Linear Logic and... the Complexity of Flow Analysis

Harry Mairson
David Van Horn
Brandeis University

Introduction

We investigate the *precision* of static, compile-time analysis, and the necessary analytic *tradeoff* with the computational *resources* that go into the analysis.

- Why kCFA as the subject of the investigation?
  Some form of CFA is used in most forms of analyses for higher-order languages.

  (Heintze and McAllester 1997)

- Why a complexity theoretic investigation?
  Program analysis is still far from being able to precisely relate ingredients of different approaches to one another [. . .].

  (Nielson et al. 1999)

Mathematics Genealogy Project

Olin Shivers

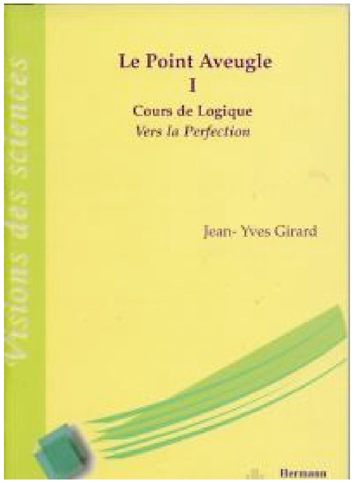
MathSciNet

Ph.D. Carnegie Mellon University 1991

Dissertation: *Control Flow Analysis of High Order Languages*

Advisor 1: Peter Lee
Advisor 2: Allen Newell

Le point aveugle *(à l'américain)*



Compiler implementors have been using control flow analysis of higher-order programs for decades, but nobody really seems to know what the cost is, or what the complexity is of the problem being solved.

Deuxième point aveugle

If your program is running for *exponential time*, and you have a *polynomial-time* control flow analyzer, **what can it possibly observe?** *(An analytical, not empirical answer...)*

Deuxième point aveugle

If your program is running for *exponential time*, and you have a *polynomial-time* control flow analyzer, **what can it possibly observe?** *(An analytical, not empirical answer...)*

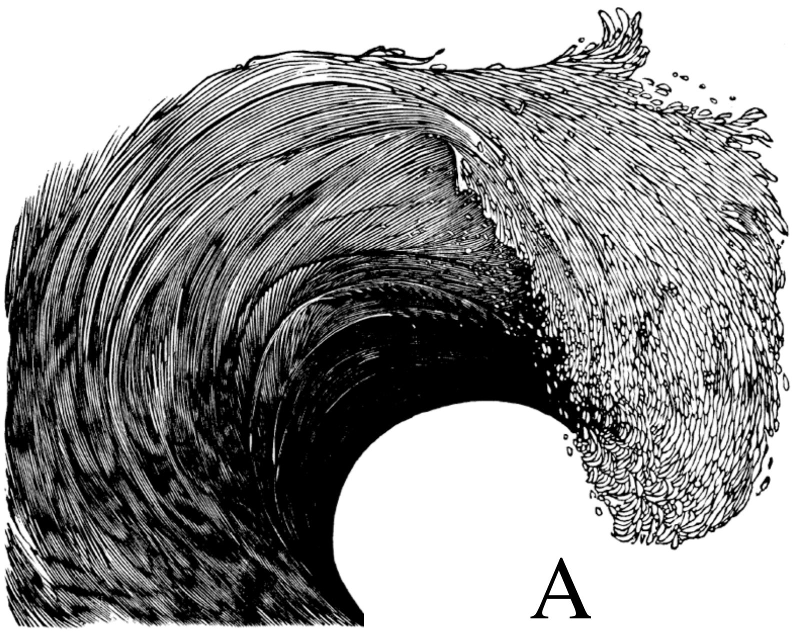
Wouldn't it be like reading *Moby-Dick?*

CALL ME ISHMAEL ...

Deuxième point aveugle

If your program is running for *exponential time*, and you have a *polynomial-time* control flow analyzer, **what can it possibly observe?** *(An analytical, not empirical answer...)*

Wouldn't it be like reading *Moby-Dick*, only with all the consonants removed?



A E I AE

Outline

- Some ideas from CFA -- what's it about?
- PTIME-completeness of 0CFA
  Linearity and non-affine Boolean circuits
- EXPTIME-completeness of kCFA
  (k>0 constant)
  Nonlinearity and Cartesian products
  (a weird exponential...)

CFA Primer

1. For every application, which abstractions can be applied?
2. For every abstraction, to which arguments can it be applied?

Question— can call site A eventually call program P in context C? (What's a context?)

Intuition— the more information we compute about contexts (whatever they are), the more precisely we can answer the question. *But this takes work*.

*Exact answer via **geometry of interaction.***
*Approximate answer via **CFA**, with false positives...*

*Exact analysis via **geometry of interaction***
*(**CFA:** approximation, with false positives...)*

Type directed flow analysis (example: CIL/System I [Boston U.])
bounded rank intersection types x: $\alpha\wedge\beta$ *describes*
sharing by variable occurrences
non-idempotency $\alpha\wedge\alpha \neq \alpha$ *and control flow*

... but:
type inference = normalization in every instance
(Mairson/Neergaard, ICFP 2004)
and with very weak expressiveness:
normalization $\lesssim$ *k finite developments*

+ typed LC (idem.), or linear (non-idem.)LC?

Preliminaries: the Language

The language:

$$e ::= t^{\ell} \qquad \text{expressions (labeled terms)}$$

$$t ::= x \mid (e\ e) \mid (\lambda x.e) \qquad \text{terms (unlabeled expressions)}$$

For example:

$$((\lambda f.((f^1 f^2)^3 (\lambda y.y^4)^5)^6)^7 (\lambda x.x^8)^9)^{10}$$

Preliminaries: Contours

Contours are strings of @-labels of length $\leq k$.
Contour environments map variable names to contours.

$$\begin{array}{rcccl} \delta & \in & \Delta & = & \mathbf{Lab}^{\leq k} \\ ce & \in & \mathbf{CEnv} & = & \mathbf{Var} \rightarrow \Delta \end{array}$$

Contours describe the context in which a term evaluates.
$(e_0([(\lambda x.e_1)]e_2)^{\ell_1})^{\ell_2} \Rightarrow$ "e_1 *evaluates in contour* $\ell_2\ell_1$."

Contour environments describe the context in which a variable was bound.
$(e_0([(\lambda x.e_1)]e_2)^{\ell_1})^{\ell_2} \Rightarrow x \mapsto \ell_2\ell_1 \Rightarrow$ "x *bound in contour* $\ell_2\ell_1$."

A poor cousin of Lévy's labelled λ-calculus...

Polyvariance

During reduction, a function may copy its argument:

$$((\lambda f. \cdots (f e_1)^{\ell_1} \cdots (f e_2)^{\ell_2} \cdots)(\lambda x.e))$$

Contours and environments let us talk about e in each of the distinct calling contexts.

Preliminaries: the Analysis

An analysis is a table $\widehat{\mathsf{C}}$ that maps a label and contour to a set of abstract closures.

$$\widehat{\mathsf{C}} : \mathbf{Lab} \times \Delta \rightarrow \mathcal{P}(\mathbf{Term} \times \mathbf{CEnv})$$

$$\widehat{\mathsf{C}}(\ell, \delta) = \{\langle \lambda y, ce \rangle, \langle \lambda z, ce' \rangle\}$$

In contour δ, the term labeled ℓ evaluates to either the closure $\langle \lambda y, ce \rangle$, or $\langle \lambda z, ce' \rangle$.

As we'll see, this "or" is one that expresses false positives...

Decision problem

Control Flow Problem (kCFA): Given a closure and a label ℓ and contour δ, does that closure flow into the program point labeled ℓ under δ?

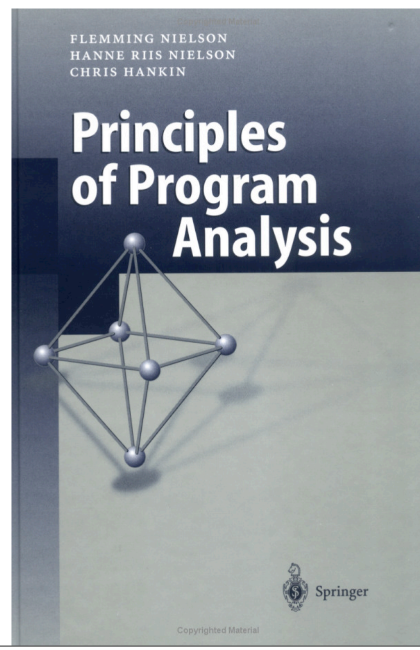
$$\langle \lambda x, ce \rangle \in \widehat{\mathsf{C}}(\ell, \delta)?$$

Acceptability

The analysis is acceptable for e, closed by ce, in contour δ:

$$\widehat{\mathsf{C}} \models_{\delta}^{ce} e$$

The analysis is acceptable for the copy of e that occurs in context described by δ, closed by the environment ce which says what copy of a term each variable is bound to.



Acceptability

$$\widehat{\mathsf{C}} \models_{\delta}^{ce} x^{\ell} \quad \text{iff} \quad \widehat{\mathsf{C}}(x, ce(x)) \subseteq \widehat{\mathsf{C}}(\ell, \delta)$$

$$\widehat{\mathsf{C}} \models_{\delta}^{ce} (\lambda x.e)^{\ell} \quad \text{iff} \quad \langle(\lambda x.e), ce_0\rangle \in \widehat{\mathsf{C}}(\ell, \delta)$$

where $ce_0 = ce|\mathbf{fv}(\lambda x.e_0)$

$$\widehat{\mathsf{C}} \models_{\delta}^{ce} (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell} \quad \text{iff} \quad \widehat{\mathsf{C}} \models_{\delta}^{ce} t_1^{\ell_1} \land \widehat{\mathsf{C}} \models_{\delta}^{ce} t_2^{\ell_2} \land$$

$$\forall\langle(\lambda x.t_0^{\ell_0}), ce_0\rangle \in \widehat{\mathsf{C}}(\ell_1, \delta) :$$

$$\widehat{\mathsf{C}} \models_{\delta_0}^{ce_0'} t_0^{\ell_0} \land$$

$$\widehat{\mathsf{C}}(\ell_2, \delta) \subseteq \widehat{\mathsf{C}}(x, \delta_0) \land$$

$$\widehat{\mathsf{C}}(\ell_0, \delta_0) \subseteq \widehat{\mathsf{C}}(\ell, \delta)$$

where $\delta_0 = \lceil \delta, \ell \rceil_k$ and $ce_0' = ce_0[x \mapsto \delta_0]$

Acceptability

$\widehat{\mathsf{C}} \models^{ce}_{\delta} x^{\ell}$ iff $\widehat{\mathsf{C}}($ … (ℓ, δ)

$\widehat{\mathsf{C}} \models^{ce}_{\delta} (\lambda x.e)^{\ell}$ … $e_0)$

$\widehat{\mathsf{C}} \models^{ce}_{\delta} (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell}$ … $\delta)$:

… $x, \delta_0) \wedge$

Mr. Yuck: Ingesting formalisms may cause *rigor mortis*

where $\delta_0 = \lceil \delta, \ell \rceil_k$ and $ce_0' = ce_0[x \mapsto \delta_0]$

Evaluator

Evaluate the term t, which is closed under environment ce.

$$\mathcal{E}[\![t^{\ell}]\!]^{ce}_{\delta}$$

Write the result into location (ℓ, δ) of the table.

Turning the CFA constraints into an evaluator is merely the construction of a so-called *abstract interpretation.*

Evaluator (exact)

$$
\begin{array}{lcl}
\mathcal{E}[\![x^\ell]\!]^{ce}_\delta & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \widehat{\mathsf{C}}(x, ce(x)) \\
\mathcal{E}[\![(\lambda x.e_0)^\ell]\!]^{ce}_\delta & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \langle \lambda x.e_0, ce_0 \rangle \\
 & & \quad \text{where } ce_0 = ce|\mathbf{fv}(\lambda x.e_0) \\
\mathcal{E}[\![(t_1^{\ell_1} t_2^{\ell_2})^\ell]\!]^{ce}_\delta & = & \mathcal{E}[\![t_1^{\ell_1}]\!]^{ce}_\delta ; \mathcal{E}[\![t_2^{\ell_2}]\!]^{ce}_\delta ; \\
 & & \text{let } \langle \lambda x.t_0^{\ell_0}, ce_0 \rangle = \widehat{\mathsf{C}}(\ell_1,\delta) \text{ in} \\
 & & \quad \widehat{\mathsf{C}}(x, \underline{\delta, \ell}) \leftarrow \widehat{\mathsf{C}}(\ell_2,\delta); \\
 & & \quad \mathcal{E}[\![t_0^{\ell_0}]\!]^{ce_0[x \mapsto \delta,\ell]}_{\delta,\ell} ; \\
 & & \quad \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \widehat{\mathsf{C}}(\ell_0, \underline{\delta, \ell})
\end{array}
$$

X ← Y means X := X ∪ Y

If e has an exact kCFA analysis, then $\mathcal{E}[\![e]\!]^{[\,]}_\epsilon$ constructs it.

Exact vs Inexact analysis

An *exact* analysis is a table $\widehat{\mathsf{C}}$ that maps label-contour pairs to *an* abstract closure.

$$\widehat{\mathsf{C}}(\ell, \delta) = \{\langle \lambda y, ce \rangle\}$$

In contour δ, the term labeled ℓ evaluates to either the closure $\langle \lambda y, ce \rangle$.

Exact vs Inexact analysis

An *inexact analysis* is a table $\widehat{\mathsf{C}}$ that maps label-contour pairs to *sets of* abstract closures.

$$\widehat{\mathsf{C}}(\ell, \delta) = \{\langle \lambda y, ce \rangle, \langle \lambda z, ce' \rangle\}$$

In contour δ, the term labeled ℓ evaluates to either the closure $\langle \lambda y, ce \rangle$, or $\langle \lambda z, ce' \rangle$.

This "or" is one that expresses false positives...

Aside: Variant of the Halting problem

For a given program, e,

$$\exists k.e \text{ has an exact } k\text{CFA?}$$

All normalizing programs have an exact kCFA (for some k), so clearly this is just the halting problem.

Evaluator (inexact)

$$
\begin{array}{lcl}
\mathcal{E}[\![x^\ell]\!]^{ce}_\delta & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \widehat{\mathsf{C}}(x, ce(x)) \\
\mathcal{E}[\![(\lambda x.e_0)^\ell]\!]^{ce}_\delta & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \{\langle \lambda x.e_0, ce_0\rangle\} \\
 & & \quad \text{where } ce_0 = ce|\mathbf{fv}(\lambda x.e_0) \\
\mathcal{E}[\![(t_1^{\ell_1} t_2^{\ell_2})^\ell]\!]^{ce}_\delta & = & \mathcal{E}[\![t_1^{\ell_1}]\!]^{ce}_\delta ; \mathcal{E}[\![t_2^{\ell_2}]\!]^{ce}_\delta ; \\
 & & \forall \langle \lambda x.t_0^{\ell_0}, ce_0\rangle \in \widehat{\mathsf{C}}(\ell_1,\delta) : \\
 & & \quad \widehat{\mathsf{C}}(x, \lceil \delta,\ell\rceil_k) \leftarrow \widehat{\mathsf{C}}(\ell_2,\delta); \\
 & & \quad \mathcal{E}[\![t_0^{\ell_0}]\!]^{ce_0[x\mapsto \lceil \delta,\ell\rceil_k]}_{\lceil \delta,\ell\rceil_k}; \\
 & & \quad \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \widehat{\mathsf{C}}(\ell_0, \lceil \delta,\ell\rceil_k)
\end{array}
$$

The kCFA analysis of e is constructed by iterating $\mathcal{E}[\![e]\!]^{[\,]}_\epsilon$ until $\widehat{\mathsf{C}}$ reaches a fixed point.

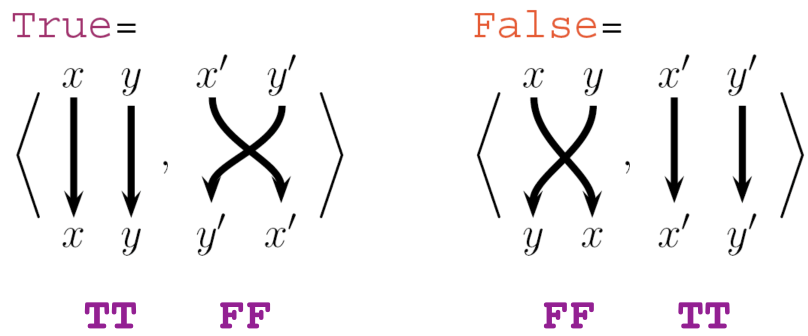
Evaluator (exact, $k = 0$)

$$
\begin{array}{lcl}
\mathcal{E}[\![x^\ell]\!] & = & \widehat{\mathsf{C}}(\ell) \leftarrow \widehat{\mathsf{C}}(x) \\
\mathcal{E}[\![(\lambda x.e_0)^\ell]\!] & = & \widehat{\mathsf{C}}(\ell) \leftarrow (\lambda x.e_0) \\
\mathcal{E}[\![(t_1^{\ell_1} t_2^{\ell_2})^\ell]\!] & = & \mathcal{E}[\![t_1^{\ell_1}]\!]; \mathcal{E}[\![t_2^{\ell_2}]\!]; \\
 & & \text{let } (\lambda x.t_0^{\ell_0}) = \widehat{\mathsf{C}}(\ell_1) \text{ in} \\
 & & \quad \widehat{\mathsf{C}}(x) \leftarrow \widehat{\mathsf{C}}(\ell_2, \delta); \\
 & & \quad \mathcal{E}[\![t_0^{\ell_0}]\!]; \\
 & & \quad \widehat{\mathsf{C}}(\ell) \leftarrow \widehat{\mathsf{C}}(\ell_0)
\end{array}
$$

This is an evaluator, but for what language?

The linear λ-calculus.

Booleans, the non-affine variation




```
- fun TT (x:'a,y:'a)= (x,y);
val TT = fn : 'a * 'a -> 'a * 'a
- fun FF (x:'a,y:'a)= (y,x);
val FF = fn : 'a * 'a -> 'a * 'a

- val True= (TT: ('a * 'a -> 'a * 'a), FF: ('a * 'a -> 'a * 'a));
val True = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
- val False= (FF: ('a * 'a -> 'a * 'a), TT: ('a * 'a -> 'a * 'a));
val False = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
```

(This is also how to do Boolean circuit computation in MLL.
And how to do non-affine, simply-typed Boolean calculation...)

Why study MLL normalization?

Multiplicative Linear Logic is a baby programming language:

$\otimes$ is a linear pairing of
expressions (**`cons`**)
expression and continuation (@)

$\parr$ is a linear unpairing of
expressions (π, π')
expression and continuation (λ)

complexity of normalization = complexity of interpreter

Symmetric logic gates

And (p,p′)(q,q′) = (p∧q,p′∨q′) = (p∧q,∼(p∧q))

```
- fun And (p,p') (q,q')=
        let val ((u,v),(u',v')) = (p (q,FF), p' (TT,q'))
       in (u,Compose (Compose (u',v),Compose (v',FF))) end;
val And = fn
  : ('a * ('b * 'b -> 'b * 'b) -> 'c * ('d -> 'e))
    * ((('f * 'f -> 'f * 'f) * 'g -> ('e -> 'h) * ('i * 'i -> 'd))
    -> 'a * 'g -> 'c * ('i * 'i -> 'h)

- And True False;
val it = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)

- Show (And True True);
val it = (true,false) : bool * bool
- Show (And True False);
val it = (false,true) : bool * bool
- Show (And False True);
val it = (false,true) : bool * bool
- Show (And False False);
val it = (false,true) : bool * bool
```

Why is there no garbage?

$$\texttt{And (p,p')(q,q') = } (p\wedge q, p'\vee q') = (p\wedge q, \sim(p\wedge q))$$

```
fun And (p,p') (q,q')=
    let val ((u,v),(u',v')) = (p (q,FF), p' (TT,q'))
    in (u,Compose (Compose (u',v),Compose (v',FF))) end;
```

```
(u,v)   = (p (q,FF))
(u',v') = (p'(TT,q'))
```

When `p=TT`,

```
(u,v)   = (q, FF)
(u',v') = (q',TT)
```

When `p=FF`,

```
(u,v)   = (FF,q)
(u',v') = (TT,q')
```

Thus `{v,v'}={TT,FF},` and

`Compose (v,Compose(v',FF))= TT` (identity function)
`Compose (Compose (u',v),Compose (v',FF))= u'`

"Symmetric garbage is self-annihilating"

Symmetric logic gates (2)

Or (p,p′)(q,q′) = (p∨q,p′∧q′) = (p∨q,∼(p∨q))

```
- fun Or (p,p') (q,q')=
        let val ((u,v),(u',v')) = (p (TT,q), p' (q',FF))
        in (u,Compose (Compose (u',v),Compose (v',FF))) end;
val Or = fn
  : (('a * 'a -> 'a * 'a) * 'b -> 'c * ('d -> 'e))
    * ('f * ('g * 'g -> 'g * 'g) -> ('e -> 'h) * ('i * 'i -> 'd))
    -> 'b * 'f -> 'c * ('i * 'i -> 'h)

- Or True False;
val it = (fn,fn) :  ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)

- Show (Or True True);
val it = (true,false) : bool * bool
- Show (Or True False);
val it = (true,false) : bool * bool
- Show (Or False True);
val it = (true,false) : bool * bool
- Show (Or False False);
val it = (false,true) : bool * bool
```

Symmetric logic gates (3)

```
- fun Not (x,y)= (y,x);
val Not = fn : 'a * 'b -> 'b * 'a

- Not True;
val it = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)

- Show (Not True);
val it = (false,true) : bool * bool
- Show (Not False);
val it = (true,false) : bool * bool
```

Symmetric logic gates (4)

```
- fun Copy (p,p')= (p (TT,FF), p' (FF,TT));
val Copy = fn
  : (('a * 'a -> 'a * 'a) * ('b * 'b -> 'b * 'b) -> 'c)
    * (('d * 'd -> 'd * 'd) * ('e * 'e -> 'e * 'e) -> 'f)
    -> 'c * 'f
```

Set `'a = 'b = 'd = 'e` and `'c = 'f = ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)`:

```
  (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) ->
   ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
* (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) ->
   ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
->
  (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
* (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
```

`[p= TT]:   Copy (p,p')= ((TT,FF), (TT,FF))` [second component reversed]
`[p= FF]:   Copy (p,p')= ((FF,TT), (FF,TT))` [first component reversed]

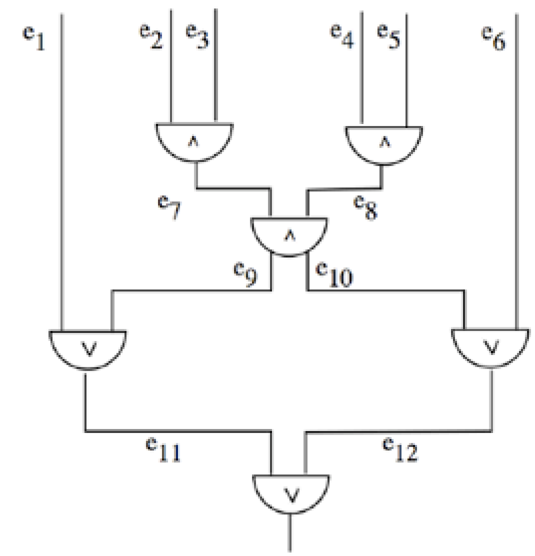
```
- let val (p,q)= Copy True in (Show p, Show q) end;
val it = ((true,false),(true,false)) : (bool * bool) * (bool * bool)
- let val (p,q)= Copy False in (Show p, Show q) end;
val it = ((false,true),(false,true)) : (bool * bool) * (bool * bool)
```

Boolean computations with non-affine linear terms

```
- fun Andgate p q k= ···
- fun Orgate p q k= ···
- fun Notgate p k= ···
- fun Copygate p k= ···
```

Straight-line code:

```
- fun Circuit e1 e2 e3 e4 e5 e6=
   (Andgate e2 e3 (fn e7=>
   (Andgate e4 e5 (fn e8=>
   (Andgate e7 e8 (fn f=>
   (Copygate f (fn (e9,e10)=>
   (Orgate e1 e9 (fn e11=>
   (Orgate e10 e6 (fn e12=>
   (Orgate e11 e12 (fn Output=> Output)))))))))))));
val Circuit = fn : < big type... >
```



The Widget

Let $E =$

```
let val (u,u')= [] in
let val ((x,y),(x',y'))= (u (f,g), u' (f',g')) in
    ((x a, y b),(x' a', y' b')) end end;
```

In $E[\phi\vec{v}]$: Does a flow as an argument to f?

- Clearly, a flows as an argument to x.
- So does f flow into x?
- (f,g) flows into (x,y) iff TT flows into u.
- TT flows into u iff $\phi\vec{v} =$ True.
- Either $\widehat{\mathsf{C}}(\mathtt{x}) = \{\mathtt{f}\}$ or $\widehat{\mathsf{C}}(\mathtt{x}) = \{\mathtt{g}\}$, but not $\widehat{\mathsf{C}}(\mathtt{x}) = \{\mathtt{f}, \mathtt{g}\}$

Evaluator (inexact, $k = 0$)

$$
\begin{array}{lcl}
\mathcal{E}[\![x^\ell]\!] & = & \widehat{\mathsf{C}}(\ell) \leftarrow \widehat{\mathsf{C}}(x) \\
\mathcal{E}[\![(\lambda x.e_0)^\ell]\!] & = & \widehat{\mathsf{C}}(\ell) \leftarrow \{(\lambda x.e_0)\} \\
\mathcal{E}[\![(t_1^{\ell_1} t_2^{\ell_2})^\ell]\!] & = & \mathcal{E}[\![t_1^{\ell_1}]\!]; \mathcal{E}[\![t_2^{\ell_2}]\!]; \\
 & & \forall (\lambda x.t_0^{\ell_0}) \in \widehat{\mathsf{C}}(\ell_1): \\
 & & \quad \widehat{\mathsf{C}}(x) \leftarrow \widehat{\mathsf{C}}(\ell_2); \\
 & & \quad \mathcal{E}[\![t_0^{\ell_0}]\!]; \\
 & & \quad \widehat{\mathsf{C}}(\ell) \leftarrow \widehat{\mathsf{C}}(\ell_0)
\end{array}
$$

- Read $\widehat{\mathsf{C}}(\ell) \leftarrow \{\lambda x, \lambda y\}$ as $\widehat{\mathsf{C}}(\ell) := \widehat{\mathsf{C}}(\ell) \cup \{\lambda x, \lambda y\}$.
- Many terms may flow into an application: apply them all.
- Iterate $\mathcal{E}$ until the table reaches a fixed point.

PTIME Completeness of 0CFA

- Hardness: LOGSPACE-reducible to Circuit Value.
- Inclusion: Well known, eg. PPA Nielson et al. (1999):
  - 0CFA computes a binary relation over a *fixed structure* (the graph description of a program).
  - The computation of the relation is *monotone*: begins empty and is added to incrementally.
  - A *fixed point* must be reached by this incremental computation (structure is finite).
  - The binary relation can be at most *polynomial in size*, and each increment is *computed in polynomial time*.

0CFA is PTIME-complete

*k*CFA

"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 have 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)

The CFA bottleneck: ***closures***

Because CFA makes approximations, many closures can flow to a single program point and contour. In 1CFA, for example,

$$(\lambda w.w x_1 x_2 \ldots x_n)$$

has n free variables, with an exponential number of ***possible*** associated environments mapping these variables to program points (contours of length 1).

Padding and polyvariance

Recall: $((\lambda f. \cdots (f e_1)^{\ell_1} \cdots (f e_2)^{\ell_2} \cdots)(\lambda x.e))$

In our constructions, we evaluate the same code π over and over, but in multiple contexts.

How can we make sure the contour doesn't reveal the context in which the code is evaluated?

$\pi \Rightarrow ((\lambda x.x)((\lambda x.x)((\lambda x.x)\ldots((\lambda x.x)\ P)^{\ell_k}\ldots)^{\ell_3})^{\ell_2})^{\ell_1}$

The iterated $((\lambda x.x)\ldots)^{\ell_i}$ removes stuff from the enclosing context, replacing it with *uniform trash* $\ell_k \ell_{k-1} \ell_{k-2} \ldots$

Exactness and complexity

Hardness of 1CFA relies on two insights:

1. Program points are approximated by an exponential number of closures.
2. *Inexactness* of analysis engenders *reevaluation* which provides *computational power*.
   A less precise analysis "yields coarser approximations, and thus induces more merging. More merging leads to more propagation, which in turn leads to more reevaluation."

(Wright and Jagannathan 1998)

1CFA as SAT solver

$$
\begin{array}{l}
(\lambda f_1.(f_1\ \texttt{True})(f_1\ \texttt{False})) \\
(\lambda x_1. \\
\quad (\lambda f_2.(f_2\ \texttt{True})(f_2\ \texttt{False})) \\
\quad (\lambda x_2. \\
\qquad (\lambda f_3.(f_3\ \texttt{True})(f_3\ \texttt{False})) \\
\qquad (\lambda x_3. \\
\qquad\qquad \cdots \\
\qquad\qquad (\lambda f_n.(f_n\ \texttt{True})(f_n\ \texttt{False})) \\
\qquad\qquad (\lambda x_n. \\
\qquad\qquad\quad E[(\lambda v.\phi\ v)\boxed{(\lambda w.w x_1 x_2 \cdots x_n)}]) \cdots))))
\end{array}
$$

Approximation allows us to bind each x_i to either of the closed λ-terms for `True` and `False`.

The Widget, Again

$E =$

```
let val (u,u')= [] in
let val ((x,y),(x',y'))= (u (f,g), u' (f',g')) in
    ((x a, y b),(x' a', y' b')) end end;
```

In $E[(\lambda v.\phi\ v)(\lambda w.wx_1x_2 \cdots x_n)]$:

`f` is applied to `a` iff ϕ is satisfiable.

1CFA is NP-hard

$(k > 1)$CFA is just as hard. The construction is “padded” to undo the added precision of longer contours.

kCFA is NP-hard

Naïve exponential algorithm for kCFA

- The $\widehat{\mathsf{C}}$ table is finite and has n^{k+1} entries.
- Each entry contains a set of closures.
- The environment maps p free variables to any one of n^k contours.
- There are n possible λx terms and n^{kp} environments, so each entry contains at most n^{1+kp} closures.
- Approximate evaluation is monotonic, and there are at most $n^{1+(k+1)p}$ updates to $\widehat{\mathsf{C}}$
- $p \leq n$ so kCFA $\in$ EXPTIME

kCFA in NP?

$$
\begin{array}{lcl}
\mathcal{E}[\![x^{\ell}]\!]_{\delta}^{ce} & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \widehat{\mathsf{C}}(x, ce(x)) \\
\mathcal{E}[\![(\lambda x.e_0)^{\ell}]\!]_{\delta}^{ce} & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \langle \lambda x.e_0, ce_0 \rangle \\
 & & \quad \text{where } ce_0 = ce|\mathbf{fv}(\lambda x.e_0) \\
\mathcal{E}[\![(t_1^{\ell_1} t_2^{\ell_2})^{\ell}]\!]_{\delta}^{ce} & = & \mathcal{E}[\![t_1^{\ell_1}]\!]_{\delta}^{ce}; \mathcal{E}[\![t_2^{\ell_2}]\!]_{\delta}^{ce}; \\
 & & \forall \langle \lambda x.t_0^{\ell_0}, ce_0 \rangle \in \widehat{\mathsf{C}}(\ell_1,\delta) : \\
 & & \quad \widehat{\mathsf{C}}(x, \lceil \delta, \ell \rceil_k) \leftarrow \widehat{\mathsf{C}}(\ell_2, \delta); \\
 & & \quad \mathcal{E}[\![t_0^{\ell_0}]\!]_{\lceil \delta, \ell \rceil_k}^{ce_0[x \mapsto \lceil \delta, \ell \rceil_k]}; \\
 & & \quad \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \widehat{\mathsf{C}}(\ell_0, \lceil \delta, \ell \rceil_k)
\end{array}
$$

Can we guess our way through the computation to answer the kCFA decision problem? No!

$(\lambda f.\ (f\ \mathrm{True})\ (f\ \mathrm{False}))$

$(\lambda x.$

$\quad(\lambda p.\ p\ (\lambda u.\ p\ (\lambda v.\ (\supset u\ v))))\ (\lambda w.\ w\ x))$

A toy calculation, with insights

[⊃ is just classical (TT) implication]

The expression $p\ (\lambda\ u.\ p\ (\lambda\ v.\ (\supset u\ v)))$ is evaluated *twice* -- with x bound to True, then to False. The variable p appears ***nonlinearly***.

$(\lambda f.\ (f\ True)\ (f\ False))$

$(\lambda x.$

$\quad (\lambda p.\ p\ (\lambda u.\ p\ (\lambda v.\ (\supset u\ v))))\ (\lambda w.\ w\ x))$

A toy calculation, with insights

[$\supset$ is just classical (TT) implication]

The expression $p\ (\lambda\ u.\ p\ (\lambda\ v.\ (\supset u\ v)))$ is evaluated *twice* -- with f bound to True, then to False. The variable p appears ***nonlinearly***.

Two closures flow out at the program point for $(\lambda w.\ w\ x)$.

Question: What values flow out at the program point for $(\supset u\ v)$?

$(\lambda f.\ (f\ \mathrm{True})\ (f\ \mathrm{False}))$

$(\lambda x.$

$\quad (\lambda p.\ p\ (\lambda u.\ p\ (\lambda v.\ (\supset u\ v))))\ (\lambda w.\ w\ x))$

A toy calculation, with insights

[⊃ is just classical (TT) implication]

The expression $p\ (\lambda\ u.\ p\ (\lambda\ v.\ (\supset u\ v)))$ is evaluated *twice* -- with f bound to True, then to False. The variable p appears ***nonlinearly***.

Two closures flow out at the program point for $(\lambda w.\ w\ x)$.

Question: What values flow out at the program point for $(\supset u\ v)$?

Answer: both True *and* False. Not true in normalization!

A toy calculation, with insights

$(\lambda f.\ (f\ \mathrm{True})\ (f\ \mathrm{False}))$
$(\lambda x.$
$\quad (\lambda p.\ p\ (\lambda u.\ p\ (\lambda v.\ (\supset u\ v))))\ (\lambda w.\ w\ x))$

[$\supset$ is just classical (TT) implication]

The expression $p\ (\lambda\ u.\ p\ (\lambda\ v.\ (\supset u\ v)))$ is evaluated *twice* -- with f bound to True, then to False. The variable p appears ***nonlinearly***.

Two closures flow out at the program point for $(\lambda\ w.\ w\ x)$.

Question: What values flow out at the program point for $(\supset u\ v)$?

Answer: both True *and* False. Not true in normalization!

Important observation: We got the two separate closures to "talk to" each other in a single (approximate) overlapped computation. This *isn't* normalization---it's *computing with the approximation.*

Goal: exploit this phenomenon to simulate EXPTIME.

Coding: split up Turing machine ID into many pieces

<T S H C b> (which we'll code as a tuple $\lambda w.wTSHCb$ ...)

"At time T, Turing machine M was in state S, the tape position was at cell H, and cell C held contents b."

T, S, H, C are **blocks** of bits with **fixed** size
(*polynomial* in length, representing *exponential* values).
One ID is represented by an *exponential* number of tuples (varying C and b).
Now define a *binary* (piecemeal) transition function δ
whose iteration gives a *fixed point* (of *tuples*, and in ***cache***):

Coding: split up Turing machine ID into many pieces

<T S H C b> (which we'll code as a tuple *λw.wTSHCb* ...)

"At time T, Turing machine M was in state S, the tape position was at cell H, and cell C held contents b."

T, S, H, C are **blocks** of bits with **fixed** size
(*polynomial* in length, representing *exponential* values).
One ID is represented by an *exponential* number of tuples (varying C and b).
Now define a *binary* (piecemeal) transition function δ
whose iteration gives a *fixed point* (of *tuples*, and in ***cache***):

δ <**T** S H H b> <**T** S' H' C' b'> = <**T+1** δ_Q(S,b) δ_{LR}(S,H,b) δ_Σ(S,b)>
(*Compute:* transition to next ID -- *head, cell address coincide*)

δ <**T+1** S H C b> <**T** S' H' C' b'> = <**T+1** S H C' b'>
(*Communicate:* copy state and head position to the other tuples)

δ <**anything else**> <**anything else**> = *<some goofy null value>*

All Boolean functions---and we have all the Boolean logic.
We need to compute *cross product* on ID fragments for *binary* δ, and set up the initial ID.

Setting up initial ID, iterator, and test

$(\lambda f_1.\ (f_1\ 0)\ (f_1\ 1))$

$(\lambda z_1.$

$(\lambda f_2.\ (f_2\ 0)\ (f_2\ 1))$

$(\lambda z_2.$

...

$(\lambda f_k.\ (f_k\ 0)\ (f_k\ 1))$

$(\lambda z_k.$

(let $\Phi =$ *coding of transition function of TM*

in Widget [Extract

$(2^n\ \Phi\ (\lambda w.\ w\ 0 \ldots 0\ Q_0\ H_0\ z_1 z_2 \ldots z_k\ 0))])\ )\ \ldots\ ))$

< T S H C b>

2^n is a fixed point operator (Y), or exponential function composer

Extract extracts final ID (with time stamp!) and checks if it codes accepting state, returning True or False accordingly

Widget is our standard control flow test...

$(\lambda f_1.\ (f_1\ 0)\ (f_1\ 1))$

(...

$(\lambda z_k.$

(let Φ= *coding of transition function of TM*

in Widget [Extract

$(2^n\ \Phi\ (\lambda w.\ w\ 0 \ldots 0\ Q_0\ H_0\ z_1 z_2 \ldots z_k\ 0))])\ )\ \ldots\ ))$

< T S H C b>

2^n is a fixed point operator (Y), or exponential function composer

Extract extracts final ID (with time stamp!) and checks if it codes accepting state, returning True or False accordingly

Widget is our standard control flow test:

```
let val (u,u')= [-] in
let val ((x,y),(x',y'))= (u (f,g) u' (f',g')) in
    ((x a, y b), (x' a', y' b')) end end;
```

Theorem. In CFA, a flows as an argument to f iff TM accepts in 2^n steps.

Corollary. The CFA decision problem is complete for EXPTIME.

Transition function Φ

$$\Phi = (\lambda p.\ p\ (\lambda x_1.\lambda x_2 \ldots \lambda x_m.\ p\ (\lambda y_1.\lambda y_2 \ldots \lambda y_m.\ (\varphi\ x_1 x_2 \ldots x_m\ y_1 y_2 \ldots y_m))))$$

$\varphi = \lambda x_1.\lambda x_2 \ldots \lambda x_m.\lambda y_1.\lambda y_2 \ldots \lambda y_m.$

<copy all the inputs appropriately>

$(\lambda w.w\ (\varphi_T\ \mathbf{x}_1\mathbf{y}_1)\ (\varphi_S\ \mathbf{x}_2\mathbf{y}_2)\ \ldots\ (\varphi_b\ \mathbf{x}_m\mathbf{y}_m))$

$(\lambda w_T.\lambda w_S.\lambda w_H.\lambda w_C.\lambda w_b.$

$w_T(\lambda z_1.\lambda z_2 \ldots \lambda z_T.$

$w_S(\lambda z_{T+1}.\lambda z_{T+2} \ldots \lambda z_{T+S}.$

$\ldots$

$w_b(\lambda z_{C+1}.\lambda z_{C+2} \ldots \lambda z_{C+b=m}.$

$\lambda w.w z_1 z_{2 \ldots} z_m)))\ldots)$

$\lambda w.w z_1 z_{2 \ldots} z_m$ is the closure returned as the value of Φ

Lunatic fringe extensions of this result

$p(n)$CFA is complete for EXPTIME, where $p(n)$ is any polynomial in program length

$2^{p(n)}$CFA is complete for 2EXPTIME, where $p(n)$ is any polynomial in program length

... and so on ...

Note the "exponential jump"...

Q: Where is this "exponential jump" coming from?

A: The cardinality of the *environments* in closures [f(n)= contour length, n= program length]:

$$|\,\mathrm{Var} \rightarrow \mathrm{Contour}\,| = (n^{f(n)})^n = 2^{f(n) n \log n}$$

This *cardinality of environments* effectively determines the *size of the universe of values* for the abstract interpretation realized by CFA.

(Our lower bound is also one on the complexity of realizing abstract interpretation.)

Q: Why only EXPTIME, if we're iterating with Y?

The EXPTIME coding (why not 2EXPTIME?) is the "limit" because with a polynomial-length tuple (as constrained by a logspace reduction), you can only code an exponential number of closures.

Why linearity is important in static analysis

Simple type inference: all bound variables have the same type. (Also ML, with qualifications.)

CFA: shared occurrences of the same closure have overlapping flow information.

Linearity removes these constraints, and lets us study the problems as if they are simply normalization.

Conclusions

What's hard about CFA is not the *length of the contours* (information about calling contexts), but the *existence of closures*. As soon as there are closures, you're done for.

Linear logic and *programming linearly* is the right way to think about *polyvariance* in control flow analysis---if information about multiple variable occurrences is *merged* in analysis, linear terms make CFA equivalent to normalization. And nonlinearity can be used to get "nonstandard" computations.