fr} & ::= & \bullet\, e \mid v\, \bullet \mid \mathit{op}(v_1, \ldots, v_i, \bullet, e_1, \ldots, e_j) \mid \textbf{if}\ \bullet\ e_2\ e_3 \mid \\
 & & & \mathsf{ifc}(\bullet)^{l_1,l_2} \mid \| \bullet \|^{l_1,l_2} \\
\textbf{Continuations} & C & ::= & \bullet \mid C{::}\mathit{Fr} \\
 & & & \\
\textbf{Machine states} & \mathit{st} & ::= & \langle C, l, e\rangle
\end{array}
$$

Figure 2 Syntax definitions for the CLAM abstract machine.

$$
\begin{array}{lll}
\langle C, l, e_1\ \ e_2\rangle & \longrightarrow \langle C{::}\bullet\ \ e_2, l, e_1\rangle & \textsc{AppStart} \\
\langle C{::}\bullet\ \ e_2, l, \|\lambda x.\ e_1\|^{l_1,\ldots,l_n}\rangle & \longrightarrow \langle C{::}\|\lambda x.\ e_1\|^{l_1,\ldots,l_n} \bullet, l, e_2\rangle & \textsc{AppMiddle} \\
\langle C{::}\|\lambda x.\ e_1\|^{l_1,\ldots,l_n} \bullet, l, \|u_2\|^{l'_1,\ldots,l'_m}\rangle & \longrightarrow \langle C, l, \|e_1\{\|u_2\|^{l'_1,\ldots,l'_m,l_n,\ldots,l_1}/x\}\|^{l_1,\ldots,l_n}\rangle & \textsc{AppEnd} \\
\langle C, l, \mathit{op}(e_1, e_2, \ldots, e_n)\rangle & \longrightarrow \langle C{::}\mathit{op}(\bullet, e_2, \ldots, e_n), l, e_1\rangle & \textsc{OpStart} \\
\langle C{::}\mathit{op}(v_1, \ldots, v_i, \bullet, e_1, e_2, \ldots, e_j), l, v\rangle & \longrightarrow \langle C{::}\mathit{op}(v_1, \ldots, v_i, v, \bullet, e_2, \ldots, e_j), l, e_1\rangle & \textsc{OpMiddle} \\
\langle C{::}\mathit{op}(\|u_1\|^{\overrightarrow{l_1}}, \ldots, \|u_i\|^{\overrightarrow{l_i}}, \bullet), l, \|u\|^{\overrightarrow{l}}\rangle & \longrightarrow \langle C, l, [\![\mathit{op}]\!]\,(u_1, \ldots, u_i, u)\rangle & \textsc{OpEnd} \\
\langle C, l, \textbf{if}\ \ e_1\ \ e_2\ \ e_3\rangle & \longrightarrow \langle C{::}\textbf{if}\ \bullet\ \ e_2\ e_3, l, e_1\rangle & \textsc{IfStart} \\
\langle C{::}\textbf{if}\ \bullet\ \ e_2\ e_3, l, \|\mathsf{true}\|^{l_1,\ldots,l_n}\rangle & \longrightarrow \langle C, l, e_2\rangle & \textsc{IfEndTrue} \\
\langle C{::}\textbf{if}\ \bullet\ \ e_2\ e_3, l, \|\mathsf{false}\|^{l_1,\ldots,l_n}\rangle & \longrightarrow \langle C, l, e_3\rangle & \textsc{IfEndFalse} \\
\langle C, l, u\rangle & \longrightarrow \langle C, l, \|u\|^{l,l}\rangle & \textsc{ColorValue} \\
\langle C, l, \|e\|^{l_1,l_2}\rangle & \longrightarrow \langle C{::}\| \bullet \|^{l_1,l_2}, l, e\rangle \text{ when } e \neq \|u\|^{\ldots,l'} & \textsc{ColorStart} \\
\langle C{::}\| \bullet \|^{l_1,l_2}, l, v\rangle & \longrightarrow \langle C, l, \|v\|^{l_1,l_2}\rangle & \textsc{ColorEnd} \\
\langle C, l, \mathsf{ifc}(e)^{l_1,l_2}\rangle & \longrightarrow \langle C{::}\mathsf{ifc}(\bullet)^{l_1,l}, l_1, e\rangle & \textsc{IfcStart} \\
\langle C{::}\mathsf{ifc}(\bullet)^{l_1,l_2}, l, \|k\|^{l'_1,\ldots,l'_n}\rangle & \longrightarrow \langle C, l_2, \|k\|^{l'_1,\ldots,l'_n,l_2}\rangle & \textsc{IfcEndBase} \\
\langle C{::}\mathsf{ifc}(\bullet)^{l_1,l_2}, l, \|\lambda x.\ e\|^{l'_1,\ldots,l'_n}\rangle & \longrightarrow \langle C, l_2, \lambda x.\ \mathsf{ifc}(\|\lambda x.\ e\|^{l'_1,\ldots,l'_n}\ \mathsf{ifc}(x)^{l_2,l_1})^{l_1,l_2}\rangle & \textsc{IfcEndLam}
\end{array}
$$

Figure 3 Reduction semantics for the CLAM abstract machine.

2.2 Semantics

While Dimoulas et al. use small-step operational semantics to give meaning to CPCF, we use an abstract machine (Figures 2 and 3). Why? It's hard to track the "current" component in a small-step operational semantics, while an abstract machine makes it easy – essential for our graph theoretic properties (Sections 3 and 4).

We define the CLAM abstract machine in terms of *machine states st*, which are a triple of the current *continuation* C (made up of *frames* Fr), the current *component label* l, and the current term to reduce e (Figure 2). Our semantics distinguishes between colorless pre-values u and colored values v. In the CLAM calculus, it is an invariant that a colored expression's last color is the current component: for the component to be computing with a value, it must be responsible for it. Reduction rules map machine states to machine states (Figure 3). The most interesting rules are the ones for coloring and interfaces. The current component changes whenever control enters a (static) interface or (dynamic) colored expression (IFCSTART, COLORSTART); the component reverts back when control leaves (IFCEND*, COLOREND). The COLORVALUE rule colors pre-values with the current component, dynamically making the current component responsible for any values it generates.

Like in Dimoulas et al.'s semantics, interfaces treat constants and functions differently when they pass over interfaces. When a constant moves to a new component, it acquires the color of that component (IfcEndBase); when a function moves to a new component, it is wrapped in a *function proxy* that takes the interface with it (IfcEndLam). That is, a function f moving from l_1 to l_2 is wrapped in a lambda so that arguments to f in the component l_2 are sent back to l_1 before running the function, whose result is moved from l_1 to l_2. Put another way, interfaces don't add colors to lambdas at all – instead, interfaces generate wrappers that do the "real" work on the arguments. The idea here – borrowed directly from Dimoulas et al. – is that when higher-order values move from one component to another, their bodies remain in the old component and values passed in to them come from the new one.

One might wonder: why have two notions of coloring? First, it is convenient for reasoning to distinguish static interfaces $\mathsf{ifc}(\bullet)^{l_1,l_2}$ from dynamic ones $\| \bullet \|^{l_1,l_2}$. More importantly, it is technically important to treat higher-order values specially at static interfaces (IfcEndLam).

Like in Dimoulas et al.'s semantics, our β reduction rule (AppEnd) updates responsibility on substitution. Unlike their semantics, we don't add the current component to the substituted value: the current container l is the same as the final color l_n on the function and the final color l'_m on the argument, so there's no need to add colors from the context. This redundancy doesn't quite exist in their system. Since our coloring brackets have ordered pairs of colors, we have a more precise account of inner and outer colors. Furthermore, we also color the substituted argument value with the color of the lambda – as it has substituted in for the variable in the lambda – and whoever is responsible for the lambda is responsible for its input parameter. Both systems have redundancy in them: values will collect many redundant copies of the same color. These redundant colors correspond to self-loops in the interface graph between components (Section 3). Rather than treating a values accumulated colors as an arbitrary list, we could have colors form a monoid with an operation that deduplicates self loops. There's no formal benefit in our abstract setting, though, so we don't bother.

Also following Dimoulas et al., operations do not taint their outputs (OpEnd): operations strip the colors off their inputs and produce uncolored prevalues.

In general, we'll assume that there's some default outer label l_0, and to run the program e we start with the configuration $\langle \bullet, l_0, e \rangle$. We'll only do so, however, if e is well colored at l_0.

2.3 Well colored terms

An expression is well colored when its interfaces and colored subexpressions agree (Figure 4); rules for frames and continuations follow naturally, with $\vdash C : l \twoheadrightarrow l'$ meaning that the continuation C expects its hole to be filled with something labeled l and produces a term labeled l'.

Coloring enjoys the expected type-like properties of substitution and preservation. Coloring has no progress property: colors don't say anything about the correctness of the program's operations; colors say that a program's components agree at their interfaces. Ill colored programs do *not* go wrong.

▶ **Lemma 2.1** (Substitution). *If $L, x{:}l', L' \vdash e : l$ and $\emptyset \vdash v : l'$ (where $v = \|u\|^{\dots,l'}$), then $L, L' \vdash e\{v/x\} : l$.*

Proof. By induction on the coloring derivation of e, leaving L' general and relying on a weakening lemma. ◀

▶ **Lemma 2.2** (Preservation). *If $\vdash st : l'$ and $st \longrightarrow st'$ then $\vdash st : l'$.*

$$\boxed{L \vdash e : l}$$

$$\frac{x{:}l' \in L}{L \vdash x : l'}\ \textsc{Var} \qquad \frac{}{L \vdash k : l}\ \textsc{Const} \qquad \frac{L, x{:}l \vdash e_{12} : l}{L \vdash \lambda x.\ e_{12} : l}\ \textsc{Abs} \qquad \frac{L \vdash e_i : l}{L \vdash op(e_1, \ldots, e_n) : l}\ \textsc{Op}$$

$$\frac{L \vdash e_1 : l \quad L \vdash e_2 : l}{L \vdash e_1\ e_2 : l}\ \textsc{App} \qquad \frac{L \vdash e_1 : l \quad L \vdash e_2 : l \quad L \vdash e_3 : l}{L \vdash \textbf{if}\ e_1\ e_2\ e_3 : l}\ \textsc{If}$$

$$\frac{L \vdash e : l_1}{L \vdash \| e \|^{l_1,l_2} : l_2}\ \textsc{Color} \qquad \frac{L \vdash e : l_1}{L \vdash \mathsf{ifc}(e)^{l_1,l_2} : l_2}\ \textsc{Ifc}$$

$$\boxed{\vdash Fr : l_1 \twoheadrightarrow l_2} \quad \boxed{\vdash C : l_1 \twoheadrightarrow l_2} \quad \boxed{\vdash st : l'}$$

$$\frac{\emptyset \vdash e : l}{\vdash \bullet\ e : l \twoheadrightarrow l}\ \textsc{FAppL} \qquad \frac{\emptyset \vdash v : l}{\vdash v\ \bullet : l \twoheadrightarrow l}\ \textsc{FAppR} \qquad \frac{\emptyset \vdash e_2 : l \quad \emptyset \vdash e_3 : l}{\vdash \textbf{if}\ \bullet\ e_2\ e_3 : l \twoheadrightarrow l}\ \textsc{FIf}$$

$$\frac{}{\vdash \| \bullet \|^{l_1,l_2} : l_1 \twoheadrightarrow l_2}\ \textsc{FColor} \qquad \frac{}{\vdash \mathsf{ifc}(\bullet)^{l_1,l_2} : l_1 \twoheadrightarrow l_2}\ \textsc{FIfc} \qquad \frac{}{\vdash \bullet : l \twoheadrightarrow l}\ \textsc{Hole}$$

$$\frac{\vdash C : l' \twoheadrightarrow l'' \quad \vdash Fr : l \twoheadrightarrow l'}{\vdash C{::}Fr : l \twoheadrightarrow l''}\ \textsc{Frame} \qquad \frac{\vdash C : l \twoheadrightarrow l' \quad \emptyset \vdash e : l}{\vdash \langle C, l, e\rangle : l'}\ \textsc{State}$$

Figure 4 Well colored expressions have correct labels on interfaces and colored expressions. Well colored continuations transition according to well colored frames; well colored states match the term to the current component and the continuation.

Proof. By induction on the coloring derivation. Let $st = \langle C, l, e\rangle$. We go first by cases on the coloring of e as $\emptyset \vdash e : l$, considering the continuation C and its coloring $\vdash C : l \twoheadrightarrow l'$ as necessary. ◀

3 Interface and value graphs

The Clam calculus explicitly marks interfaces between components in the program so that we can reason clearly about how components communicate. We use graphs to formalize the idea: a term e has a *program graph* marking the interfaces between each colored component in e. Program graphs are undirected graphs where each color is a node; when there exists an interface between two components/colors, an edge connects their corresponding nodes.

Each term e has two kinds of program graphs (Figure 5): the interface graph and the value graph. In both graphs, the nodes are simply the set of colors used in e, i.e., e's components. In the *interface graph*, $\mathbf{ig}(e)$, we take the graph's edges from interfaces – there is an edge $\{l_1 \leftrightarrow l_2\}$ for each interface $\mathsf{ifc}(e)^{l_1,l_2}$. In the *value graph*, $\mathbf{vg}(e)$, we take the graph's edges from colorings – there is an edge $\{l_1 \leftrightarrow l_2\}$ for each coloring $\|e\|^{l_1,l_2}$. For both graphs, we assume that all nodes have self loops: for every $l \in e$, the edge $\{l \leftrightarrow l\}$ is in every program graph. Both kinds of program graphs are mostly defined homomorphically – the interesting cases come in the treatment of interfaces and colored expressions.

While an expression has a straightforward tree structure, well colored expressions will have interface *graphs*, with loops. The program e_0 from Section 2.1 has a lasso structure (Figure 6, left): there is a loop using the edges $\{l_2 \leftrightarrow l_1\}$ and $\{l_3 \leftrightarrow l_2\}$ and $\{l_1 \leftrightarrow l_3\}$. Evaluating this program yields a value whose colors trace appropriately through the interface graph (Figure 7).

$$\begin{aligned}
\mathbf{ig}(x) &= \emptyset & \mathbf{vg}(x) &= \emptyset \\
\mathbf{ig}(k) &= \emptyset & \mathbf{vg}(k) &= \emptyset \\
\mathbf{ig}(\lambda x.\ e) &= \mathbf{ig}(e) & \mathbf{vg}(\lambda x.\ e) &= \mathbf{vg}(e) \\
\mathbf{ig}(op(e_1, \ldots, e_n)) &= \textstyle\bigcup_{i=1}^{n} \mathbf{ig}(e_i) & \mathbf{vg}(op(e_1, \ldots, e_n)) &= \textstyle\bigcup_{i=1}^{n} \mathbf{vg}(e_i) \\
\mathbf{ig}(e_1\ e_2) &= \mathbf{ig}(e_1) \cup \mathbf{ig}(e_2) & \mathbf{vg}(e_1\ e_2) &= \mathbf{vg}(e_1) \cup \mathbf{vg}(e_2) \\
\mathbf{ig}(\mathbf{if}\ e_1\ e_2\ e_3) &= \textstyle\bigcup_{i=1}^{3} \mathbf{ig}(e_i) & \mathbf{vg}(\mathbf{if}\ e_1\ e_2\ e_3) &= \textstyle\bigcup_{i=1}^{3} \mathbf{vg}(e_i) \\
\mathbf{ig}(\mathsf{ifc}(e)^{l_1,l_2}) &= \{l_1 \leftrightarrow l_2\} \cup \mathbf{ig}(e) & \mathbf{vg}(\mathsf{ifc}(e)^{l_1,l_2}) &= \mathbf{vg}(e) \\
\mathbf{ig}(\|e\|^{l_1,l_2}) &= \mathbf{ig}(e) & \mathbf{vg}(\|e\|^{l_1,l_2}) &= \{l_1 \leftrightarrow l_2\} \cup \mathbf{vg}(e)
\end{aligned}$$

Figure 5 Program graphs record the connections between interfaces in a program. The interface graph **ig** records interfaces, while the value graph **vg** records coloring/responsibility relationships. Interesting rows are highlighted in lavender. Definitions for frames can be derived from definitions for expressions, where holes • have empty graphs.

$$\begin{aligned}
e_0 &= \mathsf{ifc}(\mathsf{ifc}(e_1)^{l_2,l_1}\ 1)^{l_1,l_0} & e_3 &= \mathsf{ifc}(\mathsf{ifc}(e_4)^{l_m,l_c}\ \ldots)^{l_c,l_0} \\
e_1 &= \mathsf{ifc}(\mathsf{ifc}(e_2)^{l_1,l_3})^{l_3,l_2} & e_4 &= \lambda x.\ \mathsf{let}\ db = \mathsf{ifc}(e_5)^{l_d,l_m}\ \mathsf{in} \\
e_2 &= \lambda x.\ x + 1 & &\quad \mathbf{if}\ (fst\ db = x)\ (snd\ db)\ (-1) \\
& & e_5 &= (\mathsf{password}, \mathsf{secret})
\end{aligned}$$

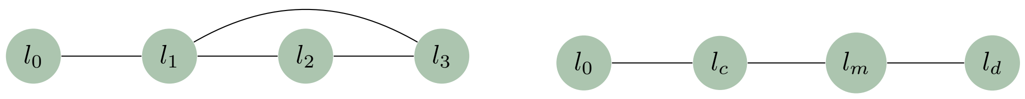

Figure 6 Example programs and their corresponding interface graphs.

But some programs are deliberately structured to avoid certain forms of communication. Say, a client should only ever talk to the database by way of the middleware, never directly (Figure 6, right). The term e_5 is a password-protected database in component l_d, containing a value secret that appears nowhere else in the program. The term e_3 is a client of the database in component l_c – it supplies a password and tries to work with the secret. The term e_4 is the middleware in component l_m; it takes a password from the client and either returns the secret database contents or signals an error by returning -1. By inspecting the interface graph, we can see that there is no direct connection from the database to the client.

4 Paths in the interface graph

Interface and value graphs let us characterize how values flow between components. Inspired by Dimoulas et al.'s proof of contract correctness, we show that (1) *as programs run, their trace is a path in the interface graph* (Lemma 4.1) and (2) *value colorings are paths in the interface graph* (Lemma 4.2). Both properties rely on a subject reduction like lemma: reducing a term yields a new term whose interface graph is a subgraph of the original term's interface graph (Lemma A.3). A similar property holds for the value graph, though the value graph of a term may gain edges as interfaces turn into colored expressions (Figure 3, IfcEnd*).

$$\begin{array}{llll}
 & \langle \bullet, & l_0, & \mathsf{ifc}(\mathsf{ifc}(\mathsf{ifc}(\mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3})^{l_3,l_2})^{l_2,l_1}\ 1)^{l_1,l_0}\rangle \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}, & l_1, & \mathsf{ifc}(\mathsf{ifc}(\mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3})^{l_3,l_2})^{l_2,l_1}\ 1\rangle \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}{::}\bullet\ 1, & l_1, & \mathsf{ifc}(\mathsf{ifc}(\mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3})^{l_3,l_2})^{l_2,l_1}\rangle \\
\longrightarrow & \langle \ldots{::}\bullet\ 1{::}\mathsf{ifc}(\bullet)^{l_2,l_1}, & l_2, & \mathsf{ifc}(\mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3})^{l_3,l_2}\rangle \\
\longrightarrow & \langle \ldots{::}\mathsf{ifc}(\bullet)^{l_2,l_1}{::}\mathsf{ifc}(\bullet)^{l_3,l_2}, & l_3, & \mathsf{ifc}(\lambda x.\ x+1)^{l_1,l_3}\rangle \\
\longrightarrow & \langle \ldots{::}\mathsf{ifc}(\bullet)^{l_3,l_2}{::}\mathsf{ifc}(\bullet)^{l_1,l_3}, & l_1, & \lambda x.\ x+1\rangle \\
\longrightarrow & \langle \ldots{::}\mathsf{ifc}(\bullet)^{l_3,l_2}{::}\mathsf{ifc}(\bullet)^{l_1,l_3}, & l_1, & \|\lambda x.\ x+1\|^{l_1,l_1}\rangle \\
\longrightarrow & \langle \ldots{::}\mathsf{ifc}(\bullet)^{l_2,l_1}{::}\mathsf{ifc}(\bullet)^{l_3,l_2}, & l_3, & \lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\rangle \\
\longrightarrow & \langle \ldots{::}\mathsf{ifc}(\bullet)^{l_2,l_1}{::}\mathsf{ifc}(\bullet)^{l_3,l_2}, & l_3, & \|\lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\|^{l_3,l_3}\rangle \\
\longrightarrow & \langle \ldots{::}\bullet\ 1{::}\mathsf{ifc}(\bullet)^{l_2,l_1}, & l_2, & \\
 & \multicolumn{3}{r}{\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\rangle} \\
\longrightarrow & \langle \ldots{::}\bullet\ 1{::}\mathsf{ifc}(\bullet)^{l_2,l_1}, & l_2, & \\
 & \multicolumn{3}{r}{\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\rangle} \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}{::}\bullet\ 1, & l_1, & \\
\multicolumn{4}{r}{\lambda w.\ \mathsf{ifc}(\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \mathsf{ifc}(\|\lambda x.\ x+1\|^{l_1,l_1}\ \mathsf{ifc}(y)^{l_3,l_1})^{l_1,l_3}\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\ \mathsf{ifc}(w)^{l_1,l_2})^{l_2,l_1}\rangle} \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}{::}\bullet\ 1, & l_1, & \\
 & \multicolumn{3}{r}{\|\lambda w.\ \mathsf{ifc}(\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \ldots\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\ \mathsf{ifc}(w)^{l_1,l_2})^{l_2,l_1}\|^{l_1,l_1}\rangle} \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}{::}\|\lambda w.\ \ldots\|^{l_1,l_1}\ \bullet, & l_1, & 1\rangle \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}{::}\|\lambda w.\ \ldots\|^{l_1,l_1}\ \bullet, & l_1, & \|1\|^{l_1,l_1}\rangle \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}, & l_1, & \\
 & \multicolumn{3}{r}{\mathsf{ifc}(\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \ldots\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\ \mathsf{ifc}(\|1\|^{l_1,l_1})^{l_1,l_2})^{l_2,l_1}\rangle} \\
\longrightarrow & \langle \mathsf{ifc}(\bullet)^{l_1,l_0}{::}\mathsf{ifc}(\bullet)^{l_2,l_1}, & l_2, & \\
 & \multicolumn{3}{r}{\|\lambda z.\ \mathsf{ifc}(\|\lambda y.\ \ldots\|^{l_3,l_3}\ \mathsf{ifc}(z)^{l_2,l_3})^{l_3,l_2}\|^{l_2,l_2}\ \mathsf{ifc}(\|1\|^{l_1,l_1})^{l_1,l_2}\rangle} \\
\longrightarrow & \langle\langle \ldots{::}\mathsf{ifc}(\bullet)^{l_2,l_1}{::}\|\lambda z.\ \ldots\|^{l_2,l_2}\ \bullet, & l_2, & \mathsf{ifc}(\|1\|^{l_1,l_1})^{l_1,l_2}\rangle\rangle \\
\longrightarrow & \langle\langle \ldots{::}\|\lambda z.\ \ldots\|^{l_2,l_2}\ \bullet{::}\mathsf{ifc}(\bullet)^{l_1,l_2}, & l_1, & \|1\|^{l_1,l_1}\rangle\rangle \\
\longrightarrow & \langle\langle \ldots{::}\mathsf{ifc}(\bullet)^{l_2,l_1}{::}\|\lambda z.\ \ldots\|^{l_2,l_2}\ \bullet, & l_2, & \|1\|^{l_1,l_2}\rangle\rangle \\
\longrightarrow & \ldots & & \\
\longrightarrow & \langle \bullet, & l_0, & \|2\|^{l_1,l_3,l_2,l_1}\rangle
\end{array}$$

Figure 7 Abbreviated evaluation trace of e_0 from Figure 6, omitting repeated colors for brevity and clarity.

4.1 Program traces

The trace of a program's evaluation is the sequence of currently executing components; such traces are paths in the value graph. To this precise, let trace $(\langle C_1, l_1, e_1\rangle \longrightarrow \langle C_2, l_2, e_2\rangle \longrightarrow \ldots \longrightarrow \langle C_n, l_n, e_n\rangle)$ be defined as $l_1, l_2, \ldots, l_n$ – the payoff of our abstract machine (Section 2.2). For our purposes, a path in a graph is a list of nodes $l_1, l_2, \ldots, l_n$ such that the edge $\{l_{i-1} \leftrightarrow l_i\}$ is in the graph.

▶ **Lemma 4.1** (Traces are paths). *If* $\vdash \langle C, l, e\rangle : l''$*, then* trace $(\langle C, l, e\rangle \longrightarrow^* \langle C', l', e'\rangle)$ *is a path in* $\mathbf{ig}(\langle C, l, e\rangle)$.

Proof. By induction on the length of the trace, observing that component changes must be edges in the interface graph (Lemma A.4) and reducing a term yields a subgraph (Lemma A.3). ◀

4.2 Value colorings

As program e evaluates, the values in e will gain colors – the colors on these values form paths in e's interface graph. We are particularly interested in source programs (with no coloring at all), but this property holds whenever $\mathbf{vg}(e) \subseteq \mathbf{ig}(e)$.

▶ **Lemma 4.2** (Value colorings are paths). *If $\vdash st : l''$ and $\mathbf{vg}(st) \subseteq \mathbf{ig}(st)$ and $st \longrightarrow^* st'$, then for any value $v = \|u\|^{l_1,\ldots,l_n}$ in st', the path $l_1, \ldots, l_n$ is in $\mathbf{ig}(st)$.*

Proof. By induction on the length of the evaluation, strengthening the induction hypothesis to show also that $\mathbf{vg}(st')$ is a subgraph of $\mathbf{ig}(st)$. If $st \longrightarrow^* st'$, then $\mathbf{vg}(st') \subseteq \mathbf{vg}(st) \cup \mathbf{ig}(st)$ (Lemma A.3) in line with our strengthened IH. ◀

4.3 Discussion

The properties we show here are intuitive and simple – arguably too simple. Preservation of well coloring (Lemma 2.2) and our coloring rules guarantee that colors are not trivial and our semantics updates colors correctly when traversing components. But who is to say that the program has interesting colors at all? Values in a "blob" program with a single module indeed trace a path in the interface graph, but that path is nothing more than a trivial self loop. There's nothing we can do about program structure: if the program has few or no modules, the program's graphs will be meager and paths will tell you little or nothing.

We have not applied our generalized analysis to show any more interesting property. It is not hard to recover the core of Dimoulas et al.'s proof: replace any $\mathsf{mon}_j^{k,l}(\lfloor \mathtt{flat}(e_c) \rfloor, e)$ with a dynamic check that e_c e yields true. Recovering more – dependent contract checking in the variety of styles – would require altering the semantics to closer match theirs. We also envision applications in object capability or other access control enforcement mechanisms. Our graph-theoretic theorems could guide an auditing process, guiding developers through the (syntactically non-obvious) series of interfaces that a value might traverse at runtime. Inspecting interfaces in the order that values will traverse them should help developers avoid classic and common issues with *confused deputies*, i.e., proxies that mediate access but can be tricked to escalate or misuse privileges.

5 Scope graphs (and other related work)

The Clam calculus is directly based on Dimoulas et al.'s work on contracts [2, 3], but there is a long line of research on components, containment, and interfaces in general and in the lambda calculus in particular.

On the theoretical side, the colors in the Clam calculus are similar to the labels in labeled lambda calculus [5]. Their labels differ from ours in a few ways: they are one-sided, while our interfaces and colored expressions are double-sided; they use over- and underlining to distinguish the parts of a β reduct; their semantics has no notion of a "current component"; they allow unlabeled values; and they work in a pure lambda calculus setting, without constants, operations, or conditional control flow. In principle, it might be possible to fit the Clam calculus into the labeled lambda calculus framework – though what advantage this would bring is unclear. The purposes are different, too: our labels are for reasoning about components, but Lévy is more interested in subtle properties of lambda calculus evaluation.

On the practical side, the informal notion of an interface or dependency graph is as old as modularity itself. Components have long been used to reason about "locality" or "containment" – for famous examples, see Morris [7], Dennis and Van Horn [1], or Miller [6]. Zdancewic et al. provide a more closely related notion of ownership [14], cited as direct inspiration by Dimoulas et al. [3].

Finally, Eelco Visser's scope graphs [8] are not dissimilar to the Clam calculus: Visser's scope graphs offer much more complete static [12] and dynamic [10, 13] accounts of binding and the flow of values in a program than either the original analysis or our extension – though scope graphs don't (by default) track the information in Clam's colors.

Compared to the Clam calculus's ifc nodes and colored expressions, scope graphs support a wide range of binding forms, including much more realistic models of modules ([8], Section 2.4). Scope graphs' resolution paths don't quite match the semantics of our colored expressions – but we can imagine extending resolution paths to track detailed path information. Such "colored resolution paths" would give a cleaner dynamic semantics [10], admitting clearly specified and meaningful static analyses [12, 11] – though they would not (without substantial work) track paths in the interface graph in quite the same way. The Clam calculus's model is not tuned for implementation (e.g., rule IfcEndLam in Figure 3), but scope graphs readily admit prototype-level implementations [10] or better [13].

6 Conclusion

The Clam calculus refines Dimoulas et al.'s imprecise dynamic analysis, relating interfaces and value to straightforward graph-theoretic concepts. We offer the Clam calculus as a reusable generalization of Dimoulas et al.'s dynamic analysis – a starting point for reasoning about modularity and runtime control.

References

1 Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9(3), March 1966. doi:10.1145/365230.365252.

2 Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *POPL*. ACM, 2011. URL: http://www.ccs.neu.edu/home/chrdimo/pubs/popl11-dfff.pdf.

3 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, volume 7211 of *LNCS*. Springer, 2012. URL: http://www.ccs.neu.edu/home/chrdimo/pubs/esop12-dthf.pdf.

4 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*. ACM, 2002. doi:10.1145/581478.581484.

5 Jean-Jacques Lévy. Tracking redexes in the lambda calculus, 2022. In submission. URL: http://pauillac.inria.fr/~levy/pubs/22trackredex.pdf.

6 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. URL: http://www.erights.org/talks/thesis/markm-thesis.pdf.

7 James H. Morris. Protection in programming languages. *CACM*, 16(1), January 1973. doi:10.1145/361932.361937.

8 Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *ESOP*, volume 9032 of *LNCS*. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

9 G.D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977. doi:10.1016/0304-3975(77)90044-5.

10 Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. In *ECOOP*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.20.

11 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *PACMPL*, 2(OOPSLA), October 2018. doi:10.1145/3276484.

12 Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *PEPM*. ACM, 2016. `doi:10.1145/2847538.2847543`.

13 Vlad Vergu, Andrew Tolmach, and Eelco Visser. Scopes and Frames Improve Meta-Interpreter Specialization. In *ECOOP*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECOOP.2019.4`.

14 Steve Zdancewic, Dan Grossman, and J. Gregory Morrisett. Principals in programming languages: A syntactic proof technique. In *ICFP*. ACM, 1999. URL: `http://www.eecs.harvard.edu/~greg/papers/pipl.pdf`.

A Proofs

▶ **Definition A.1** (Subgraphs). *A graph gr_1 is a subgraph of a graph gr_2 if the nodes of gr_1 are a subset of the nodes of gr_2 and if an edge is in gr_1 then it is in gr_2.*

▶ **Lemma A.2** (Substitution yields subgraphs).
1. $\mathbf{ig}(e\{v/x\})$ *is a subgraph of* $\mathbf{ig}(e) \cup \mathbf{ig}(v)$.
2. $\mathbf{vg}(e\{v/x\})$ *is a subgraph of* $\mathbf{vg}(e) \cup \mathbf{vg}(v)$.

Proof. By induction on e. The graphs are equal if x occurs free in e; if x doesn't occur, then the resulting graph is just e's. ◀

▶ **Lemma A.3** (Reduction yields subgraphs). *If* $\vdash st_1 : l$ *and* $st_1 \longrightarrow st_2$ *then*
1. $\mathbf{ig}(st_2)$ *is a subgraph of* $\mathbf{ig}(st_1)$*, and*
2. $\mathbf{vg}(st_2)$ *is a subgraph of* $\mathbf{ig}(st_1) \cup \mathbf{vg}(st_1)$.

Proof. By case analysis on the reduction step taken. ◀

▶ **Lemma A.4** (Component changes are edges). *If* $\langle C, l, e\rangle \longrightarrow \langle C, l', e'\rangle$*, then* $\{l \leftrightarrow l'\}$ *is in* $\mathbf{ig}(\langle C, l, e\rangle) \cup \mathbf{vg}(\langle C, l, e\rangle)$.

Proof. By case analysis on the step taken. ◀

▶ **Lemma A.5** (Subpaths). *If* $l_1, \ldots, l_n$ *is a path in a graph* gr*, then it is also a path in all supergraphs of* gr.

Proof. Given a supergraph gr', by induction on the length of the path. In the inductive case, since gr is a subgraph of gr', every edge in the path in gr must be present in the gr'. ◀