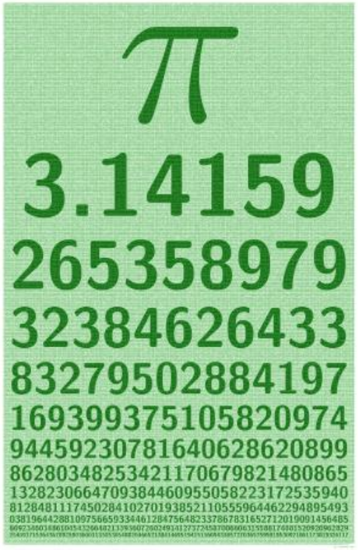
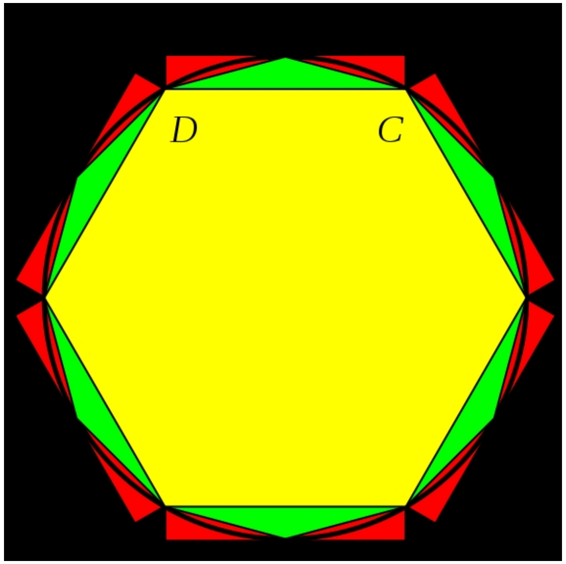
The Complexity of Flow Analysis in Higher-Order Languages

David Van Horn

Thanks

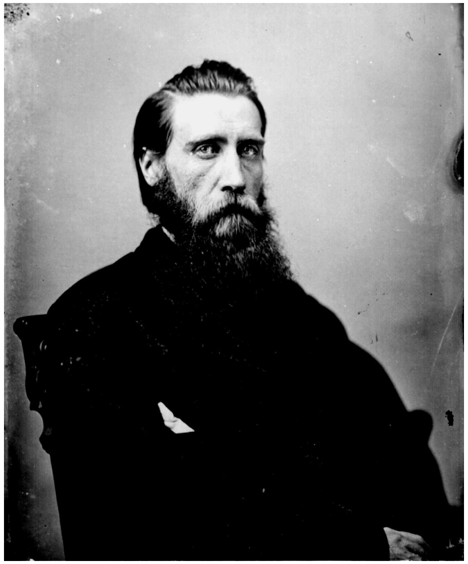
A brief history of today

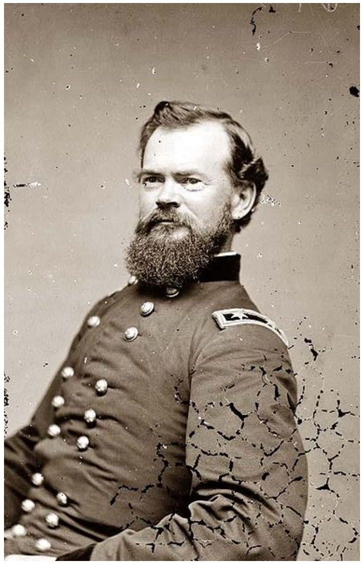




22/7 is π-approximation day.

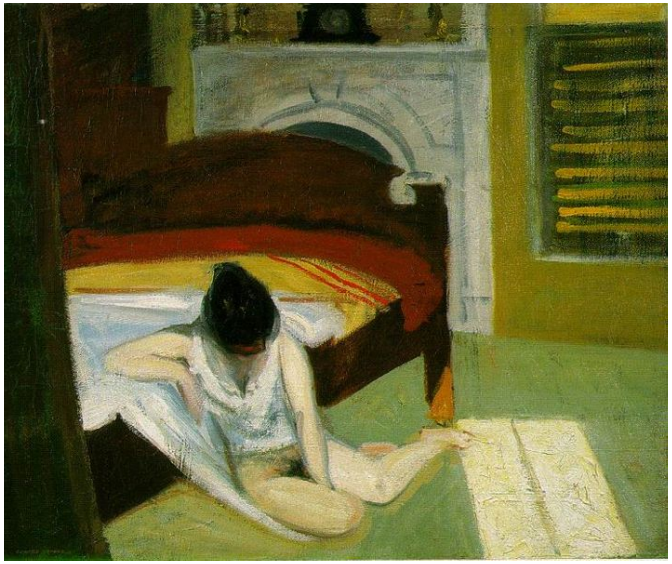
A brief history of today

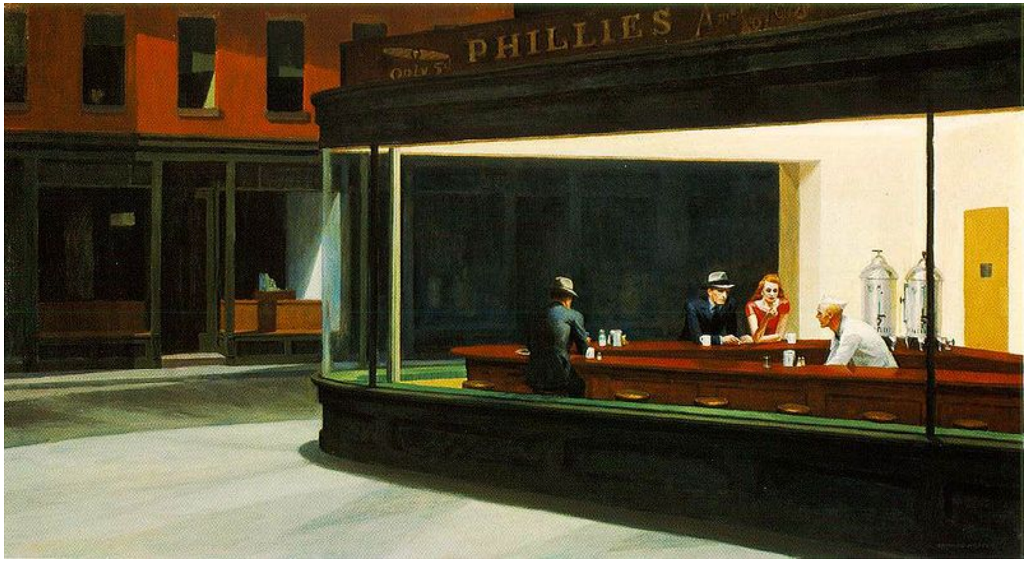




1864: The Battle of Atlanta.

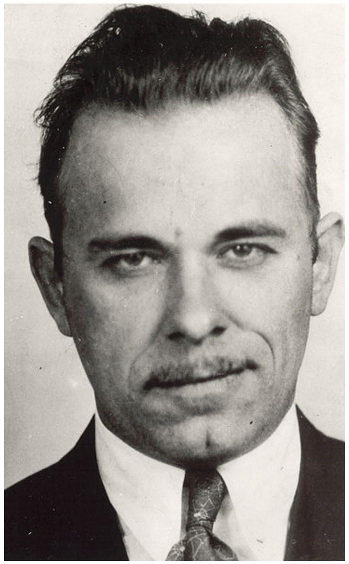
A brief history of today





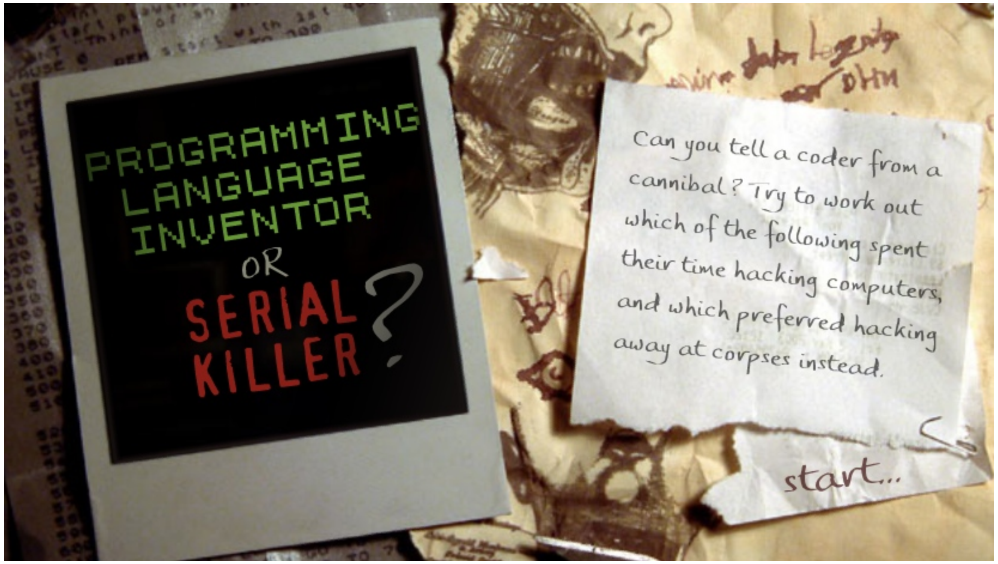
1882: Edward Hopper is born.

A brief history of today



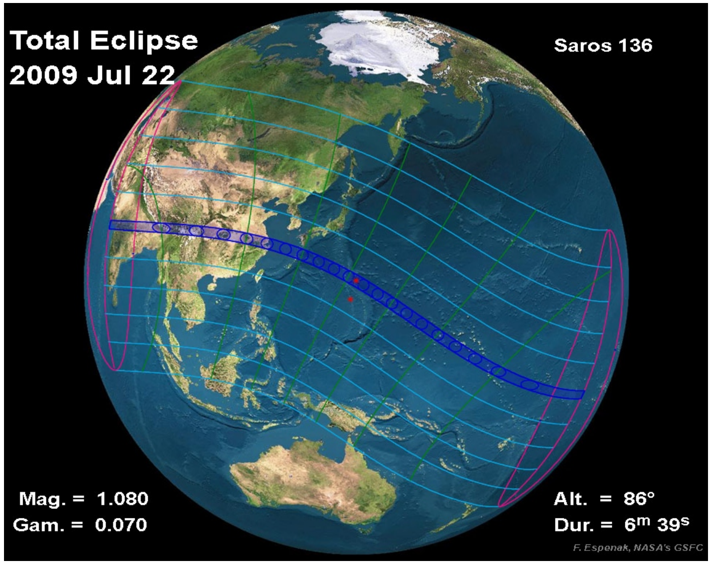
1934: John Dillinger is shot dead in front of the Biograph.

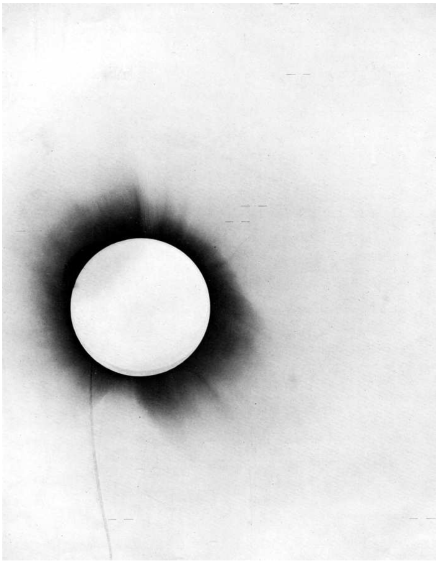
A brief history of today

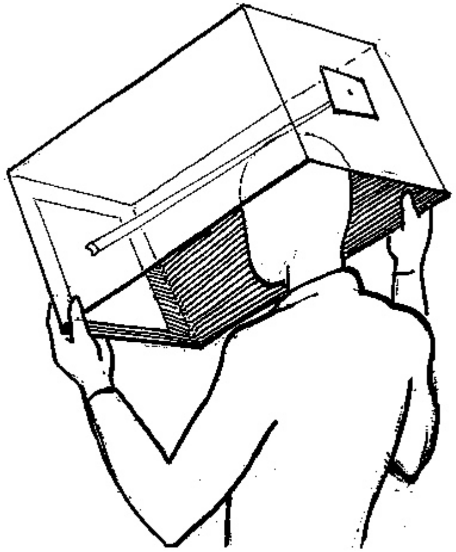


1991: Police in Milwaukee arrest serial killer Jeffrey Dahmer.

A brief history of today







2009: Total eclipse of the Sun.

in memory of William Gordon Mercer
July 22, 1927 – October 8, 2007

Assumptions

- ⋆ functional programming
- ⋆ interpreters (evaluators)
- ⋆ the λ-calculus
- ⋆ basic computational complexity theory
- ⋆ fundamentals of program analysis

What is *program analysis*?

Program analysis *predicts the future*:

"what happens when P is run?"

- $\star$ Resource usage (time, space, energy, bandwidth, . . .)
- $\star$ Errors (div. by 0, type errors, . . .)
- $\star$ Effects (I/O, missile launch, . . .)
- $\star$ Numerical correctness (overflow, rounding, . . .)

But, predicting the future is mostly undecidable.

Out of the Turing tar pit...

What to do about undecidable problems?

- ⋆ Consider *decidable subcases*
- ⋆ Use *interactive human advice*
- ⋆ Accept *nontermination*
- ⋆ Accept (sound) *approximation*

. . . and into the complexity zoo

"Researchers have expended a great deal of effort deriving clever ways to tame the cost of the analysis."

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

To what extent is this possible? What are the fundamental limitations on taming the cost of analysis?

Thesis

A complexity-theoretic investigation of flow analysis in higher-order languages provides insight into the fundamental limitations on the cost of performing analysis.

What is *flow analysis*?

Flow analysis answers questions such as:

- ⋆ For each application, which functions may be applied?

Due to first-class functions, this necessarily requires answering the more general question:

- ⋆ For each subexpression, what may it evaluate to?

In other words, flow analysis can be understood as the sound approximation to program evaluation.

Precision and complexity

Flow analysis is concerned with the sound approximation of run-time values at compile time.

Analysis answers decision problems such as:
does expression e *possibly evaluate to value* v*?*

- ⋆ The most approximate analysis always answers *yes*.
  — no resources to compute, but useless
- ⋆ The most precise analysis answers *yes* iff e evaluates to v.
  — useful, but unbounded resources to compute

For tractability, there is a necessary sacrifice of information in static analysis. (A blessing and a curse.)

Flow analysis example

$$\texttt{let } f \texttt{ = } \lambda x.x \texttt{ in } (f f)(\lambda y.y)$$

- f may be bound to $\lambda x.x$.
- x may be bound to $\lambda y.y$ or $\lambda x.x$.
- $(f f)$ may evaluate to $\lambda y.y$ or $\lambda x.x$.
- $\ldots$
- program may evaluate to $\lambda y.y$ or $\lambda x.x$.

A typical example showing the *imprecision* of analysis.

Syntax

$e ::= t^{\ell}$	expressions (or labeled terms)
$t ::= x \mid ee \mid \lambda x.e$	terms (or unlabeled expressions)

Evaluator

An *evaluator* (or *interpreter*) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.

$$
\begin{aligned}
\mathcal{E}[\![x^\ell]\!]\rho &= \rho(x) \\
\mathcal{E}[\![(\lambda x.e)^\ell]\!]\rho &= \langle \lambda x.e, \rho' \rangle \text{ where } \rho' = \rho \restriction \mathbf{fv}(\lambda x.e) \\
\mathcal{E}[\![(t^{\ell_1} t^{\ell_2})^\ell]\!]\rho &= \textbf{let } \langle \lambda x.t^{\ell_0}, \rho' \rangle = \mathcal{E}[\![t^{\ell_1}]\!]\rho \textbf{ in} \\
&\quad\; \textbf{let } v = \mathcal{E}[\![t^{\ell_2}]\!]\rho \textbf{ in} \\
&\quad\quad \mathcal{E}[\![t^{\ell_0}]\!]\rho'[x \mapsto v]
\end{aligned}
$$

Evaluation examples

$$\begin{aligned}
\mathcal{E}[\![\lambda x.x]\!]\bullet &= \langle \lambda x.x, \bullet\rangle \\
\mathcal{E}[\![(\lambda x.\lambda z.x)(\lambda y.y)]\!]\bullet &= \langle \lambda z.x, [x \mapsto \langle \lambda y.y, \bullet\rangle]\rangle \\
\mathcal{E}[\![(\lambda f.(f f)(\lambda y.y))(\lambda x.x)]\!]\bullet &= \langle \lambda y.y, \bullet\rangle
\end{aligned}$$

Instrumented evaluator

An *instrumented evaluator* (or *instrumented interpreter*) for a programming language is a procedure that, when applied to an expression of the language, performs *and records* the actions required to evaluate that expression.
Flow analysis actions:

- ⋆ Every time the value of a subexpression is computed, record its value and the context in which it was evaluated.

$$\mathsf{C} : \mathbf{Lab} \times \Delta \rightharpoonup \mathbf{Val}$$

- ⋆ Every time a variable is bound, record the value and context in which it was bound.

$$\mathsf{r} : \mathbf{Var} \times \Delta \rightharpoonup \mathbf{Val}$$

Contours

Contours describe contexts:

$$((\lambda f.(f(f\ \texttt{True})^1)^2)(\lambda y.\texttt{False}))^3$$

321 describes $((\lambda f.(f[\ ]^1)^2)(\lambda y.\texttt{False}))^3$

32 describes $((\lambda f.[\ ]^2)(\lambda y.\texttt{False}))^3$

δ	$\in$	Δ	$=$	$\mathbf{Lab}^\star$	contours
v	$\in$	$\mathbf{Val}$	$=$	$\mathbf{Term} \times \mathbf{Env}$	(contour) values
ce	$\in$	$\mathbf{Env}$	$=$	$\mathbf{Var} \rightharpoonup \Delta$	(contour) environments

Instrumented evaluator

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

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

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

C$(\ell, \delta) = v$ means t^{ℓ} evaluates to v in context δ.

Instrumented evaluator

In imperative style:

$$
\begin{array}{lcl}
\mathcal{I}[\![x^\ell]\!]^{ce}_\delta & = & \mathsf{C}(\ell,\delta) \leftarrow \mathsf{r}(x, ce(x)) \\
\mathcal{I}[\![(\lambda x.e)^\ell]\!]^{ce}_\delta & = & \mathsf{C}(\ell,\delta) \leftarrow \langle \lambda x.e, ce' \rangle \\
 & & \quad \text{where } ce' = ce \restriction \mathbf{fv}(\lambda x.e) \\
\mathcal{I}[\![(t^{\ell_1} t^{\ell_2})^\ell]\!]^{ce}_\delta & = & \mathcal{I}[\![t^{\ell_1}]\!]^{ce}_\delta;\ \mathcal{I}[\![t^{\ell_2}]\!]^{ce}_\delta; \\
 & & \text{let } \langle \lambda x.t^{\ell_0}, ce' \rangle = \mathsf{C}(\ell_1,\delta) \text{ in} \\
 & & \quad \mathsf{r}(x, \delta\ell) \leftarrow \mathsf{C}(\ell_2,\delta); \\
 & & \quad \mathcal{I}[\![t^{\ell_0}]\!]^{ce'[x \mapsto \delta\ell]}_{\delta\ell}; \\
 & & \quad \mathsf{C}(\ell,\delta) \leftarrow \mathsf{C}(\ell_0, \delta\ell)
\end{array}
$$

Abstract values and caches

Abstract values:

$$\hat{v} \in \widehat{\mathbf{Val}} = \mathcal{P}(\mathbf{Term} \times \mathbf{Env}) \quad \text{abstract values.}$$

Abstract caches:

$$\begin{aligned} \widehat{\mathsf{C}} &: \mathbf{Lab} \times \Delta \to \widehat{\mathbf{Val}} \\ \hat{\mathsf{r}} &: \mathbf{Var} \times \Delta \to \widehat{\mathbf{Val}} \end{aligned}$$

Soundness:

$$\mathcal{E}[\![P^\ell]\!] = \langle \lambda x.e, \rho \rangle \Longrightarrow \langle \lambda x.e, ce \rangle \in \widehat{\mathsf{C}}(\ell)$$

Abstract evaluator

Imperative style:

$$
\begin{array}{lcl}
\mathcal{A}_k[\![x^\ell]\!]^{ce}_\delta & = & \widehat{\mathsf{C}}(\ell,\delta) \twoheadleftarrow \hat{\mathsf{r}}(x, ce(x)) \\
\mathcal{A}_k[\![(\lambda x.e)^\ell]\!]^{ce}_\delta & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \{\langle \lambda x.e, ce'\rangle\} \\
 & & \quad \text{where } ce' = ce \restriction \mathbf{fv}(\lambda x.e) \\
\mathcal{A}_k[\![(t^{\ell_1} t^{\ell_2})^\ell]\!]^{ce}_\delta & = & \mathcal{A}_k[\![t^{\ell_1}]\!]^{ce}_\delta ; \mathcal{A}_k[\![t^{\ell_2}]\!]^{ce}_\delta ; \\
 & & \textbf{for each } \langle \lambda x.t^{\ell_0}, ce'\rangle \textbf{ in } \widehat{\mathsf{C}}(\ell_1,\delta) \textbf{ do} \\
 & & \quad \hat{\mathsf{r}}(x, \lceil \delta\ell \rceil_k) \twoheadleftarrow \widehat{\mathsf{C}}(\ell_2,\delta); \\
 & & \quad \mathcal{A}_k[\![t^{\ell_0}]\!]^{ce'[x \mapsto \lceil \delta\ell \rceil_k]}_{\lceil \delta\ell \rceil_k} ; \\
 & & \quad \widehat{\mathsf{C}}(\ell,\delta) \twoheadleftarrow \widehat{\mathsf{C}}(\ell_0, \lceil \delta\ell \rceil_k)
\end{array}
$$

Flow analysis decision problem

Given an expression e, an abstract value v, and a pair (ℓ, δ), does v flow into (ℓ, δ) by this flow analysis?

Complexity basics

Problems are categories into classes:

- ⋆ *inclusion*: no harder than hardest problems in class
- ⋆ *hardness*: no easier than hardest problems in class
- ⋆ *completeness*: both included and hard for the class

A *lower bound* establishes the minimum computational requirements it takes to solve a class of problems.

The classes used today:

- ⋆ **LOGSPACE**: space efficient
- ⋆ **PTIME**: feasible, but inherently sequential
- ⋆ **EXPTIME**: intractable

Plan

- ⋆ Monovariant analysis; 0CFA and friends
- ⋆ Flow analysis and linear logic
- ⋆ kCFA
- ⋆ Conclusion

Monovariant analysis

Approximation of monovariance

- ⋆ Closures approximated by code component
- ⋆ All occurrences of bound variables are merged

0CFA algorithm

$$
\begin{array}{lcl}
\mathcal{A}[\![x^\ell]\!] & = & \widehat{\mathsf{C}}(\ell) \twoheadleftarrow \hat{\mathsf{r}}(x) \\
\mathcal{A}[\![(\lambda x.e)^\ell]\!] & = & \widehat{\mathsf{C}}(\ell) \leftarrow \{\lambda x.e\} \\
\mathcal{A}[\![(t^{\ell_1} t^{\ell_2})^\ell]\!] & = & \mathcal{A}[\![t^{\ell_1}]\!];\ \ \mathcal{A}[\![t^{\ell_2}]\!]; \\
 & & \textbf{for each } \lambda x.t^{\ell_0} \textbf{ in } \widehat{\mathsf{C}}(\ell_1) \textbf{ do} \\
 & & \quad \hat{\mathsf{r}}(x) \twoheadleftarrow \widehat{\mathsf{C}}(\ell_2);\ \mathcal{A}[\![t^{\ell_0}]\!];\ \widehat{\mathsf{C}}(\ell) \twoheadleftarrow \widehat{\mathsf{C}}(\ell_0)
\end{array}
$$

- $\widehat{\mathsf{C}}(\ell) \leftarrow \hat{v}$ means update $\widehat{\mathsf{C}}$ so $\widehat{\mathsf{C}}(\ell) = \hat{v}$
- $\widehat{\mathsf{C}}(\ell) \twoheadleftarrow \hat{v}$ means update $\widehat{\mathsf{C}}$ so $\widehat{\mathsf{C}}(\ell) = \hat{v} \cup \widehat{\mathsf{C}}(\ell)$

A subversive approach to lower bound

“We can regard almost any program as the evaluator for some language.”

Abelson and Sussman, SICP

Idea:

- ⋆ Identify the sources of approximation
- ⋆ Hack expressive computations avoiding these sources
- ⋆ Analysis will *evaluate* the computation

From *May* to *Must*

Single and loving it, Jagannathan, et al. POPL'98:

$$\widehat{\mathsf{C}}(\ell) = \{\lambda x.e\}$$

"Subexpression ℓ ~~may~~ *must* eval to $\lambda x.e$."

(An idea also used by Might and Shivers, ICFP'06, "abstract counting".)

Linearity and evaluation

Since in a *linear* λ-term,

- $\star$ each abstraction can be applied to at most one argument
- $\star$ each variable can be bound to at most one value

Analysis of a linear term coincides exactly with its evaluation.

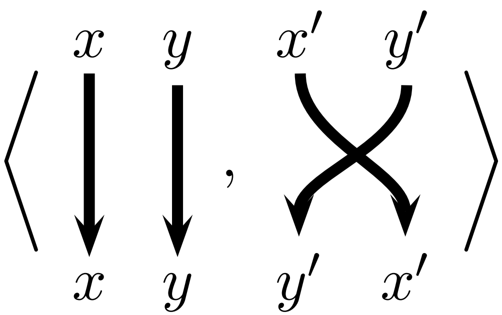
Theorem 0. *If e is linear and $\widehat{\mathsf{C}}$ is an analysis of e, $\widehat{\mathsf{C}}$ is a* complete *description of running the program, i.e. $\widehat{\mathsf{C}} \approx \mathsf{C}$.*

Symmetric logic gates

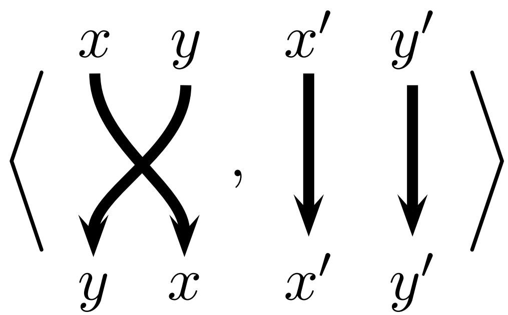
```
- fun TT(x,y)= (x,y);
- fun FF(x,y)= (y,x);

- val True=  (TT, FF);
- val False= (FF, TT);
```

Booleans built out of constants TT, FF

`True=`



`False=`



To *twist*, or not to *twist*: that is the question.

Symmetric copy gate

```
- fun Copy (p,p')= (p (TT,FF), p' (FF,TT));
```

Symmetric copy gate

```
- fun Copy (p,p')= (p (TT,FF), p' (FF,TT));
```

```
[p= TT]: Copy (p,p') = ((TT,FF), (TT,FF))
```

[second component reversed]

Symmetric copy gate

```
- fun Copy (p,p')= (p (TT,FF), p' (FF,TT));
```

```
[p= TT]: Copy (p,p') = ((TT,FF), (TT,FF))
```

[second component reversed]

```
[p= FF]: Copy (p,p') = ((FF,TT), (FF,TT))
```

[first component reversed]

Symmetric garbage self-annihilates

$\texttt{And (p,p') (q,q')} \equiv \texttt{(p}\wedge\texttt{q, p'}\vee\texttt{q')} \equiv \texttt{(p}\wedge\texttt{q,} \neg\texttt{(p'}\wedge\texttt{q'))}$

```
- 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;
```

Symmetric garbage self-annihilates

$\texttt{And (p,p') (q,q')} \equiv \texttt{(p}\wedge\texttt{q, p'}\vee\texttt{q')} \equiv \texttt{(p}\wedge\texttt{q,} \neg\texttt{(p'}\wedge\texttt{q'))}$

```
- 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;
```

When p=TT (identity),

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

When p=FF (twist),

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

Symmetric garbage self-annihilates

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;
```

When p=TT (identity),

(u,v) = (q,FF)
(u′,v′) = (q′,TT)

When p=FF (twist),

(u,v) = (FF,q)
(u′,v′) = (TT,q′)

So, {v,v′}={TT,FF}, and

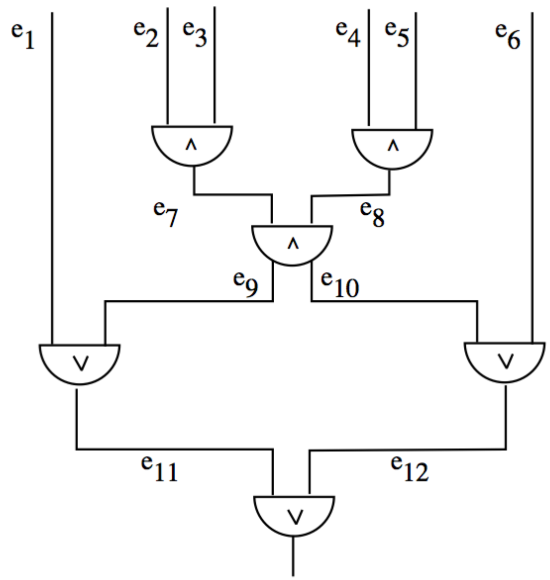
Compose (v,Compose(v′,FF)) = TT

Compose (Compose (u′,v),Compose (v′,FF)) = u′

Symmetric logic gates

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

} linear ML terms



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... >
```

Lower bound on 0CFA

Circuit Value Problem:

Given a Boolean circuit C of n inputs and one output, and truth values $\vec{x} = x_1, \ldots, x_n$, is $\vec{x}$ accepted by C?

Theorem 1. *If analysis corresponds to evaluation on linear terms, it is* $\mathbf{PTIME}$*-hard.*

Theorem 2. *0CFA is complete for* $\mathbf{PTIME}$*.*

Further approximations

The best 0CFA algorithm is nearly cubic.

Several further approximations to 0CFA have been developed:

- ⋆ Henglein's simple closure analysis
- ⋆ Ashley and Dybvig's Sub-0CFA
- ⋆ Heintze and McAllester's "subtransitive" flow analysis
- ⋆ Mossin's flow graphs

But. . . *on linear programs* they are all equivalent to each other and to evaluation.

Theorem 3. *All of the above analyses are complete for* $\mathbf{PTIME}$*.*

Flow analysis and linear logic

Outline

- ⋆ An introduction to Sharing Graphs
- ⋆ 0CFA in graphical form
- ⋆ Linear graphs and normalization
- ⋆ A simple (re-)proof of correspondence between 0CFA and evaluation for linear terms
- ⋆ First-class control and direct analysis
- ⋆ η-expansion of simply-typed programs
- ⋆ Normalization in **LOGSPACE** (Mairson and Terui)
- ⋆ Adaptation to 0CFA of non-linear programs

kCFA

kCFA

For any $k > 0$, we prove the flow analysis decision problem is complete for deterministic exponential time ($\mathbf{EXPTIME}$).

This theorem:

- ⋆ gives an exact characterization of the computational complexity of the kCFA hierarchy
- ⋆ validates empirical observations that kCFA is intractable

Proving lower bounds

A lower bound establishes the minimum computational requirements it takes to solve a class of problems.

kCFA is *provably intractable* (**EXPTIME**-hard)

The *proof* goes by construction:

Proving lower bounds

A lower bound establishes the minimum computational requirements it takes to solve a class of problems.

kCFA is *provably intractable* (**EXPTIME**-hard)

The *proof* goes by construction:

- ⋆ given the description of a Turing machine and its input,

Proving lower bounds

A lower bound establishes the minimum computational requirements it takes to solve a class of problems.

kCFA is *provably intractable* (**EXPTIME**-hard)

The *proof* goes by construction:

- ⋆ given the description of a Turing machine and its input,
- ⋆ produce an instance of the kCFA problem,

Proving lower bounds

A lower bound establishes the minimum computational requirements it takes to solve a class of problems.

kCFA is *provably intractable* ($\mathbf{EXPTIME}$-hard)

The *proof* goes by construction:

- ⋆ given the description of a Turing machine and its input,
- ⋆ produce an instance of the kCFA problem,
- ⋆ whose analysis faithfully simulates the TM on the input

Proving lower bounds

A lower bound establishes the minimum computational requirements it takes to solve a class of problems.

kCFA is *provably intractable* (**EXPTIME**-hard)

The *proof* goes by construction:

- ⋆ given the description of a Turing machine and its input,
- ⋆ produce an instance of the kCFA problem,
- ⋆ whose analysis faithfully simulates the TM on the input
- ⋆ for an exponential number of steps.

Proving lower bounds

A lower bound establishes the minimum computational requirements it takes to solve a class of problems.

kCFA is *provably intractable* (**EXPTIME**-hard)

The *proof* goes by construction:

- ⋆ given the description of a Turing machine and its input,
- ⋆ produce an instance of the kCFA problem,
- ⋆ whose analysis faithfully simulates the TM on the input
- ⋆ for an exponential number of steps.

A compiler!

Proving lower bounds

A lower bound establishes the minimum computational requirements it takes to solve a class of problems.

kCFA is *provably intractable* (**EXPTIME**-hard)

The *proof* goes by construction:

- ⋆ given the description of a Turing machine and its input,
- ⋆ produce an instance of the kCFA problem,
- ⋆ whose analysis faithfully simulates the TM on the input
- ⋆ for an exponential number of steps.

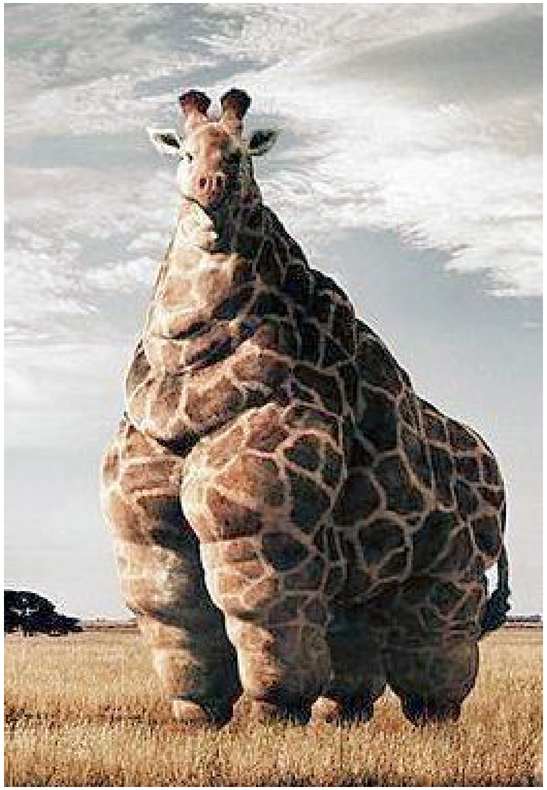
A (weird) compiler!

Strange animal

A compiler:

- ⋆ Source language: exponential TMs with input
- ⋆ Target language: the λ-calculus
- ⋆ Interpreter: kCFA (as TM simulator)

$\therefore$ kCFA is complete for $\mathbf{EXPTIME}$.



More strange animals

Other compilers (ICFP'07):

- ⋆ Source language: Boolean formulas
- ⋆ Target language: the λ-calculus
- ⋆ Interpreter: kCFA (as SAT solver)

$\therefore$ kCFA is $\mathbf{NP}$-hard.

More strange animals

Other compilers:

- $\star$ Source language: circuit with inputs
- $\star$ Target language: the linear λ-calculus
- $\star$ Interpreter: 0CFA (as λ evaluator)

$\therefore$ 0CFA is complete for $\mathbf{PTIME}$.

More strange animals

Other compilers (Mairson, JFP'04):

- $\star$ Source language: circuit with inputs
- $\star$ Target language: the linear λ-calculus
- $\star$ Interpreter: type inference (as λ evaluator)

$\therefore$ Simple type inference is complete for $\mathbf{PTIME}$.

More strange animals

Other compilers (Neergaard and Mairson, ICFP'04):

- $\star$ Source language: elementary TMs with input
- $\star$ Target language: the λ calculus
- $\star$ Interpreter: rank-k $\wedge$-type inference (as λ evaluator)

$\therefore$ Rank-k $\wedge$-type inference is complete for $\mathbf{DTIME}(\mathbf{K}(k,n))$.

More strange animals

Other compilers (Mairson, POPL'90):

- $\star$ Source language: exponential TMs with input
- $\star$ Target language: ML
- $\star$ Interpreter: type inference (as ML evaluator)

$\therefore$ ML type inference is complete for **EXPTIME**.

More strange animals

Other compilers (Henglein and Mairson, POPL'91):

- $\star$ Source language: non-elementary TMs with input
- $\star$ Target language: System F$_\omega$
- $\star$ Interpreter: type inference (as System F_ω evaluator)

$\therefore$ F$_\omega$ type inference has a non-elementary lower bound.

A complexity zoo of static analysis

0CFA $\equiv$ Simple closure analysis $\equiv$ Sub-0CFA $\equiv$ Simple type inference $\equiv$ Linear λ-calculus $\equiv$ MLL...

$\subset$

kCFA $\equiv$ ML type inference...

$\subset$

Rank-k intersection type inference...

$\subset$

Exact CFA $\equiv$ Simply typed λ-calculus...

$\subset$

∞CFA $\equiv$ The λ-calculus...

"Program analysis is still far from being able to precisely relate ingredients of different approaches to one another."

Nielson et al., Principles of Program Analysis (1999)

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 (strings of application labels) let us talk about e in each of the distinct calling contexts.

kCFA

Intuition— the more information we compute about contexts, the more precisely we can answer flow questions.
But this takes work.

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

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

kCFA

An *abstraction* of the instrumented evaluator:

$$\widehat{\mathsf{C}} \in \widehat{\mathbf{Cache}} = \mathbf{Lab} \times \mathbf{Lab}^{\leq k} \rightarrow \mathcal{P}(\mathbf{Term} \times \mathbf{Env})$$

$$\begin{array}{rcl} \mathcal{A}[\![(t^{\ell_1} t^{\ell_2})^{\ell}]\!]^{ce}_{\delta} & = & \mathcal{A}[\![t^{\ell_1}]\!]^{ce}_{\delta}; \mathcal{A}[\![t^{\ell_2}]\!]^{ce}_{\delta}; \\ & & \text{foreach } \langle \lambda x.t^{\ell_0}, ce' \rangle \in \widehat{\mathsf{C}}(\ell_1, \delta) : \\ & & \quad \hat{\mathsf{r}}(x, \lceil \delta\ell \rceil_k) \leftarrow \widehat{\mathsf{C}}(\ell_2, \delta); \\ & & \quad \mathcal{A}[\![t^{\ell_0}]\!]^{ce'[x \mapsto \lceil \delta\ell \rceil_k]}_{\lceil \delta\ell \rceil_k}; \\ & & \quad \widehat{\mathsf{C}}(\ell, \delta) \leftarrow \widehat{\mathsf{C}}(\ell_0, \lceil \delta\ell \rceil_k) \end{array}$$

Boolean logic

Coding Boolean logic in linear λ-calculus:

$$\texttt{TT} \equiv \lambda p.\text{let } \langle x, y\rangle = p \text{ in } \langle x, y\rangle \qquad \texttt{True} \equiv \langle \texttt{TT}, \texttt{FF}\rangle$$
$$\texttt{FF} \equiv \lambda p.\text{let } \langle x, y\rangle = p \text{ in } \langle y, x\rangle \qquad \texttt{False} \equiv \langle \texttt{FF}, \texttt{TT}\rangle$$

$$\texttt{Copy} \equiv \lambda b.\text{let } \langle u, v\rangle = b \text{ in } \langle u\langle \texttt{TT}, \texttt{FF}\rangle, v\langle \texttt{FF}, \texttt{TT}\rangle\rangle$$

$$\begin{aligned}\texttt{Implies} \equiv\ & \lambda b_1.\lambda b_2.\\ & \text{let } \langle u_1, v_1\rangle = b_1 \text{ in}\\ & \text{let } \langle u_2, v_2\rangle = b_2 \text{ in}\\ & \text{let } \langle p_1, p_2\rangle = u_1\langle u_2, \texttt{TT}\rangle \text{ in}\\ & \text{let } \langle q_1, q_2\rangle = v_1\langle \texttt{FF}, v_2\rangle \text{ in}\\ & \quad \langle p_1, q_1 \circ p_2 \circ q_2 \circ \texttt{FF}\rangle\end{aligned}$$

Approximation as power tool

Hardness of kCFA 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*.

Abstract closures

Many closures can flow to a single program point:

$$(\lambda w.wx_1x_2\ldots x_n)$$

- $\star$ n free variables
- $\star$ an *exponential* number of possible associated environments mapping these variables to program points (contours of length 1 in 1CFA).

Cache-bound computations

What goes in the Cache...
stays in the Cache.

Toy calculation, with insights

Consider the following *non-linear* example

$$(\lambda f.(f\ \texttt{True})(f\ \texttt{False}))$$
$$(\lambda x.$$
$$(\lambda p.p(\lambda u.p(\lambda v.(\texttt{Implies}\ u\ v))))(\lambda w.wx))$$

Toy calculation, with insights

Consider the following *non-linear* example

$$(\lambda f.(f\ \texttt{True})(f\ \texttt{False}))$$
$$(\lambda x.$$
$$(\lambda p.p(\lambda u.p(\lambda v.(\texttt{Implies}\ u\ v))))(\lambda w.wx))$$

Q: What does $\texttt{Implies}\ u\ v$ evaluate to?

Toy calculation, with insights

Consider the following *non-linear* example

$$(\lambda f.(f\ \texttt{True})(f\ \texttt{False}))$$
$$(\lambda x.$$
$$(\lambda p.p(\lambda u.p(\lambda v.(\texttt{Implies}\ u\ v))))(\lambda w.wx))$$

Q: What does $\texttt{Implies}\ u\ v$ evaluate to ?

A: $\texttt{True}$: it is equivalent to $\texttt{Implies}\ x\ x$, a tautology.

Toy calculation, with insights

Consider the following *non-linear* example

$$(\lambda f.(f\ \texttt{True})(f\ \texttt{False}))$$
$$(\lambda x.$$
$$(\lambda p.p(\lambda u.p(\lambda v.(\texttt{Implies}\ u\ v))))(\lambda w.wx))$$

Q: What does $\texttt{Implies}\ u\ v$ evaluate to ?
A: $\texttt{True}$: it is equivalent to $\texttt{Implies}\ x\ x$, a tautology.

Q: What flows out of $\texttt{Implies}\ u\ v$?

Toy calculation, with insights

Consider the following *non-linear* example

$$(\lambda f.(f\ \texttt{True})(f\ \texttt{False}))$$
$$(\lambda x.$$
$$(\lambda p.p(\lambda u.p(\lambda v.(\texttt{Implies}\ u\ v))))(\lambda w.wx))$$

Q: What does `Implies` $u\ v$ evaluate to ?
A: `True`: it is equivalent to `Implies` $x\ x$, a tautology.

Q: What flows out of `Implies` $u\ v$?
A: **both** `True` and `False`: *Not true evaluation!*

Toy calculation, with insights

Consider the following *non-linear* example

$$(\lambda f.(f\ \texttt{True})(f\ \texttt{False}))$$
$$(\lambda x.$$
$$(\lambda p.p(\lambda u.p(\lambda v.(\texttt{Implies}\ u\ v))))(\lambda w.wx))$$

Q: What does $\texttt{Implies}\ u\ v$ evaluate to?
A: $\texttt{True}$: it is equivalent to $\texttt{Implies}\ x\ x$, a tautology.

Q: What flows out of $\texttt{Implies}\ u\ v$?
A: **both** $\texttt{True}$ and $\texttt{False}$: *Not true evaluation!*

We are *computing with the approximation* (spurious flows).

Jigsaw puzzles, Machines

The idea:

- ⋆ Break machine ID into an exponential number of pieces
- ⋆ Do piecemeal transitions on pairs of puzzle pieces

$$\langle T, S, H, C, b \rangle$$

"At time T, machine is in state S, the head is at cell H, and cell C holds symbol b"

Jigsaw puzzles, Machines

$\langle T, S, H, C, b \rangle$: *"At time T, machine is in state S, the head is at cell H, and cell C holds symbol b"*

1) Compute:

$$\delta \langle T, S, H, H, b \rangle \langle T, S', H', C', b' \rangle =$$
$$\langle T + 1, \delta_Q(S, b), \delta_{LR}(S, H, b), H, \delta_\Sigma(S, b) \rangle$$

Jigsaw puzzles, Machines

$\langle T, S, H, C, b\rangle$: *"At time T, machine is in state S, the head is at cell H, and cell C holds symbol b"*

1) Compute:

$$\delta\langle T, S, H, H, b\rangle\langle T, S', H', C', b'\rangle =$$
$$\langle T+1, \delta_Q(S,b), \delta_{LR}(S,H,b), H, \delta_\Sigma(S,b)\rangle$$

2) Communicate:

$$\delta\langle T+1, S, H, C, b\rangle\langle T, S', H', C', b'\rangle = \langle T+1, S, H, C', b'\rangle$$
$$(H' \neq C')$$

Jigsaw puzzles, Machines

$\langle T, S, H, C, b\rangle$: *"At time T, machine is in state S, the head is at cell H, and cell C holds symbol b"*

1) Compute:

$$\delta\langle T, S, H, H, b\rangle\langle T, S', H', C', b'\rangle = \langle T+1, \delta_Q(S,b), \delta_{LR}(S,H,b), H, \delta_\Sigma(S,b)\rangle$$

2) Communicate:

$$\delta\langle T+1, S, H, C, b\rangle\langle T, S', H', C', b'\rangle = \langle T+1, S, H, C', b'\rangle$$

$$(H' \neq C')$$

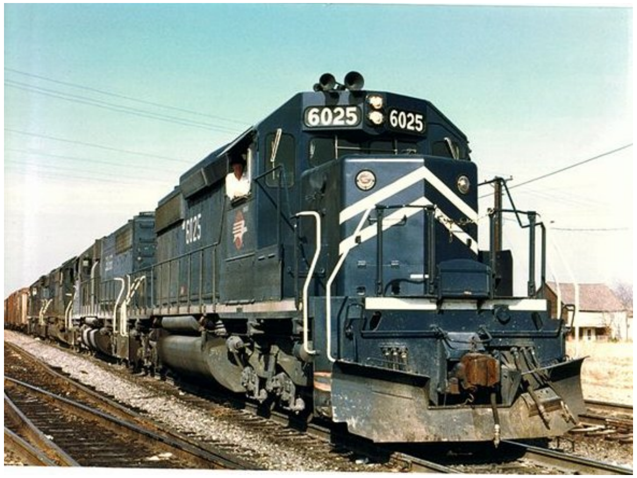
3) Otherwise:

$\delta\langle T, S, H, C, b\rangle\langle T', S', H', C', b'\rangle = \langle$some goofy  null value$\rangle$

$$(T \neq T' \text{ and } T \neq T'+1)$$

The real deal

Setting up initial ID, iterator, and test:



$(\lambda f_1.(f_1\ \mathbf{0})(f_1\ \mathbf{1}))$

$(\lambda z_1.$

$(\lambda f_2.(f_2\ \mathbf{0})(f_2\ \mathbf{1}))$

$(\lambda z_2.$

$\ldots$

$(\lambda f_N.(f_N\ \mathbf{0})(f_N\ \mathbf{1}))$

$(\lambda z_N.$

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

$\texttt{Widget}[\texttt{Extract}(Y\ \Phi\ (\lambda w.w\ \mathbf{0}\ldots\mathbf{0}\ Q_0\ H_0\ z_1 z_2 \ldots z_N\ \mathbf{0}))])) \ldots))$

$\langle T, S, H, \qquad C, b \rangle$

The real deal

$\ldots$ let $\Phi =$ *coding of transition function of TM* in

$$\texttt{Widget}[\texttt{Extract}(Y\ \Phi\ (\lambda w.w\ \mathbf{0}\ldots\mathbf{0}\ Q_0\ H_0\ z_1 z_2 \ldots z_N\ \mathbf{0}))]\ldots$$

$$\langle T, S, H, \qquad C, b\rangle$$

$$\Phi \equiv (\lambda p.p(\lambda x_1.\lambda x_2.\ldots.\lambda x_m.p(\lambda y_1.\lambda y_2\ldots\lambda y_m.$$
$$(\phi x_1 x_2 \ldots x_m y_1 y_2 \ldots y_m))))$$

The real deal

$\ldots$ let $\Phi =$ *coding of transition function of TM* in

$$\texttt{Widget}[\texttt{Extract}(Y\ \Phi\ (\lambda w.w\ \mathbf{0}\ldots\mathbf{0}\ Q_0\ H_0\ z_1 z_2 \ldots z_N\ \mathbf{0}))]\ldots$$

$$\langle T, S, H, \qquad C, b\rangle$$

$$\Phi \equiv (\lambda p.p(\lambda x_1.\lambda x_2.\ldots\lambda x_m.p(\lambda y_1.\lambda y_2\ldots\lambda y_m. (\phi x_1 x_2 \ldots x_m y_1 y_2 \ldots y_m))))$$

$\texttt{Widget}[E] \equiv \ldots f \ldots a \ldots$, where a flows as an argument to f iff a True value flows out of E.

The real deal

$\ldots$ let $\Phi =$ *coding of transition function of TM* in

$$\texttt{Widget}[\texttt{Extract}(Y\ \Phi\ (\lambda w.w\ \mathbf{0}\ldots\mathbf{0}\ Q_0\ H_0\ z_1 z_2 \ldots z_N\ \mathbf{0}))]\ldots$$

$$\langle T, S, H, \qquad C, b\rangle$$

$$\Phi \equiv (\lambda p.p(\lambda x_1.\lambda x_2.\ldots\lambda x_m.p(\lambda y_1.\lambda y_2\ldots\lambda y_m.$$
$$(\phi x_1 x_2 \ldots x_m y_1 y_2 \ldots y_m))))$$

$\texttt{Widget}[E] \equiv \ldots f \ldots a \ldots$, where a flows as an argument to f iff a True value flows out of E.

Theorem 4. In kCFA, a flows to f iff TM accept in 2^n steps.

The real deal

$\ldots$ let $\Phi =$ *coding of transition function of TM* in

$$\texttt{Widget}[\texttt{Extract}(Y\ \Phi\ (\lambda w.w\ \mathbf{0}\ldots\mathbf{0}\ Q_0\ H_0\ z_1 z_2 \ldots z_N\ \mathbf{0}))]\ldots$$

$$\langle T, S, H, \qquad C, b\rangle$$

$$\Phi \equiv (\lambda p.p(\lambda x_1.\lambda x_2.\ldots\lambda x_m.p(\lambda y_1.\lambda y_2\ldots\lambda y_m.$$
$$(\phi x_1 x_2 \ldots x_m y_1 y_2 \ldots y_m))))$$

$\texttt{Widget}[E] \equiv \ldots f \ldots a \ldots$, where a flows as an argument to f iff a True value flows out of E.

Theorem 4. In kCFA, a flows to f iff TM accept in 2^n steps.
Theorem 5. kCFA decision problem is complete for $\mathbf{EXPTIME}$ when $k > 0$.

What makes kCFA hard?

This is not just a replaying of the previous proofs.

What makes kCFA hard?

This is not just a replaying of the previous proofs.

- ⋆ If the analysis were simulating evaluation,

What makes kCFA hard?

This is not just a replaying of the previous proofs.

- ⋆ If the analysis were simulating evaluation,
- ⋆ there would be one entry in each cache location,

What makes kCFA hard?

This is not just a replaying of the previous proofs.

- $\star$ If the analysis were simulating evaluation,
- $\star$ there would be one entry in each cache location,
- $\star$ therefore bounded by a polynomial!

What makes kCFA hard?

This is not just a replaying of the previous proofs.

- $\star$ If the analysis were simulating evaluation,
- $\star$ there would be one entry in each cache location,
- $\star$ therefore bounded by a polynomial!

Might and Shivers' observation:
improved precision leads to analyzer speedups.

What makes kCFA hard?

This is not just a replaying of the previous proofs.

- ⋆ If the analysis were simulating evaluation,
- ⋆ there would be one entry in each cache location,
- ⋆ therefore bounded by a polynomial!

Might and Shivers' observation:
improved precision leads to analyzer speedups.

Analytic understanding:

What you pay for in kCFA is the junk (spurious flows).

Work harder to learn less.

Conclusions

Conclusions

A complexity-theoretic investigation of flow analysis in higher-order languages provides insight into the fundamental limitations on the cost of performing analysis.

Conclusions

A complexity-theoretic investigation of flow analysis in higher-order languages provides insight into the fundamental limitations on the cost of performing analysis.

- ⋆ 0CFA and its approximations are inherently sequential.

Conclusions

A complexity-theoretic investigation of flow analysis in higher-order languages provides insight into the fundamental limitations on the cost of performing analysis.

- ⋆ 0CFA and its approximations are inherently sequential.
- ⋆ Analysis and evaluation of linear programs are equivalent.

Conclusions

A complexity-theoretic investigation of flow analysis in higher-order languages provides insight into the fundamental limitations on the cost of performing analysis.

- ⋆ 0CFA and its approximations are inherently sequential.
- ⋆ Analysis and evaluation of linear programs are equivalent.
- ⋆ 0CFA of η-expanded, simply-typed programs can be done space efficiently.

Conclusions

A complexity-theoretic investigation of flow analysis in higher-order languages provides insight into the fundamental limitations on the cost of performing analysis.

- ⋆ 0CFA and its approximations are inherently sequential.
- ⋆ Analysis and evaluation of linear programs are equivalent.
- ⋆ 0CFA of η-expanded, simply-typed programs can be done space efficiently.
- ⋆ kCFA is provably intractable when $k > 0$.

Conclusions

A complexity-theoretic investigation of flow analysis in higher-order languages provides insight into the fundamental limitations on the cost of performing analysis.

- ⋆ 0CFA and its approximations are inherently sequential.
- ⋆ Analysis and evaluation of linear programs are equivalent.
- ⋆ 0CFA of η-expanded, simply-typed programs can be done space efficiently.
- ⋆ kCFA is provably intractable when $k > 0$.
- ⋆ Closures and spurious flows make kCFA hard.

The End

Thank you.