Flow analysis, linearity, and PTIME

David Van Horn and Harry G. Mairson

Department of Computer Science
Brandeis University
Waltham, Massachusetts 02454
{dvanhorn,mairson}@cs.brandeis.edu

Abstract. We examine the computational complexity of flow analyses that approximate Shivers' 0CFA by trading precision for faster computation. As a motivating example, we consider Henglein's simple closure analysis, which forfeits the notion of directionality in flows and enjoys an "almost linear" time algorithm, whereas the best known algorithm for 0CFA is cubic. We show that despite near linear time computability, simple closure analysis, like 0CFA, is complete for PTIME.

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 subpolynomial algorithm for these problems would require (unlikely) breakthrough results in complexity, such as PTIME = LOGSPACE.

1 Introduction

Flow analysis [1–3] is concerned with providing sound approximations to the question of "does a given value flow into a given program point during evaluation?" The most approximate analysis will answer *yes* everywhere, which takes no resources to compute—and is useless. On the other hand, the most precise analysis will answer *yes* if and only if the given value flows into the program point during evaluation, which is useful, albeit uncomputable. So every static analysis necessarily gives up valuable information for the sake of computing an answer within bounded resources. Designing a static analyzer, therefore, requires making trade-offs between precision and complexity. But what exactly is the analytic relationship between forfeited information and resource usage for any given design decision? In other words:

What are the computationally potent ingredients in a static analysis?

The best known algorithms to compute Shivers' 0CFA [4], a canonical flow analysis for higher-order programs, are cubic in the size of the program, and there is strong evidence to suggest that in general, this cannot be improved [5]. But if we are willing to give up information in the service of quickly computing a—necessarily less precise—analysis, we can avoid the "cubic bottleneck". For example, if we are willing to give up the *direction* of flows, as in Henglein's simple closure analysis [6], we can enjoy algorithms that run in near linear time. Likewise, if we explicitly bound the number of passes over the program the analyzer is allowed, as in Ashley and Dybvig's sub-0CFA [7], 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 sub-cubic analyses *require*—and are therefore complete for—polynomial time, just like 0CFA [8], despite algorithms that compute them in linear or almost linear time.

What is flow analysis? Flow analysis is a fundamental and ubiquitous static analysis of higher-order programs. It answers fundamental questions such as *what values can a given subexpression possibly evaluate to at run-time?* Each subexpression is identified with a unique superscripted *label* ℓ, which serves to index that program point. The result of a flow analysis is a *cache* $\widehat{\mathsf{C}}$ that maps program points (and variables) to sets of values. These analyses are *conservative* in the following sense: if v is in $\widehat{\mathsf{C}}(\ell)$, then the subexpression label ℓ *may* evaluate to v when the program is run (likewise, if v is in $\widehat{\mathsf{C}}(x)$, x may be bound to v when the program is run). But if v is *not* in $\widehat{\mathsf{C}}(\ell)$, we know that e *cannot* evaluate to v and conversely if e evaluates to v, v *must* be in $\widehat{\mathsf{C}}(\ell)$.

For the purposes of this paper and all of the analyses considered herein, values are (possibly open) lambda abstractions. During evaluation, functional values are denoted with *closures*—a lambda abstraction together with an *environment* that closes it. Values considered in the analysis approximate run-time denotations in the sense that if a subexpression labeled ℓ evaluates to the closure $\langle \lambda x.e, \rho \rangle$, then $\lambda x.e$ is in $\widehat{\mathsf{C}}(\ell)$.

The *acceptability* of a flow analysis is often specified as a set of (in)equations on program fragments. The most naive way to compute a satisfying analysis is to initialize the cache with the flow sets being empty everywhere. Successive passes are then made over the program, monotonically updating the cache as needed, until the least fixed point is reached. The more approximate the analysis, the faster this algorithm converges on a fixed point. The key to a fruitful analysis, then, is [7] "to accelerate the analysis without losing too much information."

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*, and this paper serves to demonstrate that this hammer hits several of the most approximate flow analyses that exist in the literature.

Linearity and approximation in static analysis: Linearity, the Curry-Howard programming counterpart of linear logic [9], plays an important role in understanding static analyses. The reason is straightforward: because static analysis has to be tractable, it typically approximates normalization, instead of simulating it, because running the program may take too long. For example, in the analysis of simple types—surely a kind of static analysis—the approximation is that all occurrences of a bound variable must have the same type. (As a consequence, perfectly good programs are rejected). A comparable but not identical thing happens in the case of type inference for ML and bounded-rank intersection types—but note that when the program is linear, there is no approximation, and type inference becomes evaluation under another name.

In the case of flow analysis, similarly, a cache is computed in the course of an approximate evaluation, which is only an approximation because each evaluation of an abstraction body causes the same piece of the cache to be (monotonically) updated. Again, if the term is linear, then there is only one evaluation of the abstraction body, and the flow analysis becomes synonymous with normalization.

Organization of this paper: The next section introduces Shivers' 0CFA in order to provide intuitions and a point of reference for comparisons with subsequent analyses. Section 3 specifies and provides an algorithm for computing Henglein's simple closure analysis. Section 4 develops a correspondence between evaluation and analysis for linear programs. It is shown that, when the program is linear, normalization and analysis are synonymous. As a consequence the normal form of a program can be *read back* from the least analysis. We then show in Section 5.1 how to simulate circuits, the canonical PTIME-hard problem, using linear terms. This establishes the PTIME-hardness of the analysis. Finally, we discuss other sub-cubic analyses and sketch why these analyses remain complete for PTIME and provide conclusions and perspective.

2 Shivers' 0CFA

As a starting point, we consider Shivers *zeroth order control flow analysis* [4].

The Language: A countably infinite set of labels (**Lab**) is assumed, where the set of variable names (**Var**) are included in **Lab**. The syntax of the language is given by the following grammar:

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

All of the syntactic categories are implicitly understood to be restricted to the finite set of terms, labels, variables, etc. that occur in the *program of interest*—the program being analyzed. The set of labels, which includes variable names, in a program fragment is denoted $\mathbf{lab}(e)$. As a convention, programs are assumed to have distinct bound variable names.

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:

$$\widehat{\mathsf{C}} \in \mathbf{Lab} \to \mathcal{P}(\mathbf{Term}) \qquad \text{abstract caches}$$

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, *et al.* [10]. Each subexpression is identified with a unique superscripted *label* ℓ, which marks that program point; $\widehat{\mathsf{C}}(\ell)$ stores all possible values flowing to point ℓ. An *acceptable* control flow analysis for an expression e is written $\widehat{\mathsf{C}} \models e$:

$$\begin{aligned}
&\widehat{\mathsf{C}} \models x^{\ell} \text{ iff } \widehat{\mathsf{C}}(x) \subseteq \widehat{\mathsf{C}}(\ell)\\
&\widehat{\mathsf{C}} \models (\lambda x.e)^{\ell} \text{ iff } \lambda x.e \in \widehat{\mathsf{C}}(\ell)\\
&\widehat{\mathsf{C}} \models (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell} \text{ iff } \widehat{\mathsf{C}} \models t_1^{\ell_1} \wedge \widehat{\mathsf{C}} \models t_2^{\ell_2} \wedge \forall \lambda x.t_0^{\ell_0} \in \widehat{\mathsf{C}}(\ell_1) :\\
&\qquad \widehat{\mathsf{C}} \models t_0^{\ell_0} \wedge \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(x) \wedge \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell)
\end{aligned}$$

The $\models$ relation needs to be coinductively defined since verifying a judgment $\widehat{\mathsf{C}} \models e$ may obligate verification of $\widehat{\mathsf{C}} \models e'$ which in turn may require verification of $\widehat{\mathsf{C}} \models e$. 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 $\widehat{\mathsf{C}} \models t^{\ell}$ 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:

$$\widehat{\mathsf{C}} \sqsubseteq \widehat{\mathsf{C}}' \text{ iff } \forall \ell : \widehat{\mathsf{C}}(\ell) \subseteq \widehat{\mathsf{C}}'(\ell).$$

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

[1] See [10] for details and a thorough discussion of coinduction in specifying static analyses.

$$
\begin{array}{ll}
\mathcal{A}[\![x^{\ell}]\!] & = \widehat{\mathsf{C}}[\ell \mapsto^{+} \widehat{\mathsf{C}}(x)] \\
\mathcal{A}[\![(\lambda x.e)^{\ell}]\!] & = \widehat{\mathsf{C}}[\ell \mapsto \{\lambda x.e\}] \\
\mathcal{A}[\![(t_1^{\ell_1}\ t_2^{\ell_2})^{\ell}]\!] & = \mathcal{A}[\![t_1^{\ell_1}]\!];\ \ \mathcal{A}[\![t_2^{\ell_2}]\!]; \\
 & \quad \textbf{for each } \lambda x.t_0^{\ell_0} \textbf{ in } \widehat{\mathsf{C}}(\ell_1) \textbf{ do} \\
 & \quad\ \ \widehat{\mathsf{C}}[x \mapsto^{+} \widehat{\mathsf{C}}(\ell_2)];\ \ \mathcal{A}[\![t_0^{\ell_0}]\!];\ \ \widehat{\mathsf{C}}[\ell \mapsto^{+} \widehat{\mathsf{C}}(\ell_0)]
\end{array}
$$

The abstract evaluator $\mathcal{A}[\![\cdot]\!]$ is iterated until the finite cache reaches a fixed point.[2] Since the cache size is polynomial in the program size, so is the running time, as the cache is *monotonic*—we put values in, but never take them out. Thus the analysis and any decision problems answered by the analysis are clearly computable within 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:

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

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

3 Henglein's simple closure analysis

Simple closure analysis follows from an observation by Henglein some 15 years ago: 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 [6]. The analysis bears a strong resemblance to simple typing—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 datastructure.

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

[2] A single iteration of $\mathcal{A}[\![e]\!]$ may in turn make a recursive call $\mathcal{A}[\![e]\!]$ with no change in the cache, so care must be taken to avoid looping. This amounts to appealing to the coinductive hypothesis $\widehat{\mathsf{C}} \models e$ in verifying $\widehat{\mathsf{C}} \models e$. 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.

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:

$$\begin{aligned}
&\widehat{\mathsf{C}} \models x^{\ell} \text{ iff } \widehat{\mathsf{C}}(x) = \widehat{\mathsf{C}}(\ell)\\
&\widehat{\mathsf{C}} \models (\lambda x.e)^{\ell} \text{ iff } \lambda x.e \in \widehat{\mathsf{C}}(\ell)\\
&\widehat{\mathsf{C}} \models (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell} \text{ iff } \widehat{\mathsf{C}} \models t_1^{\ell_1} \wedge \widehat{\mathsf{C}} \models t_2^{\ell_2} \wedge \forall \lambda x.t_0^{\ell_0} \in \widehat{\mathsf{C}}(\ell_1) :\\
&\qquad \widehat{\mathsf{C}} \models t_0^{\ell_0} \wedge \widehat{\mathsf{C}}(\ell_2) = \widehat{\mathsf{C}}(x) \wedge \widehat{\mathsf{C}}(\ell_0) = \widehat{\mathsf{C}}(\ell)
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{A}[\![x^{\ell}]\!] &= \widehat{\mathsf{C}}[\ell \leftrightarrow x]\\
\mathcal{A}[\![(\lambda x.e)^{\ell}]\!] &= \widehat{\mathsf{C}}[\ell \mapsto^{+} \{\lambda x.e\}]\\
\mathcal{A}[\![(t_1^{\ell_1} t_2^{\ell_2})^{\ell}]\!] &= \mathcal{A}[\![t_1^{\ell_1}]\!];\ \ \mathcal{A}[\![t_2^{\ell_2}]\!];\\
&\qquad \textbf{for each } \lambda x.t_0^{\ell_0} \textbf{ in } \widehat{\mathsf{C}}(\ell_1) \textbf{ do}\\
&\qquad\quad \widehat{\mathsf{C}}[x \leftrightarrow \ell_2];\ \ \mathcal{A}[\![t_0^{\ell_0}]\!];\ \ \widehat{\mathsf{C}}[\ell \leftrightarrow \ell_0]
\end{aligned}$$

The abstract evaluator $\mathcal{A}[\![\cdot]\!]$ is iterated until a fixed point is reached.[3] By similar reasoning to that given for 0CFA, simple closure analysis is clearly computable within polynomial time.

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):

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

[3] The fine print of FN 2 applies as well.

4 Linearity and normalization

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

First, consider an evaluator for our language, $\mathcal{E}[\![\cdot]\!]$:

$$\mathcal{E}[\![\cdot]\!] : \mathbf{Exp} \to \mathbf{Env} \rightharpoonup \langle \mathbf{Term}, \mathbf{Env} \rangle$$

$$\begin{aligned}
\mathcal{E}[\![x^\ell]\!][x \mapsto c] &= c \\
\mathcal{E}[\![(\lambda x.e)^\ell]\!]\rho &= \langle \lambda x.e, \rho \rangle \\
\mathcal{E}[\![(e_1\ e_2)^\ell]\!]\rho &= \textbf{let } \langle \lambda x.e_0, \rho' \rangle = \mathcal{E}[\![e_1]\!]\rho \restriction \mathbf{fv}(e_1) \textbf{ in} \\
&\quad\ \ \textbf{let } c = \mathcal{E}[\![e_2]\!]\rho \restriction \mathbf{fv}(e_2) \textbf{ in} \\
&\quad\ \ \ \ \mathcal{E}[\![e_0]\!]\rho'[x \mapsto c]
\end{aligned}$$

We use ρ to range over *environments*, $\mathbf{Env} = \mathbf{Var} \rightharpoonup \langle \mathbf{Term}, \mathbf{Env} \rangle$, and let c range over *closures*, each comprising a term and an environment that closes the term. The set of labels in a closure is defined as $\mathbf{lab}(t, \rho) = \mathbf{lab}(t) \cup \mathbf{lab}(\rho)$, where $\mathbf{lab}(\rho) = \bigcup_{x \in \mathbf{dom}(\rho)} \mathbf{lab}(\rho(x))$.

Notice that the evaluator "tightens" the environment when evaluating an application, thus maintaining throughout evaluation that the domain of the environment is exactly the set of free variables in the expression. So, when evaluating a variable occurrence, there is only mapping in the environment, and likewise, when constructing a closure the environment does not need to be restricted—it already is.

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 two (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:*

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

Lemma 1. *If* ρ *linearly closes* t *and* $\mathcal{E}[\![t^\ell]\!]\rho = c$*, then* $|c| \leq |t, \rho|$*.*

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

This is the environment-based analog to the easy observation that each β-step *strictly* decreases the size of a linear term.

Definition 2. *A cache* $\widehat{\mathsf{C}}$ respects $\langle t, \rho\rangle$ *(written* $\widehat{\mathsf{C}} \vdash t, \rho$*) when,*

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

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

Assume that the imperative algorithm $\mathcal{A}[\![\cdot]\!]$ of Section 3 is written in the obvious "cache-passing" functional style.

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

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 $\widehat{\mathsf{C}} \vdash x, \rho$, ρ linearly closes x, thus $\rho = [x \mapsto \langle t', \rho'\rangle]$ and ρ' linearly closes t'. By definition,
  $$\mathcal{E}[\![x^\ell]\!]\rho = \langle t', \rho'\rangle, \text{ and}$$
  $$\mathcal{A}[\![x^\ell]\!]\widehat{\mathsf{C}} = \widehat{\mathsf{C}}[x \leftrightarrow \ell].$$
  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,
  $$\mathcal{E}[\![(\lambda x.e_0)^\ell]\!]\rho = \langle \lambda x.e_0, \rho\rangle,$$
  $$\mathcal{A}[\![(\lambda x.e_0)^\ell]\!]\widehat{\mathsf{C}} = \widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}],$$
  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
  $$\mathcal{E}[\![t_1]\!]\rho \restriction \mathbf{fv}(t_1^{\ell_1}) = \langle v_1, \rho_1\rangle = \langle \lambda x.t_0^{\ell_0}, \rho_1\rangle,$$
  $$\mathcal{E}[\![t_2]\!]\rho \restriction \mathbf{fv}(t_2^{\ell_2}) = \langle v_2, \rho_2\rangle,$$
  $$\mathcal{A}[\![t_1]\!]\widehat{\mathsf{C}} = \widehat{\mathsf{C}}_1, \text{ and}$$
  $$\mathcal{A}[\![t_2]\!]\widehat{\mathsf{C}} = \widehat{\mathsf{C}}_2.$$

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

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

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's clear that $\widehat{\mathsf{C}}_3 \models t_i^{\ell_i}$. Furthermore,

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

By Lemma 1, $|v_i, \rho_i| \leq |t_i, \rho \restriction \mathbf{fv}(t_i)|$, therefore

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

Let

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

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 a least simple closure analysis of a linear program t^{ℓ}, then $\mathcal{E}[\![t^{\ell}]\!]\emptyset = \langle v, \rho'\rangle$ 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 a least simple closure analysis of e iff $\widehat{\mathsf{C}}$ is a least 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 all correspond. And, as we'll see, when an analysis is *exact* for linear terms, the analysis will have a PTIME-lower bound. The key to designing sub-PTIME analyses therefore is to do something less precise on linear terms.

5 Lower bound for the linear λ-calculus

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 sub-cubic 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 of complexity for a very 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.

5.1 Boolean encodings in linear λ-calculus

The canonical PTIME-complete problem is the Circuit Value Problem [11]:

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

A problem is PTIME-hard if any instance of it can be compiled using only $O(\ln |x|)$ space into an instance of the circuit value problem. We now show how to program circuits using linear terms, proving simple closure analysis to be PTIME-hard.

We code in a linear subset of Scheme and rely on the usual Church-encoding of pairs, but use Scheme's **syntax-rules** facility to define some syntactic sugar and thus sweeten the presentation a bit. The connectives from linear logic are used to emphasize the connection: ⊗ constructs pairs, e.g., (⊗ *x y*), ⅋ linearly unpairs them, e.g., (⅋ ((*x y*) *p*) *e*)—this destructures a pair *p*, binding the names *x* and *y* in the scope of *e*, where by linearity, *x* and *y* must appear exactly once. Function composition, ∘, is defined as usual albeit as sugar (((to spare some parentheses))).

```
(define-syntax ∘
  (syntax-rules ()
    [(∘ f g) (λ (x) (f (g x)))]))

(define-syntax ⊗
  (syntax-rules ()
    [(⊗ x y) (λ (z) ((z x) y))]))

(define (tt p) (⅋ ((x y) p) (⊗ x y)))
(define (ff p) (⅋ ((x y) p) (⊗ y x)))
(define T (⊗ tt ff))
(define F (⊗ ff tt))

(define-syntax ⅋
  (syntax-rules ()
    [(⅋ ((x y) p) e)
     (p (λ (x) (λ (y) e)))]))

(define and
  (λ (a) (λ (b)
    (⅋ ((p p̄) a)
    (⅋ ((q q̄) b)
    (⅋ ((u v) (p (⊗ q ff)))
    (⅋ ((ū v̄) (p̄ (⊗ tt q̄)))
      (⊗ u (∘ ū (∘ v (∘ v̄ ff))))))))))))

(define or
  (λ (a) (λ (b)
    (⅋ ((p p̄) a)
    (⅋ ((q q̄) b)
    (⅋ ((u v) (p (⊗ tt q)))
    (⅋ ((ū v̄) (p̄ (⊗ q̄ ff)))
      (⊗ u (∘ ū (∘ v (∘ v̄ ff))))))))))))
```

```
(define (copy p)
  (⅋ ((u v) p) (⊗ (u (⊗ tt ff)
                     (v (⊗ ff tt))))))

(define (not p)
  (⅋ ((u v) p) (⊗ v u)))
```

Briefly, the logic gates work like this: *tt* is the identity on pairs, and *ff* is the swap. Boolean values are either *(tt,ff)* or *(ff,tt)*, where the first component is the "real" value, and the second component is the negation. Conjunction works by computing pairs *(p,p')*, *(q,q')* where the first is the *and*, and the second is (exploiting deMorgan duality) the *or* on the complements. Then the answer is *(p,q)*, and we are *guaranteed* that $ff \circ p' \circ q' = tt$. This latter computation represents the *garbage*, which needs disposal, easy since *tt* is the identity function. This hacking 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.

Once continuation-passing style variants of the logic gates are defined,

```
(define and-gate (λ (a) (λ (b) (λ (k) (k ((and a) b))))))
```

and similarly for the other gates, circuits can be written as straight-line code—and also compiled in logarithmic space into the corresponding Scheme program:

```
(define circuit
  (λ (e1) (λ (e2) (λ (e3) (λ (e4) (λ (e5) (λ (e6)
    (((and-gate e2) e3) (λ (e7)
    (((and-gate e4) e5) (λ (e8)
    (((and-gate e7) e8) (λ (f)
    ((copy-gate f) (λ (e9) (λ (e10)
    (((or-gate e1) e9) (λ (e11)
    (((or-gate e10) e6) (λ (e12)
    (((or-gate e11) e12) (λ (out) out)))))))))))))))))))))))
```

We further assume that the definition of the logical gates and boolean values are inlined, the program is α-converted to have distinct bound variable names, and is uniquely labeled. Note that inlining only increases the size of the program by a constant factor. We now formalize the flow analysis decision problem described colloquially in Section 1:

The Decision Problem

Given a closed expression e, a term $\lambda x.e'$, and label ℓ, is $\lambda x.e' \in \widehat{\mathsf{C}}(\ell)$ in a least analysis of e?

We know from Theorem 1 that normalization and analysis of linear programs are synonymous, and our encoding of circuits will faithfully simulate a given circuit on its inputs, evaluating to T iff the circuit accepts its inputs. But there may be many syntactic instances of T after inlining. In other words, if the program is labeled ℓ, we know $\widehat{\mathsf{C}}(\ell) = T$ iff the circuit accepts, but we don't know which T to ask about for the purposes of the decision problem.

We thus use the following construction, dubbed The Widget, to isolate a particular flow that can be asked about with respect to the decision problem:

```
(define (widget b)
  (⅋ ((u ū) b)
  (⅋ ((m m̄) (u (⊗ (λ (c) c) (λ (c̄) c̄))))
  (⅋ ((w w̄) (ū (⊗ (λ (ē) ē) (λ (e) e))))
    (⊗ (⊗ (m (λ (t?) t?)) (m̄ (λ (f?) f?)))
       (⊗ (w (λ (t̄?) t̄?)) (w̄ (λ (f̄?) f̄?)))))))))
```

The Widget inputs a boolean value b, destructures it into its constituent, dual functions on pairs. So u is the identity function on pairs when b is true, otherwise it's the swap function. Then u is applied to a pair and the result destructured. Thus m is bound to $(\lambda\ (c)\ c)$ iff b is true, in which case, when m is applied to the closed term $(\lambda\ (t?)\ t?)$, it flows into c. The rest of the widget performs symmetric operations in order to maintain linearity.

So by Theorem 1, either $(\lambda\ (t?)\ t?) \in \widehat{\mathsf{C}}(c)$ or $(\lambda\ (f?)\ f?) \in \widehat{\mathsf{C}}(c)$, not both.

Theorem 2. *Simple closure analysis is* PTIME*-complete.*

6 Other sub-cubic analyses

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

6.1 Ashley and Dybvig's sub-0CFA

In [7], Ashley and Dybvig develop a general framework for specifying and computing flow analyses, which can be instantiated to obtain Shivers' 0CFA or Jagannathan and Weeks polynomial 1CFA [12], for example. They also develop a class of instantiations of their framework dubbed *sub-0CFA* that is 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 lambda 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 we give up and say all lambda abstractions flow out of this point. But in linear terms of course, the cache is only updated at most once for each program point. Thus we can 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 be the number of passes 0CFA makes over the given the program). We can further clarify this relationship by noting that for all programs that are linear, all analyses in the sub-0CFA class are identical to 0CFA (and thus simple closure analysis).

6.2 Subtransitive 0CFA

Heintze and McAllester [5] have shown that the "cubic bottleneck" of computing full 0CFA—that is, computing all the flows in a program—cannot be avoided in general without combinatorial breakthroughs: the problem is 2NPDA-hard, for which the "the cubic time decision procedure [...] has not been improved since its discovery in 1968."

Given the unlikeliness of improving the situation in general, Heintze and McAllester [13] identify several simpler flow questions (including the decision problem discussed in the paper, which is the simplest; answers to any of the other questions imply an answer to this problem). They give algorithms for simply typed terms that answer these restricted flow problems, which under certain conditions, compute in less than cubic time.

Their analysis is linear with respect to a program's graph, which 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 we remove this bound assumption, though, it is clear that even these simplified flow problems (and even their bidirectional-flow analogs), are complete for PTIME(observe every linear term is simply typable, however the size of this type 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 with respect to the program size—in which case the standard cubic algorithm is preferred.

Independently, Mossin [14] 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.* [15] 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's not surprising then, that given a much richer graph structure, the analysis can be computed quickly. In this light, recent results [8] on 0CFA of η-expanded, simply typed programs can be seen as an improvement of the substransitive flow analysis since it works equally well for languages with first-class control and is done in LOGSPACE (or in other words, PTIME = LOGSPACE upto η).

7 Conclusions and perspective

What can be said about designing sub-PTIME-hard analyses is that it necessarily involves making approximations on linear programs. 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 [16] developed an "analysis" which is exact for simply typed terms and is *ipso facto* non-elementary recursive [17]. But what is the usefulness of an analysis if there is no difference between analyzing and running a program?

It remains open whether there are useful analyses for linear programs that can be computed with less resources than it takes to evaluate the program.

We should be clear about what's being said, and not said. There is a considerable difference in practice between linear algorithms (nominally considered efficient) and 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). Similarly, 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 similarly.

A fundamental question we need to be able to answer is this: what can be deduced about a long-running program with a time-bounded analyzer? When we statically analyze exponential-time programs with a polynomial-time method, there should be a analytic bound on what we can learn at compile-time: a theorem delineating how exponential time is being viewed through the compressed, myopic lens of polynomial time computation.

For example, there is a theorem due to Rick Statman [17] that says this: let $\mathbf{P}$ be a property of simply-typed λ-terms that we'd like to detect by static analysis, where $\mathbf{P}$ is preserved by reduction (normalization), and is computable in elementary time (polynomial, or exponential, or doubly-exponential, or...). Then $\mathbf{P}$ is a *trivial* property: for any type τ, $\mathbf{P}$ is satisfied by *all* or *none* of the programs of type τ.

We'd like to prove some analogs of Statman's theorem, with or without the typing condition, but weakening the condition of "preserved by reduction" to some *approximation* analogous to the approximations of control flow analysis, as described above. We're motivated as well by yardsticks such as Shannon's theorem [18] from information theory: specify a bandwidth for communication and an error rate, and Shannon's results give bounds on the channel capacity. We too have essential measures: the time complexity of our analysis, the asymptotic differential between that bound and the time bound of the program we're analyzing. There ought to be a fundamental result about what information can be yielded as a function of that differential. At one end, if the program and analyzer take the same time, the analyzer can just run the program to find out everything. At the other end, if the analyzer does no work (or a constant amount of work), nothing can be learned. Analytically speaking, what is in between?

Acknowledgments: We are grateful to Olin Shivers and Matt Might for a long, fruitful, and ongoing dialogue on flow analysis. The first author also thanks the researchers of the Northeastern University Programming Research Lab for the hospitality and engaging discussions had as a visiting lecturer over the last year.

References

1. Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Proceedings of the 8th Colloquium on Automata, Languages and Programming, London, UK, Springer-Verlag (1981) 114–128
2. Sestoft, P.: Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark (October 1988) Master's thesis no. 254.
3. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (May 1991) Technical Report CMU-CS-91-145.
4. Shivers, O.: Control flow analysis in Scheme. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, New York, NY, USA, ACM (1988) 164–174
5. Heintze, N., McAllester, D.: On the cubic bottleneck in subtyping and flow analysis. In: LICS '97: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, IEEE Computer Society (1997) 342
6. Henglein, F.: Simple closure analysis. DIKU Semantics Report D-193 (1992)
7. Ashley, J.M., Dybvig, R.K.: A practical and flexible flow analysis for higher-order languages. ACM Trans. Program. Lang. Syst. **20**(4) (1998) 845–868
8. Van Horn, D., Mairson, H.G.: Relating complexity and precision in control flow analysis. In: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM Press (2007) 85–96
9. Girard, J.Y.: Linear logic: its syntax and semantics. In: Proceedings of the workshop on Advances in linear logic, Cambridge University Press (1995)
10. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
11. Ladner, R.E.: The circuit value problem is log space complete for P. SIGACT News **7**(1) (1975) 18–20
12. Jagannathan, S., Weeks, S.: A unified treatment of flow analysis in higher-order languages. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1995) 393–407
13. Heintze, N., McAllester, D.: Linear-time subtransitive control flow analysis. In: PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, New York, NY, USA, ACM (1997) 261–272
14. Mossin, C.: Higher-order value flow graphs. Nordic J. of Computing **5**(3) (1998) 214–234
15. Hankin, C., Nagarajan, R., Sampath, P.: Flow analysis: games and nets. In: The essence of computation: complexity, analysis, transformation. Springer-Verlag New York, Inc., New York, NY, USA (2002) 135–156
16. Mossin, C.: Exact flow analysis. In: SAS '97: Proceedings of the 4th International Symposium on Static Analysis, London, UK, Springer-Verlag (1997) 250–264
17. Statman, R.: The typed λ-calculus is not elementary recursive. Theor. Comput. Sci. **9** (1979) 73–81
18. Shannon, C.E.: A mathematical theory of communication. Bell System Technical Journal **27** (1948)