Under consideration for publication in J. Functional Programming

Computational complexity of kCFA

DAVID VAN HORN
Northeastern University, Boston, MA, USA
and HARRY G. MAIRSON
Brandeis University, Waltham, MA, USA

Abstract

We give exact characterizations of the computational complexity of the *k*CFA hierarchy and related analyses. In each case, we precisely capture both the *expressiveness* and *feasibility* of the analysis, identifying the elements responsible for the trade-off.

0CFA is complete for polynomial time. This result relies on the insight that when a program is linear (each bound variable occurs exactly once), the analysis makes no approximation; abstract and concrete interpretation coincide, and therefore program analysis becomes evaluation under another guise. Moreover, this is true not only for 0CFA, but for a number of *further approximations* to 0CFA. In each case, we derive polynomial time completeness results.

For any $k > 0$, *k*CFA is complete for exponential time. Even when $k = 1$, the distinction in binding contexts results in a limited form of *closures*, which do not occur in 0CFA. This theorem validates empirical observations that *k*CFA is intractably slow for any $k > 0$. There is, in the worst case, no way to tame the cost of the analysis. Exponential time is required. The empirically observed intractability of this analysis can be understood as being *inherent in the approximation problem being solved*, rather than reflecting unfortunate gaps in our programming abilities.

1 Introduction

We analyze the computational complexity of flow analysis for higher-order languages, yielding a number of novel insights: *k*CFA is provably intractable; 0CFA and its approximations are inherently sequential; and analysis and evaluation of linear programs are equivalent.

1.1 Overview

Semantics-based program analysis (Cousot & Cousot, 1977; Cousot & Cousot, 1992; Muchnick & Jones, 1981; Nielson *et al.*, 1999) aims to discover the run-time behavior of a program *without actually running it* [page xv] (Muchnick & Jones, 1981). But as a natural consequence of Rice's theorem (1953), a perfect prediction is almost always impossible. So *tractable* program analysis must necessarily trade exact evaluation for a safe, computable approximation to it. This trade-off induces a fundamental dichotomy at play in the design of program analyzers and optimizing compilers. On the one hand, the more an analyzer can discover about what will happen when the program is run, the more optimizations the compiler can perform. On the other, compilers are generally valued not only for producing fast code, but doing so

quickly and efficiently; some optimizations may be forfeited because the requisite analysis is too difficult to do in a timely or space-efficient manner.

As an example in the extreme, if we place *no limit* on the resources consumed by the compiler, it can perfectly predict the future—the compiler can simply simulate the running of the program, watching as it goes. When (and if) the simulation completes, the compiler can optimize with perfect information about what will happen when the program is run. With good reason, this seems a bit like cheating.

So at a minimum, we typically expect a compiler will eventually finish working and produce an optimized program. (In other words, we expect the compiler to compute within bounded resources of time and space). After all, what good is an optimizing compiler that never finishes?

But by requiring an analyzer to compute within bounded resources, we have necessarily and implicitly limited its ability to predict the future.

As the analyzer works, it *must* use some form of approximation; knowledge must be given up for the sake of computing within bounded resources. Further resource-efficiency requirements may entail further curtailing of knowledge that a program analyzer can discover. But the relationship between approximation and efficiency is far from straightforward. Perhaps surprisingly, as has been observed empirically by researchers (Wright & Jagannathan, 1998; Jagannathan *et al.*, 1998; Might & Shivers, 2006b), added precision may avoid needless computation induced by approximation in the analysis, resulting in computational *savings*—that is, better information can often be produced faster than poorer information. So what exactly is the analytic relationship between forfeited information and resource usage for any given design decision?

In trading exact evaluation for a safe, computable approximation to it, analysis negotiates a compromise between complexity and precision. But what exactly are the trade-offs involved in this negotiation? For any given design decision, what is given up and what is gained? What makes an analysis rich and expressive? What makes an analysis fast and resource-efficient?

We examine these questions in the setting of *flow analysis* (Jones, 1981; Sestoft, 1988; Sestoft, 1989; Shivers, 1988; Shivers, 1991; Midtgaard, 2007), a fundamental and ubiquitous static analysis for higher-order programming languages. It forms the basis of almost all other analyses and is a much-studied component of compiler technology.

Flow analysis answers basic questions such as "what functions can be applied?," and "to what arguments?" These questions specify well-defined, significant *decision problems*, quite apart from any algorithm proposed to solve them. This paper examines the inherent computational difficulty of deciding these problems.

If we consider the most useful analysis the one which yields complete and perfectly accurate information about the running of a program, then clearly this analysis is intractable—it consumes the same computational resources as running the program. At the other end of the spectrum, if the least useful analysis yields no information about the running of a program, then this analysis is surely feasible, but useless.

If the design of software is really a science, we have to understand the trade-offs between the running time of static analyzers, and the accuracy of their computations.

There is substantial empirical experience, which gives a partial answer to these questions. However, despite being the fundamental analysis of higher-order programs, despite

being the subject of investigation for over twenty-five years, and the great deal of expended effort deriving clever ways to tame the cost, there has remained a poverty of analytic knowledge on the complexity of flow analysis, the essence of how it is computed, and where the sources of approximation occur that make the analysis work.

This paper is intended to repair such lacunae in our understanding.

1.2 Summary of Results

1.2.1 Linearity, Analysis and Normalization

- Normalization and analysis are equivalent for linear programs.

Although variants of flow analysis abound, we identify a core language, the linear λ-calculus, for which all of these variations coincide. In other words, for *linear* programs—those written to use each bound variable exactly once—all known flow analyses will produce equivalent information.

It is straightforward to observe that in a *linear* λ-term, 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. Generalizing this observation, analysis of a linear λ-term coincides exactly with its evaluation. So not only are the varying analyses equivalent to each other on linear terms, they are all equivalent to evaluation.

Linearity is an equalizer among variations of static analysis, and a powerful tool in proving lower bounds.

1.2.2 Monovariance and PTIME

- 0CFA and other monovariant flow analyses are complete for PTIME.

By definition, a *monovariant* analysis (e.g. 0CFA), does not distinguish between occurrences of the same variable bound in different calling contexts. But the distinction is needless for linear programs and analysis becomes evaluation under another name. This opens the door to proving lower bounds on the complexity of the analysis by writing—to the degree possible—computationally intensive, linear programs, which will be faithfully executed by the analyzer rather than the interpreter.

We rely on a symmetric coding of Boolean logic in the linear λ-calculus to simulate circuits and reduce the 0CFA decision problem to the canonical PTIME problem, the circuit value problem. This shows, since the inclusion is well-known, that 0CFA is complete for PTIME. Consequently, 0CFA is inherently sequential and there is no fast parallel algorithm for 0CFA (unless PTIME $=$ NC). Moreover, this remains true for a number of *further approximations* to 0CFA.

The best known algorithms for computing 0CFA are often not practical for large programs. Nonetheless, information can be given up in the service of quickly computing a necessarily less precise analysis. For example, by forfeiting 0CFA's notion of directionality, algorithms for Henglein's simple closure analysis run in near linear time (Henglein, 1992). Similarly, by explicitly bounding the number of passes the analyzer is allowed over the program, as in Ashley and Dybvig's sub-0CFA (Ashley & Dybvig, 1998), we can recover running times that are linear in the size of the program. But the question remains: Can

we do better? For example, is it possible to compute these less precise analyses in logarithmic space? We show that without profound combinatorial breakthroughs (PTIME = LOGSPACE), the answer is no. Simple closure analysis, sub-0CFA, and other analyses that approximate or restrict 0CFA, *require*—and are therefore, like 0CFA, complete for—polynomial time.

1.2.3 0CFA with η-Expansion and LOGSPACE

- 0CFA of typed, η-expanded programs is complete for LOGSPACE.

We identify a restricted class of functional programs whose 0CFA decision problem may be simpler—namely, complete for LOGSPACE. Consider programs that are simply typed, and where a variable in the function position or the argument position of an application is fully η-expanded. This case—especially, but not only when the programs are linear—strongly resembles multiplicative linear logic with *atomic* axioms.

We rely on the resemblance to bring recent results on the complexity of normalization in linear logic to bear on the analysis of η-expanded programs resulting in a LOGSPACE-complete variant of 0CFA.

1.2.4 kCFA and EXPTIME

- kCFA is complete for EXPTIME for all $k > 0$.

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.

A fairly straightforward calculation shows that kCFA can be computed in exponential time. We show that the naive algorithm is essentially the best one. There is, in the worst case—and plausibly, in practice—no way to tame the cost of the analysis. Exponential time is required.

2 Monovariant Analysis and PTIME

The monovariant form of flow analysis defined over the pure λ-calculus has emerged as a fundamental notion of flow analysis for higher-order languages, and some form of flow analysis is used in most analyses for higher-order languages (Heintze & McAllester, 1997a).

In this chapter, we examine several of the most well-known variations of monovariant flow analysis: Shivers' 0CFA (1988), Henglein's simple closure analysis (1992), Heintze and McAllester's subtransitive flow analysis (1997a), Ashley and Dybvig's sub-0CFA (1998), Mossin's single source/use analysis (1998), and others.

In each case, evaluation and analysis are proved equivalent for the class of linear programs and a precise characterization of the computational complexity of the analysis, namely PTIME-completeness, is given.

2.1 The Approximation of Monovariance

To ensure tractability of any static analysis, there has to be an *approximation* of something, where information is deliberately *lost* in the service of providing what is left in a reasonable amount of time. A good example of what is lost during *monovariant* static analysis is that the information gathered for each occurrence of a bound variable is merged. When variable f occurs twice in function position with two different arguments,

$$(f\ v_1)\cdots(f\ v_2)$$

a monovariant flow analysis will blur which copy of the function is applied to which argument. If a function $\lambda z.e$ flows into f in this example, the analysis treats occurrences of z in e as bound to *both* v_1 and v_2.

Shivers' 0CFA is among the most well-known forms of monovariant flow analysis; however, the best known algorithm for 0CFA requires nearly cubic time in proportion to the size of the analyzed program.

It is natural to wonder whether it is possible to do better, avoiding this bottleneck, either by improving the 0CFA algorithm in some clever way or by *further* approximation for the sake of faster computation.

Consequently, several analyses have been designed to approximate 0CFA by trading precision for faster computation. Henglein's simple closure analysis, for example, forfeits the notion of directionality in flows. Returning to the earlier example,

$$f(\lambda x.e')\cdots f(\lambda y.e'')$$

simple closure analysis, like 0CFA, will blur $\lambda x.e'$ and $\lambda y.e''$ as arguments to f, causing z to be bound to both. But unlike 0CFA, a *bidirectional* analysis such as simple closure analysis will identify two λ-terms with each other. That is, because both are arguments to the same function, by the bi-directionality of the flows, $\lambda x.e'$ may flow out of $\lambda y.e''$ and *vice versa*.

Because of this further curtailing of information, simple closure analysis enjoys an "almost linear" time algorithm. But in making trade-offs between precision and complexity, what has been given up and what has been gained? Where do these analyses differ and where do they coincide?

We identify a core language—the linear λ-calculus—where 0CFA, simple closure analysis, and many other known approximations or restrictions to 0CFA are rendered identical. Moreover, for this core language, analysis corresponds with (instrumented) evaluation. Because analysis faithfully captures evaluation, and because the linear λ-calculus is complete for PTIME, we derive PTIME-completeness results for all of these analyses.

Proof of this lower bound relies on the insight that linearity of programs subverts the approximation of analysis and renders it equivalent to evaluation. We establish a correspondence between Henglein's simple closure analysis and evaluation for linear terms. In doing so, we derive sufficient conditions effectively characterizing not only simple closure analysis, but many known flow analyses computable in less than cubic time, such as Ashley and Dybvig's sub-0CFA, Heintze and McAllester's subtransitive flow analysis, and Mossin's single source/use analysis.

By using a nonstandard, symmetric implementation of Boolean logic within the linear lambda calculus, it is possible to simulate circuits at analysis-time, and as a consequence, we prove that all of the above analyses are complete for PTIME. Any sub-polynomial algorithm for these problems would require (unlikely) breakthrough results in complexity, such as PTIME = LOGSPACE.

We may continue to wonder whether it is possible to do better, either by improving the 0CFA algorithm in some clever way or by further approximation for faster computation. However these theorems demonstrate the limits of both avenues. 0CFA is inherently sequential, and so is *any* algorithm for it, no matter how clever. Designing a provably efficient parallel algorithm for 0CFA is as hard as parallelizing all polynomial time computations. On the other hand, further approximations, such as simple closure analysis and most other variants of monovariant flow analysis, make no approximation on a linear program. This means they too are inherently sequential and no easier to parallelize.

2.2 0CFA

Something interesting happens when $k = 0$. Notice in the application rule of the kCFA abstract evaluator of **??** that environments are extended as $\rho[x \mapsto \lceil \delta\ell \rceil_k]$. When $k = 0$, $\lceil \delta\ell \rceil_0 = \varepsilon$. All contour environments map to the empty contour, and therefore carry no contextual information. As such, 0CFA is a "monovariant" analysis, analogous to simple-type inference, which is a monovariant type analysis.

Since there is only one constant environment (the "everywhere ε" environment), environments of **??** can be eliminated from the analysis altogether and the cache no longer needs a contour argument. Likewise, the set of abstract values collapses from $\mathscr{P}(\mathbf{Term} \times \mathbf{Env})$ into $\mathscr{P}(\mathbf{Term})$.

The result of 0CFA is an *abstract cache* that maps each program point (i.e., label) to a set of lambda abstractions which potentially flow into this program point at run-time:

$$\begin{array}{lcl} \widehat{\mathsf{C}} & : & \mathbf{Lab} \to \mathscr{P}(\mathbf{Term}) \\ \hat{\mathsf{r}} & : & \mathbf{Var} \to \mathscr{P}(\mathbf{Term}) \end{array}$$

$$\widehat{\mathbf{Cache}} \quad = \quad (\mathbf{Lab} \to \mathscr{P}(\mathbf{Term})) \times (\mathbf{Var} \to \mathscr{P}(\mathbf{Term}))$$

Caches are extended using the notation $\widehat{\mathsf{C}}[\ell \mapsto s]$, and we write $\widehat{\mathsf{C}}[\ell \mapsto^+ s]$ to mean $\widehat{\mathsf{C}}[\ell \mapsto (s \cup \widehat{\mathsf{C}}(\ell))]$. It is convenient to sometimes think of caches as mutable tables (as we do in the algorithm below), so we abuse syntax, letting this notation mean both functional extension and destructive update. It should be clear from context which is implied.

The Analysis: We present the specification of the analysis here in the style of Nielson, Nielson, and Hankin (1999). Each subexpression is identified with a unique superscript *label* ℓ, which marks that program point; $\widehat{\mathsf{C}}(\ell)$ stores all possible values flowing to point ℓ, $\hat{\mathsf{r}}(x)$ stores all possible values flowing to the definition site of x. An *acceptable* 0CFA analysis for an expression e is written $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \models e$ and derived according to the scheme given in Figure 1.

The $\models$ relation needs to be coinductively defined since verifying a judgment $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \models e$ may obligate verification of $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \models e'$ which in turn may require verification of $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \models e$.

$$
\begin{array}{lll}
\widehat{\mathsf{C}},\hat{\mathsf{r}} \models x^{\ell} & \text{iff} & \hat{\mathsf{r}}(x) \subseteq \widehat{\mathsf{C}}(\ell) \\
\widehat{\mathsf{C}},\hat{\mathsf{r}} \models (\lambda x.e)^{\ell} & \text{iff} & \lambda x.e \in \widehat{\mathsf{C}}(\ell) \\
\widehat{\mathsf{C}},\hat{\mathsf{r}} \models (t^{\ell_1}\ t^{\ell_2})^{\ell} & \text{iff} & \widehat{\mathsf{C}},\hat{\mathsf{r}} \models t^{\ell_1} \wedge \widehat{\mathsf{C}},\hat{\mathsf{r}} \models t^{\ell_2} \wedge \\
 & & \quad \forall \lambda x.t^{\ell_0} \in \widehat{\mathsf{C}}(\ell_1) : \\
 & & \quad\quad \widehat{\mathsf{C}}(\ell_2) \subseteq \hat{\mathsf{r}}(x) \wedge \\
 & & \quad\quad \widehat{\mathsf{C}},\hat{\mathsf{r}} \models t^{\ell_0} \wedge \\
 & & \quad\quad \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell)
\end{array}
$$

Fig. 1. 0CFA abstract cache acceptability.

The above specification of acceptability, when read as a table, defines a functional, which is monotonic, has a fixed point, and $\models$ is defined coinductively as the greatest fixed point of this functional.[1]

Writing $\widehat{\mathsf{C}},\hat{\mathsf{r}} \models t^{\ell}$ means "the abstract cache contains all the flow information for program fragment t at program point ℓ." The goal is to determine the *least* cache solving these constraints to obtain the most precise analysis. Caches are partially ordered with respect to the program of interest:

$$
\begin{array}{lcll}
\widehat{\mathsf{C}} & \sqsubseteq & \widehat{\mathsf{C}}' & \text{iff} \quad \forall \ell : \widehat{\mathsf{C}}(\ell) \subseteq \widehat{\mathsf{C}}'(\ell) \\
\hat{\mathsf{r}} & \sqsubseteq & \hat{\mathsf{r}}' & \text{iff} \quad \forall x : \hat{\mathsf{r}}(x) \subseteq \hat{\mathsf{r}}'(x)
\end{array}
$$

These constraints can be thought of as an abstract evaluator— $\widehat{\mathsf{C}},\hat{\mathsf{r}} \models t^{\ell}$ simply means *evaluate* t^{ℓ}, which serves *only* to update an (initially empty) cache.

$$
\begin{array}{lcl}
\mathscr{A}_0[\![x^{\ell}]\!] & = & \widehat{\mathsf{C}}(\ell) \leftarrow \hat{\mathsf{r}}(x) \\
\mathscr{A}_0[\![(\lambda x.e)^{\ell}]\!] & = & \widehat{\mathsf{C}}(\ell) \leftarrow \{\lambda x.e\} \\
\mathscr{A}_0[\![(t^{\ell_1}\ t^{\ell_2})^{\ell}]\!] & = & \mathscr{A}_0[\![t^{\ell_1}]\!];\ \mathscr{A}_0[\![t^{\ell_2}]\!]; \\
 & & \textbf{for each } \lambda x.t^{\ell_0} \textbf{ in } \widehat{\mathsf{C}}(\ell_1) \textbf{ do} \\
 & & \quad \hat{\mathsf{r}}(x) \leftarrow \widehat{\mathsf{C}}(\ell_2); \\
 & & \quad \mathscr{A}_0[\![t^{\ell_0}]\!]; \\
 & & \quad \widehat{\mathsf{C}}(\ell) \leftarrow \widehat{\mathsf{C}}(\ell_0)
\end{array}
$$

Fig. 2. Abstract evaluator $\mathscr{A}_0$ for 0CFA, imperative style.

The abstract evaluator $\mathscr{A}_0[\![\cdot]\!]$ is iterated until the finite cache reaches a fixed point.

Fine Print: A single iteration of $\mathscr{A}_0[\![e]\!]$ may in turn make a recursive call $\mathscr{A}_0[\![e]\!]$ with no change in the cache, so care must be taken to avoid looping. This amounts to appealing to the coinductive hypothesis $\widehat{\mathsf{C}},\hat{\mathsf{r}} \models e$ in verifying $\widehat{\mathsf{C}},\hat{\mathsf{r}} \models e$. However, we consider this inessential detail, and it can safely be ignored for the purposes of obtaining our main results in which this behavior is *never triggered*.

Since the cache size is polynomial in the program size, so is the running time, as the cache is *monotonic*—values are put in, but never taken out. Thus the analysis and any

[1] See Nielson, Nielson, and Hankin (1999) for details and a thorough discussion of coinduction in specifying static analyses.

decision problems answered by the analysis are clearly computable within polynomial time.

Lemma 1
The control flow problem for 0CFA is contained in PTIME.

Proof
0CFA computes a binary relation over a fixed structure. The computation of the relation is monotone: it begins as empty and is added to incrementally. Because the structure is finite, a fixed point must be reached by this incremental computation. The binary relation can be at most polynomial in size, and each increment is computed in polynomial time. □

An Example: Consider the following program, which we will return to discuss further in subsequent analyses:

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

The least 0CFA is given by the following cache:

$$\begin{array}{lll}
\widehat{\mathsf{C}}(1)=\{\lambda x\} & \widehat{\mathsf{C}}(6) &= \{\lambda x, \lambda y\} \\
\widehat{\mathsf{C}}(2)=\{\lambda x\} & \widehat{\mathsf{C}}(7) &= \{\lambda f\} \qquad \widehat{\mathsf{r}}(f)=\{\lambda x\} \\
\widehat{\mathsf{C}}(3)=\{\lambda x, \lambda y\} & \widehat{\mathsf{C}}(8) &= \{\lambda x, \lambda y\} \qquad \widehat{\mathsf{r}}(x)=\{\lambda x, \lambda y\} \\
\widehat{\mathsf{C}}(4)=\{\lambda y\} & \widehat{\mathsf{C}}(9) &= \{\lambda x\} \qquad \widehat{\mathsf{r}}(y)=\{\lambda y\} \\
\widehat{\mathsf{C}}(5)=\{\lambda y\} & \widehat{\mathsf{C}}(10) &= \{\lambda x, \lambda y\}
\end{array}$$

where we write λx as shorthand for $\lambda x.x^8$, etc.

2.3 Henglein's Simple Closure Analysis

Simple closure analysis follows from an observation by Henglein some 15 years ago "in an influential though not often credited technical report" [page 4] (Midtgaard, 2007): he noted that the standard control flow analysis can be computed in dramatically less time by changing the specification of flow constraints to use equality rather than containment (Henglein, 1992). The analysis bears a strong resemblance to simple-type inference—analysis can be performed by emitting a system of equality constraints and then solving them using *unification*, which can be computed in almost linear time with a union-find data structure.

Consider a program with both $(f\ x)$ and $(f\ y)$ as subexpressions. Under 0CFA, whatever flows into x and y will also flow into the formal parameter of all abstractions flowing into f, but it is not necessarily true that whatever flows into x *also* flows into y and *vice versa*. However, under simple closure analysis, this is the case. For this reason, flows in simple closure analysis are said to be *bidirectional*.

The Analysis: The specification of the analysis is given in Figure 3.

The Algorithm: We write $\widehat{\mathsf{C}}[\ell \leftrightarrow \ell']$ to mean $\widehat{\mathsf{C}}[\ell \mapsto^+ \widehat{\mathsf{C}}(\ell')][\ell' \mapsto^+ \widehat{\mathsf{C}}(\ell)]$.

$$
\begin{array}{lll}
\widehat{\mathsf{C}},\hat{\mathsf{r}} \models x^{\ell} & \text{iff} & \hat{\mathsf{r}}(x) = \widehat{\mathsf{C}}(\ell) \\
\widehat{\mathsf{C}},\hat{\mathsf{r}} \models (\lambda x.e)^{\ell} & \text{iff} & \lambda x.e \in \widehat{\mathsf{C}}(\ell) \\
\widehat{\mathsf{C}},\hat{\mathsf{r}} \models (t^{\ell_1}\ t^{\ell_2})^{\ell} & \text{iff} & \widehat{\mathsf{C}},\hat{\mathsf{r}} \models t^{\ell_1} \wedge \widehat{\mathsf{C}},\hat{\mathsf{r}} \models t^{\ell_2} \wedge \\
 & & \quad \forall \lambda x.t^{\ell_0} \in \widehat{\mathsf{C}}(\ell_1) : \\
 & & \quad\quad \widehat{\mathsf{C}}(\ell_2) = \hat{\mathsf{r}}(x) \wedge \\
 & & \quad\quad \widehat{\mathsf{C}},\hat{\mathsf{r}} \models t^{\ell_0} \wedge \\
 & & \quad\quad \widehat{\mathsf{C}}(\ell_0) = \widehat{\mathsf{C}}(\ell)
\end{array}
$$

Fig. 3. Simple closure analysis abstract cache acceptability.

$$
\begin{array}{lcl}
\mathscr{A}_0[\![x^{\ell}]\!] & = & \widehat{\mathsf{C}}(\ell) \hookleftarrow \hat{\mathsf{r}}(x) \\
\mathscr{A}_0[\![(\lambda x.e)^{\ell}]\!] & = & \widehat{\mathsf{C}}(\ell) \leftarrow \{\lambda x.e\} \\
\mathscr{A}_0[\![(t_1^{\ell_1} t_2^{\ell_2})^{\ell}]\!] & = & \mathscr{A}_0[\![t_1^{\ell_1}]\!];\ \mathscr{A}_0[\![t_2^{\ell_2}]\!]; \\
 & & \textbf{for each } \lambda x.t_0^{\ell_0} \textbf{ in } \widehat{\mathsf{C}}(\ell_1) \textbf{ do} \\
 & & \quad \hat{\mathsf{r}}(x) \hookleftarrow \widehat{\mathsf{C}}(\ell_2); \\
 & & \quad \mathscr{A}_0[\![t_0^{\ell_0}]\!]; \\
 & & \quad \widehat{\mathsf{C}}(\ell) \hookleftarrow \widehat{\mathsf{C}}(\ell_0)
\end{array}
$$

The abstract evaluator $\mathscr{A}_0[\![\cdot]\!]$ is iterated until a fixed point is reached.[2] By similar reasoning to that given for 0CFA, simple closure analysis is clearly computable within polynomial time.

Lemma 2
The control flow problem for simple closure analysis is contained in PTIME.

An Example: Recall the example program of the previous section:

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

Notice that $\lambda x.x$ is applied to itself and then to $\lambda y.y$, so x will be bound to both $\lambda x.x$ and $\lambda y.y$, which induces an equality between these two terms. Consequently, everywhere that 0CFA was able to deduce a flow set of $\{\lambda x\}$ or $\{\lambda y\}$ will be replaced by $\{\lambda x, \lambda y\}$ under a simple closure analysis. The least simple closure analysis is given by the following cache (new flows are underlined):

$$
\begin{array}{llll}
\widehat{\mathsf{C}}(1) = \{\lambda x, \underline{\lambda y}\} & \widehat{\mathsf{C}}(6) = \{\lambda x, \lambda y\} & & \\
\widehat{\mathsf{C}}(2) = \{\lambda x, \underline{\lambda y}\} & \widehat{\mathsf{C}}(7) = \{\lambda f\} & \hat{\mathsf{r}}(f) = \{\lambda x, \underline{\lambda y}\} & \\
\widehat{\mathsf{C}}(3) = \{\lambda x, \lambda y\} & \widehat{\mathsf{C}}(8) = \{\lambda x, \lambda y\} & \hat{\mathsf{r}}(x) = \{\lambda x, \lambda y\} & \\
\widehat{\mathsf{C}}(4) = \{\lambda y, \underline{\lambda x}\} & \widehat{\mathsf{C}}(9) = \{\lambda x, \underline{\lambda y}\} & \hat{\mathsf{r}}(y) = \{\lambda y, \underline{\lambda x}\} & \\
\widehat{\mathsf{C}}(5) = \{\lambda y, \underline{\lambda x}\} & \widehat{\mathsf{C}}(10) = \{\lambda x, \lambda y\} & &
\end{array}
$$

2.4 Linearity: Analysis is Evaluation

It is straightforward to observe that in a *linear* λ-term, 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

[2] The fine print of subsection 2.2 applies as well.

$$
\begin{array}{lcl}
\mathscr{E}' & : & \mathbf{Exp} \times \mathbf{Env} \rightharpoonup \mathbf{Val} \\
\\
\mathscr{E}'[\![x^\ell]\!][x \mapsto v] & = & v \\
\mathscr{E}'[\![(\lambda x.e)^\ell]\!]\rho & = & \langle \lambda x.e, \rho \rangle \\
\mathscr{E}'[\![(e_1\ e_2)^\ell]\!]\rho & = & \textbf{let } \langle \lambda x.e_0, \rho' \rangle = \mathscr{E}'[\![e_1]\!]\rho \restriction \mathbf{fv}(e_1) \textbf{ in} \\
 & & \textbf{let } v = \mathscr{E}'[\![e_2]\!]\rho \restriction \mathbf{fv}(e_2) \textbf{ in} \\
 & & \quad \mathscr{E}'[\![e_0]\!]\rho'[x \mapsto v]
\end{array}
$$

Fig. 4. Evaluator $\mathscr{E}'$.

argument.[3] Generalizing this observation, analysis of a linear λ-term coincides exactly with its evaluation. So not only are the analyses equivalent on linear terms, but they are also synonymous with evaluation.

A natural and expressive class of such linear terms are the ones which implement Boolean logic. When analyzing the coding of a Boolean circuit and its inputs, 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 the circuit value problem (Ladner, 1975), which is complete for PTIME, can be reduced to an instance of the 0CFA control flow problem, we conclude this control flow problem is PTIME-hard. Further, as 0CFA can be computed in polynomial time, the control flow problem for 0CFA is PTIME-complete.

One way to realize the computational potency of a static analysis is to subvert this loss of information, making the analysis an *exact* computational tool. Lower bounds on the expressiveness of an analysis thus become exercises in hacking, armed with this newfound tool. Clearly the more approximate the analysis, the less we have to work with, computationally speaking, and the more we have to do to undermine the approximation. But a fundamental technique has emerged in understanding expressivity in static analysis—*linearity*.

In this section, we show that when the program is *linear*—every bound variable occurs exactly once—analysis and evaluation are synonymous.

First, we start by considering an alternative evaluator, given in Figure 4, which is slightly modified from the one given in **??**. Notice that this evaluator "tightens" the environment in the case of an application, thus maintaining throughout evaluation that the domain of the environment is exactly the set of free variables in the expression. When evaluating a variable occurrence, there is only one mapping in the environment: the binding for this variable. Likewise, when constructing a closure, the environment does not need to be restricted: it already is.

This alternative evaluator $\mathscr{E}'$ will be useful in reasoning about linear programs, but it should be clear that it is equivalent to the original, standard evaluator $\mathscr{E}$ of **??**.

Lemma 3
$\mathscr{E}[\![e]\!]\rho \Longleftrightarrow \mathscr{E}'[\![e]\!]\rho$, when $\mathbf{dom}(\rho) = \mathbf{fv}(e)$.

[3] 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.

In a linear program, each mapping in the environment corresponds to the single occurrence of a bound variable. So when evaluating an application, this tightening *splits* the environment ρ into (ρ_1, ρ_2), where ρ_1 closes the operator, ρ_2 closes the operand, and $\mathbf{dom}(\rho_1) \cap \mathbf{dom}(\rho_2) = \emptyset$.

Definition 1
Environment ρ *linearly closes* t (or $\langle t, \rho \rangle$ is a *linear closure*) iff t is linear, ρ closes t, and for all $x \in \mathbf{dom}(\rho)$, x occurs exactly once (free) in t, $\rho(x)$ is a linear closure, and for all $y \in \mathbf{dom}(\rho), x$ does not occur (free or bound) in $\rho(y)$. The *size* of a linear closure $\langle t, \rho \rangle$ is defined as:

$$\begin{aligned} |t, \rho| &= |t| + |\rho| \\ |x| &= 1 \\ |(\lambda x.t^{\ell})| &= 1 + |t| \\ |(t_1^{\ell_1}\ t_2^{\ell_2})| &= 1 + |t_1| + |t_2| \\ |[x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]| &= n + \sum_i |c_i| \end{aligned}$$

The following lemma states that evaluation of a linear closure cannot produce a larger value. This is the environment-based analog to the easy observation that β-reduction *strictly* decreases the size of a linear term.

Lemma 4
If ρ linearly closes t and $\mathscr{E}'[\![t^{\ell}]\!]\rho = c$, then $|c| \leq |t, \rho|$.

Proof
Straightforward by induction on $|t, \rho|$, reasoning by case analysis on t. Observe that the size strictly decreases in the application and variable case, and remains the same in the abstraction case. □

The function $\mathbf{lab}(\cdot)$ is extended to closures and environments by taking the union of all labels in the closure or in the range of the environment, respectively.

Definition 2
The set of labels in a given term, expression, environment, or closure is defined as follows:

$$\begin{aligned} \mathbf{lab}(t^{\ell}) &= \mathbf{lab}(t) \cup \{\ell\} & \mathbf{lab}(e_1\ e_2) &= \mathbf{lab}(e_1) \cup \mathbf{lab}(e_2) \\ \mathbf{lab}(x) &= \{x\} & \mathbf{lab}(\lambda x.e) &= \mathbf{lab}(e) \cup \{x\} \\ \mathbf{lab}(t, \rho) &= \mathbf{lab}(t) \cup \mathbf{lab}(\rho) & \mathbf{lab}(\rho) &= \bigcup_{x \in \mathbf{dom}(\rho)} \mathbf{lab}(\rho(x)) \end{aligned}$$

Definition 3
A cache $\widehat{\mathsf{C}}, \hat{\mathsf{r}}$ *respects* $\langle t, \rho \rangle$ (written $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \vdash t, \rho$) when,

1. ρ linearly closes t,
2. $\forall x \in \mathbf{dom}(\rho).\rho(x) = \langle t', \rho' \rangle \Rightarrow \hat{\mathsf{r}}(x) = \{t'\}$ and $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \vdash t', \rho'$,
3. $\forall \ell \in \mathbf{lab}(t), \widehat{\mathsf{C}}(\ell) = \emptyset$, and
4. $\forall x \in \mathbf{bv}(t), \hat{\mathsf{r}}(x) = \emptyset$.

Clearly, $\emptyset \vdash t, \emptyset$ when t is closed and linear, i.e. t is a linear

Figure 5 gives a "cache-passing" functional algorithm for $\mathscr{A}_0[\![\cdot]\!]$ of subsection 2.3. It is equivalent to the functional style abstract evaluator of **??** specialized by letting $k = 0$. We now state and prove the main theorem of this section in terms of this abstract evaluator.

$$
\begin{array}{lcl}
\mathscr{A}_0 & : & \mathbf{Exp} \times \widehat{\mathbf{Cache}} \to \widehat{\mathbf{Cache}} \\
\\
\mathscr{A}_0[\![x^\ell]\!]\ \widehat{\mathsf{C}},\hat{\mathsf{r}} & = & \widehat{\mathsf{C}}[\ell \mapsto \hat{\mathsf{r}}(x)],\hat{\mathsf{r}} \\
\mathscr{A}_0[\![(\lambda x.e)^\ell]\!]\ \widehat{\mathsf{C}},\hat{\mathsf{r}} & = & \widehat{\mathsf{C}}[\ell \mapsto \{\lambda x.e\}],\hat{\mathsf{r}} \\
\mathscr{A}_0[\![(t^{\ell_1}\ t^{\ell_2})^\ell]\!]\ \widehat{\mathsf{C}},\hat{\mathsf{r}} & = & \widehat{\mathsf{C}}_3[\ell \mapsto \widehat{\mathsf{C}}_3(\ell_0)],\hat{\mathsf{r}}_3, \text{ where} \\
 & & \quad \delta' = \lceil \delta\ell \rceil_k \\
 & & \quad \widehat{\mathsf{C}}_1,\hat{\mathsf{r}}_1 = \mathscr{A}_0[\![t^{\ell_1}]\!]\ \widehat{\mathsf{C}},\hat{\mathsf{r}} \\
 & & \quad \widehat{\mathsf{C}}_2,\hat{\mathsf{r}}_2 = \mathscr{A}_0[\![t^{\ell_2}]\!]\ \widehat{\mathsf{C}}_1,\hat{\mathsf{r}}_1 \\
 & & \quad \widehat{\mathsf{C}}_3,\hat{\mathsf{r}}_3 = \\
 & & \quad \bigsqcup_{\lambda x.t^{\ell_0}}^{\widehat{\mathsf{C}}_2(\ell)} \left(\mathscr{A}_0[\![t^{\ell_0}]\!]\ \widehat{\mathsf{C}}_2,\hat{\mathsf{r}}_2[x \mapsto \widehat{\mathsf{C}}_2(\ell_2)]\right)
\end{array}
$$

Fig. 5. Abstract evaluator $\mathscr{A}_0$ for 0CFA, functional style.

Theorem 1
If $\widehat{\mathsf{C}},\hat{\mathsf{r}} \vdash t,\rho$, $\widehat{\mathsf{C}}(\ell) = \emptyset$, $\ell \notin \mathbf{lab}(t,\rho)$, $\mathscr{E}'[\![t^\ell]\!]\rho = \langle t',\rho'\rangle$, and $\mathscr{A}_0[\![t^\ell]\!]\widehat{\mathsf{C}},\hat{\mathsf{r}} = \widehat{\mathsf{C}}',\hat{\mathsf{r}}'$, then $\widehat{\mathsf{C}}'(\ell) = \{t'\}$, $\widehat{\mathsf{C}}' \vdash t',\rho'$, and $\widehat{\mathsf{C}}',\hat{\mathsf{r}}' \models t^\ell$.

An important consequence is noted in Corollary 1.

Proof
By induction on $|t,\rho|$, reasoning by case analysis on t.

- Case $t \equiv x$.
  Since $\widehat{\mathsf{C}} \vdash x,\rho$ and ρ linearly closes x, thus $\rho = [x \mapsto \langle t',\rho'\rangle]$ and ρ' linearly closes t'. By definition,

$$
\begin{array}{rcl}
\mathscr{E}'[\![x^\ell]\!]\rho & = & \langle t',\rho'\rangle, \text{ and} \\
\mathscr{A}_0[\![x^\ell]\!]\widehat{\mathsf{C}} & = & \widehat{\mathsf{C}}[x \leftrightarrow \ell].
\end{array}
$$

  Again since $\widehat{\mathsf{C}} \vdash x,\rho$, $\widehat{\mathsf{C}}(x) = \{t'\}$, with which the assumption $\widehat{\mathsf{C}}(\ell) = \emptyset$ implies

$$\widehat{\mathsf{C}}[x \leftrightarrow \ell](x) = \widehat{\mathsf{C}}[x \leftrightarrow \ell](\ell) = \{t'\},$$

  and therefore $\widehat{\mathsf{C}}[x \leftrightarrow \ell] \models x^\ell$. It remains to show that $\widehat{\mathsf{C}}[x \leftrightarrow \ell] \vdash t',\rho'$. By definition, $\widehat{\mathsf{C}} \vdash t',\rho'$. Since x and ℓ do not occur in t',ρ' by linearity and assumption, respectively, it follows that $\widehat{\mathsf{C}}[x \mapsto \ell] \vdash t',\rho'$ and the case holds.
- Case $t \equiv \lambda x.e_0$.
  By definition,

$$
\begin{array}{rcl}
\mathscr{E}'[\![(\lambda x.e_0)^\ell]\!]\rho & = & \langle \lambda x.e_0,\rho\rangle, \\
\mathscr{A}_0[\![(\lambda x.e_0)^\ell]\!]\widehat{\mathsf{C}} & = & \widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}],
\end{array}
$$

  and by assumption $\widehat{\mathsf{C}}(\ell) = \emptyset$, so $\widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}](\ell) = \{\lambda x.e_0\}$ and therefore $\widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}] \models (\lambda x.e_0)^\ell$. By assumptions $\ell \notin \mathbf{lab}(\lambda x.e_0,\rho)$ and $\widehat{\mathsf{C}} \vdash \lambda x.e_0,\rho$, it follows that $\widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}] \vdash \lambda x.e_0,\rho$ and the case holds.
- Case $t \equiv t_1^{\ell_1}\ t_2^{\ell_2}$. Let

$$
\begin{array}{rcl}
\mathscr{E}'[\![t_1]\!]\rho \restriction \mathbf{fv}(t_1^{\ell_1}) & = & \langle v_1,\rho_1\rangle = \langle \lambda x.t_0^{\ell_0},\rho_1\rangle, \\
\mathscr{E}'[\![t_2]\!]\rho \restriction \mathbf{fv}(t_2^{\ell_2}) & = & \langle v_2,\rho_2\rangle,
\end{array}
$$

$$\begin{aligned}\mathscr{A}_0[\![t_1]\!]\widehat{\mathsf{C}} &= \widehat{\mathsf{C}}_1, \text{ and}\\ \mathscr{A}_0[\![t_2]\!]\widehat{\mathsf{C}} &= \widehat{\mathsf{C}}_2.\end{aligned}$$

Clearly, for $i \in \{1,2\}$, $\widehat{\mathsf{C}} \vdash t_i, \rho \upharpoonright \mathbf{fv}(t_i)$ and

$$1 + \sum_i |t_i^{\ell_i}, \rho \upharpoonright \mathbf{fv}(t_i^{\ell_i})| = |(t_1^{\ell_1}\ t_2^{\ell_2}), \rho|.$$

By induction, for $i \in \{1,2\} : \widehat{\mathsf{C}}_i(\ell_i) = \{v_i\}, \widehat{\mathsf{C}}_i \vdash \langle v_i, \rho_i\rangle$, and $\widehat{\mathsf{C}}_i \models t_i^{\ell_i}$. From this, it is straightforward to observe that $\widehat{\mathsf{C}}_1 = \widehat{\mathsf{C}} \cup \widehat{\mathsf{C}}'_1$ and $\widehat{\mathsf{C}}_2 = \widehat{\mathsf{C}} \cup \widehat{\mathsf{C}}'_2$ where $\widehat{\mathsf{C}}'_1$ and $\widehat{\mathsf{C}}'_2$ are disjoint. So let $\widehat{\mathsf{C}}_3 = (\widehat{\mathsf{C}}_1 \cup \widehat{\mathsf{C}}_2)[x \leftrightarrow \ell_2]$. It is clear that $\widehat{\mathsf{C}}_3 \models t_i^{\ell_i}$. Furthermore,

$$\begin{aligned}\widehat{\mathsf{C}}_3 &\vdash t_0, \rho_1[x \mapsto \langle v_2, \rho_2\rangle],\\ \widehat{\mathsf{C}}_3(\ell_0) &= \emptyset, \text{ and}\\ \ell_0 &\notin \mathbf{lab}(t_0, \rho_1[x \mapsto \langle v_2, \rho_2\rangle]).\end{aligned}$$

By Lemma 4, $|v_i, \rho_i| \leq |t_i, \rho \upharpoonright \mathbf{fv}(t_i)|$, therefore

$$|t_0, \rho_1[x \mapsto \langle v_2, \rho_2\rangle]| < |(t_1^{\ell_1}\ t_2^{\ell_2})|.$$

Let

$$\begin{aligned}\mathscr{E}'[\![t_0^{\ell_0}]\!]\rho_1[x \mapsto \langle v_2, \rho_2\rangle] &= \langle v', \rho'\rangle,\\ \mathscr{A}_0[\![t_0^{\ell_0}]\!]\widehat{\mathsf{C}}_3 &= \widehat{\mathsf{C}}_4,\end{aligned}$$

and by induction, $\widehat{\mathsf{C}}_4(\ell_0) = \{v'\}$, $\widehat{\mathsf{C}}_4 \vdash v', \rho'$, and $\widehat{\mathsf{C}}_4 \models v'$. Finally, observe that $\widehat{\mathsf{C}}_4[\ell \leftrightarrow \ell_0](\ell) = \widehat{\mathsf{C}}_4[\ell \leftrightarrow \ell_0](\ell_0) = \{v'\}$, $\widehat{\mathsf{C}}_4[\ell \leftrightarrow \ell_0] \vdash v', \rho'$, and $\widehat{\mathsf{C}}_4[\ell \leftrightarrow \ell_0] \models (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell}$, so the case holds.

□

We can now establish the correspondence between analysis and evaluation.

Corollary 1
If $\widehat{\mathsf{C}}$ is the simple closure analysis of a linear program t^{ℓ}, then $\mathscr{E}'[\![t^{\ell}]\!]\emptyset = \langle v, \rho'\rangle$ where $\widehat{\mathsf{C}}(\ell) = \{v\}$ and $\widehat{\mathsf{C}} \vdash v, \rho'$.

By a simple replaying of the proof substituting the containment constraints of 0CFA for the equality constraints of simple closure analysis, it is clear that the same correspondence can be established, and therefore 0CFA and simple closure analysis are identical for linear programs.

Corollary 2
If e is a linear program, then $\widehat{\mathsf{C}}$ is the simple closure analysis of e iff $\widehat{\mathsf{C}}$ is the 0CFA of e.

Discussion: Returning to our earlier question of the computationally potent ingredients in a static analysis, we can now see that when the term is linear, whether flows are directional and bidirectional is irrelevant. For these terms, simple closure analysis, 0CFA, and evaluation are equivalent. And, as we will see, when an analysis is *exact* for linear terms, the analysis will have a PTIME-hardness bound.

2.5 Lower Bounds for Flow Analysis

There are at least two fundamental ways to reduce the complexity of analysis. One is to compute more approximate answers, the other is to analyze a syntactically restricted language.

We use *linearity* as the key ingredient in proving lower bounds on analysis. This shows not only that simple closure analysis and other flow analyses are PTIME-complete, but the result is rather robust in the face of analysis design based on syntactic restrictions. This is because we are able to prove the lower bound via a highly restricted programming language—the linear λ-calculus. So long as the subject language of an analysis includes the linear λ-calculus, and is exact for this subset, the analysis must be at least PTIME-hard.

The decision problem answered by flow analysis, described in **??**, is formulated for monovariant analyses as follows:

Flow Analysis Problem: Given a closed expression e, a term v, and label ℓ, is $v \in \widehat{\mathsf{C}}(\ell)$ in the analysis of e?

Theorem 2
If analysis corresponds to evaluation on linear terms, it is PTIME-hard.

The proof is by reduction from the canonical PTIME-complete problem of circuit evaluation (Ladner, 1975):

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?

An instance of the circuit value problem can be compiled, using only logarithmic space, into an instance of the flow analysis problem. The circuit and its inputs are compiled into a linear λ-term, which simulates C on $\vec{x}$ via *evaluation*—it normalizes to true if C accepts $\vec{x}$ and false otherwise. But since the analysis faithfully captures evaluation of linear terms, and our encoding is linear, the circuit can be simulated by flow analysis.

The encodings work like this: `tt` is the identity on pairs, and `ff` is the swap. Boolean values are either $\langle \mathtt{tt}, \mathtt{ff} \rangle$ or $\langle \mathtt{ff}, \mathtt{tt} \rangle$, where the first component is the "real" value, and the second component is the complement.

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

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

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

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

The And connective is defined as follows:

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

Conjunction works by computing pairs $\langle p_1, p_2\rangle$ and $\langle q_1, q_2\rangle$. The former is the usual conjunction on the first components of the Booleans b_1, b_2: $u_1\langle u_2, \texttt{ff}\rangle$ can be read as "if u_1 then u_2, otherwise false (ff)." The latter is (exploiting De Morgan duality) the disjunction of the complement components of the Booleans: $v_1\langle \texttt{tt}, v_2\rangle$ is read as "if v_1 (i.e. if not u_1) then true (tt), otherwise v_2 (i.e. not u_2)." The result of the computation is equal to $\langle p_1, q_1\rangle$, but this leaves p_2, q_2 unused, which would violate linearity. However, there is symmetry to this *garbage*, which allows for its disposal. Notice that, while we do not know whether p_2 is tt or ff and similarly for q_2, we do know that *one of them is* tt *while the other is* ff. Composing the two together, we are guaranteed that $p_2 \circ q_2 = \texttt{ff}$. Composing this again with another twist (ff) results in the identity function $p_2 \circ q_2 \circ \texttt{ff} = \texttt{tt}$. Finally, composing this with q_1 is just equal to q_1, so $\langle p_1, q_1 \circ p_2 \circ q_2 \circ \texttt{ff}\rangle = \langle p_1, q_1\rangle$, which is the desired result, but the symmetric garbage has been *annihilated*, maintaining linearity.

Similarly, we define truth-table implication:

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

Let us work through the construction once more: 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. 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 De Morgan 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. De Morgan duality ensures that one is tt, and the other is ff (though we do not know which), so they always compose to ff.

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.

This hacking, with its self-annihilating garbage, is an improvement over that given by Mairson (2004) and allows Boolean computation without K-redexes, making the lower bound stronger, but also preserving all flows. In addition, it is the best way to do circuit computation in multiplicative linear logic, and is how you compute similarly in non-affine typed λ-calculus (Mairson, 2006b).

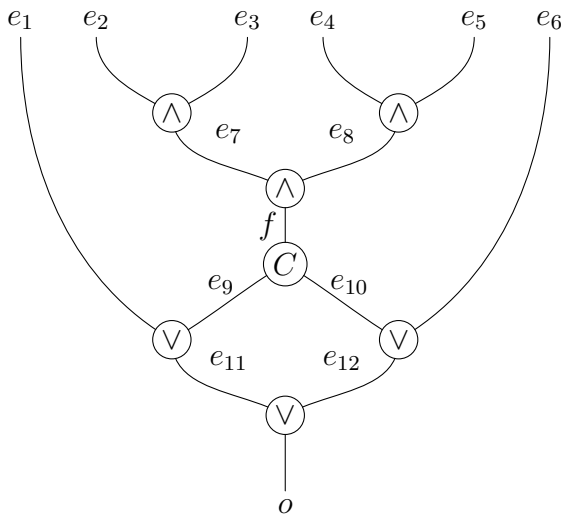


Fig. 6. An example circuit.

By writing continuation-passing style variants of the logic gates, we can encode circuits that look like straight-line code. For example, define CPS logic gates as follows:

$$\begin{aligned}
\texttt{Andgate} &\equiv \lambda b_1.\lambda b_2.\lambda k.k(\texttt{And}\ b_1\ b_2)\\
\texttt{Orgate} &\equiv \lambda b_1.\lambda b_2.\lambda k.k(\texttt{Or}\ b_1\ b_2)\\
\texttt{Impliesgate} &\equiv \lambda b_1.\lambda b_2.\lambda k.k(\texttt{Implies}\ b_1\ b_2)\\
\texttt{Notgate} &\equiv \lambda b.\lambda k.k(\texttt{Not}\ b)\\
\texttt{Copygate} &\equiv \lambda b.\lambda k.k(\texttt{Copy}\ b)
\end{aligned}$$

Continuation-passing style code such as $\texttt{Andgate}\ b_1\ b_2\ (\lambda r.e)$ can be read colloquially as a kind of low-level, straight-line assembly language: "compute the And of registers b_1 and b_2, write the result into register r, and goto e."

An example circuit is given in Figure 6, which can be encoded as:

$$\begin{aligned}
\texttt{Circuit} \equiv\ & \lambda e_1.\lambda e_2.\lambda e_3\lambda e_4.\lambda e_5.\lambda e_6.\\
& \texttt{Andgate}\ e_2\ e_3\ (\lambda e_7.\\
& \texttt{Andgate}\ e_4\ e_5\ (\lambda e_8.\\
& \texttt{Copygate}\ f\ (\lambda e_9.\lambda e_{10}.\\
& \texttt{Orgate}\ e_1\ e_9\ (\lambda e_{11}.\\
& \texttt{Orgate}\ e_{10}\ e_6\ (\lambda e_{12}.\\
& \texttt{Orgate}\ e_{11}\ e_{12}\ (\lambda o.o))))))
\end{aligned}$$

Notice that each variable in this CPS encoding corresponds to a wire in the circuit.

The above code says:

- compute the And of e_2 and e_3, putting the result in register e_7,
- compute the And of e_4 and e_5, putting the result in register e_8,
- compute the And of e_7 and e_8, putting the result in register f,

- make two copies of register f, putting the values in registers e_9 and e_{10},
- compute the `Or` of e_1 and e_9, putting the result in register e_{11},
- compute the `Or` of e_{10} and e_6, putting the result in register e_{12},
- compute the `Or` of e_{11} and e_{12}, putting the result in the o ("output") register.

We know from corollary 1 that evaluation and analysis of linear programs are synonymous, and our encoding of circuits will faithfully simulate a given circuit on its inputs, evaluating to true iff the circuit accepts its inputs. But it does not immediately follow that the circuit value problem can be reduced to the flow analysis problem. Let $||C,\vec{x}||$ be the encoding of the circuit and its inputs. It is tempting to think the instance of the flow analysis problem could be stated:

$$\text{is } \texttt{True} \text{ in } \widehat{\mathsf{C}}(\ell) \text{ in the analysis of } ||C,\vec{x}||^{\ell}?$$

The problem with this is there may be many syntactic instances of "`True`." Since the flow analysis problem must ask about a particular one, this reduction will not work. The fix is to use a context which expects a Boolean expression and induces a particular flow (that can be asked about in the flow analysis problem) iff that expression evaluates 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, `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 `Widget` b. If a false value flows out of b, then $\texttt{False}_W$ flows out of `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 `Widget` b?" without much ado.

$$\begin{aligned}\texttt{Widget} \quad \equiv \quad & \lambda b. \\ & \quad \text{let } \langle u, v\rangle = b \text{ in} \\ & \quad \text{let } \langle x, y\rangle = u\langle f, g\rangle \text{ in} \\ & \quad \text{let } \langle x', y'\rangle = u'\langle f', g'\rangle \text{ in} \\ & \quad\quad \langle\langle xa, yn\rangle, \langle x'a', y'b'\rangle\rangle\end{aligned}$$

Because the circuit value problem is complete for PTIME, we conclude:

Theorem 3
The control flow problem for 0CFA is complete for PTIME.

Corollary 3
The control flow problem for simple closure analysis is complete for PTIME.

2.6 Other Monovariant Analyses

In this section, we survey some of the existing monovariant analyses that either approximate or restrict 0CFA to obtain faster analysis times. In each case, we sketch why these analyses are complete for PTIME.

Shivers (2004) noted in his retrospective on control flow analysis that "in the ensuing years [since 1988], researchers have expended a great deal of effort deriving clever ways to tame the cost of the analysis." Such an effort prompts a fundamental question: to what extent is this possible?

Algorithms to compute 0CFA were long believed to be at least cubic in the size of the program, proving impractical for the analysis of large programs, and Heintze and McAllester (1997c) provided strong evidence to suggest that in general, this could not be improved. They reduced the problem of computing 0CFA to that of deciding two-way nondeterministic push-down automata acceptance (2NPDA); a problem whose best known algorithm was cubic and had remained so since its discovery (Aho *et al.*, 1968)—or so it was believed; see section 8 for a discussion.

In the face of this likely insurmountable bottleneck, researchers derived ways of further approximating 0CFA, thereby giving up information in the service of quickly computing a necessarily less precise analysis in order to avoid the "cubic bottleneck."

Such further approximations enjoy linear or near linear algorithms and have become widely used for the analysis of large programs where the more precise 0CFA would be to expensive to compute. But it is natural to wonder if the algorithms for these simpler analyses could be improved. Owing to 0CFA's PTIME-lower bound, its algorithms are unlikely to be effectively parallelized or made memory efficient. But what about these other analyses?

2.6.1 Ashley and Dybvig's Sub-0CFA

(Ashley & Dybvig, 1998) developed a general framework for specifying and computing flow analyses; instantiations of the framework include 0CFA and the polynomial 1CFA of (Jagannathan & Weeks, 1995), for example. They also developed a class of instantiations, dubbed *sub-0CFA*, that are faster to compute, but less accurate than 0CFA.

This analysis works by explicitly bounding the number of times the cache can be updated for any given program point. After this threshold has been crossed, the cache is updated with a distinguished *unknown* value that represents all possible λ-abstractions in the program. Bounding the number of updates to the cache for any given location effectively bounds the number of passes over the program an analyzer must make, producing an analysis that is $O(n)$ in the size of the program. Empirically, Ashley and Dybvig observe that setting the bound to 1 yields an inexpensive analysis with no significant difference in enabling optimizations with respect to 0CFA.

The idea is the cache gets updated once (n times in general) before giving up and saying all λ-abstractions flow out of this point. But for a linear term, the cache is only updated at most once for each program point. Thus we conclude even when the sub-0CFA bound is 1, the problem is PTIME-complete.

As Ashley and Dybvig note, for any given program, there exists an analysis in the sub-0CFA class that is identical to 0CFA (namely by setting n to the number of passes 0CFA makes over the given program). We can further clarify this relationship by noting that for all linear programs, all analyses in the sub-0CFA class are identical to 0CFA (and thus simple closure analysis).

2.6.2 Subtransitive 0CFA

(Heintze & McAllester, 1997c) have shown the "cubic bottleneck" of computing full 0CFA—that is, computing all the flows in a program—cannot be avoided in general without combi-

natorial breakthroughs: the problem is 2NPDA-hard, for which the "the cubic time decision procedure [...] has not been improved since its discovery in 1968."

Forty years later, that decision procedure was improved to be slightly subcubic by (Chaudhuri, 2008). However, given the strong evidence at the time that the situation was unlikely to improve in general, (Heintze & McAllester, 1997a) identified several simpler flow questions[4] and designed algorithms to answer them for simply-typed programs. Under certain typing conditions, namely that the type is within a bounded size, these algorithms compute in less than cubic time.

The algorithm constructs a graph structure and runs in time linear in a program's graph. The graph, in turn, is bounded by the size of the program's type. Thus, bounding the size of a program's type results in a linear bound on the running times of these algorithms.

If this type bound is removed, though, it is clear that even these simplified flow problems (and their bidirectional-flow analogs), are complete for PTIME: observe that every linear term is simply typable, however in our lower bound construction, the type size is proportional to the size of the circuit being simulated. As they point out, when type size is not bounded, the flow graph may be exponentially larger than the program, in which case the standard cubic algorithm is preferred.

Independently, (Mossin, 1998) developed a type-based analysis that, under the assumption of a constant bound on the size of a program's type, can answer restricted flow questions such as single source/use in linear time with respect to the size of the explicitly typed program. But again, removing this imposed bound results in PTIME-completeness.

As (Hankin *et al.*, 2002) point out: both Heintze and McAllester's and Mossin's algorithms operate on type structure (or structure isomorphic to type structure), but with either implicit or explicit η-expansion. For simply-typed terms, this can result in an exponential blow-up in type size. It is not surprising then, that given a much richer graph structure, the analysis can be computed quickly.

In this light, the results of section 3 on 0CFA of η-expanded, simply-typed programs can be seen as an improvement of the subtransitive flow analysis since it works equally well for languages with first-class control and can be performed with only a fixed number of pointers into the program structure, i.e. it is computable in LOGSPACE (and in other words, PTIME = LOGSPACE up to η).

2.7 Conclusions

When an analysis is *exact*, it will be possible to establish a correspondence with evaluation. The richer the language for which analysis is exact, the harder it will be to compute the analysis. As an example in the extreme, (Mossin, 1997a) developed a flow analysis that is exact for simply-typed terms. The computational resources that may be expended to compute this analysis are *ipso facto* not bounded by any elementary recursive function (Statman, 1979). However, most flow analyses do not approach this kind of expressivity. By way of comparison, 0CFA only captures PTIME, and yet researchers have still expending a great deal of effort deriving approximations to 0CFA that are faster to compute. But

[4] Including the decision problem discussed in this dissertation, which is the simplest; answers to any of the other questions imply an answer to this problem

as we have shown for a number of them, they all coincide on linear terms, and so they too capture PTIME.

We should be clear about what is being said, and not said. There is a considerable difference in practice between linear algorithms (nominally considered efficient) and cubic—or near cubic—algorithms (still feasible, but taxing for large inputs), even though both are polynomial-time. PTIME-completeness does not distinguish the two. But if a sub-polynomial (e.g., LOGSPACE) algorithm was found for this sort of flow analysis, it would depend on (or lead to) things we do not know (LOGSPACE = PTIME).

Likewise, were a parallel implementation of this flow analysis to run in logarithmic time (i.e., NC), we would consequently be able to parallelize every polynomial time algorithm. PTIME-complete problems are considered to be the least likely to be in NC. This is because logarithmic-space reductions (such as our compiler from circuits to λ-terms) preserve parallel complexity, and so by composing this reduction with a (hypothetical) logarithmic-time 0CFA analyzer (or equivalently, a logarithmic-time linear λ-calculus evaluator) would yield a fast parallel algorithm for *all* problems in PTIME, which are by definition, logspace-reducible to the circuit value problem [page 377] (Papadimitriou, 1994).

The practical consequences of the PTIME-hardness result is that we can conclude any analysis which is exact for linear programs, which includes 0CFA, and many further approximations, does not have a fast parallel algorithm unless PTIME = NC.

3 Linear Logic and Static Analysis

If you want to understand exactly how and where static analysis is computationally difficult, you need to know about linearity. In this chapter, we develop an alternative, graphical representation of programs that makes explicit both non-linearity and control, and is suitable for static analysis.

This alternative representation offers the following benefits:

- It provides clear intuitions on the essence of 0CFA and forms the basis for a transparent proof of the correspondence between 0CFA and evaluation for linear programs.
- As a consequence of symmetries in the notation, it is equally well-suited for representing programs with first-class control.
- It based on the technology of linear logic. Insights gleaned from linear logic, viewed through the lens of a Curry-Howard correspondence, can inform program analysis and *vice versa*.
- As an application of the above, a novel and efficient algorithm for analyzing typed programs (subsection 3.4) is derived from recent results on the efficient normalization of linear logic proofs.

We give a reformulation of 0CFA in this setting and then transparently *reprove* the main result of subsection 2.4: analysis and evaluation are synonymous for linear programs.

3.1 Sharing Graphs for Static Analysis

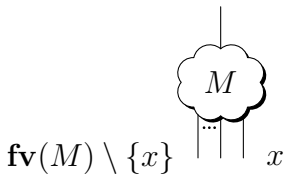
In general, the sharing graph of a term will consist of a distinguished *root* wire from which the rest of the term's graph "hangs."

At the bottom of the graph, the dangling wires represent free variables and connect to occurrences of the free variable within in term.

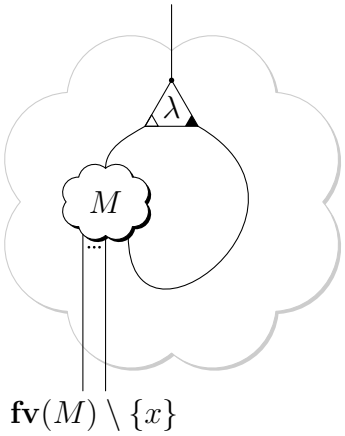
Graphs consist of ternary abstraction (λ), apply (@), sharing ($\triangledown$) nodes, and unary weakening ($\odot$) nodes. Each node has a distinguished *principal* port. For unary nodes, this is the only port. The ternary nodes have two *auxiliary* ports, distinguished as the *white* and *black* ports.

- A variable occurrence is represented simply as a wire from the root to the free occurrence of the variable.

- Given the graph for M, where x occurs free,

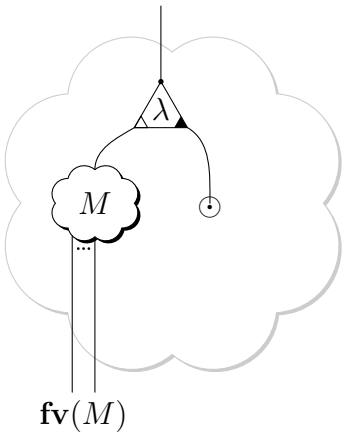


the abstraction $\lambda x.M$ is formed as,



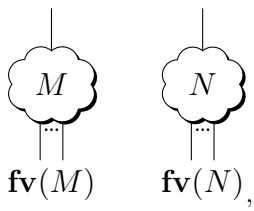
$\lambda x.M, x \in \mathbf{fv}(M)$

Supposing x does not occur in M, the weakening node ($\odot$) is used to "plug" the λ variable wire.



$\lambda x.M, x \notin \mathbf{fv}(M)$

- Given graphs for M and N,



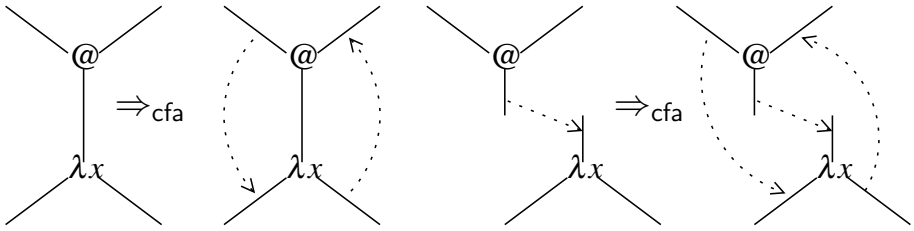
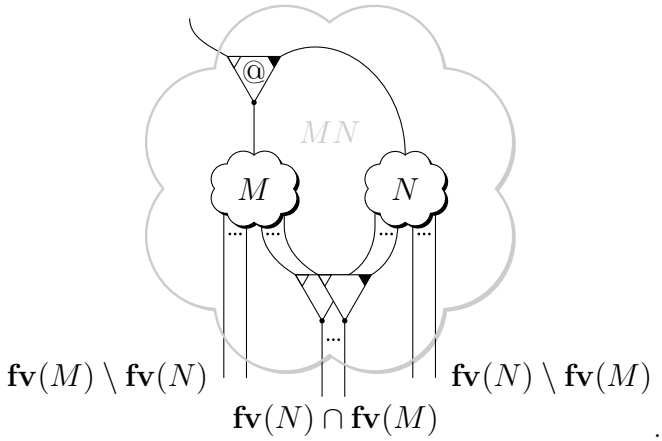


Fig. 7. CFA virtual wire propagation rules.

the application *MN* is formed as,



An application node is introduced. The operator *M* is connected to the function port and the operand *N* is connected to the argument port. The continuation wire becomes the root wire for the application. Free variables shared between both *M* and *N* are fanned out with sharing nodes.

3.2 Graphical 0CFA

We now describe an algorithm for performing control flow analysis that is based on the graph coding of terms. The graphical formulation consists of generating a set of *virtual paths* for a program graph. Virtual paths describe an approximation of the real paths that will arise during program execution.

Figure 7 defines the virtual path propagation rules. Note that a wire can be identified by its label or a variable name.[5] The left hand rule states that a virtual wire is added from the continuation wire to the body wire and from the variable wire to the argument wire of each β-redex. The right hand rule states analogous wires are added to each *virtual* β*-redex*—an apply and lambda node connected by a virtual path. There is a *virtual path* between two wires ℓ and ℓ', written $\ell \rightsquigarrow \ell'$ in a CFA-graph iff:

1. $\ell \equiv \ell'$.

[5] We implicitly let ℓ range over both in the following definitions.

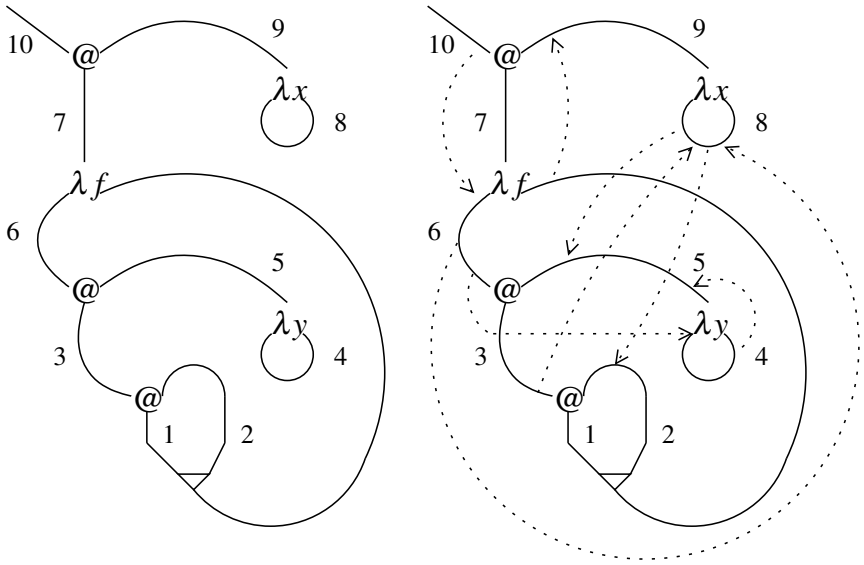


Fig. 8. Graph coding and CFA graph of $(\lambda f.ff(\lambda y.y))(\lambda x.x)$.

2. There is a virtual wire from ℓ to ℓ'.
3. ℓ connects to an auxiliary port and ℓ' connects to the root port of a sharing node.
4. There is a virtual path from ℓ to ℓ'' and from ℓ'' and ℓ'.

Reachability: Some care must be taken to ensure leastness when propagating virtual wires. In particular, wires are added only when there is a virtual path between a *reachable* apply and a lambda node. An apply node is reachable if it is on the spine of the program, i.e., if $e = (\cdots((e_0 e_1)^{\ell_1} e_2)^{\ell_2} \cdots e_n)^{\ell_n}$ then the apply nodes with continuation wires labeled $\ell_1, \ldots, \ell_n$ are reachable, or it is on the spine of an expression with a virtual path from a reachable apply node.

Reachability is usually explained as a known improvement to flow analysis; precision is increased by avoiding parts of the program that cannot be reached (Ayers, 1993; Palsberg & Schwartzbach, 1995; Biswas, 1997; Heintze & McAllester, 1997b; Midtgaard & Jensen, 2008; Midtgaard & Jensen, 2009).

But reachability can also be understood as an analysis analog to weak normalization. Reachability says roughly: "don't analyze under λ until the analysis determines it may be applied." On the other hand, weak normalization says: "don't evaluate under λ until the evaluator determines it is applied." The analyzers of **??** implicitly include reachability since they are based on a evaluation function that performs weak normalization.

The graph-based analysis can now be performed in the following way: construct the CFA graph according to the rules in Figure 7, then define $\widehat{\mathsf{C}}(\ell)$ as $\{(\lambda x.e)^{\ell'} \mid \ell \rightsquigarrow \ell'\}$ and $\hat{\mathsf{r}}(x)$ as $\{(\lambda x.e)^{\ell} \mid x \rightsquigarrow \ell\}$. It is easy to see that the algorithm constructs answers that satisfy the acceptability relation specifying the analysis. Moreover, this algorithm constructs least solutions according to the partial order given in **??**.

Lemma 5
$\widehat{\mathsf{C}}', \hat{\mathsf{r}}' \models e$ implies $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \sqsubseteq \widehat{\mathsf{C}}', \hat{\mathsf{r}}'$ for $\widehat{\mathsf{C}}, \hat{\mathsf{r}}$ constructed for e as described above.

We now consider an example of use of the algorithm. Consider the labeled program:

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

Figure 8 shows the graph coding of the program and the corresponding CFA graph. The CFA graph is constructed by adding virtual wires $10 \rightsquigarrow 6$ and $f \rightsquigarrow 9$, induced by the actual β-redex on wire 7. Adding the virtual path $f \rightsquigarrow 9$ to the graph creates a virtual β-redex via the route $1 \rightsquigarrow f$ (through the sharing node), and $f \rightsquigarrow 9$ (through the virtual wire). This induces $3 \rightsquigarrow 8$ and $8 \rightsquigarrow 2$. There is now a virtual β-redex via $3 \rightsquigarrow 8 \rightsquigarrow 2 \rightsquigarrow f \rightsquigarrow 9$, so wires $6 \rightsquigarrow 8$ and $8 \rightsquigarrow 5$ are added. This addition creates another virtual redex via $3 \rightsquigarrow 8 \rightsquigarrow 2 \rightsquigarrow 5$, which induces virtual wires $6 \rightsquigarrow 4$ and $4 \rightsquigarrow 5$. No further wires can be added, so the CFA graph is complete. The resulting abstract cache gives:

$$\begin{array}{llll}
\widehat{\mathsf{C}}(1) = \{\lambda x\} & \widehat{\mathsf{C}}(6) & = \{\lambda x, \lambda y\} & \\
\widehat{\mathsf{C}}(2) = \{\lambda x\} & \widehat{\mathsf{C}}(7) & = \{\lambda f\} & \hat{\mathsf{r}}(f) = \{\lambda x\} \\
\widehat{\mathsf{C}}(3) = \{\lambda x, \lambda y\} & \widehat{\mathsf{C}}(8) & = \{\lambda x, \lambda y\} & \hat{\mathsf{r}}(x) = \{\lambda x, \lambda y\} \\
\widehat{\mathsf{C}}(4) = \{\lambda y\} & \widehat{\mathsf{C}}(9) & = \{\lambda x\} & \hat{\mathsf{r}}(y) = \{\lambda y\} \\
\widehat{\mathsf{C}}(5) = \{\lambda y\} & \widehat{\mathsf{C}}(10) & = \{\lambda x, \lambda y\} &
\end{array}$$

3.3 Multiplicative Linear Logic

The Curry-Howard isomorphism states a correspondence between logical systems and computational calculi (Howard, 1980). The fundamental idea is that data types are theorems and typed programs are proofs of theorems.

> *It begins with the observation that an implication $A \to B$ corresponds to a type of functions from A to B, because inferring B from $A \to B$ and A can be seen as* applying *the first assumption to the second one—just like a function from A to B applied to an element of A yields an element of B. [p. v] (Sørensen & Urzyczyn, 2006)*

For the functional programmer, the most immediate correspondence is between proofs in propositional intuitionistic logic and simply typed λ-terms. But the correspondence extends considerably further.

> *Virtually all proof-related concepts can be interpreted in terms of computations, and virtually all syntactic features of various lambda-calculi and similar systems can be formulated in the language of proof theory.*

In this section we want to develop the "proofs-as-programs" correspondence for linear programs, an important class of programs to consider for lower bounds on program analysis. Because analysis and evaluation are synonymous for linear programs, insights from proof evaluation can guide new algorithms for program analysis.

The correspondence between simply typed (nonlinear) terms and intuitionistic logic can be seen by looking at the familiar typing rules:

$$\textsc{Var}\ \frac{}{\Gamma, x : A \vdash x : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

If you ignore the "proof terms" (i.e. the programs), you get intuitionsitic sequent calculus:

$$\text{Ax}\ \frac{}{\Gamma, A \vdash A} \qquad \to\text{I}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \to\text{E}\ \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

Likewise, *linear programs* have their own logical avatar, namely *multiplicative linear logic*.

3.3.1 Proofs

Each atomic formula is given in two forms: positive (A) and negative ($A^\perp$) and the *linear negation* of A is $A^\perp$ and *vice versa*. Negation is extended to compound formulae via De Morgan laws:

$$(A \otimes B)^\perp = A^\perp \parr B^\perp \qquad (A \parr B)^\perp = A^\perp \otimes B^\perp$$

A two sided sequent

$$A_1, \ldots, A_n \vdash B_1, \ldots, B_m$$

is replaced by

$$\vdash A_1^\perp, \ldots, A_n^\perp, B_1, \ldots, B_m$$

The interested reader is referred to (Girard, 1987) for more details on linear logic.

For each derivation in MLL, there is a proofnet, which abstracts away much of the needless sequentialization of sequent derivations, "like the order of application of independent logical rules: for example, there are many inessintailly different ways to obtain $\vdash A_1 \parr A_2, \ldots A_{n-1} \parr A_n$ from $\vdash A_1, \ldots A_n$, while there is only one proof net representing all these derivations" (Di Cosmo *et al.*, 2003). There is strong connection with calculus of explicit substitutions (Di Cosmo *et al.*, 2003).

The sequent rules of multiplicative linear logic (MLL) are given in Figure 9.

$$\text{Ax}\ \frac{}{A, A^\perp} \qquad \text{Cut}\ \frac{\Gamma, A \quad A^\perp, \Delta}{\Gamma, \Delta} \qquad \parr\ \frac{\Gamma, A, B}{\Gamma, A \parr B} \qquad \otimes\ \frac{\Gamma, A \quad \Delta, B}{\Gamma, \Delta, A \otimes B}$$

Fig. 9. MLL sequent rules.

3.3.2 Programs

These rules have an easy functional programming interpretation as the types of a linear programming language (eg. linear ML), following the intuitions of the Curry-Howard correspondence (Girard *et al.*, 1989; Sørensen & Urzyczyn, 2006).[6]

[6] For a more detailed discussion of the C.-H. correspondence between linear ML and MLL, see (Mairson, 2004).

(These are written in the more conventional (to functional programmers) two-sided sequents, but just remember that $A^\perp$ on the left is like A on the right).

$$\frac{}{x:A \vdash x:A} \qquad \frac{\Gamma \vdash M:A \quad \Delta \vdash N:B}{\Gamma,\Delta \vdash (M,N):A \otimes B} \qquad \frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x.M : A \multimap B}$$

$$\frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma,\Delta \vdash MN : B} \qquad \frac{\Gamma \vdash M : A \otimes B \quad \Delta, x:A, y:B \vdash N : C}{\Gamma,\Delta \vdash \text{let } \langle x,y\rangle = M \text{ in } N : C}$$

The AXIOM rule says that a variable can be viewed simultaneously as a continuation ($A^\perp$) or as an expression (A)—one man's ceiling is another man's floor. Thus we say "input of type A" and "output of type $A^\perp$" interchangeably, along with similar dualisms. We also regard $(A^\perp)^\perp$ synonymous with A: for example, `Int` is an integer, and $\texttt{Int}^\perp$ is a request (need) for an integer, and if you need to need an integer—$(\texttt{Int}^\perp)^\perp$—then you have an integer.

The CUT rule says that if you have two computations, one with an output of type A, another with an input of type A, you can plug them together.

The $\otimes$-rule is about pairing: it says that if you have separate computations producing outputs of types A and B respectively, you can combine the computations to produce a paired output of type $A \otimes B$. Alternatively, given two computations with A an output in one, and B an input (equivalently, continuation $B^\perp$ an output) in the other, they get paired as a *call site* "waiting" for a function which produces an *output* of type B with an *input* of type A. Thus $\otimes$ is both `cons` and function call (@).

The $\parr$-rule is the linear unpairing of this $\otimes$-formation. When a computation uses inputs of types A and B, these can be combined as a single input pair, e.g., `let (x,y)=p in...`. Alternatively, when a computation has an input of type A (output of continuation of type $A^\perp$) and an output of type B, these can be combined to construct a function which inputs a call site pair, and unpairs them appropriately. Thus $\parr$ is both unpairing and λ.

3.4 η-Expansion and LOGSPACE

3.4.1 Atomic versus Non-Atomic Axioms

The above AXIOM rule does not make clear whether the formula A is an atomic type variable or a more complex type formula. When a *linear* program only has atomic formulas in the "axiom" position, then we can evaluate (normalize) it in logarithmic space. When the program is not linear, we can similarly compute a 0CFA analysis in LOGSPACE. Moreover, these problems are complete for LOGSPACE.

MLL proofs with non-atomic axioms can be easily converted to ones with atomic axioms using the following transformation, analogous to η-expansion:

$$\frac{}{\alpha \otimes \beta, \alpha^\perp \parr \beta^\perp} \quad \Rightarrow \quad \frac{\dfrac{\dfrac{}{\alpha,\alpha^\perp} \quad \dfrac{}{\beta,\beta^\perp}}{\alpha \otimes \beta, \alpha^\perp, \beta^\perp}}{\alpha \otimes \beta, \alpha^\perp \parr \beta^\perp}$$

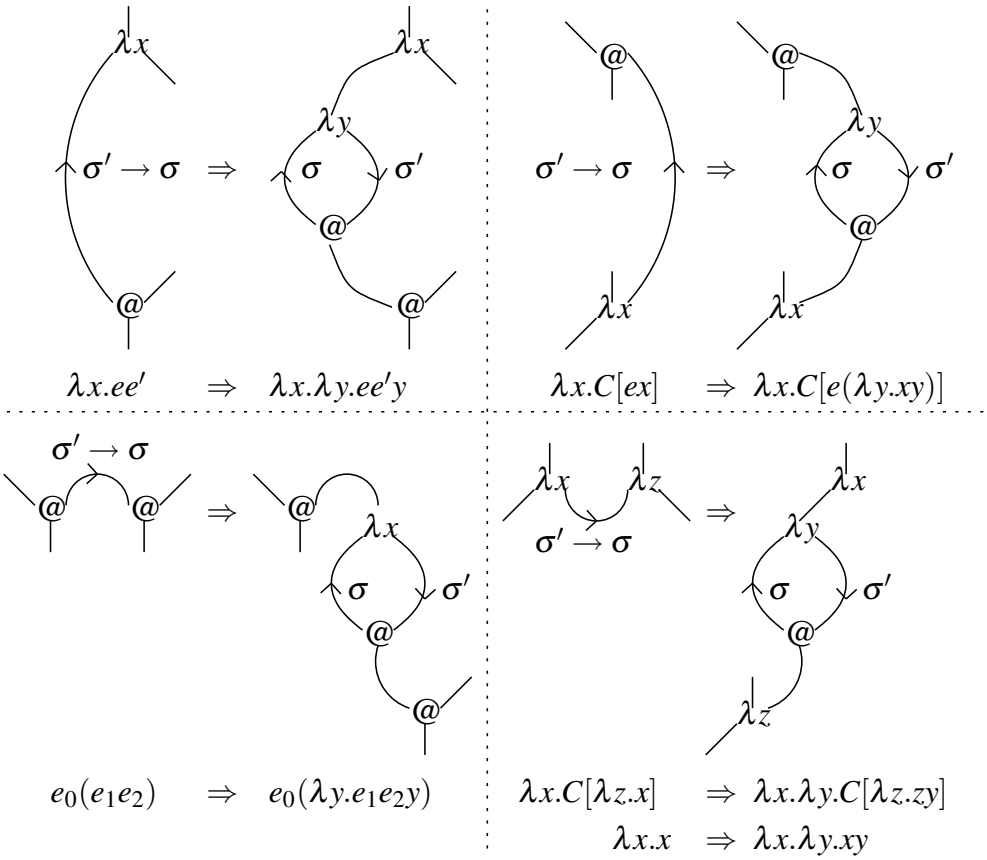


Fig. 10. Expansion algorithm.

This transformation can increase the size of the proof. For example, in the circuit examples of the previous section (which are evidence for PTIME-completeness), η-expansion causes an exponential increase in the number of proof rules used.[7] A LOGSPACE evaluation is then polynomial-time and -space in the original circuit description.

The program transformation corresponding to the above proof expansion is a version of η-expansion: see Figure 10. The left hand expansion rule is simply η, dualized in the unusual right hand rule. The right rule is written with the @ above the λ only to emphasis its duality with the left rule. Although not shown in the graphs, but implied by the term rewriting rules, an axiom may pass through any number of sharing nodes.

3.4.2 Proof Normalization with Non-Atomic Axioms: PTIME

A normalized *linear* program has no redexes. From the type of the program, one can reconstruct—in a totally syntax-directed way—what the structure of the term is (Mairson, 2004). It is only the position of the *axioms* that is not revealed. For example, both `TT` and `FF`

[7] It is linear in the formulas used, whose length increases exponentially (not so if the formulas are represented by directed acyclic graphs).

from the above circuit example have type `'a * 'a -> 'a * 'a`.[8] From this type, we can see that the term is a λ-abstraction, the parameter is unpaired—and then, are the two components of type `a` repaired as before, or "twisted"? To twist or not to twist is what distinguishes `TT` from `FF`.

An MLL *proofnet* is a graphical analogue of an MLL proof, where various sequentialization in the proof is ignored. The proofnet consists of axiom, cut, $\otimes$, and $\parr$ nodes with various dangling edges corresponding to conclusions. Rules for proofnet formation (Figure 11) follow the rules for sequent formation (Figure 9) almost identically.

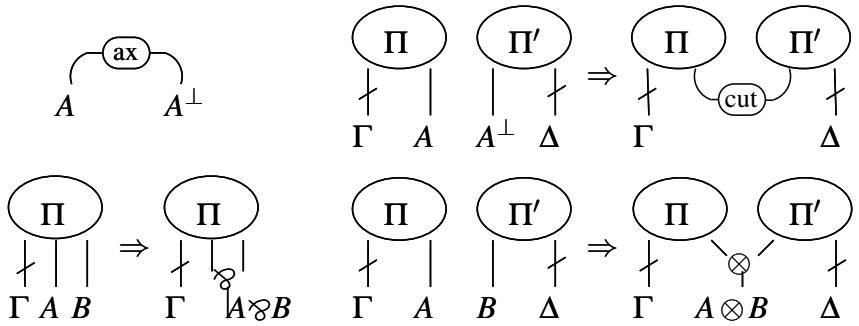


Fig. 11. MLL proofnets.

A binary axiom node has two dangling edges, typed A and $A^\perp$. Given two disjoint proofnets with dangling edges (conclusions) typed Γ, A and $A^\perp, \Delta$, the edges typed $A, A^\perp$ can be connected to a binary cut node, and the resulting connected proofnet has dangling edges typed Γ, Δ. Given a connected proofnet with dangling wires typed Γ, A, B, the edges typed A, B can be connected to the two auxiliary port of a $\parr$ node and the dangling edge connected to the principal port will have type $A \parr B$. Finally, given two disjoint proofnets with dangling edges typed Γ, A and Δ, B, the edges typed A, B can be connected to the two auxiliary ports of a ternary $\otimes$ node; the principal port then has a dangling wire of type $A \otimes B$. The intuition is that $\otimes$ is pairing and $\parr$ is linear unpairing.

The geometry of interaction (Girard, 1989; Gonthier *et al.*, 1992)—the semantics of linear logic—and the notion of paths provide a way to calculate normal forms, and may be viewed as the logician's way of talking about static program analysis.[9] To understand how this analysis works, we need to have a graphical picture of what a linear functional program looks like.

Without loss of generality, such a program has a type ϕ. Nodes in its graphical picture are either λ or linear unpairing ($\parr$ in MLL), or application/call site or linear pairing ($\otimes$ in MLL). We draw the graphical picture so that axioms are on top, and cuts (redexes, either β-redexes or pair-unpair redexes) are on the bottom as shown in Figure 12.

Because the axioms all have atomic type, the graph has the following nice property:

[8] The linear logic equivalent is $(\alpha^\perp \parr \alpha^\perp) \parr (\alpha \otimes \alpha)$. The λ is represented by the outer $\parr$, the unpairing by the inner $\parr$, and the `cons`ing by the $\otimes$.

[9] See (Mairson, 2002) for an introduction to context semantics and normalization by static analysis in the geometry of interaction.

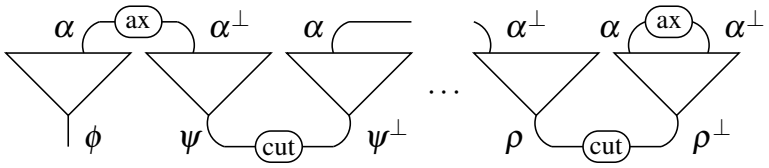


Fig. 12. MLL proofnet with atomic axioms.

Lemma 6
Begin at an axiom α and "descend" to a cut-link, saving in an (initially empty) stack whether nodes are encountered on their left or right auxiliary port. Once a cut is reached, "ascend" the accompanying structure, popping the stack and continuing left or right as specified by the stack token. Then (1) the stack empties exactly when the next axiom α' is reached, and (2) if the k-th node from the start traversed is a $\otimes$, the k-th node from the end traversed is a $\parr$, and vice versa.

The path traced in the Lemma, using the stack, is geometry of interaction (GoI), also known as static analysis. The correspondence between the k-th node from the start and end of the traversal is precisely that between a *call site* ($\otimes$) and a *called function* ($\parr$), or between a `cons` ($\otimes$) and a linear unpairing ($\parr$).

3.4.3 Proof Normalization with Atomic Axioms: LOGSPACE

A sketch of the "four finger" normalization algorithm: The stack height may be polynomial, but we do not need the stack! Put fingers α, β on the axiom where the path begins, and iterate over all possible choices of another two fingers α', β' at another axiom. Now move β and β' towards the cut link, where if β encounters a node on the left (right), then β' must move left (right) also. If α', β' were correctly placed initially, then when β arrives at the cut link, it must be met by β'. If β' isn't there, or got stuck somehow, then α', β' were incorrectly placed, and we iterate to another placement and try again.

Lemma 7
Any path from axiom α to axiom α' traced by the stack algorithm of the previous lemma is also traversed by the "four finger" normalization algorithm.

Normalization by static analysis is synonymous with traversing these paths. Because these fingers can be stored in logarithmic space, we conclude (Terui, 2002; Mairson, 2006a; Mairson, 2006b):

Theorem 4
Normalization of linear, simply-typed, and fully η-expanded functional programs is contained in LOGSPACE.

That 0CFA is then contained in LOGSPACE is a casual byproduct of this theorem, due to the following observation: if application site χ calls function ϕ, then the $\otimes$ and $\parr$ (synonymously, @ and λ) denoting call site and function are in distinct trees connected by a CUT link. As a consequence the 0CFA computation is a subcase of the four-finger algorithm: traverse the two paths from the nodes to the cut link, checking that the paths are

isomorphic, as described above. The full 0CFA calculation then iterates over all such pairs of nodes.

Corollary 4
0CFA of linear, simply-typed, and fully η-expanded functional programs is contained in LOGSPACE.

3.4.4 0CFA in LOGSPACE

Now let us remove the linearity constraint, while continuing to insist on full η-expansion as described above, and simple typing. The normalization problem is no longer contained in LOGSPACE, but rather non-elementary recursive, (Statman, 1979; Mairson, 1992b; Asperti & Mairson, 1998). However, 0CFA remains contained in LOGSPACE, because it is now an *approximation*. This result follows from the following observation:

Lemma 8
Suppose $(t^{\ell}\ e)$ occurs in a simply typed, fully η-expanded program and $\lambda x.e \in \widehat{\mathsf{C}}(\ell)$. Then the corresponding $\otimes$ and $\parr$ occur in adjacent trees connected at their roots by a CUT-link and on dual, isomorphic paths modulo placement of sharing nodes.

Here "modulo placement" means: follow the paths to the cut—then we encounter $\otimes$ (resp., $\parr$) on one path when we encounter $\parr$ (resp., $\otimes$) on the other, on the same (left, right) auxiliary ports. We thus *ignore* traversal of sharing nodes on each path in judging whether the paths are isomorphic. (Without sharing nodes, the $\otimes$ and $\parr$ would annihilate—i.e., a β-redex—during normalization.)

Theorem 5
0CFA of a simply-typed, fully η-expanded program is contained in LOGSPACE.

Observe that 0CFA defines an *approximate* form of normalization which is suggested by simply *ignoring* where sharing occurs. Thus we may define the *set* of λ-terms to which that a term might evaluate. Call this *0CFA-normalization*.

Theorem 6
For fully η-expanded, simply-typed terms, 0CFA-normalization can be computed in *nondeterministic* LOGSPACE.

Conjecture 1
For fully η-expanded, simply-typed terms, 0CFA-normalization is complete for *nondeterministic* LOGSPACE.

The proof of the above conjecture likely depends on a coding of arbitrary directed graphs and the consideration of commensurate path problems.

Conjecture 2
An algorithm for 0CFA normalization can be realized by *optimal reduction*, where sharing nodes *always* duplicate, and never annihilate.

3.4.5 LOGSPACE-hardness of Normalization and 0CFA: linear, simply-typed, fully η-expanded programs

That the normalization and 0CFA problem for this class of programs is as hard as any LOGSPACE problem follows from the LOGSPACE-hardness of the *permutation problem*: given a permutation π on $1,\ldots,n$ and integer $1 \leq i \leq n$, are 1 and i on the same cycle in π? That is, is there a k where $1 \leq k \leq n$ and $\pi^k(1) = i$?

Briefly, the LOGSPACE-hardness of the permutation problem is as follows.[10] Given an arbitrary LOGSPACE Turing machine M and an input x to it, visualize a graph where the nodes are machine IDs, with directed edges connecting successive configurations. Assume that M always accepts or rejects in unique configurations. Then the graph has two connected components: the "accept" component, and the "reject" component. Each component is a directed tree with edges pointing towards the root (final configuration). Take an Euler tour around each component (like tracing the fingers on your hand) to derive two *cycles*, and thus a *permutation* on machine IDs. Each cycle is polynomial size, because the configurations only take logarithmic space. The equivalent permutation problem is then: does the initial configuration and the accept configuration sit on the same cycle?

The following linear ML code describes the "target" code of a transformation of an instance of the permutation problem. For a permutation on n letters, we take here an example where $n = 3$. Begin with a vector of length n set to `False`, and a permutation on n letters:

```
- val V= (False,False,False);
val V = ((fn,fn),(fn,fn),(fn,fn))
  : (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
```

Denote as ν the type of vector `V`.

```
- fun Perm (P,Q,R)= (Q,R,P);
val Perm = fn : ν -> ν
```

The function `Insert` *linearly* inserts `True` in the first vector component, using all input exactly once:

```
- fun Insert ((p,p'),Q,R)= ((TT,Compose(p,p')),Q,R);
val Insert = fn : ν -> ν
```

The function `Select` *linearly* selects the third vector component:

```
- fun Select (P,Q,(r,r'))=
    (Compose (r,Compose (Compose P, Compose Q)),r');
val Select = fn
  : ν -> (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
```

Because `Perm` and `Insert` have the same flat type, they can be composed iteratively in ML without changing the type. (This clearly is *not* true in our coding of circuits, where the size of the type increases with the circuit. A careful coding limits the type size to be polynomial in the circuit size, regardless of circuit depth.)

[10] This presentation closely follows (Mairson, 2006b).

Fig. 13. Graph coding of `call/cc` and example CFA graph.

Lemma 9
Let π be coded as permutation `Perm`. Define `Foo` to be

```
Compose(Insert,Perm)
```

composed with itself n times. Then 1 and i are on the same cycle of π iff `Select (Foo V)` normalizes to `True`.

Because 0CFA of a linear program is identical with normalization, we conclude:

Theorem 7
0CFA of a simply-typed, fully η-expanded program is complete for LOGSPACE.

The usefulness of η-expansion has been noted in the context of partial evaluation (Jones *et al.*, 1993; Danvy *et al.*, 1996). In that setting, η-redexes serve to syntactically embed binding-time coercions. In our case, the type-based η-expansion does the trick of placing the analysis in LOGSPACE by embedding the type structure into the syntax of the program.[11]

3.5 Graphical Flow Analysis and Control

(Shivers, 2004) argues that "CPS provide[s] a uniform representation of control structure," allowing "this machinery to be employed to reason about context, as well," and that "without CPS, separate contextual analyses and transforms must be also implemented—redundantly," in his view. Although our formulation of flow analysis is a "direct-style" formulation, a graph representation enjoys the same benefits of a CPS representation, namely that control structures are made explicit—in a graph a continuation is simply a wire. Control constructs such as `call/cc` can be expressed directly (Lawall & Mairson, 2000) and our graphical formulation of control flow analysis carries over without modification.

(Lawall & Mairson, 2000) derive graph representations of programs with control operators such as `call/cc` by first translating programs into continuation passing style (CPS). They observed that when edges in the CPS graphs carrying answer values (of type $\perp$) are eliminated, the original (direct-style) graph is regained, modulo placement of boxes and croissants that control sharing. Composing the two transformations results in a direct-style graph coding for languages with `call/cc` (hereafter, $\lambda_{\mathscr{K}}$). The approach applies equally well to languages such as Filinski's symmetric λ-calculus (1989), Parigot's λ_μ calculus (1992), and most any language expressible in CPS.

Languages such as λ_ξ, which contains the "delimited control" operators *shift* and *reset* (Danvy & Filinski, 1990), are not immediately amenable to this approach since the direct-style transformation requires all calls to functions or continuations be in tail position. Adapting this approach to such languages constitutes an open area of research.

The left side of Figure 13 shows the graph coding of `call/cc`. Examining this graph, we can read of an interpretation of `call/cc`, namely: `call/cc` is a function that when applied, copies the current continuation ($\triangle$) and applies the given function f to a function ($\lambda v \ldots$)

[11] Or, in slogan form: LOGSPACE = PTIME upto η.

that when applied abandons the continuation at that point ($\odot$) and gives its argument v to a copy of the continuation where `call/cc` was applied. If f never applies the function it is given, then control returns "normally" and the value f returns is given to the other copy of the continuation where `call/cc` was applied.

The right side of Figure 13 gives the CFA graph for the program:

$$(\texttt{call/cc}\ \ (\lambda k.(\lambda x.\overline{1})(k\overline{2})))^{\ell}$$

From the CFA graph we see that $\widehat{\mathsf{C}}(\ell) = \{\overline{1}, \overline{2}\}$, reflecting the fact that the program will return $\overline{1}$ under a call-by-name reduction strategy and $\overline{2}$ under call-by-value. Thus, the analysis is indifferent to the reduction strategy. Note that whereas before, approximation was introduced through nonlinearity of bound variables, approximation can now be introduced via nonlinear use of continuations, as seen in the example. In the same way that 0CFA considers all occurrences of a bound variable "the same", 0CFA considers all continuations obtained with each instance of `call/cc` "the same".

Note that we can ask new kinds of interesting questions in this analysis. For example, in Figure 13, we can compute which continuations are potentially *discarded*, by computing which continuations flow into the weakening node of the `call/cc` term. (The answer is the continuation $((\lambda x.\overline{1})[\,])$.) Likewise, it is possible to ask which continuations are potentially *copied*, by computing which continuations flow into the principal port of the sharing node in the `call/cc` term (in this case, the top-level empty continuation $[\,]$). Because continuations are used linearly in `call/cc`-free programs, the questions were uninteresting before—the answer is always *none*.

Our proofs for the PTIME-completeness of 0CFA for the untyped λ-calculus carry over without modification languages such as $\lambda_{\mathscr{K}}$, λ_{μ} and the symmetric λ-calculus. In other words, first-class control operators such as `call/cc` increase the expressivity of the language, but add nothing to the computational complexity of control flow analysis. In the case of simply-typed, fully η-expanded programs, the same can be said. A suitable notion of "simply-typed" programs is needed, such as that provided by (Griffin, 1990) for $\lambda_{\mathscr{K}}$. The type-based expansion algorithm of Figure 10 applies without modification and lemma 8 holds, allowing 0CFA for this class of programs to be done in LOGSPACE. Linear logic provides a foundation for (classical) λ-calculi with control; related logical insights allow control flow analysis in this setting.

The graph coding of terms in our development is based on the technology of *sharing graphs* in the untyped case, and *proof nets* in the typed case (Lafont, 1995). The technology of proofnets have previously been extended to intersection types (Regnier, 1992; Møller Neergaard, 2004), which have a close connection to flow analysis (Amtoft & Turbak, 2000; Palsberg & Pavlopoulou, 2001; Wells *et al.*, 2002; Banerjee & Jensen, 2003).

The graph codings, CFA graphs, and virtual wire propagation rules share a strong resemblance to the "pre-flow" graphs, flow graphs, and graph "closing rules", respectively, of (Mossin, 1997b). Casting the analysis in this light leads to insights from linear logic and optimal reduction. For example, as [page 78] (Mossin, 1997b) notes, the CFA virtual paths computed by 0CFA are an approximation of the actual run-time paths and correspond exactly to the "well-balanced paths" of (Asperti & Laneve, 1995) as an approximation to "legal paths" (Lévy, 1978) and results on proof normalization in linear logic (Mairson & Terui, 2003) informed the novel flow analysis algorithms presented here.

4 *k*CFA and EXPTIME

In this chapter, we give an exact characterization of the computational complexity of the *k*CFA 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.

4.1 Shivers' kCFA

As noted in subsection 1.1, practical flow analyses must negotiate a compromise between complexity and precision, 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 (section 2).

As described in section 2, a monovariant analysis is one that approximates at points of nonlinearity. When a variable appears multiple times, flow information is merged together for all sites.

So for example, in analyzing the program from subsection 2.2,

$$(\lambda f.(f f)(\lambda y.y))(\lambda x.x),$$

a monovariant analysis such as 0CFA or simple closure analysis will merge the flow information for the two occurrences of f. Consequently both $\lambda x.x$ and $\lambda y.y$ are deemed to flow out of the whole expression.

More precise analyses can be obtained by incorporating context-sensitivity to distinguish multiple closures over the same λ-term, resulting in "finer grained approximations, expending more work to gain more information" (Shivers, 1988; Shivers, 1991). This context-sensitivity will allow the two occurrences of f to be analyzed independently. Consequently, such an analysis will determine that only $\lambda y.y$ flows out of the expression.

To put it another way, a context-sensitive analysis is capable of evaluating this program.

As a first approximation to understanding, the added precision of *k*CFA can be thought of as the ability to do partial reductions before analysis. If were to first reduce all of the apparent redexes in the program, and *then* do 0CFA on the residual, our example program would look like

$$(\lambda x_1.x_1)(\lambda x_2.x_2)(\lambda y.y).$$

Being a linear program, 0CFA is sufficient to prove only $\lambda y.y$ flows out of this residual. The polyvariance of *k*CFA is powerful enough to prove the same, however it is important to note that it is *not* done by a bounded reduction of the program. Instead, the *k*CFA hierarchy uses the last k calling contexts to distinguish closures.

The increased precision comes with an empirically observed increase in cost. As Shivers noted in his retrospective on the *k*CFA 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.[12] For example, it seemed plausible (at least, to us) that the kCFA problem could be in NPTIME by *guessing* flows appropriately during analysis.

As this dissertation shows, the naive algorithm is essentially the best one, and 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.

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.
  This connection between theory and experience contrasts with the similar result for ML-type inference (Mairson, 1990): 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.[13]
  But static flow analysis really has been costly, and this theorem explains why.
- The theorem is proved *by* functional programming.
  We take the view that the analysis itself is a functional programming language, 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.

[12] 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.

[13] (Kuan & MacQueen, 2007) have recently provided a refined perspective on the complexity of ML-type inference that explains why it works so quickly in practice.

We proceed by first bounding the complexity of *k*CFA from above, showing that *k*CFA can be solved in exponential time (subsection 4.2). This is easy to calculate and is known (Nielson *et al.*, 1999). Next, we bound the complexity from below by using *k*CFA as a SAT-solver. This shows *k*CFA is at least NPTIME-hard (subsection 4.3). The intuitions developed in the NPTIME-hardness proof can be improved to construct a kind of exponential iterator. A small, elucidative example is developed in subsection 4.4. These ideas are then scaled up and applied in subsection 4.5 to close the gap between the EXPTIME upper bound and NPTIME lower bound by giving a construction to simulate Turing machines for an exponential number of steps using *k*CFA, thus showing *k*CFA to be complete for EXPTIME.

4.2 kCFA is in EXPTIME

Recall the definition of *k*CFA from **??**. The cache, $\widehat{\mathsf{C}},\hat{\mathsf{r}}$, is a finite mapping and has n^{k+1} entries. Each entry contains a set of closures. The environment component of each closure maps p free variables to any one of n^k contours. There are n possible λ-terms and n^{kp} environments, so each entry contains at most n^{1+kp} closures. Analysis is monotonic, and there are at most $n^{1+(k+1)p}$ updates to the cache. Since $p \leq n$, we conclude:

Lemma 10
The control flow problem for *k*CFA is contained in EXPTIME.

It is worth noting that this result shows, from a complexity perspective, the flatness of the *k*CFA hierarchy: *for any constant k*, *k*CFA is decidable in exponential time. It is not the case, for example, that 1CFA requires exponential time (for all j, DTIME(2^{n^j}) $\subseteq$ EXPTIME), while 2CFA requires *doubly* exponential time (for all j, DTIME($2^{2^{n^j}}$) $\subseteq$ 2EXPTIME), 3CFA requires *triply* exponential time, etc. There are strict separation results for these classes, EXPTIME $\subset$ 2EXPTIME $\subset$ 3EXPTIME, etc., so we know from the above lemma there is no need to go searching for lower bounds greater than EXPTIME.

4.3 kCFA is NPTIME-hard

Because *k*CFA makes approximations, many closures can flow to a single program point and contour. In 1CFA, for example, $\lambda w.wx_1x_2\cdots x_n$ has n free variables, with an exponential number of possible associated environments mapping these variables to program points (contours of length 1). Approximation allows us to bind each x_i, independently, to either of the closed λ-terms for `True` or `False` that we saw in the PTIME-completeness proof for 0CFA. In turn, application to an n-ary Boolean function necessitates computation of all 2^n such bindings in order to compute the flow out from the application site. The term `True` can only flow out if the Boolean function is satisfiable by some truth valuation. For an appropriately chosen program point (label) ℓ, the cache location $\widehat{\mathsf{C}}(v,\ell)$ will contain the set of all possible closures which are approximated to flow to v. This set is that of all closures

$$\langle(\lambda w.wx_1x_2\cdots x_n),\rho\rangle$$

where ρ ranges over all assignments of `True` and `False` to the free variables (or more precisely assignments of locations in the table containing `True` and `False` to the free

$$
\begin{array}{l}
(\lambda f_1.(f_1\ \texttt{True})(f_1\ \texttt{False})) \\
(\lambda x_1. \\
\quad (\lambda f_2.(f_2\ \texttt{True})(f_2\ \texttt{False})) \\
\quad (\lambda x_2. \\
\qquad (\lambda f_3.(f_3\ \texttt{True})(f_3\ \texttt{False})) \\
\qquad (\lambda x_3. \\
\qquad\quad \cdots \\
\qquad\quad (\lambda f_n.(f_n\ \texttt{True})(f_n\ \texttt{False})) \\
\qquad\quad (\lambda x_n. \\
\qquad\qquad C[(\lambda v.\phi\ v)(\lambda w.wx_1x_2\cdots x_n)])\cdots))))
\end{array}
$$

Fig. 14. NPTIME-hard construction for kCFA.

variables). The Boolean function ϕ is completely linear, as in the PTIME-completeness proof; the context C uses the Boolean output(s) as in the conclusion to that proof: mixing in some ML, the context is:

```
- 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;
```

Again, `a` can only flow as an argument to `f` if `True` flows to `(u,u')`, leaving `(f,g)` unchanged, which can only happen if *some* closure $\langle(\lambda w.wx_1x_2\cdots x_n),\rho\rangle$ provides a satisfying truth valuation for ϕ. We have as a consequence:

Theorem 8
The control flow problem for 1CFA is NPTIME-hard.

Having established this lower bound for 1CFA, we now argue the result generalizes to all values of $k > 0$. Observe that by going from kCFA to $(k+1)$CFA, further context-sensitivity is introduced. But, this added precision can be undone by inserting an identity function application at the point relevant to answering the flow question. This added calling context consumes the added bit of precision in the analysis and renders the analysis of rest of the program equivalently to the courser analysis. Thus, it is easy to insert an identity function into the above construction such that 2CFA on this program produces the same results as 1CFA on the original. So for any $k > 1$, we can construct an NPTIME-hard computation by following the above construction and inserting $k-1$ application sites to eat up the precision added beyond 1CFA. The result is equivalent to 1CFA on the original term, so we conclude:

Theorem 9
The control flow problem for kCFA is NPTIME-hard, for any $k > 0$.

At this point, there is a tension in the results. On the one hand, kCFA is contained in EXPTIME; on the other, kCFA requires at least NPTIME-time to compute. So a gap remains; either the algorithm for computing kCFA can be improved and put into NPTIME, or the lower bound can be strengthened by exploiting more computational power from the analysis.

We observe that while the computation of the *entire* cache requires exponential time, perhaps the existence of a *specific* flow in it may well be computable in NPTIME. A nondeterministic algorithm might compute using the "collection semantics" $\mathscr{E}[\![t^\ell]\!]^\rho_\delta$, but rather than compute entire sets, *choose* the element of the set that bears witness to the flow. If so we could conclude kCFA is NPTIME-complete.

However, this is not the case. We show that the lower bound can be improved and kCFA is complete for EXPTIME. The improvement relies on simulating an exponential iterator using analysis. The following section demonstrates the core of the idea.

4.4 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:

$$(\lambda f.(f\ \texttt{True})(f\ \texttt{False}))$$
$$(\lambda x.\texttt{Implies}\,x\,x)$$

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 nonlinearly. The approximation permits one occurrence of x be bound to `True` while the other occurrence is bound to `False`; and somewhat alarmingly, `ImpliesTrueFalse` 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:

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

Here, rather than pass x directly to `Implies`, we construct a unary tuple $\lambda w.wx$. The tuple is used nonlinearly, 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 NPTIME-hardness of the 1CFA control flow problem.

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.

4.5 *k*CFA is EXPTIME-hard

4.5.1 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 as described in subsection 4.3.

4.5.2 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:

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

Fig. 15. Generalization of toy calculation for kCFA.

In the inner 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.)

4.5.3 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:

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

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 $T+1$:

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

(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.

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

Clearly, these three rules can be coded by a single Boolean circuit, and we have all the required Boolean logic at our disposal from subsection 2.5.

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 in Figure 16. The `Copy`

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

Fig. 16. Turing machine transition function construction.

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 $\lambda x.\texttt{Implies}xx$ 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.

4.5.4 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`. In this code, the $\lambda x.x$ (with label ℓ' on its application) serve

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

Fig. 17. EXPTIME-hard construction for *k*CFA.

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,

$$2^n\ \Phi\ \lambda w.w\mathbf{0}\ldots\mathbf{0}\cdots Q_0\cdots H_0\cdots z_1z_2\ldots z_N\mathbf{0},$$

where Φ is a coding of transition function for M. The λ-term 2^n is a fixed point operator for *k*CFA, 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 without loss of generality 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 11
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-constructable, closed, labeled λ-term e with distinguished label ℓ such that in the *k*CFA analysis of e ($k > 0$), `True` flows into ℓ iff M accepts x.

Theorem 10
The control flow problem for *k*CFA is complete for EXPTIME for any $k > 0$.

4.6 Exact kCFA is PTIME-complete

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* evaluation. To get a good lower bound, this is necessary: it turns out there is a dearth of computation power when *k*CFA corresponds with evaluation, i.e. when the analysis is exact.

As noted earlier, approximation arises from the truncation of contours during analysis. Consequently, if truncation never occurs, the instrumented interpreter and the abstract interpreter produce identical results for the given program. But what can we say about the complexity of these programs? In other words, what kind of computations can *k*CFA analyze exactly when k is a constant, independent of the program analyzed? What is the intersection between the abstract and concrete interpreter?

An answer to this question provides another point in the characterization of the expressiveness of an analysis. For 0CFA, the answer is PTIME since the evaluation of linear terms is captured. For *k*CFA, the answer remains the same.

For any fixed k, kCFA can only analyze polynomial time programs exactly, since, in order for an analysis to be exact, there can only one entry in each cache location, and there are only n^{k+1} locations. But from this it is clear that only through the use of approximation that a exponential time computation can be simulated, but this computation has little to do with the actual running of the program. A program that runs for exponential time cannot be analyzed exactly by kCFA for any constant k.

Contrast this with ML-typability, for example, where the evaluation of programs that run for exponential time can be simulated via type inference.

Note that if the contour is never truncated, 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 a characterization of the computational complexity (or expressivity) of the language evaluated by the instrumented evaluator $\mathscr{E}$ of section **??** as a function of the contour length.

It also 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 & Jagannathan, 1998). (Might & Shivers, 2006b) make a similar observation: "imprecision reinforces itself during a flow analysis through an ever-worsening feedback loop." This ever-worsening feedback loop, in which we can make `False` (spuriously) flow out of `Implies`xx, is the critical ingredient in our EXPTIME lower bound.

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

4.7 Discussions

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 **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.

The idea behind kCFA is that the precision of could *dialed up*, but there are essentially two settings to the kCFA hierarchy: *high* ($k > 0$, EXPTIME) and *low* ($k = 0$). We can see, from a computational complexity perspective, that 0CFA is strictly less expressive than kCFA. In turn, kCFA is strictly less expressive than, for example, Mossin's flow analysis (1997a). 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?

This argues that the relationship between dial level N and $N+1$ should be exact. This is the case with say simple-typing and ML-typing. (ML = simple + let reduction). There is no analogous relationship known between k and $k+1$CFA. A major computational expense in kCFA is the approximation engendering further approximation and re-evaluation. Perhaps by staging analysis into polyvariance and approximation phases, the feedback loop of spurious flows can be avoided.

If you had an analysis that did some kind of exact, bounded, evaluation of the program and then analyzed the residual with 0CFA, you may have a far more usable analysis than with the kCFA hierarchy.

The precision of kCFA is highly sensitive to syntactic structure. Simple program refactorings such as η-expansion have drastic effects on the results of kCFA and can easily undermine the added work of a more and more precise analysis. Indeed, we utilize these simple refactorings to undermine the added precision of kCFA to generalize the hardness results from the case of 1CFA to all $k > 0$ CFA. But an analysis that was robust in the face of these refactorings could undermine these lower bounds.

In general, techniques that lead to increased precision will take computational power *away* from our lower bound constructions. For instance, it is not clear what could be said about lower bounds on the complexity of a variant of kCFA that employed abstract garbage collection (Might & Shivers, 2006b), which allows for the safe removal of values from the cache during computation. It is critical in the lower bound construction that what goes into the cache, stays in the cache.

Lévy's notion of labeled reduction (1978; 1980) provides a richer notion of "instrumented evaluation" coupled with a richer theory of exact flow analysis, namely the geometry of interaction (Girard, 1989; Gonthier *et al.*, 1992). 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.

4.8 Conclusions

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 nonlinearity 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.

This detailed accounting of the ingredients that combine to make kCFA hard, when $k > 0$, should provide guidance in designing new abstractions that avoid computationally expensive components of analysis. A lesson learned has been that *closures*, as they exist when $k > 0$, result in an exponential value space that can be harnessed for the EXPTIME lower-bound construction.

This dissertation draws upon several large veins of research. At the highest level, this includes complexity, semantics, logic, and program analysis. This chapter surveys related work to sketch applications and draw parallels with existing work.

5 Monovariant Flow Analysis

In the setting of first-order programming languages, (Reps, 1996) gives a complexity investigation of program analyses and shows interprocedural slicing to be complete for PTIME and that obtaining "meet-over-all-valid-paths" solutions of distributive data-flow analysis problems (Hecht, 1977) is PTIME-hard in general, and PTIME-complete when there are only a finite number of data-flow facts. A circuit-value construction by interprocedural data-flow analysis is given using Boolean circuitry encoded as call graph gadgets, similar in spirit to our constructions in section 2.

In the setting of higher-order programming languages, (Melski & Reps, 2000) give a complexity investigation of 0CFA-like, inclusion-based monovariant flow analysis for a functional language with pattern matching. The analysis takes the form of a constraint satisfaction problem and this satisfaction problem is shown to be complete for PTIME. See section 7 for further discussion.

The impact of pattern matching on analysis complexity is further examined by (Heintze & McAllester, 1997b), which shows how deep pattern matching affects monovariant analysis, making it complete for EXPTIME.

6 Linearity and Static Analysis

(Jagannathan *et al.*, 1998) observe that flow analysis, which is a *may* analysis, can be adapted to answer *must* analysis questions by incorporating a "per-program-point *variable cardinality map*, which indicates whether all reachable environments binding a variable x hold the same value. If so, x is marked single at that point; otherwise x is marked multiple." The resulting must-alias information facilities program optimization such as lightweight closure conversion (Steckler & Wand, 1997). This must analysis is a simple instance of tracking linearity information in order to increase the precision of the analysis. (Might & Shivers, 2006b) use a similar approach of *abstract counting*, which distinguish singleton and non-singleton flow sets, to improve flow analysis precision.

Something similar can be observed in 0CFA without cardinality maps; singleton flow sets $\widehat{\mathsf{C}}(\ell) = \{\lambda x.e\}$, which are interpreted as "the expression labelled ℓ *may* evaluate to one of $\{\lambda x.e\}$," convey *must* information. The expression labelled ℓ either diverges or evaluates to $\lambda x.e$. When $\lambda x.e$ is linearly closed—the variables map to singleton sets containing linear closures—then the run-time value produced by the expression labelled ℓ can be determined completely at analysis time. The idea of taking this special case of *must* analysis within a *may* analysis to its logical conclusion is the basis of section 2.

(Damian & Danvy, 2003) have investigated the impact of linear β-reduction on the result of flow analysis and show how leastness is preserved. The result is used to show that leastness is preserved through CPS and administrative reductions, which are linear.

An old, but key, observation about the type inference problem for simply typed λ-terms is that, when the term is linear (every bound variable occurs exactly once), the most general type and normal form are isomorphic (Hindley, 1989; Hirokawa, 1991; Henglein & Mairson, 1991; Mairson, 2004).[14]

The observation translates to flow analysis, as shown in section 2, but in a typed setting, it also scales to richer systems. The insight leads to an elegant reproof of the EXPTIME-hardness of ML-type inference result from (Mairson, 1990) (Henglein, 1990). It was used to prove novel lower bounds on type inference for System F_ω (Henglein & Mairson, 1991) and rank-bound intersection type inference (Møller Neergaard & Mairson, 2004). See section 14 for further discussion.

7 Context-Free-Language Reachability

(Melski & Reps, 2000) show the interconvertibility between a number of set-constraint problems and the context-free-language (CFL) reachability problem, which is known to be complete for PTIME (Ullman & van Gelder, 1986). (Heintze, 1994) develops a set-based approach to flow analysis for a simple untyped functional language with functions, applications, pattern-matching, and recursion. The analysis works by making a pass over the program, generating set constraints, which can then be solved to compute flow analysis results. Following Melski and Reps, we refer to this constraint system as ML set-

[14] The seed of inspiration for this work came from a close study of (Mairson, 2004) in the Spring of 2005 for a seminar presentation given in a graduate course on advanced topics in complexity theory at the University of Vermont.

constraints. For the subset of the language considered in this dissertation, solving these constraints computes a monovariant flow analysis that coincides with 0CFA.

In addition to the many set-constraint problems considered, which have applications to static analysis of first-order programming languages, [section 5] (Melski & Reps, 2000) also investigate the problem of solving the ML set-constraints used by Heintze. They show this class of set-constraint problems can be solved in cubic time with respect to the size of the input constraints. Since (Heintze, 1994) gave a $O(n^3)$ algorithm for solving these constraints, Melski and Reps' result demonstrates the conversion to CFL-reachability preserves cubic-solvability, while allowing CFL-reachability formulations of static analyses, such as program slicing and shape analysis, to be brought to bear on higher-order languages, where previously they had only been applied in a first-order setting.

After showing ML set-constraints can be solved using CFL-reachability, [section 6] (Melski & Reps, 2000) also prove the converse holds: CFL-reachability problems can be solved by reduction to ML set-constraint problems while preserving the worse-case asymptotic complexity. By the known PTIME-hardness of CFL-reachability, this implies ML set-constraint satisfaction is PTIME-complete. It does not follow, however, that 0CFA is also PTIME-complete.

It is worth noting that Melski and Reps are concerned with constraint satisfaction, and not directly with flow analysis—the two are intimately related, but the distinction is important. It follows as a corollary that since ML set-constraints can be solved, through a reduction to CFL-reachability, flow analysis can be performed in cubic time. [page 314] (Heintze, 1994) observes that the size of the set-constraint problem generated by the initial pass of the program is linear in the size of the program being analyzed. Therefore it is straightforward to derive from the ML set-constraint to CFL-reachability reduction the (known) inclusion of 0CFA in PTIME.

In the other direction, it is not clear that it follows from the PTIME-hardness of ML set-constraint satisfaction that flow analysis of Heintze's subject language is PTIME-hard. Melski and Reps use the constraint language directly in their encoding of CFL-reachability. What remains to be seen is whether there are programs which could be constructed that would induce these constraints. Moreover, their reduction relies soley on the "case" constraints of Heintze, which are set constraints induced by pattern matching expressions in the source language.

If the source language lacks pattern matching, the Boolean circuit machinery of Melski and Reps can no longer be constructed since no expressions induce the needed "case" constraints. For this language, the PTIME-hardness of constraint satisfaction and 0CFA does not follow from the results of Melski and Reps.

This reiterates the importance of Reps' own observation that analysis problems should be formulated in "trimmed-down form," which both leads to a wider applicability of the lower bounds and "allows one to gain greater insight into exactly what aspects of an [...] analysis problem introduce what computational limitations on algorithms for these problems," [section 2] (Reps, 1996).

By considering only the core subset of every higher-order programming language and relying on the specification of analysis, rather than its implementation technology, the 0CFA PTIME-completeness result implies as an immediate corollary the PTIME-completeness of the ML set-constraint problem considered by Melski and Reps. Moreover, as we have

seen, our proof technique of using linearity to subvert approximation is broadly applicable to further analysis approximations, whereas CFL-reachability reductions must be replayed *mutatis mutandis*.

8 2NPDA and the Cubic Bottleneck

The class 2PNDA contains all languages that are recognizable by a two-way non-deterministic push-down automaton.[15] The familiar PDAs found in undergraduate textbooks (Martin, 1997), both deterministic and non-deterministic, are one-way: consuming their input from left-to-right. In contrast, two-way NPDAs accept their input on a read-only input tape marked with special begin and end markers, on which they can move the read-head forwards, backwards, or not at all.

Over a decade ago, (Heintze & McAllester, 1997c) proved deciding a monovariant flow analysis problem to be at least as hard as 2PNDA, and argued this provided evidence the "cubic bottleneck" of flow analysis was unlikely to be overcome since the best known algorithm for 2PNDA was cubic and had not been improved since its formulation by (Aho *et al.*, 1968). This statement was made by several other papers (Neal, 1989; Heintze & McAllester, 1997c; Heintze & McAllester, 1997a; Melski & Reps, 2000; McAllester, 2002; Van Horn & Mairson, 2008). Yet collectively, this is simply an oversight in the history of events; (Rytter, 1985) improved the cubic bound by a logarithmic factor.

Since then, Rytter's technique has been used in various contexts: in diameter verification, in Boolean matrix multiplication, and for the all pairs shortest paths problem (Basch *et al.*, 1995; Zwick, 2006; Chan, 2007), as well as for reachability in recursive state machines (Chaudhuri, 2008), and for maximum node-weighted k-clique (Vassilevska, 2009) to name a few. In particular, (Chaudhuri, 2008) recently used Rytter's techniques to formulate a subcubic algorithm for the related problem of context-free language (CFL) reachability. Perhaps unknown to most, indirectly this constitutes the first subcubic inclusion-based flow analysis algorithm when combined with a reduction due to (Melski & Reps, 2000).

The logarithmic improvement can be carried over to the flow analysis problem directly, by applying the same known set compression techniques (Rytter, 1985) applies to improve deciding 2PNDA. Moreover, refined analyses similar to (Heintze & McAllester, 1997b) that incorporate notions of reachability to improve precision remain subcubic. See (Midtgaard & Van Horn, 2009) for details.

0CFA is complete for both 2PNDA (Heintze & McAllester, 1997c) and PTIME (section 2). Yet, it is not clear what relation these class have to each other.

The 2PNDA inclusion proof of Heintze and McAllester is sensitive to representation choices and problem formulations. They use an encoding of programs that requires a non-standard bit string labelling scheme in which identical subterms have the same labels. The authors remark that without this labelling scheme, the problem "appears not to be in 2PNDA."

Moreover, the notions of reduction employed in the definitions of 2PNDA-hardness and PTIME-hardness rely on different computational models. For a problem to be 2PNDA-hard, all problems in the class must be reducible to it in $O(nR(\log n))$ time on a RAM,

[15] This section is derived from material in (Midtgaard & Van Horn, 2009).

where R is a polynomial. Whereas for a problem to be PTIME-hard, all problems in the class must be reducible to it using a $O(\log n)$ space work-tape on a Turing machine.

9 kCFA

Our coding of Turing machines is descended from 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.

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.

10 Class Analysis

Flow analysis of functional languages is complicated by the fact that *computations are expressible values*. This makes basic questions about control flow undecidable in the general case. But the same is true in object-oriented programs—computations may be package up as values, passed as arguments, stored in data-structures, etc.—and so program analyses in object-oriented settings often deal with the same issues as flow analysis. A close analogue of flow analysis is *class analysis*.

Expressions in object-oriented languages may have a declared class (or type) but, at run-time, they can evaluate to objects of every subclass of the class. Class analysis computes the actual set of classes that an expression can have at run-time (Johnson *et al.*, 1988; Chambers & Ungar, 1990; Palsberg & Schwartzbach, 1991; Bacon & Sweeney, 1996). Class analysis is sometimes called receiver class analysis, type analysis, or concrete type inference; it informs static method resolution, inlining, and other program optimizations.

An object-oriented language is higher-order in the same way as a language with first-class functions and exactly the same circularity noted by Shivers occurs in the class analysis of an object-oriented language.

(Grove & Chambers, 2001):

In object-oriented languages, the method invoked by a dynamically dispatched message send depends on the class of the object receiving the message; in languages with function

values, the procedure invoked by the application of a computed function value is determined by the function value itself. In general, determining the flow of values needed to build a useful call graph requires an interprocedural data and control flow analysis of the program. But interprocedural analysis in turn requires that a call graph be built prior to the analysis being performed.

Ten years earlier, [page 6] (Shivers, 1991)[16] had written essentially the same:

So, if we wish to have a control-flow graph for a piece of Scheme code, we need to answer the following question: for every procedure call in the program, what are the possible lambda expressions that call could be a jump to? But this is a flow analysis question! So with regard to flow analysis in an HOL, we are faced with the following unfortunate situation:

- *In order to do flow analysis, we need a control-flow graph.*
- *In order to determine control-flow graphs, we need to do flow analysis.*

Class analysis is often presented using the terminology of type inference, however these type systems typically more closely resemble flow analysis: types are finite sets of classes appearing syntactically in the program and subtyping is interpreted as set inclusion.

In other words, objects are treated much like functions in the flow analysis of a functional language—typically both are approximated by a set of definition sites, i.e. an object is approximated by a set of class names that appear in the program; a function is approximated by a set of λ occurrences that appear in the program. In an object-oriented program, we may ask of a subexpression, what classes may the subexpression evaluate to? In a functional language we may ask, what λ terms may this expression evaluate to? Notice both are general questions that analysis must answer in a higher order setting if you want to know about control flow. To know where control may transfer to from $(f\ x)$ we have to know what f may be. To know where control may transfer to from `f.apply(x)` we have to know what `f` may be. In both cases, if we approximate functions by sets of λs and objects by sets of class names, we may determine a set of possible places in code where control may transfer, but we will not know about the *environment* of this code, i.e. the environment component of a closure or the record component of an object.

(Spoto & Jensen, 2003) give a reformulation of several class analyses, including that of (Palsberg & Schwartzbach, 1991; Bacon & Sweeney, 1996; Diwan *et al.*, 1996), using abstract interpretation.

(DeFouw *et al.*, 1998) presents a number of variations on the theme of monovariant class analysis. They develop a framework that can be instantiated to obtain inclusion, equality, and optimistic based class analyses with close analogies to 0CFA, simple closure analysis, and rapid type analysis (Bacon & Sweeney, 1996), respectively. Each of these instantiations enjoy the same asymptotic running times as their functional language counterparts; cubic, near linear, and linear, respectively.

Although some papers give upper bounds for the algorithms they present, there are very few lower bound results in the literature.[17]

[16] It is a testament to Shivers' power as a writer that his original story has been told over and over again in so many places, usually with half the style.

[17] I was able to find zero papers that deal directly with lower bounds on class analysis complexity.

```
new Fun<Fun<B,List<B>>,List<B>>() {
 public List<B> apply(Fun<B,List<B>> f1) {
   f1.apply(true);
   return f1.apply(false);
 }
}.apply(new Fun<B,List<B>>() {
  public List<B> apply(final B x1) {
    return
      new Fun<Fun<B,List<B>>,List<B>>() {
       public List<B> apply(Fun<B,List<B>> f2) {
         f2.apply(true);
         return f2.apply(false);
       }
      }.apply(new Fun<B,List<B>>() {
        public List<B> apply(final B x2) {
          return
            ...
            new Fun<Fun<B,List<B>>,List<B>>() {
             public List<B> apply(Fun<B,List<B>> fn) {
               fn.apply(true);
               return fn.apply(false);
             }
            }.apply(new Fun<B,List<B>>() {
              public List<B> apply(final B xn) {
                return
                  new List<B>{x1,x2,...xn};}}
```

Fig. 18. Translation of *k*CFA EXPTIME-construction into an object-oriented language.

Class analysis is closely related to *points-to* analysis in object-oriented languages. "*Points-to analysis* is a fundamental static analysis used by optimizing compilers and software engineering tools to determine the set of objects whose addresses may be stored in reference variables and reference fields of objects," (Milanova *et al.*, 2005). When a points-to analysis is *flow-sensitive*—"analyses take into account the flow of control between program points inside a method, and compute separate solutions for these points," (Milanova *et al.*, 2005)—the analysis necessarily involves some kind of class analysis.

In object-oriented languages, context-sensitive is typically distinguished as being object-sensitive (Milanova *et al.*, 2005), call-site sensitive (Grove & Chambers, 2001), or partially flow sensitivity (Rinetzky *et al.*, 2008).

(Grove & Chambers, 2001) provide a framework for a functional and object-oriented hybrid language that can be instantiated to obtain a *k*CFA analysis and an object-oriented analogue called *k-l-CFA*. There is a discussion and references in Section 9.1. In this discussion, (Grove & Chambers, 2001) cite (Oxhøj *et al.*, 1992) as giving "1-CFA extension to Palsberg and Schwartzbach's algorithm," although the paper develops the analysis as a type inference problem. Grove and Chambers also cite (Vitek *et al.*, 1992) as one of several "adaptations of *k*CFA to object-oriented programs," and although this paper actually has analogies to *k*CFA in an object-oriented setting (they give a call-string approach to call graph context sensitivity in section 7), it seems to be developed completely independently of Shivers' *k*CFA work or any functional flow analysis work.

The construction of Figure 15 can be translated in an object-oriented language such as Java, as given in Figure 18.[18] Functions are simulated as objects with an apply method. The crucial subterm in Figure 18 is the construction of the list `{x1,x2,..xn}`, where $\mathtt{x}_i$ occur free with the context of the innermost "lambda" term, `new Fun() {...}`. To be truly faithful to the original construction, lists would be Church-encoded, and thus represented with a function of one argument, which is applied to `x1` through `xn`. An analysis with a similar context abstraction to 1CFA will approximate the term representing the list `x1,x2,...,xn` with an abstract object that includes 1 bit of context information for each *instance variable*, and thus there would be 2^n values flowing from this program point, one for each mapping $\mathtt{x}_i$ to the calling context in which it was bound to either true or false for all possible combinations. (Grove & Chambers, 2001) develop a framework for call-graph construction which can be instantiated in the style of 1CFA and the construction above should be adaptable to show this instantiation is EXPTIME-hard.

A related question is whether the insights about linearity can be carried over to the setting of pointer analysis in a first-order language to obtain simple proofs of lower bounds. If so, is it possible higher-order constructions can be transformed systematically to obtain first-order constructions?

Type hierarchy analysis is a kind of class analysis particularly relevant to the discussion in **??** and the broader applicability of the approach to proving lower bounds employed in section 2. Type hierarchy analysis is an analysis of statically typed object-oriented languages that bounds the set of procedures a method invocation may call by examining the type hierarchy declarations for method overrides. "Type hierarchy analysis does not examine what the program actually does, just its type and method declarations," (Diwan *et al.*, 1996). It seems unlikely that the technique of **??** can be applied to prove lower bounds about this analysis since it has nothing to do with approximating evaluation.

11 Pointer Analysis

Just as flow analysis plays a fundamental role in the analysis of higher-order functional programs, *pointer analysis*[19] plays a fundamental role in imperative languages with pointers (Landi, 1992a) and object-oriented languages, and informs later program analyses such as live variables, available expressions, and constant propagation. Moreover, flow and alias analysis variants are often developed along the same axes and have natural analogues with each other.

For example, Henglein's (1992) simple closure analysis and Steensgaard's (1996) points-to analysis are natural analogues. Both operate in near linear time by relying on equality-based (rather than inclusion-based) set constraints, which can be implemented using a union-find data-structure. Steensgaard algorithm "is inspired by Henglein's (1991) binding time analysis by type inference," which also forms the conceptual basis for (Henglein, 1992). Palsberg's (1995) and Heintze's (1994) constraint-based flow analysis and Andersen's

[18] This translation is Java except for the made up list constructor and some abbreviation in type names for brevity, i.e. `B` is shorthand for `Boolean`.

[19] Also known as *alias* and *points-to* analysis.

(1994) pointer analysis are similarly analogous and bear a strong resemblance in their use of subset constraints.

To get a full sense of the correspondence between pointer analysis and flow analysis, read their respective surveys in parallel (Hind, 2001; Midtgaard, 2007). These comprise major, mostly independent, lines of research. Given the numerous analogies, it is natural to wonder what the pointer analysis parallels are to the results presented in this dissertation. The landscape of the pointer analysis literature is much like that of flow analysis; there are hundreds of papers; similar, over-loaded, and abused terminology is frequently used; it concerns a huge variety of tools, frameworks, notations, proof techniques, implementation techniques, etc. Without delving into too much detail, we recall some of the fundamental concepts of pointer analysis, cite relevant results, and try to more fully develop the analogies between flow analysis and pointer analysis.

A pointer analysis attempts to statically determine the possible run-time values of a pointer. Given a program and two variables p and q, points-to analysis determines if p can point to q (Chakaravarthy, 2003). It is clear that in general, like all interesting properties of programs, it is not decidable if p can point q. A traditional assumption in this community is that all paths in the program are executable. However, even under this conservative assumption, the problem is undecidable. The history of pointer analysis can be understood largely in terms of the trade-offs between complexity and precision.

Analyses are characterized along several dimensions (Hind, 2001), but of particular relevance are those of:

- *Equality-based*: assignment is treated as an undirected flow of values.
- *Subset-based*: assignment is treated as a directed flow of values.
- *Flow sensitivity*
  A points-to analysis is *flow-sensitive* analysis if it is given the control flow graph for the analyzed program. The control flow graphs informs the paths considered when determining the points-to relation. A *flow-insensitive* analysis is not given the control flow graph and it is assumed statements can be executed in any order. See also section 4.4 of (Hind, 2001) and section 2.3 of (Rinetzky *et al.*, 2008).
- *Context sensitivity*
  calling context is considered when analyzing a function so that calls return to their caller. See also section 4.4 of (Hind, 2001).
  (Bravenboer & Smaragdakis, 2009) remark:

  > *In full context-sensitive pointer analysis, there is an ongoing search for context abstractions that provide precise pointer information, and do not cause massive redundant computation.*[20]

The complexity of pointer analysis has been deeply studied (Myers, 1981; Landi & Ryder, 1991; Landi, 1992a; Landi, 1992b; Choi *et al.*, 1993; Ramalingam, 1994; Horwitz, 1997; Muth & Debray, 2000; Chatterjee *et al.*, 2001; Chakaravarthy & Horwitz, 2002; Chakaravarthy, 2003; Rinetzky *et al.*, 2008).

[20] That search has been reflected in the functional community as well, see for example, (Shivers, 1991; Jagannathan & Weeks, 1995; Banerjee, 1997; Faxén, 1997; Nielson & Nielson, 1997; Sereni, 2007; Ashley & Dybvig, 1998; Wright & Jagannathan, 1998; Might & Shivers, 2006a; Might, 2007).

Flow sensitive points-to analysis with dynamic memory is not decidable (Landi, 1992b; Ramalingam, 1994; Chakaravarthy, 2003). Flow sensitive points-to analysis without dynamic memory is PSPACE-hard (Landi, 1992a; Muth & Debray, 2000), even when pointers are well-typed and restricted to only two levels of dereferencing (Chakaravarthy, 2003). Context-sensitive pointer analysis can be done efficiently in practice (Emami *et al.*, 1994; Wilson & Lam, 1995). Flow and context-sensitive points-to analysis for Java can be efficient and practical even for large programs (Milanova *et al.*, 2005).

See (Muth & Debray, 2000; Chakaravarthy, 2003) for succinct overview of complexity results and open problems.

12 Logic Programming

(McAllester, 2002) argues "bottom-up logic program presentations are clearer and simpler to analyze, for both correctness and *complexity*" and provides theorems for characterizing their run-time. McAllester argues bottom-up logic programming is especially appropriate for static analysis algorithms. The paper gives a bottom-up logic presentation of evaluation (Fig. 4) and flow analysis (Fig 5.) for the λ-calculus with pairing and uses the run-time theorem to derive a cubic upper bound for the analysis.

Recent work by (Bravenboer & Smaragdakis, 2009) demonstrates how Datalog can be used to specify and efficiently implement pointer analysis. By the PTIME-completeness of Datalog, any analysis that can be specified is included in PTIME.

This bears a connection to the implicit computational complexity program, which has sought to develop syntactic means of developing programming languages that capture some complexity class (Hofmann, 1998; Leivant, 1993; Hofmann, 2003; Kristiansen & Niggl, 2004). Although this community has focused on general purpose programming languages—with only limited success in producing usable systems—it seems that restricting the domain of interest to program analyzers may be a fruitful line of work to investigate.

The EXPTIME construction of subsection 4.5 has a conceptual basis in Datalog complexity research (Hillebrand *et al.*, 1995; Gaifman *et al.*, 1993). See section 9 for a discussion.

13 Termination Analysis

Termination analysis of higher-order programs (Jones & Bohr, 2008; Sereni & Jones, 2005; Giesl *et al.*, 2006; Sereni, 2007) is inherently tied to some underlying flow analysis.

Recent work by Sereni and Jones on the termination analysis of higher-order languages has relied on an initial control flow analysis of a program, the result of which becomes input to the termination analyzer (Sereni & Jones, 2005; Sereni, 2007). Once a call-graph is constructed, the so-called "size-change" principle[21] can be used to show that there is no infinite path of decreasing size through through the program's control graph, and therefore the program eventually produces an answer. This work has noted the inadequacies of 0CFA for producing precise enough graphs for proving most interesting programs terminating.

[21] The size-change principle has enjoyed a complexity investigation in its own right (Lee *et al.*, 2001; Ben-Amram & Lee, 2007).

Motivated by more powerful termination analyses, these researchers have designed more powerful (i.e., more precise) control flow analyses, dubbed k-limited CFA. These analyses are parametrized by a fixed bound on the depth of environments, like Shivers' kCFA. So for example, in 1-limited CFA, each variable is mapped to the program point in which it is bound, but no information is retained about this value's environment. But unlike kCFA, this "limited" analysis is not polyvariant (context-sensitive) with respect to the most recent k calling contexts.

A lesson of our investigation into the complexity of kCFA is that it is *not* the polyvariance that makes the analysis difficult to compute, but rather the environments. Sereni notes that the k-limited CFA hierarchy "present[s] different characteristics, in particular in the aspects of precision and complexity" (Sereni, 2007), however no complexity characterization is given.

14 Type Inference and Quantifier Elimination

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 (Mairson, 1990; Henglein, 1990; Henglein & Mairson, 1991; Mairson, 2004). 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.[22] This observation motivates the analysis of type inference described by (Mairson, 1990; Mairson, 2004).

Compared to flow analysis, type reconstruction has enjoyed a much more thorough complexity analysis.

A key observation about the type inference problem for simply typed λ-terms is that, when the term is linear (every bound variable occurs exactly once), the most general type and normal form are isomorphic (Hindley, 1989; Hirokawa, 1991; Henglein & Mairson, 1991; Mairson, 2004). So given a linear term in normal form, we can construct its most general type (no surprise there), but conversely, when given a most general type, we can construct the normal form of all terms with that type.

This insight becomes the key ingredient in proving the lower bound complexity of simple-type inference—when the program is linear, static analysis is effectively "running" the program. Lower bounds, then, can be obtained by simply hacking within the linear λ-calculus.

[22] An aberrant case of this phenomenon is examined by (Møller Neergaard & 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.

Aside: *The normal form of a linear program can be "read back" from its most general type in the following way: given a type $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_k \to \alpha$, where α is a type variable, we can conclude the normal form has the shape $\lambda x_1.\lambda x_2.\ldots\lambda x_k.e$. Since the term is linear, and the type is most general, every type variable occurs exactly twice: once positively and once negatively. Furthermore, there exists a unique $\sigma_i \equiv \tau_1 \to \tau_2 \to \ldots \to \tau_m \to \alpha$, so x_i must be the head variable of the normal form, i.e., we now know: $\lambda x_1.\lambda x_2.\ldots\lambda x_k.x_1 e'$, and x_i is applied to m arguments, each with type $\tau_1, \ldots, \tau_m$, respectively. But now, by induction, we can recursively construct the normal forms of the arguments. The base case occurs when we get to a base type (a type variable); here the term is just the occurrence of the λ-bound variable that has this (unique) type. In other words, a negative type-variable occurrence marks a λ-binding, while the corresponding positive type-variable occurrence marks the single occurrence of the bound variable. The rest of the term structure is determined in a syntax-directed way by the arrow structure of the type.*

It has been known for a long time that type reconstruction for the simply typed λ-calculus is decidable (Curry, 1969; Hindley, 1969), i.e. it is decidable whether a term of the untyped λ-calculus is the image under type-erasing of a term of the simply typed λ-calculus.[23] (Wand, 1987) gave the first direct reduction to the unification problem (Herbrand, 1930; Robinson, 1965; Dwork *et al.*, 1984; Kanellakis *et al.*, 1991). (Henglein, 1991; Henglein, 1992) used unification to develop efficient type inference for binding time analysis and flow analysis, respectively. This work directly inspired the widely influential (Steensgaard, 1996) algorithm.[24]

A lower bound on the complexity of type inference can often be leveraged by the combinatorial power behind a quantifier elimination procedure (Mairson, 1992a). These procedures are syntactic transformations that map programs into potentially larger programs that can been typed in a simpler, quantifier-free setting.

As an example, consider the case of ML polymorphism. The universal quantification introduced by `let`-bound values can be eliminated by reducing all `let`-redexes. The residual program is simply-typable if, and only if, the original program is ML-typable.

This is embodied in the following inference rule:[25]

$$\frac{\Gamma \vdash M : \tau_0 \quad \Gamma \vdash [M/x]N : \tau_1}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : \tau_1}$$

The residual may be exponentially larger due to nested `let` expressions that must all be eliminated. From a Curry-Howard perspective, this can be seen as a form of cut-elimination. From a computational perspective, this can be seen as a bounded running of the program at compile time. From a software engineering perspective, this can be seen as code-reuse—the ML-type inference problem has been reduced to the simple-type inference problem,

[23] See (Tiuryn, 1990) for a survey of type inference problems, cited in (Cardone & Hindley, 2006).

[24] See section 11 for more on the relation of pointer analysis and flow analysis.

[25] In the survey, *Type systems for programming languages*, (Mitchell, 1990) attributes this observation to Albert Meyer. [page 122] (Henglein & Mairson, 1991) point out in a footnote that it also appears in the thesis of (Damas, 1985), and is the subject of a question on the 1985 postgraduate examination in computing at Edinburgh University.

and thus to first-order unification. But the price is that an exponential amount of work may now be required.

Full polymorphism is undecidable, but ML offers a limit form of outermost universal quantification. But this restriction relegates polymorphic functions to a second-class citizenship, so in particular, functions passed as arguments to functions (a staple of higher-order programming) can only be used monomorphically.

Intersection types restore first-class polymorphism by offering a finite form of explicit quantification over simple types. The type $\tau_1 \wedge \tau_2$ is used for a term that is typable as both τ_1 and τ_2. This can formalized as the following inference rule for $\wedge$:[26]

$$\frac{\Gamma_1 \vdash M : \tau_1 \quad \Gamma_2 \vdash M : \tau_2}{\Gamma_1 \wedge \Gamma_2 \vdash M : \tau_1 \wedge \tau_2}$$

where $\wedge$ is lifted to environments in a straightforward way. Notice that this allows expressions such as,

$$(\lambda f.\lambda z.z(f\ 2)(f\ \mathbf{false}))\ (\lambda x.x),$$

to be typed where x has type $\mathbf{int} \wedge \mathbf{bool}$.

The inference rule, as stated, breaks syntax-directed inference. (van Bakel, 1992) observed that by limiting the rule to the arguments of function application, syntax-direction can be recovered without changing the set of typable terms (although some terms will have fewer typings). Such systems are called *strict intersections* since the $\wedge$ can occur only on the left of a function type.

The finite $\wedge$-quantifiers of strict intersections too have an elimination procedure, which can be understood as a program transformation that eliminates $\wedge$-quantification by *rank*. A type is rank r if there are no occurrences of $\wedge$ to the left of r occurrences of an arrow. The highest rank intersections can be eliminated by performing a *minimal complete development*.

Every strongly normalizing term has an intersection type, so type inference in general is undecidable. However, decidable fragments can be regained by a standard approach of applying a *rank* restriction, limiting the depth of $\wedge$ to the left of a function type.

By bounding the rank, inference becomes decidable; if the rank is bound at k, k developments suffice to eliminate all intersections. The residual program is simply-typable if, and only if, the original program is rank-k intersection typable. Since each development can cause the program to grow by an exponential factor, iteratively performing k-MCD's results in an elementary lower bound (Kfoury *et al.*, 1999; Møller Neergaard & Mairson, 2004).

The special case of rank-2 intersection types have proved to be an important case with applications to modular flow analysis, dead-code elimination, and typing polymorphic recursion, local definitions, conditionals and pattern matching (Damiani & Prost, 1998; Damiani, 2003; Banerjee & Jensen, 2003; Damiani, 2007).

System F, the polymorphic typed λ-calculus (Reynolds, 1974; Girard *et al.*, 1989), has an undecidable Curry-style inference problem (Wells, 1999). Partial inference in a Church-

[26] This presentation closely follows the informal presentation of intersection types in Chapter 4 of (Møller Neergaard, 2004).

style system is investigated by (Boehm, 1985; Pfenning, 1993) and Pfenning's result shows even partial inference for a simple predicative fragment is undecidable.

The quantifier-elimination approach to proving lower bounds was extended to System F_ω by (Henglein & Mairson, 1991). They prove a sequence of lower bounds on recognizing the System F_k-typable terms, where the bound for F_{k+1} is exponentially larger than that for F_k. This is analogous to intersection quantifier elimination via complete developments at the term level. The essence of (Henglein & Mairson, 1991) is to compute developments at the kind level to shift from System F_{k+1} to System F_k typability. This technique led to lower bounds on System F_i and the non-elementary bound on System F_ω (Henglein & Mairson, 1991). (Urzyczyn, 1997) showed Curry-style inference for System F_ω is undecidable.

There are some interesting open complexity problems in the realm of type inference and quantifier elimination. Bounded polymorphic recursion has recently been investigated (Comini *et al.*, 2008), and is decidable but with unknown complexity bounds, nor quantifier elimination procedures. Typed Scheme (Tobin-Hochstadt & Felleisen, 2008), uses explicit annotations, but with partial inference and flow sensitivity. It includes intersection rules for function types. Complexity bounds on type checking and partial inference are unknown.

The simple algorithm of (Wand, 1987), which generates constraints for type reconstruction, can also be seen as compiler for the linear λ-calculus. It compiles a linear term into a "machine language" of first-order constraints of the form $a = b$ and $c = d \to e$. This machine language is the computational analog of logic's own low-level machine language for first-order propositional logic, the machine-oriented logic of (Robinson, 1965).

Unifying these constraints effectively runs the machine language, evaluating the original program, producing an answer in the guise of a solved form of the type, which is isomorphic to the normal form of the program.

Viewed from this perspective, this is an instance of normalization-by-evaluation for the linear λ-calculus. A linear term is mapped into the domain of first-order logic, where unification is used to evaluate to a canonical solved form, which can be mapped to the normal form of the term. Constraint-based formulations of monovariant flow analyses analogously can be seen as instances of *weak* normalization-by-evaluation functions for the linear λ-calculus.

15 Contributions

Flow analysis is a fundamental static analysis of functional, object-oriented, and other higher-order programming languages; it is a ubiquitous and much-studied component of compiler technology with nearly thirty years of research on the topic. This dissertation has investigated the computational complexity of flow analysis in higher-order programming languages, yielding novel insights into the fundamental limitations on the cost of performing flow analysis.

Monovariant flow analysis, such as 0CFA, is complete for polynomial time. Moreover, many further approximations to 0CFA from the literature, such as Henglein's simple closure analysis, remain complete for polynomial time. These theorems rely on the fact that when a program is linear (each bound variable occurs exactly once), the analysis

makes no approximation; abstract and concrete interpretation coincide. More generally, we conjecture *any* abstract and concrete interpretation will have some sublanguage of coincidence, and this sublanguage may be useful in proving lower bounds.

The linear λ-calculus has been identified as an important language subset to study in order to understand flow analysis. Linearity is an equalizer among variants of static analysis, and a powerful tool in proving lower bounds. Analysis of linear programs coincide under both equality and inclusion-based flow constraints, and moreover, concrete and abstract interpretation coincide for this core language. The inherently sequential nature of flow analysis can be understood as a consequence of a lack of abstraction on this language subset.

Since linearity plays such a fruitful role in the study of program analysis, we developed connections with linear logic and the technology of sharing graphs. Monovariant analysis can be formulated graphically, and the technology of graph reduction and optimal evaluation can be applied to flow analysis. The explicit control representation of sharing graphs makes it easy to extend flow analysis to languages with first-class control.

Simply-typed, η-expanded programs have a potentially simpler 0CFA problem, which is complete for logarithmic space. This discovery is based on analogies with proof normalization for multiplicative linear logic with *atomic* axioms.

Shivers' polyvariant kCFA, for any $k > 0$, is complete for deterministic exponential time. This theorem validates empirical observations that such control flow analysis is intractable. A fairly straightforward calculation shows that kCFA can be computed in exponential time. We show that the naive algorithm is essentially the best one. There is, in the worst case—and plausibly, in practice—no way to tame the cost of the analysis. Exponential time is required.

Collectively, these results provide general insight into the complexity of abstract interpretation and program analysis.

16 Future Work

We end by outlining some new directions and open problems worth pursuing, in approximately ascending order of ambition and import.

16.1 Completing the Pointer Analysis Complexity Story

Compared with flow analysis, pointer analysis has received a much more thorough complexity investigation. A series of important refinements have been made by (Landi & Ryder, 1991; Landi, 1992a; Landi, 1992b; Choi *et al.*, 1993; Horwitz, 1997; Muth & Debray, 2000; Chatterjee *et al.*, 2001; Chakaravarthy, 2003), yet open problems persist. (Chakaravarthy, 2003) leaves open the lower bound on the complexity of pointer analysis with well-defined types with less than two levels of dereference. We believe our insights into linearity and circuit construction can lead to an answer to this remaining problem.

16.2 Polyvariant, Polynomial Flow Analyses

To echo the remark of (Bravenboer & Smaragdakis, 2009), only adapted to the setting of flow analysis rather than pointer analysis, there is an ongoing search for polyvariant, or

context-sensitive, analyses that provide precise flow information without causing massive redundant computation. There has been important work in this area (Jagannathan & Weeks, 1995; Nielson & Nielson, 1997), but the landscape of tractable, context-sensitive flow analyses is mostly open and in need of development.

The ingredients, detailed in section 4, that combine to make kCFA hard, when $k > 0$, should provide guidance in designing new abstractions that avoid computationally expensive components of analysis. A lesson learned has been that *closures*, as they exist when $k > 0$, result in an exponential value space that can be harnessed for the EXPTIME lower-bound construction. It should be possible to design alternative closure abstractions while remaining both polyvariant and polynomial (more below).

16.3 An Expressive Hierarchy of Flow Analyses

From the perspective of computational complexity, the kCFA hierarchy is flat (for any fixed k, kCFA is in EXPTIME; see subsection 4.2). On the other hand, there are far more powerful analyses such as those of (Burn *et al.*, 1985) and (Mossin, 1998). How can we systematically bridge the gap between these analyses to obtain a real expressivity hierarchy?

Flow analyses based on rank-bounded intersection types offers one approach. It should also be possible to design such analyses by composing notions of precise but bounded computation—such as partial evaluation or a series of complete developments—followed by course analysis of residual programs. The idea is to stage analysis into two phases: the first eliminates the need for polyvariance in analysis by transforming the original program into an equivalent, potentially larger, residual program. The subsequent stage performs a course (monovariant) analysis of the residual program. By staging the analysis in this manner—first computing a precise but bounded program evaluation, *then* an imprecise evaluation approximation—the "ever-worsening feedback loop" (Might & Shivers, 2006b) is avoided. By using a sufficiently powerful notion of bounded evaluation, it should be possible to construct flow analyses that form a true hierarchy from a complexity perspective. By using a sufficiently weak notion of bounded evaluation, it should be possible to construct flow analyses that are arbitrarily polyvariant, but computable in polynomial time.

16.4 Truly Subcubic Inclusion-Based Flow Analysis

This dissertation has focused on lower bounds, however recent upper bound improvements have been made on the "cubic bottleneck" of inclusion-based flow analyses such as 0CFA (Midtgaard & Van Horn, 2009). These results have shown known set compression techniques can be applied to obtain direct 0CFA algorithms that run in $O(n^3/\log n)$ time on a unit cost random-access memory model machine. While these results do provide a logarithmic improvement, it is natural to wonder if there is a $O(n^c)$ algorithm for 0CFA and related analyses, where $c < 3$.

At the same time, there have been recent algorithmic breakthroughs on the all-pairs shortest path problem resulting in truly subcubic algorithms. Perhaps the graphical formulation of flow analysis from section 3 can be adapted to exploit these breakthroughs.

16.5 Toward a Fundamental Theorem of Static Analysis

A theorem due to (Statman, 1979) says this: let **P** be a property of simply-typed λ-terms that we would like to detect by static analysis, where **P** is invariant under reduction (normalization), and is computable in elementary time (polynomial, or exponential, or doubly-exponential, or...). Then **P** is a *trivial* property: for any type τ, **P** is satisfied by *all* or *none* of the programs of type τ. (Henglein & Mairson, 1991) have complemented these results, showing that if a property is invariant under β-reduction for a class of programs that can encode all Turing Machines solving problems of complexity class F using reductions from complexity class G, then any superset is either F-complete or trivial. Simple typability has this property for linear and linear affine λ-terms (Henglein & Mairson, 1991; Mairson, 2004), and these terms are sufficient to code all polynomial-time Turing Machines.

We would like to prove some analogs of these theorems, with or without the typing condition, but weakening the condition of "invariant under reduction" to "invariant under abstract interpretation."

References

Aho, Alfred V., Hopcroft, John E., & Ullman, Jeffrey D. (1968). Time and tape complexity of pushdown automaton languages. *Information and control*, **13**(3), 186–206.

Alpuente, María, & Vidal, Germán (eds). (2008). *Static analysis, 15th international symposium, sas 2008, valencia, spain, july 16-18, 2008. proceedings*. Lecture Notes in Computer Science, vol. 5079. Springer.

Amtoft, Torben, & Turbak, Franklyn A. (2000). Faithful translations between polyvariant flows and polymorphic types. *Pages 26–40 of: Esop '00: Proceedings of the 9th european symposium on programming languages and systems*. London, UK: Springer-Verlag.

Andersen, Lars Ole. 1994 (May). *Program analysis and specialization for the C programming language*. Ph.D. thesis, DIKU, University of Copenhagen.

Ashley, J. Michael, & Dybvig, R. Kent. (1998). A practical and flexible flow analysis for higher-order languages. *Acm trans. program. lang. syst.*, **20**(4), 845–868.

Asperti, Andrea, & Laneve, Cosimo. (1995). Paths, computations and labels in the λ-calculus. *Theor. comput. sci.*, **142**(2), 277–297.

Asperti, Andrea, & Mairson, Harry G. (1998). Parallel beta reduction is not elementary recursive. *In:* (POPL 1998, 1998).

Ayers, Andrew Edward. (1993). *Abstract analysis and optimization of Scheme*. Ph.D. thesis, Cambridge, Massachusetts, USA.

Bacon, David F., & Sweeney, Peter F. (1996). Fast static analysis of C++ virtual function calls. *In:* (OOPSLA 1996, 1996).

Banerjee, Anindya. (1997). A modular, polyvariant and type-based closure analysis. *In:* (Berman, 1997).

Banerjee, Anindya, & Jensen, Thomas. (2003). Modular control-flow analysis with rank 2 intersection types. *Mathematical. structures in comp. sci.*, **13**(1), 87–124.

Basch, Julien, Khanna, Sanjeev, & Motwani, Rajeev. (1995). *On diameter verification and Boolean matrix multiplication.* Tech. rept. Stanford University, Stanford, California, USA.

Ben-Amram, Amir M., & Lee, Chin Soon. (2007). Program termination analysis in polynomial time. *Acm trans. program. lang. syst.*, **29**(1), 5.

Berman, A. Michael (ed). (1997). *Icfp '97: Proceedings of the second acm sigplan international conference on functional programming*. New York, New York, USA: ACM.

Biswas, Sandip K. (1997). A demand-driven set-based analysis. *In:* (POPL 1997, 1997).

Boehm, Hans-J. (1985). Partial polymorphic type inference is undecidable. *Pages 339–345 of: Sfcs '85: Proceedings of the 26th annual symposium on foundations of computer science (sfcs 1985)*. Washington, DC, USA: IEEE Computer Society.

Bravenboer, Martin, & Smaragdakis, Yannis. 2009 (Oct.). Strictly declarative specification of sophisticated points-to analyses. *Oopsla '09: Proceedings of the 24th annual acm sigplan conference on object-oriented programming, systems, languages, and applications*. To appear.

Burn, G. L., Hankin, C. L., & Abramsky, S. (1985). The theory of strictness analysis for higher order functions. *Pages 42–62 of:* Ganzinger, H., & Jones, N. (eds), *Programs as data objects*. New York, New York, USA: Springer-Verlag.

Cardone, Felice, & Hindley, J. Roger. (2006). *History of lambda-calculus and combinatory logic*. Tech. rept. MRRS-05-06. Swansea University Mathematics Department Research Report. To appear in *Handbook of the History of Logic, Volume 5*, D. M. Gabbay and J. Woods, editors.

Chakaravarthy, Venkatesan T. (2003). New results on the computability and complexity of points–to analysis. *Pages 115–125 of: Popl '03: Proceedings of the 30th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

Chakaravarthy, Venkatesan T., & Horwitz, Susan. (2002). On the non-approximability of points-to analysis. *Acta informatica*, **38**(8), 587–598.

Chambers, Craig, & Ungar, David. (1990). Interactive type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. *Pages 150–164 of: Pldi '90: Proceedings of the acm sigplan 1990 conference on programming language design and implementation*. New York, New York, USA: ACM.

Chan, Timothy M. (2007). More algorithms for all-pairs shortest paths in weighted graphs. *Pages 590–598 of: Stoc '07: Proceedings of the thirty-ninth annual acm symposium on theory of computing*. New York, New York, USA: ACM.

Chatterjee, Ramkrishna, Ryder, Barbara G., & Landi, William A. (2001). Complexity of points-to analysis of Java in the presence of exceptions. *Ieee trans. softw. eng.*, **27**(6), 481–512.

Chaudhuri, Swarat. (2008). Subcubic algorithms for recursive state machines. *In:* (POPL 2008, 2008).

Choi, Jong-Deok, Burke, Michael, & Carini, Paul. (1993). Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *In:* (POPL 1993, 1993).

Comini, Marco, Damiani, Ferruccio, & Vrech, Samuel. (2008). On polymorphic recursion, type systems, and abstract interpretation. *In:* (Alpuente & Vidal, 2008).

Cousot, Patrick, & Cousot, Radhia. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Pages 238–252 of: Popl '77: Proceedings of the 4th acm sigact-sigplan symposium on principles of programming languages*. New York, New York, USA: ACM.

Cousot, Patrick, & Cousot, Radhia. (1992). Abstract interpretation frameworks. *Journal of logic and computation*, **2**(4), 511–547.

Curry, Haskell B. (1969). Modified basic functionality in combinatory logic. *Dialectica*, **23**(2), 83–92.

Damas, Luis. (1985). *Type assignment in programming languages*. Ph.D. thesis, University of Edinburgh, Department of Computer Science. Technical Report CST- 33-85.

Damian, Daniel, & Danvy, Olivier. (2003). CPS transformation of flow information, Part II: administrative reductions. *J. funct. program.*, **13**(5), 925–933.

Damiani, Ferruccio. (2003). Rank 2 intersection types for local definitions and conditional expressions. *Acm trans. program. lang. syst.*, **25**(4), 401–451.

Damiani, Ferruccio. (2007). Rank 2 intersection for recursive definitions. *Fundam. inf.*, **77**(4), 451–488.

Damiani, Ferruccio, & Prost, Frédéric. (1998). Detecting and removing dead-code using rank 2 intersection. *Pages 66–87 of: Types '96: Selected papers from the international workshop on types for proofs and programs*. London, UK: Springer-Verlag.

Danvy, Olivier, & Filinski, Andrzej. (1990). Abstracting control. *Pages 151–160 of: Lfp '90: Proceedings of the 1990 acm conference on lisp and functional programming*. New York, New York, USA: ACM Press.

Danvy, Olivier, r, Karoline Malmkjæ, & Palsberg, Jens. (1996). Eta-expansion does the trick. *Acm trans. program. lang. syst.*, **18**(6), 730–751.

DeFouw, Greg, Grove, David, & Chambers, Craig. (1998). Fast interprocedural class analysis. *In:* (POPL 1998, 1998).

Di Cosmo, Roberto, Kesner, Delia, & Polonovski, Emmanuel. (2003). Proof nets and explicit substitutions. *Mathematical. structures in comp. sci.*, **13**(3), 409–450.

Diwan, Amer, Moss, J. Eliot B., & McKinley, Kathryn S. (1996). Simple and effective analysis of statically-typed object-oriented programs. *In:* (OOPSLA 1996, 1996).

Dwork, Cynthia, Kanellakis, Paris C., & Mitchell, John C. (1984). On the sequential nature of unification. *J. log. program.*, **1**(1), 35–50.

Emami, Maryam, Ghiya, Rakesh, & Hendren, Laurie J. (1994). Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Pages 242–256 of: Pldi '94: Proceedings of the acm sigplan 1994 conference on programming language design and implementation*. New York, New York, USA: ACM.

Faxén, Karl-Filip. (1997). Polyvariance, polymorphism and flow analysis. *Pages 260–278 of: Selected papers from the 5th lomaps workshop on analysis and verification of multiple-agent languages*. London, UK: Springer-Verlag.

Filinski, Andrzej. (1989). Declarative continuations: an investigation of duality in programming language semantics. *Pages 224–249 of: Category theory and computer science*. London, UK: Springer-Verlag.

Gaifman, Haim, Mairson, Harry, Sagiv, Yehoshua, & Vardi, Moshe Y. (1993). Undecidable optimization problems for database logic programs. *J. acm*, **40**(3), 683–713.

Giesl, Jürgen, Swiderski, Stephan, Schneider-Kamp, Peter, & Thiemann, René. (2006). Automated termination analysis for Haskell: From term rewriting to programming languages. *Pages 297–312 of:* Pfenning, Frank (ed), *Rta*. Lecture Notes in Computer Science, vol. 4098. Springer.

Girard, Jean-Yves. (1987). Linear logic. *Theor. comput. sci.*, **50**(1), 1–102.

Girard, Jean-Yves. (1989). Geometry of interaction I: Interpretation of System F. *Pages 221–260 of:* Bonotto, C. (ed), *Logic colloquium '88*. North Holland.

Girard, Jean-Yves, Taylor, Paul, & Lafont, Yves. (1989). *Proofs and types*. New York, New York, USA: Cambridge University Press. Reprinted with corrections 1990.

Gonthier, Georges, Abadi, Martín, & Lévy, Jean-Jacques. (1992). The geometry of optimal lambda reduction. *Pages 15–26 of: Popl '92: Proceedings of the 19th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM Press.

Griffin, Timothy G. (1990). A formulae-as-type notion of control. *In:* (POPL 1990, 1990).

Grove, David, & Chambers, Craig. (2001). A framework for call graph construction algorithms. *Acm trans. program. lang. syst.*, **23**(6), 685–746.

Hankin, Chris, Nagarajan, Rajagopal, & Sampath, Prahladavaradan. (2002). *The essence of computation: complexity, analysis, transformation*. New York, New York, USA: Springer-Verlag New York, Inc. Chap. Flow analysis: games and nets, pages 135–156.

Hecht, Matthew S. (1977). *Flow analysis of computer programs*. New York, New York, USA: Elsevier Science Inc.

Heintze, Nevin. (1994). Set-based analysis of ML programs. *Pages 306–317 of: Lfp '94: Proceedings of the 1994 acm conference on lisp and functional programming*. New York, New York, USA: ACM Press.

Heintze, Nevin, & McAllester, David. (1997a). Linear-time subtransitive control flow analysis. *Pages 261–272 of: Pldi '97: Proceedings of the acm sigplan 1997 conference on programming language design and implementation*. New York, New York, USA: ACM Press.

Heintze, Nevin, & McAllester, David. (1997b). On the complexity of set-based analysis. *In:* (Berman, 1997).

Heintze, Nevin, & McAllester, David. (1997c). On the cubic bottleneck in subtyping and flow analysis. *Page 342 of: Lics '97: Proceedings of the 12th annual ieee symposium on logic in computer science*. Washington, DC, USA: IEEE Computer Society.

Henglein, Fritz. (1990). *Type invariant simulation: A lower bound technique for type inference*. Unpublished manuscript.

Henglein, Fritz. (1991). Efficient type inference for higher-order binding-time analysis. *Pages 448–472 of: Proceedings of the 5th acm conference on functional programming languages and computer architecture*. London, UK: Springer-Verlag.

Henglein, Fritz. 1992 (Mar.). *Simple closure analysis*. Tech. rept. DIKU Semantics Report D-193.

Henglein, Fritz, & Mairson, Harry G. (1991). The complexity of type inference for higher-order lambda calculi. *In:* (POPL 1991, 1991).

Herbrand, Jacques. (1930). *Investigations in proof theory: The properties of true propositions*. Harvard University Press. Chapter 5 of Herbrand's Ph.D. dissertation, *Recherches sur la théorie de la deémonstration*. Pages 525–581.

Hillebrand, Gerd G., Kanellakis, Paris C., Mairson, Harry G., & Vardi, Moshe Y. (1995). Undecidable boundedness problems for datalog programs. *J. logic program.*, **25**(2), 163–190.

Hind, Michael. (2001). Pointer analysis: haven't we solved this problem yet? *Pages 54–61 of: Paste '01: Proceedings of the 2001 acm sigplan-sigsoft workshop on program analysis for software tools and engineering*. New York, New York, USA: ACM.

Hindley, J. Roger. (1969). The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, Dec., 29–60.

Hindley, J. Roger. (1989). BCK-combinators and linear λ-terms have types. *Theor. comput. sci.*, **64**, 97–105.

Hirokawa, Sachio. (1991). Principal type-schemes of BCI-lambda-terms. *Pages 633–650 of: Tacs '91: Proceedings of the international conference on theoretical aspects of computer software*. London, UK: Springer-Verlag.

Hofmann, Martin. (1998). *Type systems for polynomial-time computation*. Ph.D. thesis, TU Darmstadt. Habilitation Thesis.

Hofmann, Martin. (2003). Linear types and non-size-increasing polynomial time computation. *Inf. comput.*, **183**(1), 57–85.

Horwitz, Susan. (1997). Precise flow-insensitive may-alias analysis is NP-hard. *Acm trans. program. lang. syst.*, **19**(1), 1–6.

Howard, William A. (1980). The formulae-as-types notion of construction. *In:* (Seldin & Hindley, 1980).

Jagannathan, Suresh, & Weeks, Stephen. (1995). A unified treatment of flow analysis in higher-order languages. *Pages 393–407 of: Popl '95: Proceedings of the 22nd acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM Press.

Jagannathan, Suresh, Thiemann, Peter, Weeks, Stephen, & Wright, Andrew. (1998). Single and loving it: must-alias analysis for higher-order languages. *In:* (POPL 1998, 1998).

Johnson, Ralph E., Graver, Justin O., & Zurawski, Laurance W. (1988). TS: An optimizing compiler for Smalltalk. *Pages 18–26 of: Oopsla '88: Conference proceedings on object-oriented programming systems, languages and applications*. New York, New York, USA: ACM.

Jones, Neil D. (1981). Flow analysis of lambda expressions (preliminary version). *Pages 114–128 of: Proceedings of the 8th colloquium on automata, languages and programming*. London, UK: Springer-Verlag.

Jones, Neil D., & Bohr, Nina. (2008). Call-by-value termination in the untyped λ-calculus. *Logical methods in computer science*, **4**(1), 1–39.

Jones, Neil D., Gomard, Carsten K., & Sestoft, Peter. (1993). *Partial evaluation and automatic program generation*. Upper Saddle River, New Jersey, USA: Prentice-Hall, Inc.

Kanellakis, Paris C., Mairson, Harry G., & Mitchell, John C. (1991). Unification and ML-type reconstruction. *Pages 444–478 of: Computational logic: Essays in honor of alan robinson*. MIT Press.

Kfoury, Assaf J., Mairson, Harry G., Turbak, Franklyn A., & Wells, J. B. (1999). Relating typability and expressiveness in finite-rank intersection type systems (extended abstract). *Pages 90–101 of: Icfp '99: Proceedings of the fourth acm sigplan international conference on functional programming*. New York, New York, USA: ACM.

Kristiansen, L., & Niggl, K.-H. (2004). On the computational complexity of imperative programming languages. *Theor. comput. sci.*, **318**(1-2), 139–161.

Kuan, George, & MacQueen, David. (2007). Efficient type inference using ranked type variables. *Pages 3–14 of: Ml '07: Proceedings of the 2007 workshop on ml*. New York, New York, USA: ACM.

Ladner, Richard E. (1975). The circuit value problem is log space complete for *P*. *Sigact news*, **7**(1), 18–20.

Lafont, Yves. (1995). From proof-nets to interaction nets. *Pages 225–247 of: Proceedings of the workshop on advances in linear logic*. New York, New York, USA: Cambridge University Press.

Landi, William. (1992a). *Interprocedural aliasing in the presence of pointers*. Ph.D. thesis, Rutgers University.

Landi, William. (1992b). Undecidability of static analysis. *Acm lett. program. lang. syst.*, **1**(4), 323–337.

Landi, William, & Ryder, Barbara G. (1991). Pointer-induced aliasing: a problem taxonomy. *In:* (POPL 1991, 1991).

Lawall, Julia L., & Mairson, Harry G. (2000). Sharing continuations: Proofnets for languages with explicit control. *Pages 245–259 of: Esop '00: Proceedings of the 9th european symposium on programming languages and systems*. London, UK: Springer-Verlag.

Lee, Chin Soon, Jones, Neil D., & Ben-Amram, Amir M. (2001). The size-change principle for program termination. *Pages 81–92 of: Popl '01: Proceedings of the 28th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

Leivant, Daniel. (1993). Stratified functional programs and computational complexity. *In:* (POPL 1993, 1993).

Lévy, Jean-Jacques. 1978 (Jan.). *Réductions correctes et optimales dans le lambda-calcul*. Ph.D. thesis, University of Paris VII. thése d'Etat.

Lévy, Jean-Jacques. (1980). Optimal reductions in the lambda-calculus. *In:* (Seldin & Hindley, 1980).

Mairson, Harry G. (1990). Deciding ML typability is complete for deterministic exponential time. *In:* (POPL 1990, 1990).

Mairson, Harry G. (1992a). Quantifier elimination and parametric polymorphism in programming languages. *J. functional programming*, **2**, 213–226.

Mairson, Harry G. (1992b). A simple proof of a theorem of Statman. *Theor. comput. sci.*, **103**(2), 387–394.

Mairson, Harry G. (2002). From Hilbert spaces to Dilbert spaces: Context semantics made simple. *Pages 2–17 of: Fst tcs '02: Proceedings of the 22nd conference kanpur on foundations of software technology and theoretical computer science*. London, UK: Springer-Verlag.

Mairson, Harry G. (2004). Linear lambda calculus and PTIME-completeness. *J. functional program.*, **14**(6), 623–633.

Mairson, Harry G. (2006a). *Axiom-sensitive normalization bounds for multiplicative linear logic*. Unpublished manuscript.

Mairson, Harry G. 2006b (Feb.). MLL normalization and transitive closure: circuits, complexity, and Euler tours. *Geocal (geometry of calculation): Implicit computational complexity. institut de mathematique de luminy, marseille.*

Mairson, Harry G., & Terui, Kazushige. (2003). On the computational complexity of cut-elimination in linear logic. *Pages 23–36 of:* Blundo, Carlo, & Laneve, Cosimo (eds), *Theoretical computer science, 8th italian conference, ictcs 2003, bertinoro, italy, october 13-15, 2003, proceedings*. Lecture Notes in Computer Science, vol. 2841. Springer.

Martin, John C. (1997). *Introduction to languages and the theory of computation*. McGraw-Hill Higher Education.

McAllester, David. (2002). On the complexity analysis of static analyses. *J. acm*, **49**(4), 512–537.

Melski, David, & Reps, Thomas. (2000). Interconvertibility of a class of set constraints and context-free-language reachability. *Theor. comput. sci.*, **248**(1-2), 29–98.

Midtgaard, Jan. 2007 (Dec.). *Control-flow analysis of functional programs*. Technical Report BRICS RS-07-18. DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark. To appear in revised form in ACM Computing Surveys.

Midtgaard, Jan, & Jensen, Thomas. (2008). A calculational approach to control-flow analysis by abstract interpretation. *In:* (Alpuente & Vidal, 2008).

Midtgaard, Jan, & Jensen, Thomas. (2009). Control-flow analysis of function calls and returns by abstract interpretation. *Icfp '09: Proceedings of the acm sigplan international conference on functional programming*. To appear. Extended version available as INRIA research report RR-6681.

Midtgaard, Jan, & Van Horn, David. 2009 (May). *Subcubic control-flow analysis algorithms*. Tech. rept. 125. Roskilde University, Denmark. To appear in the Symposium in Honor of Mitchell Wand.

Might, Matthew. (2007). *Environment analysis of higher-order languages*. Ph.D. thesis, Georgia Institute of Technology, Atlanta, Georgia. Adviser-Olin G. Shivers.

Might, Matthew, & Shivers, Olin. (2006a). Environment analysis via ΔCFA. *Pages 127–140 of: Popl '06: Conference record of the 33rd acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

Might, Matthew, & Shivers, Olin. 2006b (September). Improving flow analyses via ΓCFA: Abstract garbage collection and counting. *Pages 13–25 of: Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*.

Milanova, Ana, Rountev, Atanas, & Ryder, Barbara G. (2005). Parameterized object sensitivity for points-to analysis for Java. *Acm trans. softw. eng. methodol.*, **14**(1), 1–41.

Mitchell, John C. (1990). Type systems for programming languages. Cambridge, Massachusetts, USA: MIT Press.

Møller Neergaard, Peter. 2004 (October). *Complexity aspects of programming language design: From logspace to elementary time via proofnets and intersection types*. Ph.D. thesis, Brandeis University, Waltham, Massachusetts, USA.

Møller Neergaard, Peter, & Mairson, Harry G. (2004). Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. *Pages 138–149 of: Icfp '04: Proceedings*

of the ninth acm sigplan international conference on functional programming. New York, New York, USA: ACM Press.

Mossin, Christian. (1997a). Exact flow analysis. *Pages 250–264 of: Sas '97: Proceedings of the 4th international symposium on static analysis*. London, UK: Springer-Verlag.

Mossin, Christian. 1997b (January (revised August)). *Flow analysis of typed higher-order programs*. Ph.D. thesis, DIKU, University of Copenhagen.

Mossin, Christian. (1998). Higher-order value flow graphs. *Nordic j. of computing*, **5**(3), 214–234.

Muchnick, Steven S., & Jones, Neil D. (eds). (1981). *Program flow analysis: Theory and applications*. Prentice Hall.

Muth, Robert, & Debray, Saumya. (2000). On the complexity of flow-sensitive dataflow analyses. *Pages 67–80 of: Popl '00: Proceedings of the 27th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

Myers, Eugene M. (1981). A precise inter-procedural data flow algorithm. *Pages 219–230 of: Popl '81: Proceedings of the 8th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

Neal, Radford. 1989 (Dec.). *The computational complexity of taxonomic inference*. Unpublished manuscript. `ftp://ftp.cs.utoronto.ca/pub/radford/taxc.ps`.

Nielson, Flemming, & Nielson, Hanne Riis. (1997). Infinitary control flow analysis: a collecting semantics for closure analysis. *In:* (POPL 1997, 1997).

Nielson, Flemming, Nielson, Hanne Riis, & Hankin, Chris. (1999). *Principles of program analysis*. Secaucus, New Jersey, USA: Springer-Verlag New York, Inc.

OOPSLA 1996. (1996). *Oopsla '96: Proceedings of the 11th acm sigplan conference on object-oriented programming, systems, languages, and applications*. New York, New York, USA: ACM.

Oxhøj, Nicholas, Palsberg, Jens, & Schwartzbach, Michael I. (1992). Making type inference practical. *Pages 329–349 of: Ecoop '92: Proceedings of the european conference on object-oriented programming*. London, UK: Springer-Verlag.

Palsberg, Jens. (1995). Closure analysis in constraint form. *Acm trans. program. lang. syst.*, **17**(1), 47–62.

Palsberg, Jens, & Pavlopoulou, Christina. (2001). From polyvariant flow information to intersection and union types. *J. funct. program.*, **11**(3), 263–317.

Palsberg, Jens, & Schwartzbach, Michael I. (1991). Object-oriented type inference. *Pages 146–161 of: Oopsla '91: Conference proceedings on object-oriented programming systems, languages, and applications*. New York, New York, USA: ACM.

Palsberg, Jens, & Schwartzbach, Michael I. (1995). Safety analysis versus type inference. *Inf. comput.*, **118**(1), 128–141.

Papadimitriou, Christos H. (1994). *Computational complexity*. Reading, Massachusetts, USA: Addison-Wesley.

Parigot, Michel. (1992). $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. *Pages 190–201 of: Logic programming and automated reasoning,international conference lpar'92, st. petersburg, russia, july 15-20, 1992, proceedings*. Lecture Notes in Computer Science, vol. 624. Springer.

Pfenning, Frank. (1993). On the undecidability of partial polymorphic type reconstruction. *Fundam. inf.*, **19**(1-2), 185–199.

POPL 1990. (1990). *Proceedings of the 17th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

POPL 1991. (1991). *Proceedings of the 18th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

POPL 1993. (1993). *Proceedings of the 20th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

POPL 1997. (1997). *Proceedings of the 24th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

POPL 1998. (1998). *Proceedings of the 25th acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

POPL 2008. (2008). *Proceedings of the 35th annual acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

Ramalingam, G. (1994). The undecidability of aliasing. *Acm trans. program. lang. syst.*, **16**(5), 1467–1471.

Regnier, Laurent. (1992). *Lambda calcul et réseaux*. Ph.D. thesis, University of Paris VII.

Reps, Thomas W. (1996). On the sequential nature of interprocedural program-analysis problems. *Acta informatica*, **33**(8), 739–757.

Reynolds, John C. (1974). Towards a theory of type structure. *Pages 408–423 of: Programming symposium, proceedings colloque sur la programmation*. London, UK: Springer-Verlag.

Rice, Henry G. (1953). Classes of recursively enumerable sets and their decision problems. *Trans. amer. math. soc.*, **74**, 358–366.

Rinetzky, N., Ramalingam, G., Sagiv, M., & Yahav, E. (2008). On the complexity of partially-flow-sensitive alias analysis. *Acm trans. program. lang. syst.*, **30**(3), 1–28.

Robinson, J. Alan. (1965). A machine-oriented logic based on the resolution principle. *J. acm*, **12**(1), 23–41.

Rytter, Wojciech. (1985). Fast recognition of pushdown automaton and context-free languages. *Inf. control*, **67**(1-3), 12–22.

Seldin, Jonathan P., & Hindley, J. Roger (eds). (1980). *To h. b. curry: Essays on combinatory logic, lambda calculus and formalism*. London: Academic Press.

Sereni, Damien. (2007). Termination analysis and call graph construction for higher-order functional programs. *Pages 71–84 of:* Hinze, Ralf, & Ramsey, Norman (eds), *Icfp '07: Proceedings of the 12th acm sigplan international conference on functional programming, freiburg, germany, october 1–3, 2007*. New York, New York, USA: ACM.

Sereni, Damien, & Jones, Neil D. (2005). Termination analysis of higher-order functional programs. *Pages 281–297 of:* Yi, Kwangkeun (ed), *Programming languages and systems, third asian symposium, aplas 2005, tsukuba, japan, november 2-5, 2005, proceedings*. Lecture Notes in Computer Science, vol. 3780. Springer.

Sestoft, Peter. 1988 (Oct). *Replacing function parameters by global variables*. M.Phil. thesis, DIKU, University of Copenhagen, Denmark. Master's thesis no. 254.

Sestoft, Peter. (1989). Replacing function parameters by global variables. *Pages 39–53 of: Fpca '89: Proceedings of the fourth international conference on functional programming languages and computer architecture*. New York, New York, USA: ACM Press.

Shivers, Olin. (1988). Control flow analysis in Scheme. *Pages 164–174 of: Pldi '88: Proceedings of the acm sigplan 1988 conference on programming language design and implementation*. New York, New York, USA: ACM.

Shivers, Olin. (1991). *Control-flow analysis of higher-order languages*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Shivers, Olin. (2004). Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. vol. 39. New York, New York, USA: ACM.

Sørensen, Morten Heine, & Urzyczyn, Pawel. (2006). *Lectures on the curry-howard isomorphism*. Studies in Logic and the Foundations of Mathematics, vol. 149. New York, New York, USA: Elsevier Science Inc.

Spoto, Fausto, & Jensen, Thomas. (2003). Class analyses as abstract interpretations of trace semantics. *Acm trans. program. lang. syst.*, **25**(5), 578–630.

Statman, Richard. (1979). The typed λ-calculus is not elementary recursive. *Theor. comput. sci.*, **9**, 73–81.

Steckler, Paul A., & Wand, Mitchell. (1997). Lightweight closure conversion. *Acm trans. program. lang. syst.*, **19**(1), 48–86.

Steensgaard, Bjarne. (1996). Points-to analysis in almost linear time. *Pages 32–41 of: Popl '96: Proceedings of the 23rd acm sigplan-sigact symposium on principles of programming languages*. New York, New York, USA: ACM.

Terui, Kazushige. 2002 (July). *On the complexity of cut-elimination in linear logic*. Invited talk at LL2002 (LICS2002 affiliated workshop), Copenhagen.

Tiuryn, Jerzy. (1990). Type inference problems: a survey. *Pages 105–120 of: Mfcs '90: Proceedings on mathematical foundations of computer science 1990*. New York, New York, USA: Springer-Verlag New York, Inc.

Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2008). The design and implementation of Typed Scheme. *In:* (POPL 2008, 2008).

Ullman, Jeffrey D., & van Gelder, Allen. (1986). Parallel complexity of logical query programs. *Pages 438–454 of: Sfcs '86: Proceedings of the 27th annual symposium on foundations of computer science (sfcs 1986)*. Washington, DC, USA: IEEE Computer Society.

Urzyczyn, Paweł. (1997). Type reconstruction in F_ω. *Mathematical. structures in comp. sci.*, **7**(4), 329–358.

van Bakel, Steffen. (1992). Complete restrictions of the intersection type discipline. *Theor. comput. sci.*, **102**(1), 135–163.

Van Horn, David, & Mairson, Harry G. (2008). Flow analysis, linearity, and PTIME. *In:* (Alpuente & Vidal, 2008).

Vassilevska, Virginia. (2009). Efficient algorithms for clique problems. *Inf. process. lett.*, **109**(4), 254–257.

Vitek, Jan, Horspool, R. Nigel, & Uhl, James S. (1992). Compile-time analysis of object-oriented programs. *Pages 236–250 of: Cc '92: Proceedings of the 4th international conference on compiler construction*. London, UK: Springer-Verlag.

Wand, Mitchell. (1987). A simple algorithm and proof for type inference. *Fundam. inform.*, **10**, 115–122.

Wells, J. B. (1999). Typability and type checking in System F are equivalent and undecidable. *Annals of pure and applied logic*, **98**, 111–156.

Wells, J. B., Dimock, Allyn, Muller, Robert, & Turbak, Franklyn. (2002). A calculus with polymorphic and polyvariant flow types. *J. funct. program.*, **12**(3), 183–227.

Wilson, Robert P., & Lam, Monica S. (1995). Efficient context-sensitive pointer analysis for C programs. *Pages 1–12 of: Pldi '95: Proceedings of the acm sigplan 1995 conference on programming language design and implementation*. New York, New York, USA: ACM.

Wright, Andrew K., & Jagannathan, Suresh. (1998). Polymorphic splitting: an effective polyvariant flow analysis. *Acm trans. program. lang. syst.*, **20**(1), 166–207.

Zwick, Uri. (2006). A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. *Algorithmica*, **46**(2), 181–192.