The Paradox of Flow Analysis

Or: What We Talk About When We Talk About Higher-Order Flow Analysis

David Van Horn

On Flow Analysis (kCFA):

- Background
- Complexity
- Paradox
- Resolution
- Exploitation

Part I

Background

Q: What is program analysis?

A: Prediction of the future.

underpins:
verification, optimization, tranformation, understanding, ...

global register allocation, global constant subexpression elimination, loop invariant detection, redundant assignment detection, dead code elimination, constant propagation, range analysis, code hoisting, induction variable elimination, copy propagation, live variable analysis, loop unrolling, loop jamming, type inference

Q: Which are higher-order programming languages?

A: Most of them.

Object-oriented: Java, Smalltalk, C#, C++, JavaScript, Eiffel, Scala, Ruby, Perl, Clojure, Python, ...

Functional: Common Lisp, Erlang, F#, Haskell, JavaScript, Scheme, SML, OCaml, ...

Q: What is a higher-order programming language (HOPL)?

A: Computation as values.

```
var cos = deriv(sin)

[cos, sin, sqr].map(function (f) { return f(5) })
```

Q: What makes program analysis difficult in a HOPL?

A: Mutual recursion between data and control.

- To do data-flow analysis, we need a control-flow analysis.
- To do control-flow analysis, we need a data-flow analysis.

```
var j = [cos, sin, sqr][i](5)
```

Evaluation:

```
// (ℝ → ℝ) → (ℝ → ℝ)
// Produce the numerical derivative of f.
function deriv(f) {
  function (x) { (f(x+ϵ) + f(x-ϵ)) / 2*ϵ; }
};

deriv(sqr);     //=> ⟨function (x) {..}, [f ↦ sqr]⟩
deriv(sin)      //=> ⟨function (x) {..}, [f ↦ sin]⟩
```

Idealized JavaScript:

$$
\begin{array}{rcll}
E & ::= & x & x \in \{\mathtt{x}, \mathtt{y}, \ldots\} \\
 & | & \mathtt{function}\ (x)\ \{\ E\ \} & \\
 & | & E; E & \\
 & | & E(E) &
\end{array}
$$

Evaluation:

$ev : \mathsf{Exp} \times \mathsf{Env} \to \mathsf{Val}$

$ev(x, \rho) = \rho(x)$

$ev(\texttt{function (}x\texttt{) \{}E\texttt{\}}, \rho) = \langle \texttt{function (x) \{}E\texttt{\}}, \rho \rangle$

$ev(E_0\,;E_1, \rho) = ev(E_0, \rho); ev(E_1, \rho)$

$ev(E_0\texttt{(}E_1\texttt{)}, \rho) =$
$\quad \mathsf{let}\ ev(E_0, \rho) = \langle \texttt{function (}x\texttt{) \{}E_0\texttt{\}}, \rho_0 \rangle\ \mathsf{in}$
$\quad \mathsf{let}\ ev(E_1, \rho) = \mathsf{v}\ \mathsf{in}$
$\qquad ev(E_0, \rho_0[x \mapsto v])$

Labeled Idealized JavaScript:

$$
\begin{array}{rcll}
E^{\ell} & ::= & x^{\ell} & x \in \{\mathtt{x}, \mathtt{y}, \ldots\} \\
 & | & \mathtt{function}^{\ell}\ (x)\ \{\ E^{\ell}\ \} & \\
 & | & E^{\ell}; E^{\ell} & \\
 & | & E^{\ell}(E^{\ell})^{\ell} &
\end{array}
$$

Abstract Evaluation (0CFA):

Computes a cache C that maps locations ℓ and variables x to sets of "abstract" values, i.e. `function` terms.

$$\begin{aligned} C(\ell) &= \{\, \texttt{function (x) \{..\}}, \\ &\quad\ \texttt{function (y) \{..\}} \,\} \\ C(x) &= \{\, \texttt{function (x) \{..\}}, \\ &\quad\ \texttt{function (y) \{..\}} \,\} \end{aligned}$$

Soundness:

- if E^{ℓ} evaluates to $\langle \texttt{f}, \rho \rangle$, then $\texttt{f} \in C(\ell)$,
- if x is bound to $\langle \texttt{f}, \rho \rangle$, then $\texttt{f} \in C(x)$.

Abstract Evaluation (0CFA):

```
// ∀ X . X → X
// The identity function.
function id(x) {
  x
};

id(sqr);        //=> { sqr }
id(sin)         //=> { sqr, sin }
```

Abstract Evaluation (0CFA):

```
// ∀ X,Y . (X → Y) → (X → Y)
// The identity function (for functions).
function id(f) {
  function (x) { f(x) }
};

id(sqr);          //=> { function (x) {..} }
id(sin)           //=> { function (x) {..} }
```

Abstract Evaluation (0CFA):

```
// ∀ X,Y . (X → Y) → (X → Y)
// The identity function (for functions).
function id(f) {
  function (x) { f(x) }
};

id(sqr)(5);  //=> { sqr(5) }
id(sin)(7)   //=> { sin(7), sin(5), sqr(5), sqr(7) }
```

Abstract Evaluation (0CFA):

$av : \mathsf{Exp} \times \mathsf{Cache} \to \mathsf{Cache}$

$av(x^{\ell}) = C(x) \subseteq C(\ell)$

$av(\texttt{function}^{\ell}\ (x)\ \{E\}) = \{\texttt{function (x) }\{E\}\} \subseteq C(\ell)$

$av(E^{\ell_1}; E^{\ell_2}) = av(E^{\ell_1}); av(E^{\ell_2})$

$av(E^{\ell_1}\,(E^{\ell_2})^{\ell}) = av(E^{\ell_1}); av(E^{\ell_2});$
$\quad \forall(\texttt{function }(x)\ \{E^{\ell_0}\}) \in C(\ell_1) :$
$\qquad C(\ell_2) \subseteq C(x);$
$\qquad av(E^{\ell_0});$
$\qquad C(\ell_0) \subseteq C(\ell)$

The idea of kCFA:

Use calling contexts to make distinctions during analysis.

(function (f) { f(E_1)^1; f(E_2)^2 })
(function (x) { E_0 })

E_0 will be analyzed twice:

- once with x meaning E_1,
- once with x meaning E_2.

Abstract Evaluation (1CFA):

```
// ∀ X, Y . (X → Y) → (X → Y)
// The identity function (for functions).
function id(f) {
  function (x) { f(x) }
};

id(sqr)[1];      //=> {⟨function (x) {..}, [f ↦ 1]⟩ }
id(sin)[2]       //=> {⟨function (x) {..}, [f ↦ 2]⟩ }
                 //   C(f, 1) = { sqr }
                 //   C(f, 2) = { sin }
```

Abstract Evaluation (1CFA):

$// \ \forall \ X, Y \ . \ (X \ \rightarrow \ Y) \ \rightarrow \ (X \ \rightarrow \ Y)$

```
// The identity function (for functions).
function id(f) {
  function (x) { f(x) }
};

id(sqr)[1](5)[3];  //=> { sqr(5) }
id(sin)[2](7)[4]   //=> { sin(7) }
```

Abstract Evaluation (*k*CFA):

Cache C maps locations ℓ and variables x with contours δ to sets of "abstract" *closures*, i.e. `function`, contour environment pairs.

$$\begin{aligned} C(\ell, \delta) &= \{\ \langle \texttt{function (x) \{..\}}, \phi_0 \rangle, \\ &\quad\ \langle \texttt{function (y) \{..\}}, \phi_1 \rangle\ \} \\ C(x, \delta) &= \{\ \langle \texttt{function (x) \{..\}}, \phi_0 \rangle, \\ &\quad\ \langle \texttt{function (y) \{..\}}, \phi_1 \rangle\ \} \end{aligned}$$

Soundness:

- if E^{ℓ} evaluates in context δ to $\langle \texttt{f}, \rho \rangle$, then $\langle \texttt{f}, \phi \rangle \in C(\ell, \delta)$, where $\rho \sqsubseteq \phi$,
- if x is bound in context δ to $\langle \texttt{f}, \rho \rangle$, then $\langle \texttt{f}, \phi \rangle \in C(x, \delta)$, where $\rho \sqsubseteq \phi$.

Instrumented Evaluation (∞CFA):

$av : \mathsf{Exp} \times \mathsf{CEnv} \times \mathsf{Contour} \times \mathsf{Cache} \to \mathsf{Cache}$

$av(x^{\ell}, \phi, \delta) = C(x, \phi(x)) \subseteq C(\ell, \delta)$

$av(\texttt{function}^{\ell}\ (x)\ \{E\}, \phi, \delta) = \{\langle \texttt{function (x) } \{E\}, \phi \rangle\} \subseteq C(\ell, \delta)$

$av(E^{\ell_1}; E^{\ell_2}, \phi, \delta) = av(E^{\ell_1}, \phi, \delta); av(E^{\ell_2}, \phi, \delta)$

$$
\begin{aligned}
&av(E^{\ell_1}\,(E^{\ell_2})^{\ell}, \phi, \delta) = av(E^{\ell_1}, \phi, \delta); av(E^{\ell_2}, \phi, \delta); \\
&\quad \forall(\langle \texttt{function } (x)\ \{E^{\ell_0}\}, \phi_0 \rangle) \in C(\ell_1, \delta) : \\
&\qquad C(\ell_2, \delta) \subseteq C(x, \delta\ell); \\
&\qquad av(E^{\ell_0}, \phi_0[x \mapsto \delta\ell], \delta\ell); \\
&\qquad C(\ell_0, \delta\ell) \subseteq C(\ell, \delta)
\end{aligned}
$$

Abstract Evaluation (kCFA):

$av : \mathsf{Exp} \times \mathsf{CEnv} \times \mathsf{Contour} \times \mathsf{Cache} \to \mathsf{Cache}$

$av(x^{\ell}, \phi, \delta) = C(x, \phi(x)) \subseteq C(\ell, \delta)$

$av(\texttt{function}^{\ell}\ (x)\ \{E\}, \phi, \delta) = \{\langle \texttt{function (x) } \{E\}, \phi \rangle\} \subseteq C(\ell, \delta)$

$av(E^{\ell_1}; E^{\ell_2}, \phi, \delta) = av(E^{\ell_1}, \phi, \delta); av(E^{\ell_2}, \phi, \delta)$

$$
\begin{aligned}
&av(E^{\ell_1}\,(E^{\ell_2})^{\ell}, \phi, \delta) = av(E^{\ell_1}, \phi, \delta); av(E^{\ell_2}, \phi, \delta); \\
&\quad \forall(\langle \texttt{function } (x)\ \{E^{\ell_0}\}, \phi_0 \rangle) \in C(\ell_1, \delta) : \\
&\qquad C(\ell_2, \delta) \subseteq C(x, \lfloor \delta\ell \rfloor_k); \\
&\qquad av(E^{\ell_0}, \phi_0[x \mapsto \lfloor \delta\ell \rfloor_k], \lfloor \delta\ell \rfloor_k); \\
&\qquad C(\ell_0, \lfloor \delta\ell \rfloor_k) \subseteq C(\ell, \delta)
\end{aligned}
$$

“It did not take long to discover that the basic analysis, for any $k > 0$, was intractably slow for large programs.

In the ensuing years, researchers expended a great deal of effort deriving clever ways to tame the cost of the analysis.”

Olin Shivers, Higher-order control-flow analysis in retrospect: Lessons learned, lessons abandoned (2004)

Part II

Complexity

- 0CFA: Linearity and Evaluation
- kCFA: Non-Linearity and Approximation

Theorem. 0CFA is in PTIME.

Proof: C grows monotonically. Its size is $O(n^2)$. □

$$
\begin{aligned}
C(\ell_0) &= \{\texttt{function}^{\ell_i}\ \{..\},\ldots,\texttt{function}^{\ell_j}\ \{..\}\} \\
&\vdots \\
C(\ell_n) &= \{\texttt{function}^{\ell_i}\ \{..\},\ldots,\texttt{function}^{\ell_j}\ \{..\}\} \\
C(\texttt{x}_p) &= \{\texttt{function}^{\ell_i}\ \{..\},\ldots,\texttt{function}^{\ell_j}\ \{..\}\} \\
&\vdots \\
C(\texttt{x}_q) &= \{\texttt{function}^{\ell_i}\ \{..\},\ldots,\texttt{function}^{\ell_j}\ \{..\}\}
\end{aligned}
$$

Theorem. 0CFA is PTIME-hard.

Proof (sketch):

- Analysis and evaluation coincide for linear programs.
- Linear programs can express circuits.
- The circuit value problem is PTIME-hard.

```
function (x) {.. x ..}   // x occurs exactly once
```

Observe, in a *linear* program:

- each function can be applied to at most one argument,
- each variable can be bound to at most one value.

Therefore:
Analysis of a linear program coincides exactly with its evaluation.

Theorem. 1CFA is in EXPTIME.

Proof: C grows monotonically. Its size is $O(2^n)$. □

Theorem. 1CFA is EXPTIME-hard.

Proof: ...

Hardness proof of kCFA relies on two insights:

1. Terms are approximated by an exponential number of closures.
2. *Approximation* engenders reevaluation which provides *power*.

Many closures can flow to a single program point:

$$\texttt{function (z) \{ z(x}_1\texttt{)(x}_2\texttt{)...(x}_n\texttt{) \}}$$

- n free variables
- an *exponential* number of environments map these variables to contours of length 1 in 1CFA.

```
(function (f0) { f0(0); f0(1) })
(function (x0) {
  function (z) { z(x0) }})
```

$\langle$`function (z) { z(x0) }`$, [x_0 \mapsto 0]\rangle$
$\langle$`function (z) { z(x0) }`$, [x_0 \mapsto 1]\rangle$

```
(function (f0) { f0(0); f0(1) })
(function (x0) {
  (function (f1) { f1(0); f1(1) })
  (function (x1) {
    function (z) { z(x0)(x1) }})})
```

$$\langle \texttt{function (z) \{ z(x}_0\texttt{)(x}_1\texttt{) \}}, [x_0 \mapsto 0, x_1 \mapsto 0]\rangle$$
$$\langle \texttt{function (z) \{ z(x}_0\texttt{)(x}_1\texttt{) \}}, [x_0 \mapsto 1, x_1 \mapsto 0]\rangle$$
$$\langle \texttt{function (z) \{ z(x}_0\texttt{)(x}_1\texttt{) \}}, [x_0 \mapsto 0, x_1 \mapsto 1]\rangle$$
$$\langle \texttt{function (z) \{ z(x}_0\texttt{)(x}_1\texttt{) \}}, [x_0 \mapsto 1, x_1 \mapsto 1]\rangle$$

The rest is just using this idea to construct a TM simulator that runs a machine for an exponential number of steps.

(See ICFP'08 for details.)

k-CFA	EXPTIME
⋮	
2-CFA	EXPTIME
1-CFA	EXPTIME
0-CFA	PTIME
Sub0-CFA	PTIME
Simple closure analysis	PTIME
⋮	

Part III

Paradox

A unifying view of functions vs. objects:

A function is just an object with an `apply` method.

Both data- and control-flow are recursively intertwined:

```
n(o) { o.m(x) }
```

It is easy to translate the construction to Java?

```
(function (f) { f(0); f(1) })
(function (x) {
  function (z) { z(x) }})

new Fun() { apply(f) { f.apply(0);f.apply(1) }}.apply(
new Fun() { apply(x) {
   new Fun() { apply(z) { z.apply(x) }}}})
```

Object-oriented variants of kCFA are widely used...

...and they are implemented in Datalog!

$$\mathrm{PTIME} \subsetneq \mathrm{EXPTIME}$$

Part IV

Resolution

Traditionally, Java programs are analyzed in lifted form:

```
new Fun() { apply(f) { f.apply(0);f.apply(1) }}.apply(
new Fun() { apply(x) {
   new Fun() { apply(z) { z.apply(x) }}}})

class Fun1 { apply(f) { f.apply(0); f.apply(1) }};
class Fun2 { apply(x) { new Fun3(x) }};
class Fun3 {
  x;
  Fun3(x) { this.x = x };
  apply(z) { z.apply(x) }
};
new Fun1().apply(new Fun2())
```

But now objects are *explicitly* closed.

Consider:

```
class Fun3 {
  x;
  Fun3(x) { this.x = x };
  apply(z) { z.apply(x) }
};

new Fun3(p);
new Fun3(q);
```

Creates two closures over one variable.

Now consder:

```
class FunN {
  x0; ... xn;
  FunN { this.x0 = x0; ... this.xn = xn };
  apply(z) { z.apply(x0)...apply(xn) }
};

new Fun3(p1,...,pn);
new Fun3(q1,...,qn)
```

Creates 2 closures over n variables; we don't consider 2^n closures.

Theorem. kCFA of lifted object-oriented programs is in PTIME.

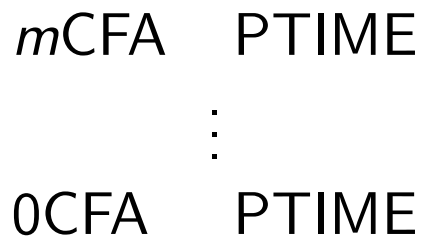
Part V

Exploitation

Key idea: simulate lifting with the representation of closures.

- $\text{Exp} \times (\text{Var} \to \text{Contour})$
- $\text{Exp} \times \text{Contour}$

At call sites, copy values of closures into the store.


mCFA PTIME
0CFA PTIME

Speed: Pathological-case benchmark

Terms	$k = 1$	$m = 1$	poly.,k=1
69	ϵ	ϵ	ϵ
123	ϵ	ϵ	ϵ
231	46 s	ϵ	2 s
447	∞	3 s	5 s
879	∞	48 s	1 m 8 s
1743	∞	51 m	∞

Speed and precision:

Prog/Terms	$k = 1$		$m = 1$		poly.,k=1	
eta / 49	ϵ	7	ϵ	7	ϵ	3
map / 157	ϵ	8	ϵ	8	ϵ	8
sat / 223	∞	-	ϵ	12	1s	12
regex / 1015	4s	25	3s	25	14s	25
scm2java / 2318	5s	86	3s	86	3s	79
interp / 4289	5s	123	4s	123	9s	123
scm2c / 6219	179s	136	143s	136	157s	131

Column 1: analysis time

Column 2: inlinings enabled

What to take home with you:

- Linearity subverts approximation
- Closures make kCFA hard
- Lifting makes object-oriented kCFA easy
- Flat closures make functional mCFA easy
- mCFA v kCFA: similar precision, lower cost