Deciding kCFA is complete for EXPTIME

David Van Horn Harry G. Mairson
Brandeis University
{dvanhorn,mairson}@cs.brandeis.edu

Abstract

We give an exact characterization of the computational complexity of the kCFA hierarchy. For any $k > 0$, we prove that the control flow decision problem is complete for deterministic exponential time. This theorem validates empirical observations that such control flow analysis is intractable. It also provides more general insight into the complexity of abstract interpretation.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Computability theory, Computational logic, Lambda calculus and related systems

General Terms Languages, Theory

Keywords flow analysis, complexity

1. Introduction

Flow analysis (Jones 1981; Sestoft 1988; Shivers 1988; Midtgaard 2007) is concerned with the sound approximation of run-time values at compile time. This analysis gives rise to natural decision problems such as: *does expression e possibly evaluate to value v at run-time?* or *does function f possibly get applied at call site ℓ?*[1] The most approximate analysis always answers *yes*. This crude "analysis" takes no resources to compute, and is useless. In complete contrast, the most precise analysis only answers *yes* by running the program to find that out. While such information is surely useful, the cost of this analysis is likely prohibitive, requiring intractable or unbounded resources. Practical flow analyses occupy a niche between these extremes, and their *expressiveness* can be characterized by the computational resources required to compute their results.

Examples of simple yet useful flow analyses include Shivers' 0CFA (1988) and Henglein's simple closure analysis (1992), which are *monovariant*—functions that are closed over the same λ-expression are identified. Their expressiveness is characterized by the class PTIME (Van Horn and Mairson 2007, 2008). More precise analyses can be obtained by incorporating context-sensitivity to distinguish multiple closures over the same λ-term. The kCFA hierarchy (1991) uses the last k calling contexts to distinguish closures, resulting in "finer grained approximations, expending more work to gain more information" (Shivers 1988).

The increased precision comes with an empirically observed increase in cost. As Shivers noted in his retrospective on the kCFA work (2004):

> *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.*

A fairly straightforward calculation—see, for example, Nielson et al. (1999)—shows that 0CFA can be computed in polynomial time, and for any $k > 0$, kCFA can be computed in exponential time. These naive upper bounds suggest that the kCFA hierarchy is essentially *flat*; researchers subsequently "expended a great deal of effort" trying to improve them.[2] For example, it seemed plausible (at least, to us) that the kCFA problem could be in NP by *guessing* flows appropriately during analysis.

In this paper, we show that the naive algorithm is essentially the best one, and that the *lower* bounds are what needed improving. We prove that for all $k > 0$, computing the kCFA analysis requires (and is thus complete for) deterministic exponential time. There is, in the worst case—and plausibly, in practice—no way to tame the cost of the analysis. Exponential time is required.

Who cares, and why should this result matter to functional programmers?

This result concerns a fundamental and ubiquitous static analysis *of* functional programs. The theorem gives an analytic, scientific characterization of the expressive power of kCFA. As a consequence, the *empirically observed* intractability of the cost of this analysis can be understood as being *inherent in the approximation problem being solved*, rather than reflecting unfortunate gaps in our programming abilities.

Good science depends on having relevant theoretical understandings of what we observe empirically in practice—otherwise, we devolve to an unfortunate situation resembling an old joke about the difference between geology (the observation of physical phenomena that cannot be explained) and geophysics (the explanation of physical phenomena that cannot be observed).

This connection between theory and experience contrasts with the similar result for ML type inference (Mairson 1990): we confess that while the problem of recognizing ML-typable terms is complete for exponential time, programmers have happily gone on programming. It is likely that their need of higher-order procedures, essential for the lower bound, is not considerable. But static flow analysis really has been costly, and our theorem explains why.

The theorem is proved *by* functional programming. We take the view that the analysis itself is a functional programming language,

[1] There is a bit more nuance to the question, which will be developed later.

[2] Even so, there is a big difference between algorithms that run in 2^n and 2^{n^2} steps, though both are nominally in EXPTIME.

albeit with implicit bounds on the available computational resources. Our result harnesses the approximation inherent in kCFA as a computational tool to hack exponential time Turing machines within this unconventional language. The hack used here is completely unlike the one used for the ML analysis, which depended on complete developments of `let`-redexes. The theorem we prove in this paper uses approximation in a way that has little to do with normalization.

2. Preliminaries

2.1 Instrumented interpretation

kCFA can be thought of as an abstraction (in the sense of a computable approximation) to an instrumented interpreter, which not only evaluates a program, but records a history of *flows*. Every time a subterm evaluates to a value, every time a variable is bound to a value, the *flow* is recorded. Consider a simple example, where e is closed and in normal form:

$$(\lambda x.x)e$$

We label the term to index its constituents:

$$((\lambda x.x^0)^1 e^2)^3$$

The interpreter will record all the flows for evaluating e (there are none, since it is in normal form), then the flow of e's value (which is e, closed over the empty environment) into label 2 (e's label) is recorded.

This value is then recorded as flowing into the binding of x. The body of the λx expression is evaluated under an extended environment with x bound to the result of evaluating e. Since it is a variable occurrence, the value bound to x is recorded as flowing into this variable occurrence, labeled 0. Since this is the result of evaluating the body of the function, it is recorded as flowing out of the λ-term labeled 1. And again, since this is the result of the application, the result is recorded for label 3.

The flow history is recorded in a *cache*, C, which maps labels and variables to values. If the cache maps a variable to a value, $\mathsf{C}(x) = v$, it means that during evaluation of the program, the variable x was bound to the value v at some point. If the cache maps a label to a value, $\mathsf{C}(\ell) = v$, it means that the subexpression with that label evaluated to that value.

Of course, a variable may be bound to any number of values during the course of evaluation. Likewise, a subexpression that occurs once syntactically may evaluate to any number of values during evaluation. So asking about the flows of a subexpression is ambiguous without further information. Our simple example does not reflect this possible ambiguity, but consider the following example, where `True` and `False` are closed and in normal form:

$$(\lambda f.f(f\ \texttt{True}))(\lambda y.\texttt{False})$$

During evaluation, y gets bound to both `True` and `False`—asking "what was y bound to?" is ambiguous. But let us label the applications in our term:

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

Notice that y is bound to different values within different contexts. That is, y is bound `True` when evaluating the application labeled 1, and to `False` when evaluating the application labeled 2. Both of these occur while evaluating the outermost application, labeled 3. A string of these application labels, called a *contour*, uniquely describes the *context* under which a subexpression evaluates.

$$\begin{array}{rll} 3\cdot 2\cdot 1 & \text{describes} & ((\lambda f.(f[\,]^1)^2)(\lambda y.\texttt{False}))^3 \\ 3\cdot 2 & \text{describes} & ((\lambda f.[\,]^2)(\lambda y.\texttt{False}))^3 \end{array}$$

So a question about what a subexpression evaluates to *within a given context* has an unambiguous answer. The interpreter, therefore, maintains an environment that maps each variable to a description of the context in which it was bound. Similarly, flow questions about a particular subexpression or variable binding must be accompanied by a description of a context. Returning to our example, we would have $\mathsf{C}(y, 3\cdot 2\cdot 1) = \texttt{True}$ and $\mathsf{C}(y, 3\cdot 2) = \texttt{False}$.

Typically, values are denoted by *closures*—a λ-term together with an environment mapping all free variables in the term to values.[3] But in the instrumented interpreter, rather than mapping variables to values, environments map a variable to a *contour*—the sequence of labels which describes the context of successive function applications in which this variable was bound:[4]

$$\begin{array}{rcrll} \delta & \in & \Delta & = & \mathbf{Lab}^n & \text{contour} \\ ce & \in & \mathbf{CEnv} & = & \mathbf{Var} \to \Delta & \text{contour environment} \end{array}$$

By consulting the cache, we can then retrieve the value. So if under typical evaluation, a term labeled ℓ evaluates to $\langle \lambda x.e, \rho\rangle$ within a context described by a string a application labels, δ, then we will have $\mathsf{C}(\ell, \delta) = \langle \lambda x.e, ce\rangle$, where the contour environment ce, like ρ, closes $\lambda x.e$. But unlike ρ, it maps each variable to a contour describing the context in which the variable was bound. So if $\rho(y) = \langle \lambda z.e', \rho'\rangle$, then $\mathsf{C}(y, ce(y)) = \langle \lambda z.e', ce'\rangle$, where ρ' is similarly related to ce'.

We can now write the instrumented evaluator. The syntax of the language is given by the following grammar:

$$\begin{array}{lll} \mathbf{Exp} & e ::= t^\ell & \text{expressions (or labeled terms)} \\ \mathbf{Term} & t ::= x \mid e\ e \mid \lambda x.e & \text{terms (or unlabeled expressions)} \end{array}$$

$\mathcal{E}[\![t^\ell]\!]^{ce}_\delta$ evaluates t and writes the result into the table C at location (ℓ, δ). The notation $\mathsf{C}(\ell, \delta) \leftarrow v$ means that the cache is updated so that $\mathsf{C}(\ell, \delta) = v$. The notation $\delta\ell$ denotes the concatenation of contour δ and label ℓ.

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

The cache constructed by

$$\mathcal{E}[\![((\lambda f.(f(f\ \texttt{True})^1)^2)(\lambda y.\texttt{False}))^3]\!]^\emptyset_\epsilon$$

includes the following entries:

$$\begin{array}{rcl} \mathsf{C}(f, 3) & = & \lambda y.\texttt{False} \\ \mathsf{C}(y, 3\cdot 2\cdot 1) & = & \texttt{True} \\ \mathsf{C}(1, 3\cdot 2) & = & \lambda y.\texttt{False} \\ \mathsf{C}(y, 3\cdot 2) & = & \texttt{False} \end{array}$$

In a more declarative style, we can write a specification of *acceptable caches*—a cache is acceptable iff it records all of the flows which occur during evaluation. The smallest cache satisfying this acceptability relation is the one that is computed by the above

[3] We abused syntax above by writing the closure of a closed term as the term itself, rather than $\langle \texttt{True}, \emptyset\rangle$, for example. We continue with the abuse.

[4] All of the syntactic categories are implicitly understood to be restricted to the finite set of terms, labels, variables, etc. that occur in the *program of interest*—the program being analyzed. As a convention, programs are assumed to have distinct bound variable names and labels.

interpreter.

$$\begin{array}{lll}
\mathsf{C} \models_\delta^{ce} x^\ell & \text{iff} & \mathsf{C}(\ell,\delta) = \mathsf{C}(x, ce(x)) \\
\mathsf{C} \models_\delta^{ce} (\lambda x.e)^\ell & \text{iff} & \mathsf{C}(\ell,\delta) = \langle \lambda x.e, ce' \rangle \\
 & & \quad \text{where } ce' = ce \restriction \mathbf{fv}(\lambda x.e) \\
\mathsf{C} \models_\delta^{ce} (t^{\ell_1}\ t^{\ell_2})^\ell & \text{iff} & \mathsf{C} \models_\delta^{ce} t^{\ell_1} \wedge \mathsf{C} \models_\delta^{ce} t^{\ell_2} \wedge \\
 & & \quad \text{let } \langle \lambda x.t^{\ell_0}, ce' \rangle = \mathsf{C}(\ell_1,\delta) \text{ in} \\
 & & \quad\quad \mathsf{C}(x, \delta\ell) = \mathsf{C}(\ell_2,\delta) \wedge \\
 & & \quad\quad \mathsf{C} \models_{\delta\ell}^{ce'[x \mapsto \delta\ell]} t^{\ell_0} \wedge \\
 & & \quad\quad \mathsf{C}(\ell,\delta) = \mathsf{C}(\ell_0, \delta\ell)
\end{array}$$

Clearly, because constructing a cache C is equivalent to evaluating a program, such a cache is not effectively computable. The next section describes kCFA as a computable *approximation.*

2.2 An abstract interpreter

kCFA is a computable approximation to this instrumented interpreter. Rather than constructing an *exact* cache C, it constructs an *abstract* cache $\widehat{\mathsf{C}}$, which maps labels and variables, not to values, but to *sets* of *abstract values*.

$$\begin{array}{rcll}
\hat{v} & \in & \widehat{\mathbf{Val}} & = \mathcal{P}(\mathbf{Term} \times \mathbf{CEnv}) \\
\widehat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}} & = (\mathbf{Lab} + \mathbf{Var}) \times \Delta \to \widehat{\mathbf{Val}}
\end{array}$$

Approximation arises from contours being bounded at length k. If during the course of instrumented evaluation, the length of the contour would exceed length k, then the kCFA abstract interpreter will truncate it to length k. In other words, only a partial description of the context can be given, which results in ambiguity. A subexpression may evaluate to two distinct values, but within contexts which are only distinguished by $k+1$ labels. Questions about which value the subexpression evaluates to can only supply k labels, so the answer must be *both*, according to a sound approximation.

When applying a function, there is now a set of possible closures that flow into the operator position. Likewise, there can be a multiplicity of arguments. What is the interpreter to do? The abstract interpreter applies all possible closures to all possible arguments.

The abstract interpreter, the imprecise analog of $\mathcal{E}$, is then:

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

We write $\widehat{\mathsf{C}}(\ell,\delta) \leftarrow \hat{v}$ to indicate an updated cache where (ℓ,δ) maps to $\widehat{\mathsf{C}}(\ell,\delta) \cup \{\hat{v}\}$. The notation $\lceil \delta \rceil_k$ denotes δ truncated to the rightmost (i.e., most recent) k labels.

Compared to the exact evaluator, contours similarly distinguish evaluation within contexts described by as many as k application sites: beyond this, the distinction is blurred. The imprecision of the analysis requires that $\mathcal{A}$ be iterated until the cache reaches a fixed point, but care must taken to avoid looping in an iteration since a single iteration of $\mathcal{A}[\![e]\!]_\delta^{ce}$ may in turn make a recursive call to $\mathcal{A}[\![e]\!]_\delta^{ce}$ under the same contour and environment. This care is the algorithmic analog of appealing to the coinductive hypothesis in judging an analysis acceptable. These judgment rules are given below.

An acceptable k-level control flow analysis for an expression e is written $\widehat{\mathsf{C}} \models_\delta^{ce} e$, which states that $\widehat{\mathsf{C}}$ is an acceptable analysis of e in the context of the current environment ce and current contour δ (for the top level analysis of a program, these will both be empty).

Just as we did in the previous section, we can write a specification of acceptable caches rather than an algorithm that computes. The resulting specification is what is found, for example, in Nielson et al. (1999):

$$\begin{array}{lll}
\widehat{\mathsf{C}} & \models_\delta^{ce} & x^\ell \text{ iff } \widehat{\mathsf{C}}(x, ce(x)) \subseteq \widehat{\mathsf{C}}(\ell,\delta) \\
\widehat{\mathsf{C}} & \models_\delta^{ce} & (\lambda x.e)^\ell \text{ iff } \langle \lambda x.e, ce' \rangle \in \widehat{\mathsf{C}}(\ell,\delta) \\
 & & \quad \text{where } ce' = ce \restriction \mathbf{fv}(\lambda x.e) \\
\widehat{\mathsf{C}} & \models_\delta^{ce} & (t^{\ell_1}\ t^{\ell_2})^\ell \text{ iff } \widehat{\mathsf{C}} \models_\delta^{ce} t^{\ell_1} \wedge \widehat{\mathsf{C}} \models_\delta^{ce} t^{\ell_2} \wedge \\
 & & \quad \forall \langle \lambda x.t^{\ell_0}, ce' \rangle \in \widehat{\mathsf{C}}(\ell_1,\delta): \\
 & & \quad\quad \widehat{\mathsf{C}}(\ell_2,\delta) \subseteq \widehat{\mathsf{C}}(x, \lceil \delta\ell \rceil_k) \wedge \\
 & & \quad\quad \widehat{\mathsf{C}} \models_{\lceil \delta\ell \rceil_k}^{ce'[x \mapsto \lceil \delta\ell \rceil_k]} t^{\ell_0} \wedge \\
 & & \quad\quad \widehat{\mathsf{C}}(\ell_0, \lceil \delta\ell \rceil_k) \subseteq \widehat{\mathsf{C}}(\ell,\delta)
\end{array}$$

The acceptability relation is given by the greatest fixed point of the functional defined according to the following clauses—and we are concerned only with least solutions.[5]

2.3 Complexity of abstract interpretation

What is the difficulty of computing within this hierarchy? What are the sources of approximation that render such analysis (in)tractable? We consider these questions by analyzing the complexity of the following decision problem:

Control Flow Problem: Given an expression e, an abstract value v, and a pair (ℓ,δ), is $v \in \widehat{\mathsf{C}}(\ell,\delta)$ in the flow analysis of e?

Obviously, we are interested in the complexity of control flow analysis, but our investigation also provides insight into a more general subject: the complexity of computing via abstract interpretation. It stands to reason that as the computational domain becomes more refined, so too should computational complexity. In this instance, the domain is the size of the abstract cache $\widehat{\mathsf{C}}$ and the values (namely, *closures*) that can be stored in the cache. As the table size and number of closures increase[6], so too should the complexity of computation. From a theoretical perspective, we would like to understand better the tradeoffs between these various parameters.

3. Linearity and Boolean logic

It is straightforward to observe that in a *linear* λ-term, where each variable occurs at most once, each abstraction $\lambda x.e$ can be applied to at most one argument, and hence the abstracted value can be bound to at most one argument. (Note that this observation is clearly untrue for the *nonlinear* λ-term $(\lambda f.f(a(fb)))(\lambda x.x)$, as x is bound to b, and also to ab.) Generalizing this observation, analysis of a linear λ-term coincides exactly with its evaluation:

LEMMA 1. *For any closed, labeled linear λ-term e, we have $\mathcal{E}[\![e]\!]_\epsilon^\emptyset = \mathcal{A}[\![e]\!]_\epsilon^\emptyset$, and thus $\mathsf{C} = \widehat{\mathsf{C}}$.*

A detailed proof of this lemma appears in Van Horn and Mairson (2008).

A natural and expressive class of such linear terms are the ones which implement Boolean logic. When we analyze the coding of a

[5] To be precise, we take as our starting point *uniform* kCFA rather than a kCFA in which $\widehat{\mathbf{Cache}} = (\mathbf{Lab} \times \mathbf{CEnv}) \to \widehat{\mathbf{Val}}$. The differences are immaterial for our purposes. See Nielson et al. (1999) for details and a discussion on the use of coinduction in specifying static analyses.

[6] Observe that since closure environments map free variables to contours, the number of closures increases when we increase the contour length k.

Boolean circuit and inputs to it, the Boolean output will flow to a predetermined place in the (abstract) cache. By placing that value in an appropriate context, we construct an instance of the control flow problem: a function f flows to a call site a iff the Boolean output is `True`.

Since we have therefore reduced the circuit value problem (Ladner 1975), which is complete for PTIME, to an instance of the 0CFA control flow problem, we conclude that the control flow problem is PTIME-hard. Further, as 0CFA can be computed in polynomial time, the control flow problem for 0CFA is PTIME-complete.

We use some standard syntactic sugar for constructing and deconstructing pairs:

$$\begin{array}{rcl} \langle u, v\rangle & \equiv & \lambda z.zuv \\ \text{let } \langle x, y\rangle = p \text{ in } e & \equiv & p(\lambda x.\lambda y.e) \end{array}$$

Booleans are built out of constants `tt` and `ff`, which are the identity and pair swap function, respectively:

$$\begin{array}{rclcrcl} \texttt{tt} & \equiv & \lambda p.\text{let } \langle x, y\rangle = p \text{ in } \langle x, y\rangle & \qquad & \texttt{True} & \equiv & \langle \texttt{tt}, \texttt{ff}\rangle \\ \texttt{ff} & \equiv & \lambda p.\text{let } \langle x, y\rangle = p \text{ in } \langle y, x\rangle & & \texttt{False} & \equiv & \langle \texttt{ff}, \texttt{tt}\rangle \end{array}$$

The simplest connective is `Not`, which is an inversion on pairs, like `ff`. A *linear* copy connective is defined as:

$$\texttt{Copy} \quad \equiv \quad \lambda b.\text{let } \langle u, v\rangle = b \text{ in } \langle u\langle \texttt{tt}, \texttt{ff}\rangle, v\langle \texttt{ff}, \texttt{tt}\rangle\rangle$$

The coding is easily explained: suppose b is `True`, then u is identity and v twists; so we get the pair $\langle \texttt{True}, \texttt{True}\rangle$. Suppose b is `False`, then u twists and v is identity; we get $\langle \texttt{False}, \texttt{False}\rangle$. We write $\texttt{Copy}_n$ to mean n-ary fan-out—a straightforward extension of the above.

Now we define truth-table implication:

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

Notice that if b_1 is `True`, then u_1 is `tt`, so p_1 is `tt` iff b_2 is `True`. And if b_1 is `True`, then v_1 is `ff`, so q_1 is `ff` iff b_2 is `False`. On the other hand, if b_1 is `False`, u_1 is `ff`, so p_1 is `tt`, and v_1 is `tt`, so q_1 is `ff`. Therefore $\langle p_1, q_1\rangle$ is `True` iff $b_1 \supset b_2$, and `False` otherwise.[7]

However, simply returning $\langle p_1, q_1\rangle$ violates linearity since p_2, q_2 go unused. We know that $p_2 = \texttt{tt}$ iff $q_2 = \texttt{ff}$ and $p_2 = \texttt{ff}$ iff $q_2 = \texttt{tt}$. We do not know which is which, but clearly $p_2 \circ q_2 = \texttt{ff} \circ \texttt{tt} = \texttt{tt} \circ \texttt{ff} = \texttt{ff}$. Composing $p_2 \circ q_2$ with `ff`, we are guaranteed to get `tt`. Therefore $q_1 \circ p_2 \circ q_2 \circ \texttt{ff} = q_1$, and we have used all bound variables exactly once. The `And` and `Or` connectives are defined similarly (as in Van Horn and Mairson (2007)).

3.1 The Widget

Consider a Boolean circuit coded as a program: it can only evaluate to a (coded) true or false value, but a flow analysis identifies terms by label, so it is possible several different `True` and `False` terms flow out of the program. But our decision problem is defined with respect to a particular term. What we want is to use flow analysis to answer questions like "does this program (possibly) evaluate to a true value?" We use The Widget to this effect. It is a term expecting a boolean value. It evaluates as though it were the identity function on Booleans, $\texttt{Widget}\ b = b$, but it induces a specific flow we can ask about. If a true value flows out of b, then $\texttt{True}_W$ flows out of $\texttt{Widget}\ b$. If a false value flows out of b, then $\texttt{False}_W$ flows out of $\texttt{Widget}\ b$, where $\texttt{True}_W$ and $\texttt{False}_W$ are distinguished terms, and the only possible terms that can flow out. We usually drop the subscripts and say "does `True` flow out of $\texttt{Widget}\ b$?" without much ado.[8]

$$\texttt{Widget} \quad \equiv \quad \lambda b.\text{let } \langle u, v\rangle = b \text{ in } \pi_1(u\langle \texttt{True}_W, \texttt{False}_W\rangle)$$

Because the circuit value problem is complete for PTIME, we conclude (Van Horn and Mairson 2007):

THEOREM 1. *Deciding the control flow problem for 0CFA is complete for* PTIME.

4. Nonlinearity and Cartesian products: a toy calculation, with insights

A good proof has, at its heart, a small and simple idea that makes it work. For our proof, the key idea is how the approximation of analysis can be *leveraged* to provide computing power *above and beyond* that provided by evaluation. The difference between the two can be illustrated by the following term:

$$\begin{array}{l} (\lambda f.(f\ \texttt{True})(f\ \texttt{False})) \\ (\lambda x.\texttt{Implies}\ x\ x) \end{array}$$

Consider evaluation: Here $\texttt{Implies}\ x\ x$ (a tautology) is evaluated twice, once with x bound to `True`, once with x bound to `False`. But in both cases, the result is `True`. Since x is bound to `True` or `False` both occurrences of x are bound to `True` or to `False`—but it is never the case, for example, that the first occurrence is bound to `True`, while the second is bound to `False`. The values of each occurrence of x is dependent on the other.

On the other hand, consider what flows out of $\texttt{Implies}\ x\ x$ according 1CFA: both `True` and `False`. Why? The approximation incurs analysis of $\texttt{Implies}\ x\ x$ for x bound to `True` and `False`, but it considers *each occurrence of x as ranging over* `True` *and* `False`*, independently*. In other words, for the set of values bound to x, we consider their *cross product* when x appears non-linearly. The approximation permits one occurrence of x be bound to `True` while the other occurrence is bound to `False`; and somewhat alarmingly, `Implies True False` causes `False` to flow out. Unlike in normal evaluation, where within a given scope we know that multiple occurrences of the same variable refer to the same value, in the approximation of analysis, multiple occurrences of the same variable range over *all* values that they are possible bound to *independent of each other*.

Now consider what happens when the program is expanded as follows:

$$\begin{array}{l} (\lambda f.(f\ \texttt{True})(f\ \texttt{False})) \\ (\lambda x.(\lambda p.p(\lambda u.p(\lambda v.\texttt{Implies}\ uv)))(\lambda w.wx)) \end{array}$$

Here, rather than pass x directly to `Implies`, we construct a unary tuple $\lambda w.wx$. The tuple is used non-linearly, so p will range over *closures* of $\lambda w.wx$ with x bound to `True` and `False`, again, independently.

A closure can be approximated by an exponential number of values. For example, $\lambda w.wz_1z_2\ldots z_n$ has n free variables, so there are an exponential number of possible environments mapping these variables to program points (contours of length 1). If we could apply a Boolean function to this tuple, we would effectively be evaluating all rows of a truth table; following this intuition leads to NP-hardness of the 1CFA control flow problem.

[7] Or, if you prefer, $u_1\langle u_2, \texttt{tt}\rangle$ can be read as "if u_1, then u_2 else `tt`"—the if-then-else description of the implication $u_1 \supset u_2$ —and $v_1\langle \texttt{ff}, v_2\rangle$ as its deMorgan dual $\neg(v_2 \supset v_1)$. Thus $\langle p_1, q_1\rangle$ is the answer we want—and we need only dispense with the "garbage" p_2 and q_2. DeMorgan duality ensures that one is `tt`, and the other is `ff` (though we do not know which), so they always compose to `ff`.

[8] This `Widget` is affine, but this is only for simplicity in presentation. A non-affine widget is given Van Horn and Mairson (2007).

Generalizing from unary to n-ary tuples in the above example, an exponential number of closures can flow out of the tuple. For a function taking two n-tuples, we can compute the function on the cross product of the exponential number of closures.

This insight is the key computational ingredient in simulating exponential time, as we describe in the following section.

5. The complexity of kCFA

5.1 Approximation and EXPTIME

Recall the formal definition of a Turing machine: a 7-tuple

$$\langle Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r \rangle$$

where Q, Σ, and Γ are finite sets, Q is the set of machine states (and $\{q_0, q_a, q_r\} \subseteq Q$), Σ is the input alphabet, and Γ the tape alphabet, where $\Sigma \subseteq \Gamma$. The states q_0, q_a, and q_r are the machine's initial, accept, and reject states, respectively. The complexity class EXPTIME denotes the languages that can be decided by a Turing machine in time exponential in the input length.

Suppose we have a deterministic Turing machine M that accepts or rejects its input x in time $2^{p(n)}$, where p is a polynomial and $n = |x|$. We want to simulate the computation of M on x by kCFA analysis of a λ-term E dependent on M, x, p, where a particular closure will flow to a specific program point iff M accepts x. It turns out that $k = 1$ suffices to carry out this simulation. The construction, computed in logarithmic space, is similar for all constant $k > 1$ modulo a certain amount of padding.

5.2 Coding machine IDs

The first task is to code machine IDs. Observe that each value stored in the abstract cache $\widehat{\mathsf{C}}$ is a *closure*—a λ-abstraction, together with an environment for its free variables. The number of such abstractions is bounded by the program size, as is the *domain* of the environment—while the number of such *environments* is exponential in the program size. (Just consider a program of size n with, say, $n/2$ free variables mapped to only 2 program points denoting bindings.)

Since a closure only has polynomial size, and a Turing machine ID has exponential size, we represent the latter by splitting its information into an exponential number of closures. Each closure represents a tuple $\langle T, S, H, C, b \rangle$, which can be read as

> *"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, and C are blocks of bits ($\mathbf{0} \equiv$ `True`, $\mathbf{1} \equiv$ `False`) of size polynomial in the input to the Turing machine. As such, each block can represent an exponential number of values. A single machine ID is represented by an exponential number of tuples (varying C and b). Each such tuple can in turn be coded as a λ-term $\lambda w.wz_1z_2\cdots z_N$, where $N = O(p(n))$.

We still need to be able to generate an exponential number of closures for such an N-ary tuple. The construction is only a modest, iterative generalization of the construction in our toy calculation above:

$$\begin{array}{l}(\lambda f_1.(f_1\ \mathbf{0})(f_1\ \mathbf{1}))\\(\lambda z_1.\\\quad(\lambda f_2.(f_2\ \mathbf{0})(f_2\ \mathbf{1}))\\\quad(\lambda z_2.\\\qquad\cdots\\\qquad\quad(\lambda f_N.(f_N\ \mathbf{0})(f_N\ \mathbf{1}))\\\qquad\quad(\lambda z_N.((\lambda x.x)(\lambda w.wz_1z_2\cdots z_N))^\ell)\cdots))\end{array}$$

In the final subterm $((\lambda x.x)(\lambda w.wz_1z_2\cdots z_N))^\ell$, the function $\lambda x.x$ acts as a very important form of *padding*. Recall that this is kCFA with $k = 1$—the expression $(\lambda w.wz_1z_2\cdots z_N)$ is evaluated an exponential number of times—to see why, normalize the term—but in each instance, the contour is always ℓ. (For $k > 1$, we would just need more padding to evade the *polyvariance* of the flow analyzer.) As a consequence, each of the (exponential number of) closures gets put in the *same* location of the abstract cache $\widehat{\mathsf{C}}$, while they are placed in unique, *different* locations of the exact cache C. In other words, the approximation mechanism of kCFA treats them as if they are all the same. (That is why they are put in the same cache location.)

5.3 Transition function

Now we define a binary transition function δ, which does a *piecemeal* transition of the machine ID. The transition function is represented by three rules, identified uniquely by the time stamps T on the input tuples.

The first *transition rule* is used when the tuples agree on the time stamp T, and the head and cell address of the first tuple coincide:

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

This rule *computes* the transition to the next ID. The first tuple has the head address and cell address coinciding, so it has all the information needed to compute the next state, head movement, and what to write in that tape cell. The second tuple just marks that this is an instance of the *computation* rule, simply indicated by having the time stamps in the tuples to be identical. The Boolean functions $\delta_Q, \delta_{LR}, \delta_\Sigma$ compute the next state, head position, and what to write on the tape.

The second *communication rule* is used when the tuples have time stamps $T+1$ and T: in other words, the first tuple has information about state and head position which needs to be communicated to every tuple with time stamp T holding tape cell information for an arbitrary such cell, as it gets updated to time stamp $T+1$:

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

(Note that when $H' = C'$, we have already written the salient tuple using the transition rule.) This rule *communicates* state and head position (for the first tuple computed with time stamp $T+1$, where the head and cell address coincided) to all the other tuples coding the rest of the Turing machine tape.

Finally, we define a *catch-all rule*, mapping any other pairs of tuples (say, with time stamps T and $T+42$) to some distinguished null value (say, the initial ID). We need this rule just to make sure that δ is a totally defined function.

$$\delta\langle T, S, H, C, b\rangle\langle T', S', H', C', b'\rangle = \texttt{Null} \quad (T \neq T' \text{ and } T \neq T'+1)$$

Clearly, these three rules can be coded by a single Boolean circuit, and we have all the required Boolean logic at our disposal.

Because δ is a binary function, we need to compute a *cross product* on the coding of IDs to provide its input. The transition function is therefore defined as:

$$\begin{array}{lcl}\Phi & \equiv & \lambda p.\\ & & \quad\text{let } \langle u_1,u_2,u_3,u_4,u_5\rangle = \texttt{Copy}_5\ p \text{ in}\\ & & \quad\text{let } \langle v_1,v_2,v_3.v_4,v_5\rangle = \texttt{Copy}_5\ p \text{ in}\\ & & \quad\quad(\lambda w.w(\phi_T u_1 v_1)(\phi_S u_2 v_2)\ldots(\phi_b u_5 v_5))\\ & & \quad\quad(\lambda w_T.\lambda w_S.\lambda w_H.\lambda w_C.\lambda w_b.\\ & & \quad\quad\quad w_T(\lambda z_1.\lambda z_2\ldots\lambda z_T.\\ & & \quad\quad\quad\quad w_S(\lambda z_{T+1}.\lambda z_{T+2}\ldots\lambda z_{T+S}.\\ & & \quad\quad\quad\quad\quad\cdots\\ & & \quad\quad\quad\quad\quad w_b(\lambda z_{C+1}.\lambda z_{C+2}\ldots\lambda z_{C+b=m}.\\ & & \quad\quad\quad\quad\quad\quad\lambda w.wz_1z_2\ldots z_m)\ldots)))\end{array}$$

The `Copy` functions just copy enough of the input for the separate calculations to be implemented in a linear way. Observe that

this λ-term is entirely linear *except* for the two occurrences of its parameter p. In that sense, it serves a function analogous to $\lambda x.\texttt{Implies}\, x\, x$ in the toy calculation. Just as x ranges there over the closures for `True` and for `False`, p ranges over all possible IDs flowing to the argument position. Since there are two occurrences of p, we have two entirely separate iterations in the kCFA analysis. These separate iterations, like nested "for" loops, create the equivalent of a cross product of IDs in the "inner loop" of the flow analysis.

5.4 Context and widget

The context for the Turing machine simulation needs to set up the initial ID and associated machinery, extract the Boolean value telling whether the machine accepted its input, and feed it into the flow widget that causes different flows depending on whether the value flowing in is `True` or `False`.

$$\begin{array}{rcl} C & \equiv & (\lambda f_1.(f_1\ \mathbf{0})(f_1\ \mathbf{1})) \\ & & (\lambda z_1. \\ & & \quad (\lambda f_2.(f_2\ \mathbf{0})(f_2\ \mathbf{1})) \\ & & \quad (\lambda z_2. \\ & & \quad\quad \cdots \\ & & \quad\quad\quad (\lambda f_N.(f_N\ \mathbf{0})(f_N\ \mathbf{1})) \\ & & \quad\quad\quad (\lambda z_N.((\lambda x.x)(\texttt{Widget}(\texttt{Extract}[\,]))^{\ell})^{\ell'})\cdots)) \end{array}$$

In this code, the $\lambda x.x$ (with label ℓ' on its application) serve as padding, so that the term within is always applied in the same contour. `Extract` extracts a final ID, with its time stamp, and checks if it codes an accepting state, returning `True` or `False` accordingly. `Widget` is our standard control flow test. The context is instantiated with the coding of the transition function, iterated over an initial machine ID,

$$2^n\ \Phi\ \lambda w.w\mathbf{0}\ldots\mathbf{0}\cdots Q_0\cdots H_0\cdots z_1 z_2\ldots z_N\mathbf{0},$$

where Φ is a coding of transition function for M. The λ-term 2^n is a fixed point operator for kCFA, which can be assumed to be either $\mathbf{Y}$, or an exponential function composer. There just has to be enough iteration of the transition function to produce a fixed point for the flow analysis.

To make the coding easy, we just assume that M starts by writing x on the tape, and then begins the generic exponential-time computation. Then we can just have all zeroes on the initial tape configuration.

LEMMA 2. *For any Turing machine M and input x of length n, where M accepts or rejects x in $2^{p(n)}$ steps, there exists a logspace-constructible, closed, labeled λ-term e with distinguished label ℓ such that in the kCFA analysis of e ($k > 0$), `True` flows into ℓ iff M accepts x.*

THEOREM 2. *Deciding the control flow problem for kCFA with $k > 0$ is complete for* EXPTIME.

5.5 Exactness and PTIME

At the heart of the EXPTIME-completeness result is the idea that the *approximation* inherent in abstract interpretation is being harnessed for computational power, quite apart from the power of *exact* normalization. To get a good lower bound, this is necessary: there is a dearth of computation power when kCFA corresponds with normalization, i.e. when the analysis is exact.

Such computations occur when each cache location contains at most one element. Every program point is now approximated by at most one closure (rather than an exponential number of closures). The size of the cache is then bounded by a polynomial in n; since the cache is computed monotonically, the analysis and the natural related decision problem is constrained by the size and use of the cache.

PROPOSITION 1. *Deciding the control flow problem for exact kCFA is complete for* PTIME.

This proposition provides an analytic understanding of the empirical observation researchers have made: computing a more precise analysis is often cheaper than performing a less precise one, which "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).

Observe that in such exact analyses, the contour is never truncated. As a consequence, this proposition can be seen as a characterization of the expressivity of the programming language evaluated by $\mathcal{E}$ when the size of the contour is bounded by a constant.

The asymptotic differential between the complexity of exact and abstract interpretation shows that abstract interpretation is strictly more expressive, for any fixed k.

5.6 Discussion

We observe an "exponential jump" between contour length and complexity of the control flow decision problem for every polynomial-length contour, including contours of constant length. Once $k = n$ (contour length equals program size), an exponential-time hardness result can be proved which is essentially a linear circuit with an exponential iterator—very much like (Mairson 1990). When the contours are exponential in program length, the decision problem is doubly exponential, and so on.

The reason for this exponential jump is the cardinality of environments in closures. This, in fact, is the bottleneck for control flow analysis—it is the reason that 0CFA (without closures) is tractable, while 1CFA is not. If $f(n)$ is the contour length and n is the program length, then

$$|\mathbf{CEnv}| = |\mathbf{Var} \to \Delta^{\leq f(n)}| = (n^{f(n)})^n = 2^{f(n)n\lg n}$$

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

When k is a constant, one might ask why the inherent complexity is exponential time, and not more—especially since one can iterate (in an untyped world) with the $\mathbf{Y}$ combinator. Exponential time is the "limit" because with a polynomial-length tuple (as constrained by a logspace reduction), you can only code an exponential number of closures.

Finally, we need to emphasize the importance of linearity in static analysis. Static analysis makes approximations to be tractable, but with linear terms, there is not approximation. We carefully admitted a certain, limited nonlinearity in order to increase the lower bound.

6. Related work

Our earlier work on the complexity of compile-time type inference is a precursor of the research insights described here, and naturally so, since type inference is a kind of static analysis. The decidability of type inference depends on the making of approximations, necessarily rejecting programs without type errors; in simply-typed λ-calculus, for instance, all occurrences of a variable must have the same type. (The same is, in effect, also true for ML, modulo the finite development implicit in `let`-bindings.) The type constraints on these multiple occurrences are solved by first-order unification.

As a consequence, we can understand the inherent complexity of type inference by analyzing the expressive power of *linear* terms, where no such constraints exist, since linear terms are always simply-typable. In these cases, type inference is synonymous with

normalization.[9] This observation motivates the analysis described in Mairson (1990, 2004).

Our coding of Turing machines is descended from our earlier work on Datalog (Prolog with variables, but without constants or function symbols), a programming language that was of considerable interest to researchers in database theory during the 1980s; see Hillebrand et al. (1995); Gaifman et al. (1993).

In kCFA and abstract interpretation more generally, an expression can evaluate to a set of values from a finite universe, clearly motivating the idiom of programming with sets. Relational database queries take as input a finite set of tuples, and compute new tuples from them; since the universe of tuples is finite and the computation is monotone, a fixed-point is reached in a finite number of iterations. The machine simulation here follows that framework very closely. Even the idea of splitting a machine configuration among many tuples has its ancestor in (Hillebrand et al. 1995), where a ternary $\texttt{cons}(A, L, R)$ is used to simulate a `cons`-cell at memory address A, with pointers L, R. It needs emphasis that the computing with sets described in this paper has little to do with normalization, and everything to do with the approximation inherent in the abstract interpretation.

This coding of Boolean logic in linear λ-calculus, which was previously given in Van Horn and Mairson (2007) and is briefly described again here for completeness, improves upon Mairson (2004) in that it allows uniform typing, and does not create garbage. The encoding in Mairson (2004) in turn is an improvement of the Church encodings in that they are linear and non-affine.

Although kCFA and ML type inference are two static analyses complete for EXPTIME (Mairson 1990), the proofs of these respective theorems is fundamentally different. The ML proof relies on type inference simulating exact normalization (analogous to the PTIME-completeness proof for 0CFA), hence subverting the approximation of the analysis. In contrast, the kCFA proof harnesses the approximation that results from nonlinearity.

7. Conclusions and perspective

Empirically observed increases in costs can be understood analytically as *inherent in the approximation problem being solved*.

We have given an exact characterization of the kCFA approximation problem. The EXPTIME lower bound validates empirical observations and shows that there is no tractable algorithm for kCFA.

The proof relies on previous insights about linearity, static analysis, and normalization (namely, when a term is linear, static analysis and normalization are synonymous); coupled with new insights about using non-linearity to realize the full computational power of approximate, or abstract, interpretation.

Shivers wrote in his best of PLDI retrospective (2004),

> *Despite all this work on formalising CFA and speeding it up, I have been disappointed in the dearth of work extending its* power.

This work has shown that work spent on speeding up kCFA is an exercise in futility; there is no getting around the exponential bottleneck of kCFA. The one-word description of the bottleneck is *closures*, which do not exist in 0CFA, because free variables in a closure would necessarily map to ϵ, and hence the environments are useless.

As for extending its power, from a complexity perspective, we can see that 0CFA is strictly less expressive than kCFA. In turn, kCFA is strictly less expressive than, for example, Mossin's flow analysis (1997). Mossin's analysis is a stronger analysis in the sense that it is exact for a larger class of programs than 0CFA or kCFA—it exact not only for linear terms, but for all simply-typed terms. In other words, the flow analysis of simply-typed programs is synonymous with running the program, and hence non-elementary. This kind of expressivity is also found in Burn-Hankin-Abramsky style strictness analysis (1985). But there is a considerable gap between kCFA and these more expressive analyses. What is in between and how can we build a real *hierarchy* of static analyses that occupy positions within this gap.

Lévy's notion of labeled reduction (1978) provides a richer notion of "instrumented evaluation" coupled with a richer theory of exact flow analysis, namely the geometry of interaction. With the proper notion of abstraction and simulated reduction, we should be able to design more powerful flow analyses, filling out the hierarchy from 0CFA up to the expressivity of Mossin's analysis in the limit.

Acknowledgments Thanks to Matt Might, Olin Shivers, and Mitch Wand for listening to preliminary presentations of this proof.

[9] An aberrant case of this phenomenon is analyzed in Neergaard and Mairson (2004), which analyzed a type system where normalization and type inference are synonymous in *every* case. The tractability of type inference thus implied a certain inexpressiveness of the language.

References

G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In H. Ganzinger and N. Jones, editors, *Programs as data objects*, pages 42–62, New York, NY, USA, 1985. Springer-Verlag. ISBN 0-387-16446-4.

Haim Gaifman, Harry Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. *J. ACM*, 40(3):683–713, 1993. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/174130.174142.

Fritz Henglein. Simple closure analysis. DIKU Semantics Report D-193, March 1992.

Gerd G. Hillebrand, Paris C. Kanellakis, Harry G. Mairson, and Moshe Y. Vardi. Undecidable boundedness problems for Datalog programs. *J. Logic Program.*, 25(2):163–190, 1995.

Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128, London, UK, 1981. Springer-Verlag. ISBN 3-540-10843-2.

Richard E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18–20, 1975. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/990518.990519.

Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Paris 7, January 1978. Thèse d'Etat.

Harry G. Mairson. Linear lambda calculus and PTIME-completeness. *J. Funct. Program.*, 14(6):623–633, 2004. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796804005131.

Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 382–401, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. doi: http://doi.acm.org/10.1145/96709.96748.

Jan Midtgaard. Control-flow analysis of functional programs. Technical Report BRICS RS-07-18, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2007.

Christian Mossin. Exact flow analysis. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 250–264, London, UK, 1997. Springer-Verlag. ISBN 3-540-63468-1.

Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. In *ICFP '04: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 138–149, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-905-5. doi: http://doi.acm.org/10.1145/1016850.1016871.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988. Master's thesis no. 254.

Olin Shivers. Control flow analysis in Scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54007.

Olin Shivers. *Control-flow Analysis of Higher-order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. *SIGPLAN Not.*, 39(4):257–269, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/989393.989421.

David Van Horn and Harry G. Mairson. Relating complexity and precision in control flow analysis. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 85–96, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-815-2. doi: http://doi.acm.org/10.1145/1291151.1291166.

David Van Horn and Harry G. Mairson. Linearity, flow analysis, and PTIME. In *The 15th International Static Analysis Symposium SAS 2008*, July 2008. To appear.

Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst.*, 20(1):166–207, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/271510.271523.