The Complexity of Flow Analysis in Higher-Order Languages

David Van Horn

The Complexity of Flow Analysis in Higher-Order Languages

A Dissertation

Presented to
The Faculty of the Graduate School of Arts and Sciences
Brandeis University
Mitchom School of Computer Science

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

by
David Van Horn
August, 2009

This dissertation, directed and approved by David Van Horn's committee, has been accepted and approved by the Graduate Faculty of Brandeis University in partial fulfillment of the requirements for the degree of:

DOCTOR OF PHILOSOPHY

Adam B. Jaffe, Dean of Arts and Sciences

Dissertation Committee:

Harry G. Mairson, Brandeis University, Chair
Olivier Danvy, University of Aarhus
Timothy J. Hickey, Brandeis University
Olin Shivers, Northeastern University

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

Acknowledgments

Harry taught me so much, not the least of which was a compelling kind of science.

It is fairly obvious that I am not uninfluenced by Olivier Danvy and Olin Shivers and that I do not regret their influence upon me.

My family provided their own weird kind of emotional support and humor.

I gratefully acknowledge the support of the following people, groups, and institutions, in no particular order: Matthew Goldfield. Jan Midtgaard. Fritz Henglein. Matthew Might. Ugo Dal Lago. Chung-chieh Shan. Kazushige Terui. Christian Skalka. Shriram Krishnamurthi. Michael Sperber. David McAllester. Mitchell Wand. Damien Sereni. Jean-Jacques Lévy. Julia Lawall. Matthias Felleisen. Dimitris Vardoulakis. David Herman. Ryan Culpepper. Richard Cobbe. Felix S Klock II. Sam Tobin-Hochstadt. Patrick Cousot. Alex Plotnick. Peter Møller Neergaard. Noel Welsh. Yannis Smaragdakis. Thomas Reps. Assaf Kfoury. Jeffrey Mark Siskind. David Foster Wallace. Timothy J. Hickey. Myrna Fox. Jeanne DeBaie. Helene Greenberg. Richard Cunnane. Larry Finkelstein. Neil D. Jones and the lecturers of the Program Analysis and Transformation Summer School at DIKU. New England Programming Languages and Systems Symposium. IBM Programming Language Day. Northeastern University Semantics Seminar and PL Jr. Seminar series. The reviewers of ICFP'07, SAS'08, and ICFP'08. Northeastern University. Portland State University. The National *Science* Foundation, grant CCF-0811297.

And of course, Jessie. Odd to thank the air one breathes, but crazy not to.

Abstract

This dissertation proves lower bounds on the inherent difficulty of deciding flow analysis problems in higher-order programming languages. We give exact characterizations of the computational complexity of 0CFA, the kCFA 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 $k > 0$, kCFA 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 kCFA is intractably slow for any $k > 0$. There is, in the worst case—and plausibly, in practice—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.

Preface

What to expect, What not to expect

This dissertation investigates lower bounds on the computational complexity of flow analysis for higher-order languages, uncovering its inherent computational costs and the fundamental limits of efficiency for *any* flow analysis algorithm. As such, I have taken existing, representative, flow analysis specifications "off the shelf" without modification. This is *not* a dissertation on the design and implementation of novel flow analyses (although it should inform such work). The reader is advised to expect no benchmarks or prototype implementations, but rather insightful proofs and theorems.

This dissertation relates existing research in order to situate the novelty and significance of this work. It does not attempt to comprehensively survey the nearly thirty years of research on flow analysis, nor the wealth of frameworks, formulations, and variants. A thorough survey on flow analysis has been undertaken by Midtgaard (2007).

Assumptions on the reader

For the reader expecting to understand the intuitions, proofs, and consequences of the results of this dissertation, I assume familiarity with the following, in roughly descending order of importance:

- functional programming.

  The reader should be at ease programming with higher-order procedures in languages such as Scheme or ML. For an introduction to programming

in Scheme specifically, *The Scheme Programming Language* by Dybvig (2002) and *Teach Yourself Scheme in Fixnum Days* by Sitaram (2004) are recommended; for ML, *Programming in Standard ML* by Harper (2005) and *ML for Working Programmer* by Paulson (1996) are recommended.

This dissertation relies only on the simplest applicative subsets of these languages.

- interpreters (evaluators).

  The reader should understand the fundamentals of writing an interpreter, in particular an environment-based interpreter (Landin 1964) for the functional core of a programming language.[1] The definitive reference is "Definitional interpreters for higher-order programming languages" by Reynolds (1972, 1998). Understanding sections 2–6 are an absolute must (and joy). For a more in-depth textbook treatment, see the gospel according to Abelson and Sussman (1996): *Structure and Interpretation of Computer Programs*, Chapter 3, Section 2, "The Environment Model of Evaluation," and Chapter 4, Section 1, "The Metacircular Evaluator." Finally, *Essentials of Programming Languages* by Friedman and Wand (2008) is highly recommended.[2]

- the λ-calculus.

  The encyclopedic reference is *The Lambda Calculus: Its Syntax and Semantics* by Barendregt (1984), which is an overkill for the purpose of understanding this dissertation. Chapters 1 and 3 of *Lectures on the Curry-Howard Isomorphism* by Sørensen and Urzyczyn (2006) offers a concise and sufficient introduction to untyped and typed λ-calculus, respectively. There are numerous others, such as *An Introduction to Lambda Calculi for Computer Scientists* by Hankin (2004), *Functional programming and lambda calculus* by Barendregt (1990), and so on. Almost any will do.[3]

- basic computational complexity theory.

  The reader should be familiar with basic notions such as complexity classes, Turing machines, undecidability, hardness, and complete problems. Papadimitriou (1994) is a standard introduction (See chapters 2–4, 7–9, 15,

[1] Note that almost every modern programming language includes a higher-order, functional core: Scheme, ML, JavaScript, Java, Haskell, Smalltalk, Ruby, C#, etc., etc.

[2] As an undergraduate, I cut my teeth on the first edition (1992).

[3] See Cardone and Hindley (2006, Footnote 1) for references to French, Japanese, and Russian overviews of the λ-calculus.

and 16). Jones (1997) is a good introduction with a stronger emphasis on programming and programming languages (See part IV and V). Almost any decent undergraduate text on complexity would suffice.

In particular, the classes LOGSPACE, PTIME, NPTIME, and EXPTIME are used. Reductions are given from canonical complete problems for these classes to various flow analysis problems. These canonical complete problems include, respectively: the permutation problem, circuit value problem (CVP), Boolean satisfiability (SAT), and a generic reduction for simulating deterministic, exponential time Turing machines.

Such a reduction from a particular complete computational problem to a corresponding flow analysis problem establishes a *lower bound* on the complexity of flow analysis: solving the flow problem is *at least as hard* as solving the corresponding computational problem (SAT, CVP, etc.), since any instance of these problems can be transformed (reduced), using very limited resources, to an instance of the flow analysis problem. In other words, an algorithm to solve one problem can be used as an algorithm to solve the other.

- fundamentals of program analysis.

  A basic understanding of program analysis would be beneficial, although I have tried to make plain the connection between analysis and evaluation, so a thorough understanding of program *interpretation* could be sufficient. Perhaps the standard text on the subject is *Principles of Program Analysis* by Nielson et al. (1999), which I have followed closely because it offers an authoritative and precise definition of flow analysis. It is thorough and rigorous, at the risk of slight *rigor mortis*. Shivers' dissertation, *Control-Flow Analysis of Higher-Order Languages*, contains the original development of kCFA and is replete with intuitions and explanations.

- logic and proof theory.

  The reader should be comfortable with the basics of propositional logic, such as De Morgan duality, modus ponens, etc. The reader is also assumed to be comfortable with sequent calculi, and in particular sequents for linear logic. Girard et al. (1989) provides a solid and accessible foundation.

  All of the theorems will be accessible, but without this background, only a small number of the more supplemental proofs will be inaccessible. Fear not if this is not your cup of meat.

Previously published material

Portions of this dissertation are derived from material previously published in the following papers, written jointly with Harry Mairson:

1. Relating Complexity and Precision in Control Flow Analysis. In *Proceedings of the 12th International Conference on Functional Programming*, Frieburg, Germany. Van Horn and Mairson (2007).

2. Flow Analysis, Linearity, and PTIME. In *The 15th International Static Analysis Symposium*, Valencia, Spain. Van Horn and Mairson (2008b).

3. Deciding kCFA is complete for EXPTIME. In *Proceedings of the 13th International Conference on Functional Programming*, Victoria, BC, Canada. Van Horn and Mairson (2008a).

Contents

List of Figures

Chapter 1

Introduction

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

1.1 Overview

Predicting the future is hard.

Nevertheless, such is the business of an optimizing compiler: it reads in an input program, predicts what will happen when that program is run, and then—based on that prediction—outputs an optimized program.

Often these predictions are founded on semantics-based program analysis (Cousot and Cousot 1977, 1992; Muchnick and Jones 1981; Nielson et al. 1999), which aims to discover the run-time behavior of a program *without actually running it* (Muchnick and Jones 1981, page xv). 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 and Jagannathan 1998; Jagannathan et al. 1998; Might and 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, 1989; Shivers 1988, 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 dissertation 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 dissertation is intended to repair such lacunae in our understanding.

1.2 Summary of Results

- Normalization and analysis are equivalent for linear programs.
- 0CFA and other monovariant flow analyses are complete for PTIME.
- 0CFA of typed, η-expanded programs is complete for LOGSPACE.
- kCFA is complete for EXPTIME for all $k > 0$.

1.3 Details

1.3.1 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.

1.3.2 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 (1992). Similarly, by explicitly bounding the number of passes the analyzer is allowed over the program, as in Ashley and Dybvig's sub-0CFA (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.

1.3.3 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.

1.3.4 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.

Chapter 2

Foundations

The aim of flow analysis is to safely approximate an answer the question:[1]

For each function application, which functions may be applied?

Analysis can easily be understood as the safe approximation to program evaluation. It makes sense, then, to first consider evaluation in detail. In the following sections, an evaluation function ($\mathcal{E}$) is defined, from which an instrumented variation ($\mathcal{I}$) is derived and abstracted to obtain the abstract evaluator ($\mathcal{A}$). Finally, we review basic concepts from complexity theory and sketch our approach to proving lower bounds.

2.1 Structure and Interpretation

The meaningful phrases of a program are called *expressions*, the process of executing or interpreting these expressions is called *evaluation*, and the result of evaluating an expression is called a *value* (Reynolds 1972).

We will consider a higher-order applicative programming language based on the λ-calculus, in which evaluation is environment based and functional values are

[1] See for example the transparencies accompanying Chapter 3 "Control Flow Analysis" of Nielson et al. (1999): http://www2.imm.dtu.dk/~riis/PPA/ppa.html

represented using closures. The syntax of the language is given by the following grammar:

$$\mathbf{Exp} \quad e ::= x \mid e\ e \mid \lambda x.e \quad \text{expressions}$$

> **Note to the reader:** This may seem a rather minimal programming language and you may wonder what the broader applicability of these results are in the face of other language features. But as noted earlier, this dissertation is concerned with *lower bounds* on static analysis. By examining a minimal, core language, all results will immediately apply to any language which includes this core. In other words, the more restricted the subject language, the broader the applicability.
>
> It may be that the lower bound can be improved in the presence of some language features, that is, the given set of features may make analysis provably harder, but it certainly can not make it any easier.[2]

Following Landin (1964), substitution is modelled using *environments*. Procedures will be represented as *closures*, a λ-term together with its lexical environment, which closes over the free variables in the term, binding variables to values.

We use ρ to range over *environments* (mappings from variables to closures), and v to range over *closures* (pairs consisting of a term and an environment that closes the term). The empty environment (undefined on all variables) is denoted $\bullet$, and we occasionally abuse syntax and write the closed term e in place of the closure $\langle e, \bullet \rangle$. Environment extension is written $\rho[x \mapsto v]$ and we write $\bullet[x \mapsto v]$ as $[x \mapsto v]$ and $[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]$ as $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$.

$$\begin{array}{ll} \mathbf{Env} & \rho \in \mathbf{Var} \rightharpoonup \mathbf{Val} \quad \text{environments} \\ \mathbf{Val} & v \in \langle \mathbf{Exp}, \mathbf{Env} \rangle \quad \text{closures} \end{array}$$

[2] This is to be contrasted with, for example, a type soundness theorem, where it is just the opposite: adding new language feature may revoke soundness.

For similar remarks, see the discussion in section 2 of Reps (1996) concerning the benefits of formulating an analysis problem in "trimmed-down form," which not only leads to a wider applicability of the lower bounds, but also "allows one to gain greater insight into exactly what aspects of an interprocedural-analysis problem introduce what computational limitations on algorithms for these problems."

In other words, lower bounds should be derived not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

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

Figure 2.1: Evaluator $\mathcal{E}$.

The meaning of an expression is given in terms of an evaluation function, or interpreter. Following Abelson and Sussman (1996, Chapter 4, "Metalinguistic Abstraction"), an interpreter is defined as follows:

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

The evaluation function for the language is given in Figure 2.1. We say e evaluates to v under environment ρ iff $\mathcal{E}[\![e]\!]\rho = v$ and computing the evaluation function defines our notion of the "running of a program." Some examples of evaluation:

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

This gives us a mostly *extensional* view of program behaviour—evaluation maps programs to values, but offers little information regarding *how* this value was computed. For the sake of program optimization it is much more useful to know about operational (or *intensional*) properties of programs. These properties are formulated by appealing to an "instrumented interpreter," which is the subject of the next section. Intuitively, the instrumented evaluator works just like the un-instrumented evaluator, but additionally maintains a complete history of the operations carried out during the running of the program.

2.2 Instrumented Interpretation

Instrumented (or concrete) interpretation carries out the running of program while maintaining a trace of the operations performed, thus providing an operational history of evaluation. A suitable reformulation of the original definition of an evaluator to incorporate instrumentation is then:

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

Exactly which actions should be record will vary the domain of any given static analysis and there is no universal notion of a program trace, but for flow analysis, the interesting actions are:

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

These actions are recorded in a *cache*, and there is one for each kind of action:

$$\begin{aligned} \mathsf{C} &: \mathbf{Lab} \times \Delta \rightharpoonup \mathbf{Val} \\ \mathsf{r} &: \mathbf{Var} \times \Delta \rightharpoonup \mathbf{Val} \\ \mathbf{Cache} &= (\mathbf{Lab} \times \Delta \rightharpoonup \mathbf{Val}) \times (\mathbf{Var} \times \Delta \rightharpoonup \mathbf{Val}) \end{aligned}$$

The C cache records the result, or returned value, of each subcomputation, and the r cache records each binding of a variable to a value. Given the label of a subexpression ($\mathbf{Lab}$) and a description of the context (Δ), C returns the value produced by that subexpression evaluated in that context. Given the name of a variable ($\mathbf{Var}$) and a description of the context (Δ), r returns the value bound to that variable in that context. The C cache is a partial function since a subexpression 1) may not produce a value, it may diverge, or 2) may not be evaluated in the given context. The r cache is partial for analogous reasons.

The set $\mathbf{Lab}$ is used to index subexpressions. It can easily be thought of as the implicit source location of the expression, but our formalism will use an explicit labelling scheme. We use ℓ to range over labels. The syntax of the source language is given by the following grammar, and programs are assumed to be uniquely labelled:

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

Irrelevant labels are frequently omitted for presentation purposes.

The set Δ consists of *contours*, which are strings of labels from application nodes in the abstract syntax of the program. A string of application labels describes the context under which the term evaluated.

A variable may be bound to any number of values during the course of evaluation. Likewise, a subexpression that occurs once syntactically may evaluate to any number of values during evaluation. So asking about the flows of a subexpression is ambiguous without further information. Consider the following example, where `True` and `False` are closed, syntactic values:

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

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

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

Notice that y is bound to different values within different contexts. That is, y is bound to `True` when evaluating the application labeled 1, and y is bound to `False` when evaluating the application labeled 2. Both of these occur while evaluating the outermost application, labeled 3. A string of these application labels, called a *contour*, uniquely describes the *context* under which a subexpression evaluates. Adopting the common convention of writing a context as an expression with a hole "[ ]" in it (Felleisen and Flatt 2009), the following contours describe the given contexts:

$$\begin{aligned} 321 \quad &\text{describes} \quad ((\lambda f.(f[\,]^1)^2)(\lambda y.\texttt{False}))^3 \\ 32 \quad &\text{describes} \quad ((\lambda f.[\,]^2)(\lambda y.\texttt{False}))^3 \end{aligned}$$

So a question about what a subexpression evaluates to *within a given context* has an unambiguous answer. The interpreter, therefore, maintains an environment that

maps each variable to a description of the context in which it was bound. Similarly, flow questions about a particular subexpression or variable binding must be accompanied by a description of a context. Returning to the example, the binding cache would give $\mathsf{r}(y, 321) = \texttt{True}$ and $\mathsf{r}(y, 32) = \texttt{False}$.

The set $\mathbf{Val}$ consists of closures, however the environment component of the closures are non-standard. Rather than mapping variables to values, these environments map variables to contours; the contour describes the context in which the variable was bound, so the value may be retrieved from the r cache. In other words, these environments include an added layer of indirection through the cache: variables are mapped not to their values but the location of their definition site, where the value can be found.

So we have the following data definitions:

$$\begin{array}{lcll} \delta & \in & \Delta & = & \mathbf{Lab}^\star & \text{contours} \\ v & \in & \mathbf{Val} & = & \mathbf{Term} \times \mathbf{Env} & \text{(contour) values} \\ \rho & \in & \mathbf{Env} & = & \mathbf{Var} \rightharpoonup \Delta & \text{(contour) environments} \end{array}$$

Note that this notation overloads the meaning of $\mathbf{Val}$, $\mathbf{Exp}$, and $\mathbf{Env}$ with that given in the previous section. It should be clear from setting which is meant, and when both meanings need to be used in the same context, the latter will be refered to as *contour values* and *contour environments*.

The cache is computed by the instrumented interpreter, $\mathcal{I}$, the instrumented, intentional analog of $\mathcal{E}$. It can be concisely and intuitively written as an imperative program that mutates an initially empty cache, as given in Figure 2.2.

$\mathcal{I}[\![t^\ell]\!]^\rho_\delta$ evaluates t and writes the result of evaluation into the C cache at location (ℓ, δ). The notation $\mathsf{C}(\ell, \delta) \leftarrow v$ means that the cache is mutated so that $\mathsf{C}(\ell, \delta) = v$, and similarly for $\mathsf{r}(x, \delta) \leftarrow v$. The type $\mathbf{Unit}$ is used here to emphasize the imperative nature of the instrumented evaluator; no meaningful return value is produced, the evaluator is run only for effect on the caches. The notation $\delta\ell$ denotes the concatenation of contour δ and label ℓ. The symbol ϵ denotes the empty contour.

We interpret $\mathsf{C}(\ell, \delta) = v$ as saying, "the expression labeled ℓ *evaluates to* v in the context described by δ," and $\mathsf{r}(x, \delta) = v$ as "the variable x is *bound to* v in the context described by δ." Conversely, we say "v *flows to* the expression labelled ℓ into the context described by δ," and "v *flows to* the binding of x in the context described by δ," respectively. We refer to a fact such as, $\mathsf{C}(\ell, \delta) = v$ or $\mathsf{r}(x, \delta) = v$,

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

Figure 2.2: Instrumented evaluator $\mathcal{I}$, imperative style.

as a *flow*. The instrumented interpreter works by accumulating a set of flows as they occur during evaluation.

Notice that this evaluator does not return values—it writes them into the cache: if the expression t^{ℓ} evaluates in the contour δ to v, then $\mathsf{C}(\ell, \delta)$ is assigned v. When the value of a subexpression is needed, as in the application case, the subexpression is first interpreted (causing its value to be written in the cache) and subsequently retrieved from the C cache. When the value of a variable is needed, it is retrieved from the r cache, using the contour environment to get the appropriate binding.

In other words, C is playing the role of a global return mechanism, while r is playing the role of a global environment.

Although the evaluator is mutating the cache, each location is written into just once. A straight-forward induction proof shows that the current label together with the current contour—which constitute the cache address that will be written into—forms a unique string.

Returning to the earlier example, the cache constructed by

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

includes the following entries:

$$\mathsf{r}(f, 3) \quad = \quad \lambda y.\texttt{False}$$

$$
\begin{aligned}
\mathsf{r}(y, 321) &= \texttt{True} \\
\mathsf{r}(y, 32) &= \texttt{False} \\
\mathsf{C}(1, 32) &= \lambda y.\texttt{False} \\
\mathsf{C}(3, \epsilon) &= \texttt{False}
\end{aligned}
$$

The evaluator can be written in a functional style by threading the cache through the computation as seen in Figure 2.3.

$$
\begin{array}{lcl}
\mathcal{I} & : & \mathbf{Exp} \times \mathbf{Env} \times \Delta \times \mathbf{Cache} \rightharpoonup \mathbf{Cache} \\
\\
\mathcal{I}[\![x^\ell]\!]^\rho_\delta\ \mathsf{C}, \mathsf{r} & = & \mathsf{C}[(\ell, \delta) \mapsto \mathsf{r}(x, \rho(x))], \mathsf{r} \\
\mathcal{I}[\![(\lambda x.e)^\ell]\!]^\rho_\delta\ \mathsf{C}, \mathsf{r} & = & \mathsf{C}[(\ell, \delta) \mapsto \langle \lambda x.e, \rho' \rangle], \mathsf{r} \\
 & & \quad \text{where } \rho' = \rho \restriction \mathbf{fv}(\lambda x.e) \\
\mathcal{I}[\![(t^{\ell_1} t^{\ell_2})^\ell]\!]^\rho_\delta\ \mathsf{C}, \mathsf{r} & = & \text{let } \mathsf{C}_1, \mathsf{r}_1 = \mathcal{I}[\![t^{\ell_1}]\!]^\delta_\rho\ \mathsf{C}, \mathsf{r} \text{ in} \\
 & & \text{let } \mathsf{C}_2, \mathsf{r}_2 = \mathcal{I}[\![t^{\ell_2}]\!]^\delta_\rho\ \mathsf{C}_1, \mathsf{r}_1 \text{ in} \\
 & & \text{let } \langle \lambda x.t^{\ell_0}, \rho' \rangle = \mathsf{C}_2(\ell_1, \delta) \text{ in} \\
 & & \text{let } \mathsf{r}_3 = \mathsf{r}_2[(x, \delta\ell) \mapsto \mathsf{C}_3(\ell_2, \delta)] \text{ in} \\
 & & \text{let } \mathsf{C}_3, \mathsf{r}_4 = \mathcal{I}[\![t^{\ell_0}]\!]^{\delta\ell}_{\rho'[x \mapsto \delta\ell]}\ \mathsf{C}_2, \mathsf{r}_3 \text{ in} \\
 & & \text{let } \mathsf{C}_4 = \mathsf{C}_3[(\ell, \delta) \mapsto \mathsf{C}_3(\ell_0, \delta\ell)] \text{ in} \\
 & & \quad \mathsf{C}_4, \mathsf{r}_4
\end{array}
$$

Figure 2.3: Instrumented evaluator $\mathcal{I}$, functional style.

In a more declarative style, we can write a specification of *acceptable caches*; a cache is acceptable iff it records at least all of the flows which occur during instrumented evaluation. The smallest cache satisfying this acceptability relation is the one that is computed by the above interpreter, clearly. The acceptability relation is given in Figure 2.4. It is same cache acceptability relation can be obtained from that given by Nielson et al. (1999, Table 3.10, page 192) for kCFA by letting k be arbitrarily large. (Looking ahead, the idea of kCFA is that the evaluator will begin to lose information and approximate evaluation after a contour has reached a length of k. If k is sufficiently large, approximation never occurs. So the acceptability relation of Figure 2.4 can also be seen as the specification of "∞CFA". For any program that terminates, there is a k such that performing kCFA results in a cache meeting the specification of Figure 2.4. In other words, for any program that halts, there is a k such that kCFA runs it.)

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

Figure 2.4: Exact cache acceptability, or Instrumented evaluator $\mathcal{I}$, declarative style.

There may be a multitude of acceptable analyses for a given program, so caches are partially ordered by:

$$
\begin{array}{lll}
\mathsf{C} \sqsubseteq \mathsf{C}' & \text{iff} & \forall \ell, \delta : \mathsf{C}(\ell, \delta) = v \Rightarrow \mathsf{C}'(\ell, \delta) = v \\
\mathsf{r} \sqsubseteq \mathsf{r}' & \text{iff} & \forall x, \delta : \mathsf{r}(x, \delta) = v \Rightarrow \mathsf{r}'(x, \delta) = v
\end{array}
$$

Generally, we are concerned only with the *least* such caches with respect to the domain of variables and labels found in the given program of interest.

Clearly, because constructing such a cache is equivalent to evaluating a program, it is not effectively computable.

All of the flow analyses considered in this dissertation can be thought of as an abstraction (in the sense of being a *computable approximation*) to this instrumented interpreter, which not only evaluates a program, but records a history of *flows*.

2.3 Abstracted Interpretation

Computing a complete program trace can produce an arbitrarily large cache. One way to regain decidability is to bound the size of the cache. This is achieved in kCFA by bounding the length of contours to k labels.

If during the course of evaluation, or more appropriately *analysis*, the contour is extended to exceed k labels, the analyzer will truncate the string, keeping the k

most recent labels.

But now that the cache size is bounded, a sufficiently large computation will exhaust the cache. Due to truncation, the uniqueness of cache locations no longer holds and there will come a point when a result needs to be written into a location that is already occupied with a different value. If the analyzer were to simply overwrite the value already in that location, the analysis would be unsound. Instead the analyzer must consider *both* values as flowing out of this point.

This in turn can lead to further approximation. Suppose a function application has two values given for flow analysis of the operator subterm and another two values given for the operand. The analyzer must consider the application of each function to each argument.

kCFA is a safe, computable approximation to this instrumented interpreter; it is a kind of abstract interpretation (Cousot and Cousot 1977; Jones and Nielson 1995; Nielson et al. 1999). Rather than constructing an *exact* cache C, r, it constructs an *abstract* cache $\widehat{\mathsf{C}}, \hat{\mathsf{r}}$:

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

$$\widehat{\mathbf{Cache}} \;=\; (\mathbf{Lab} \times \Delta \to \widehat{\mathbf{Val}}) \times (\mathbf{Var} \times \Delta \to \widehat{\mathbf{Val}})$$

which maps labels and variables, not to values, but to sets of values—*abstract values*:

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

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

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

The abstract interpreter $\mathcal{A}$, the imprecise analog of $\mathcal{I}$, is given in Figure 2.5 using the concise imperative style. We write $\widehat{\mathsf{C}}(\ell,\delta) \leftarrow \hat{v}$ (or $\hat{\mathsf{r}}(x,\delta) \leftarrow \hat{v}$) to indicate an

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

Figure 2.5: Abstract evaluator $\mathcal{A}$, imperative style.

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

There are many ways the concise imperative abstract evaluator can be written in a more verbose functional style, and this style will be useful for proofs in the following sections.

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

We interpret $\widehat{\mathsf{C}}(\ell,\delta) = \{v, \ldots\}$ as saying, "the expression labeled ℓ *may evaluate to* v in the context described by δ," and $\mathsf{r}(x,\delta) = v$ as "the variable x is *may be bound to* v in the context described by δ." Conversely, we say "v *flows to* the expression labelled ℓ into the context described by δ" and each $\{v, \ldots\}$ "*flows out* of the expression labelled ℓ in the context described by δ," and "v *flows to* the binding of x in the context described by δ," respectively. We refer to a fact

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

Figure 2.6: Abstract evaluator $\mathcal{A}$, functional style.

such as, $\widehat{\mathsf{C}}(\ell,\delta) \ni v$ or $\hat{\mathsf{r}}(x,\delta) \ni v$, as a *flow*. The abstract interpreter works by accumulating a set of flows as they occur during abstract interpretation until reaching a fixed point. Although this overloads the terminology used in describing the instrumented interpreter, the notions are compatible and the setting should make it clear which sense is intended.

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

Just as done in the previous section, we can write a specification of acceptable caches rather than an algorithm that computes. The resulting specification given in Figure 2.7 is what is found, for example, in Nielson et al. (1999).

There may be a multitude of acceptable analyses for a given program, so caches are partially ordered by:

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

Generally, we are concerned only with the *least* such caches with respect to the

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

Figure 2.7: Abstract cache acceptability, or Abstract evaluator $\mathcal{A}$, declarative style.

domain of variables and labels found in the given program of interest.

By bounding the contour length, the inductive proof that (ℓ, δ) was unique for any write into the cache was invalidated. Similarly, induction can no longer be relied upon for verification of acceptability. It may be the case that proving $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \models^{\rho}_{\delta} t^{\ell}$ obligates proofs of other propositions, which in turn rely upon verification of $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \models^{\rho}_{\delta} t^{\ell}$. Thus the acceptability relation is defined *co-inductively*, given by the greatest fixed point of the functional defined according to the following clauses of Figure 2.7. Proofs by co-induction would allow this later obligation to be dismissed by the *co-inductive hypothesis*.

Fine print: To be precise, we take as our starting point *uniform* kCFA rather than a kCFA in which,

$$\widehat{\mathbf{Cache}} \quad = \quad (\mathbf{Lab} \times \mathbf{Env} \to \widehat{\mathbf{Val}}) \times (\mathbf{Var} \times \mathbf{Env} \to \widehat{\mathbf{Val}})$$

The differences are immaterial for our purposes. See Nielson et al. (1999) for details and a discussion on the use of coinduction in specifying static analyses.

Having established the foundations of evaluation and analysis, we now turn to the foundations of our tools and techniques employed in the investigation of program analysis.

2.4 Computational Complexity

2.4.1 Why a Complexity Investigation?

Static analysis can be understood as a "technique for computing conservative approximations of solution for undecidable problems."[3] Complexity characterizations therefore offer a natural refinement of that understanding.

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.

We are motivated as well by yardsticks such as Shannon's theorem from information theory (Shannon 1948): 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 are 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?

In the closely related area of pointer analysis, computational complexity has played a prominent role in the development and evaluation of analyses.[4] It was the starting point for the widely influential Landi and Ryder (1992), according to the authors' retrospective (2004).

The theory of computational complexity has proved to be a fruitful tool in relating seemingly disparate computational problems. Through notions of logspace

[3] Quoted from Michael Schwartzbach's 2009 Static Analysis course description from University of Aarhus (http://www.brics.dk/~mis/static.html/, accessed June 3, 2009). The same sentiment is expressed in Patrick Cousot's 2005 MIT Abstract Interpretation lecture notes on undecidability, complexity, automatic abstract termination proofs by semidefinite programming (http://web.mit.edu/16.399/www/).

[4] See section 6.7 for details.

reductions[5] between problems, what may have appeared to be two totally unrelated problems can be shown to be, in fact, so closely related that a method for efficiently computing solutions to one can be used as a method for efficiently computing the other, and *vice versa*. For example, at first glance, 0CFA and circuit evaluation have little to do with each other. Yet, as shown in this dissertation, the two problems are intimately related; they are both complete for PTIME.

There are two fundamental relations a problem can have to a complexity class. The problem can be *included* in the complexity class, meaning the problem is no harder than the hardest problems in the class. Or, the problem can be a *hard* problem within the class, meaning that no other problem in the class is harder than this one. When a problem is both included and hard for a given class, it said to be *complete* for that class; it is as hard as the hardest problems in the class, but no harder.

Inclusion results establish feasibility of analysis—it tells us analysis can be performed within some upper-bound on resources. These results are proven by constructing efficient and correct program analyzers, either by solving the analysis problem directly or reducing it to another problem with a known inclusion.

Hardness results, on the other hand, establish the minimum resource requirements to perform analysis in general. They can be viewed as lower-bounds on the difficulty of analysis, telling us, for example, when no amount of cleverness in algorithm design will improve on the efficiency of performing analysis. So while inclusion results have an existential form: "there exists an algorithm such that it operates within these bounded resources," hardness results have a universal form: "for all algorithms computing the analysis, at least these resources can be consumed."

Whereas inclusion results require clever algorithmic insights applied to a program analyzer, hardness results require clever exploitation of the analysis to perform computational work.

Lower bounds are proved by giving reductions—efficient compilers—for transforming instances of some problem that is known to be hard for a class, (e.g. circuit evaluation and PTIME) into instances of a flow analysis problem. Such a reduction to a flow analysis problem establishes a *lower bound* on the complexity

[5]Logspace reductions are essentially memory efficient translations between instances of problems. The PL-theorist may be most comfortable thinking of these reductions as space-efficient compilers.

of flow analysis: solving the flow problem must be *at least as hard* as solving the original problem, which is known to be of the hardest in the class.

The aim, then, is to solve hard problems by make principled use of the analysis. From a programming language perspective, the analyzer can be regarded as an evaluator of a language, albeit a language with implicit resource bounds and a (sometimes) non-standard computational model. Lower bounds can be proved by writing computationally intensive programs in this language. That is, proving lower bounds is an exercise in clever hacking in the language of analysis.

For flow analysis, inclusion results are largely known. Simple algorithmic analysis applied to the program analyzer is often sufficient to establish an upper bound.

Much work in the literature on flow analysis has been concerned with finding more and more efficient ways to compute various program analyses. But while great effort has been expended in this direction, there is little work addressing the fundamental limits of this endeavour. Lower-bounds tell us to what extent this is possible.

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

Viewed from another perspective, hardness results can be seen as a characterization of the *expressiveness* of an analysis; it is a measure of the work an analysis is capable of doing. The complexity and expressivity of an analysis are two sides of the same coin and analyses can be compared and characterized by the class of computations each captures. In their definitive study, Nielson et al. (1999) remark, "Program analysis is still far from being able to precisely relate ingredients of different approaches to one another," but we can use computational complexity theory as an effective tool in relating analyses. Moreover, insights gleaned from this understanding of analysis can inform future analysis design. To develop rich analyses, we should expect larger and larger classes to be captured. In short: com-

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

putational complexity is a means to both organize and extend the universe of static analyses.

Other research has shown a correspondence between 0CFA and certain type systems (Palsberg and O'Keefe 1995; Heintze 1995) and a further connection has been made between intersection typing and kCFA (Mossin 1997b; Palsberg and Pavlopoulou 2001). Work has also been done on relating the various flavors of control flow analysis, such as 0CFA, kCFA, polymorphic splitting, and uniform kCFA (Nielson and Nielson 1997). Moreover, control flow analysis can be computed under a number of different guises such as set-based analysis (Heintze 1994), closure analysis (Sestoft 1988, 1989), abstract interpretation (Shivers 1991; Tang and Jouvelot 1994; Might and Shivers 2006a; Might 2007; Midtgaard and Jensen 2008, 2009), and type and effect systems (Faxén 1995; Heintze 1995; Faxén 1997; Banerjee 1997).

We believe a useful taxonomy of these and other static analyses can be derived by investigating their computational complexity. Results on the complexity of static analyses are way of understanding when two seemingly different program analyses are in fact computing the same thing.

2.4.2 Complexity Classes

In this section, we review some basic definitions about complexity classes and define the flow analysis problem.

A complexity class is specified by a model of computation, a mode of computation (e.g. deterministic, non-deterministic), the designation of a unit of *work*—a resource used up by computation (e.g. time, space), and finally, a function f that bounds the use of this resource. The complexity class is defined as the set of all languages decided by some Turing machine M that for any input x operates in the given mode within the bounds on available resources, at most $f(|x|)$ units of work (Papadimitriou 1994, page 139).

Turing machines are used as the model of computation, in both deterministic and non-deterministic mode. 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 is the tape alphabet, where $\Sigma \subseteq \Gamma$. For a deterministic machine, the transition function, δ, is a partial function that maps the current machine state and tape contents under the head position to the next machine state, a symbol to write on the tape under the head, and left or right shift operation for moving the head. For a non-deterministic machine, the transition function is actually a relation and may map each machine configuration to multiple successor configurations. The states q_0, q_a, and q_r are the machine's initial, accept, and reject states, respectively. Transitions consume one unit of time and space consumption is measured by the amount of tape used.

Definition 1. *Let $f : \mathcal{N} \to \mathcal{N}$. We say that machine M* operates within time $f(n)$ *if, for any input string x, the time required by M on x is at most $f(|x|)$ where $|x|$ denotes the length of string x. Function $f(n)$ is a* time bound *for M.*

Let $g : \mathcal{N} \to \mathcal{N}$. We say that machine M operates within space $g(n)$ *if, for any input string x, the space required for the work tape of M on x is at most $g(|x|)$. Function $g(n)$ is a* space bound *for M.*

Space bounds do not take into consideration the amount of space needed to represent the input or output of a machine, but only its working memory. To this end, one can consider Turing machines with three tapes: an input, output and work tape. (Such machines can be simulated by single tape Turing machines with an inconsequential loss of efficiency). The input tape is read-only and contains the input string. The work tape is initially empty and can be read from and written to. The output tape, where the result of computation is written, is initially empty and is write only. A space bound characterizes the size of the work only. See Papadimitriou (1994, Sections 2.3–5) or Garey and Johnson (1979, Section 7.5) for further details.

A complexity class is a set of languages representing decision problems that can be decided within some specified bound on the resources used, typically time and space. Suppose the decision problem can be decided by a deterministic Turing machine operating in time (space) $f(n)$, we say the problem is in DTIME($f(n)$) (DSPACE($f(n)$)); likewise, if a problem can be decided by a non-deterministic Turing machine operating in time $f(n)$, we say the problem is in NTIME($f(n)$).

We make use of the following standard complexity classes:

$$\begin{array}{rrcl} & \text{LOGSPACE} & = & \bigcup_{j>0} \text{DSPACE}(j \log n) \\ \subseteq & \text{PTIME} & = & \bigcup_{j>0} \text{DTIME}(n^j) \\ \subseteq & \text{NPTIME} & = & \bigcup_{j>0} \text{NTIME}(n^j) \\ \subseteq & \text{EXPTIME} & = & \bigcup_{j>0} \text{DTIME}(2^{n^j}) \end{array}$$

In addition to the above inequalities, it is known that $\text{PTIME} \subset \text{EXPTIME}$.

What is the difficulty of computing within this hierarchy? What are the sources of approximation that render such analysis tractable? We examine these questions in relation to *flow analysis problems*, which are *decision problems*, computational problems that require either a "yes" or "no" answer, and therefore are insensitive to output size (it is just one bit).

The flow analysis decision problem we examine is the following:

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

2.5 Proving Lower Bounds: The Art of Exploitation

Program exploitation—a staple of hacking—is simply a clever way of making a program do what you want it to do, even if it was designed to prevent that action (Erickson 2008, page 115). This is precisely the idea in proving lower bounds, too.

The program to be exploited, in this setting, is the static analyzer. What we would like to do varies, but generally we want to exploit analysis to solve various computationally difficult classes of problems. In some cases, what we want to do is *evaluate*, rather than approximate, the program being analyzed. In this sense, we truly subvert the analyzer by using it to carry out that which it was designed to avoid (to discover *without actually running* (Muchnick and Jones 1981, page xv)).

The approach of exploiting analysis for computation manifests itself in two ways in this dissertation:

1. Subverting abstraction

   The first stems from a observation that perhaps the languages of abstract and concrete interpretation intersect. That is, abstract interpretation makes approximations compared to concrete interpretation, but perhaps there is a subset of programs on which no approximation is made by analysis. For this class of programs, abstract and concrete interpretation are synonymous. Such a language certainly exists for all of the flow analyses examined in this dissertation. We conjecture that in general, for every useful abstract interpretation, there is a subset of the analyzed language for which abstract and concrete interpretation coincide. By identifying this class of programs, lower bounds can be proved by programming within the subset.

   One of the fundamental ideas of computer science is that "we can regard almost any program as the evaluator for some language" (Abelson and Sussman 1996, page 360). So it is natural to ask of any algorithm, *what is the language being evaluated?* The question is particularly relevant when asked of an abstract evaluator. We can gain insight into an analysis by comparing the language of the abstract evaluator to the language of the concrete evaluator.

   So a program analysis itself can be viewed as a kind of programming language, and an analyzer as a kind of evaluator. Because of the requisite decidability of analysis, these languages will come with implicit bounds on computational resources—if the analysis is decidable, these languages cannot be Turing-complete. But lower bounds can be proved by clever hacking within the unconventional language of analysis.

   This approach is the subject of chapter 3.

2. Harnessing re-evaluation

   The second way analysis can be exploited is to identify the sources of approximation in the analysis and instead of avoiding them, (turning the abstract into the concrete as above), harness them for combinatorial power. In this approach, lower bounds can be proved by programming the language of the analysis in a way that has little to do with programming in the language of concrete interpretation.

   Researchers have made empirical observations that computing a more precise analysis is often cheaper than performing a less precise one. The less precise analysis "yields coarser approximations, and thus induces more

merging. More merging leads to more propagation, which in turn leads to more reevaluation" (Wright and Jagannathan 1998). Might and Shivers (2006b) make a similar observation: "imprecision reinforces itself during a flow analysis through an ever-worsening feedback loop." For the purposes of proving lower bounds, we are able to harness this re-evaluation as a computation engine.

This approach is the subject of chapter 5.

Chapter 3

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 and 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.

3.1 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.

3.2 0CFA

Something interesting happens when $k = 0$. Notice in the application rule of the kCFA abstract evaluator of Figure 2.5 that environments are extended as $\rho[x \mapsto \lceil \delta\ell \rceil_k]$. When $k = 0$, $\lceil \delta\ell \rceil_0 = \epsilon$. 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 section 2.3 can be eliminated from the analysis altogether and the cache no longer needs a contour argument. Likewise, the set of abstract values collapses from $\mathcal{P}(\mathbf{Term} \times \mathbf{Env})$ into $\mathcal{P}(\mathbf{Term})$.

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:

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

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

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 et al. (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 3.1.

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

Figure 3.1: 0CFA abstract cache acceptability.

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 $\widehat{\mathsf{C}}, \hat{\mathsf{r}} \models e$. The above specification of acceptability, when read as a table, defines a functional, which is monotonic, has a fixed point, and $\models$ is de-

fined 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:

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

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.

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

Figure 3.2: Abstract evaluator $\mathcal{A}_0$ for 0CFA, imperative style.

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

Fine Print: A single iteration of $\mathcal{A}_0[\![e]\!]$ may in turn make a recursive call $\mathcal{A}_0[\![e]\!]$ 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 decision problems answered by the analysis are clearly computable within polynomial time.

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

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:

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

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

3.3 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" (Midtgaard 2007, page 4): 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 3.3.

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

Figure 3.3: Simple closure analysis abstract cache acceptability.

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

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

The abstract evaluator $\mathcal{A}_0[\![\cdot]\!]$ 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.

[2]The fine print of section 3.2 applies as well.

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):

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

3.4 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 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.

[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.

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

Figure 3.4: Evaluator $\mathcal{E}'$.

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 3.4, which is slightly modified from the one given in Figure 2.1. 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 $\mathcal{E}'$ will be useful in reasoning about linear programs, but it should be clear that it is equivalent to the original, standard evaluator $\mathcal{E}$ of Figure 2.1.

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

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 2. *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* $\mathcal{E}'[\![t^{\ell}]\!]\rho = c$*, then* $|c| \leq |t, \rho|$*.*

Proof. Straightforward by induction on $|t, \rho|$, reasoning by case analysis on t. Observe 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 3. *The set of labels in a given term, expression, environment, or closure is defined as follows:*

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

Definition 4. *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

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

Figure 3.5: Abstract evaluator $\mathcal{A}_0$ for 0CFA, functional style.

Figure 3.5 gives a "cache-passing" functional algorithm for $\mathcal{A}_0[\![\cdot]\!]$ of section 3.3. It is equivalent to the functional style abstract evaluator of Figure 2.6 specialized by letting $k = 0$. We now state and prove the main theorem of this section in terms of this abstract evaluator.

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

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

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,

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

  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

$$\begin{aligned} \mathcal{E}'[\![t_1]\!]\rho \restriction \mathbf{fv}(t_1^{\ell_1}) &= \langle v_1, \rho_1 \rangle = \langle \lambda x.t_0^{\ell_0}, \rho_1 \rangle, \\ \mathcal{E}'[\![t_2]\!]\rho \restriction \mathbf{fv}(t_2^{\ell_2}) &= \langle v_2, \rho_2 \rangle, \\ \mathcal{A}_0[\![t_1]\!]\widehat{\mathsf{C}} &= \widehat{\mathsf{C}}_1, \text{ and} \\ \mathcal{A}_0[\![t_2]\!]\widehat{\mathsf{C}} &= \widehat{\mathsf{C}}_2. \end{aligned}$$

  Clearly, for $i \in \{1, 2\}$, $\widehat{\mathsf{C}} \vdash t_i, \rho \restriction \mathbf{fv}(t_i)$ 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 is clear that $\widehat{\mathsf{C}}_3 \models t_i^{\ell_i}$. Furthermore,

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

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

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

Let

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

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 $\mathcal{E}'[\![t^{\ell}]\!]\emptyset = \langle v, \rho' \rangle$ 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 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.

3.5 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 chapter 2, 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.

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

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

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

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

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:

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

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).

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:

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

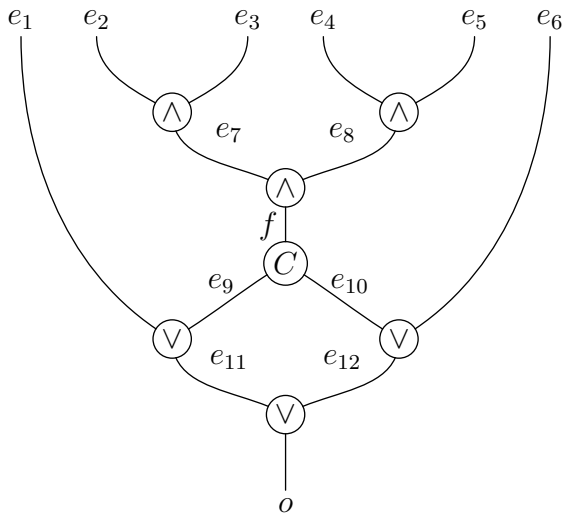


Figure 3.6: An example circuit.

Continuation-passing style code such as `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 3.6, 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.

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

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.

3.6 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 6.4 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?

3.6.1 Ashley and Dybvig's Sub-0CFA

Ashley and Dybvig (1998) developed a general framework for specifying and computing flow analyses; instantiations of the framework include 0CFA and the polynomial 1CFA of Jagannathan and 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).

3.6.2 Subtransitive 0CFA

Heintze and 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 combinatorial 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 and 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 chapter 4 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).

[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

3.7 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 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 (Papadimitriou 1994, page 377).

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.

Chapter 4

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 (section 4.4) 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 section 3.4: analysis and evaluation are synonymous for linear programs.

4.1 Sharing Graphs for Static Analysis

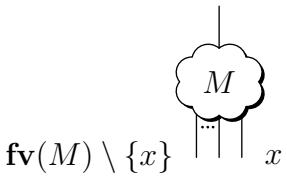
In general, the sharing graph of a term will consist of a distinguished *root* wire from which the rest of the term's graph "hangs."

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

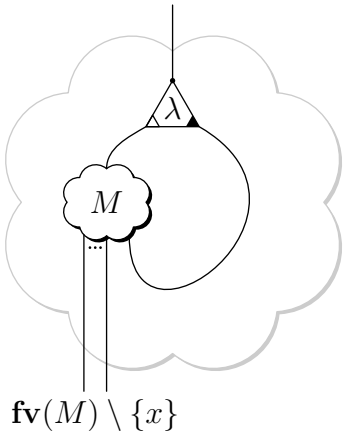
Graphs consist of ternary abstraction (λ), apply (@), sharing ($\triangledown$) nodes, and unary weakening ($\odot$) nodes. Each node has a distinguished *principal* port. For unary nodes, this is the only port. The ternary nodes have two *auxiliary* ports, distinguished as the *white* and *black* ports.

- 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,

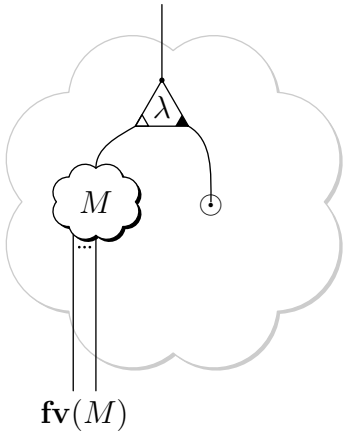


the abstraction $\lambda x.M$ is formed as,



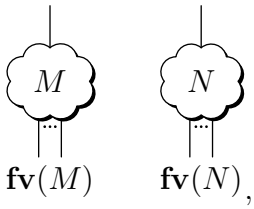
$\lambda x.M, x \in \mathbf{fv}(M)$

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,



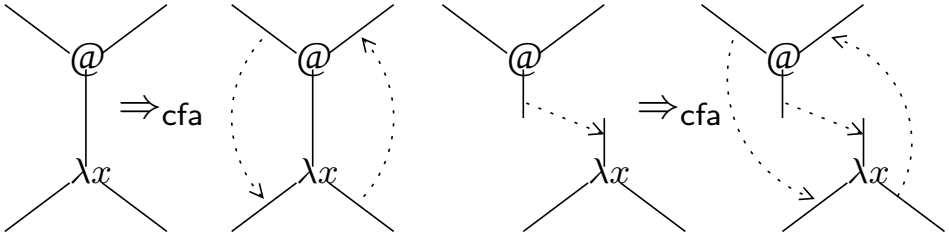
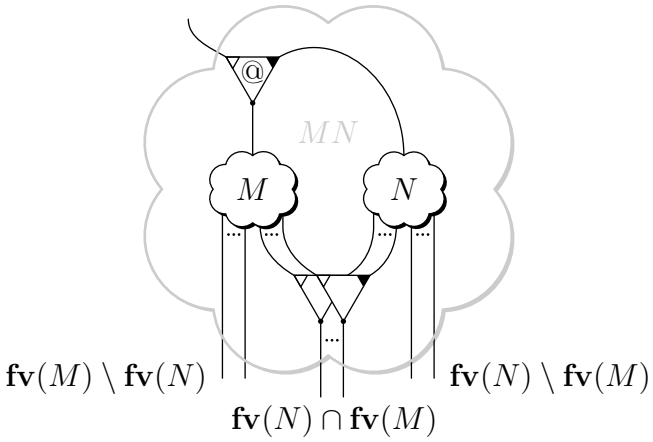


Figure 4.1: CFA virtual wire propagation rules.

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.

4.2 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 4.1 defines the virtual path propagation rules. Note that a wire can be

identified by its label or a variable name.[1] 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. $\ell \equiv \ell'$.
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 and Schwartzbach 1995; Biswas 1997; Heintze and McAllester 1997b; Midtgaard and Jensen 2008, 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 chapter 2 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 4.1, 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

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

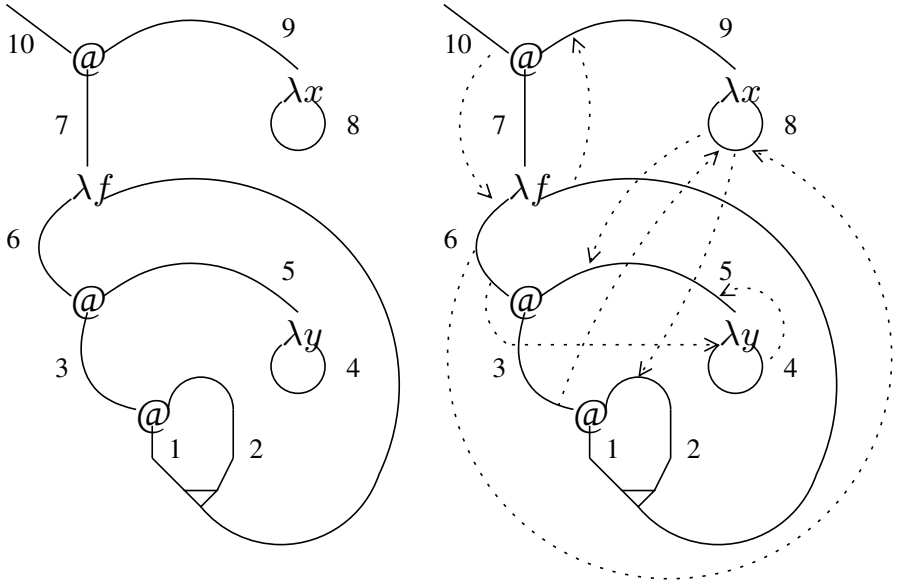


Figure 4.2: Graph coding and CFA graph of $(\lambda f.ff(\lambda y.y))(\lambda x.x)$.

algorithm constructs answers that satisfy the acceptability relation specifying the analysis. Moreover, this algorithm constructs least solutions according to the partial order given in section 2.3.

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 4.2 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:

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

4.3 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. (Sørensen and Urzyczyn 2006, p. v)

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:

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

$$\textsc{App}\ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

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

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

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

4.3.1 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 4.3.

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

Figure 4.3: MLL sequent rules.

4.3.2 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 and Urzyczyn 2006).[2]

(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`

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

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.

4.4 η-Expansion and LOGSPACE

4.4.1 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

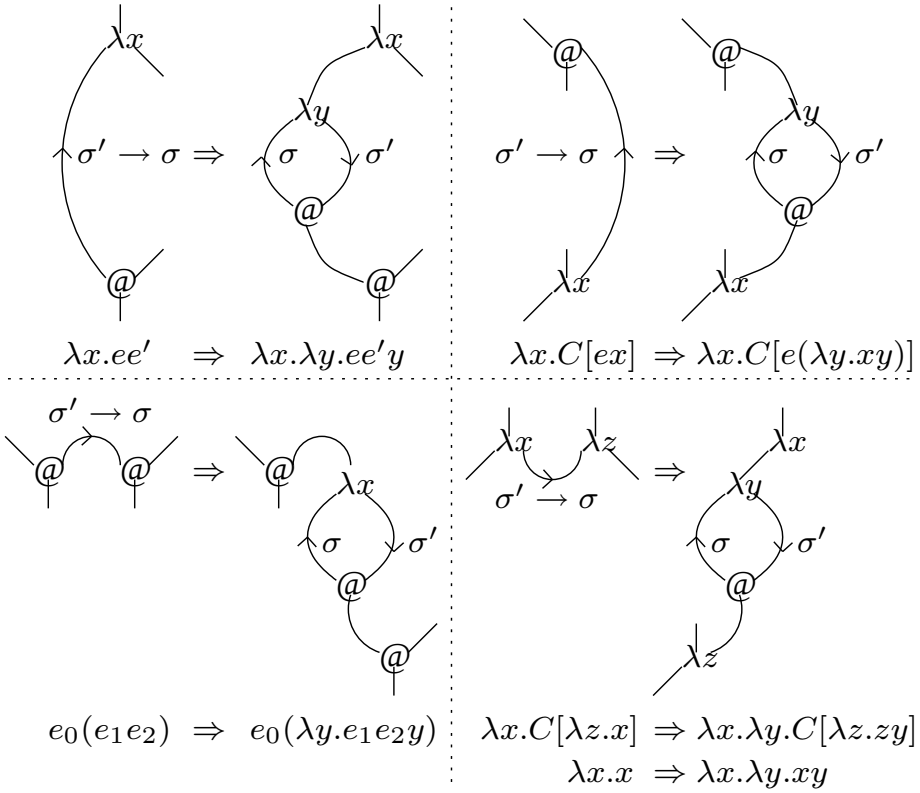


Figure 4.4: Expansion algorithm.

axioms using the following transformation, analogous to η-expansion:

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

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.[3] A LOGSPACE evaluation is then polynomial-time and -space in the original circuit description.

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

The program transformation corresponding to the above proof expansion is a version of η-expansion: see Figure 4.4. 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.

4.4.2 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` from the above circuit example have type `'a * 'a -> 'a * 'a`.[4] 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 4.5) follow the rules for sequent formation (Figure 4.3) almost identically.

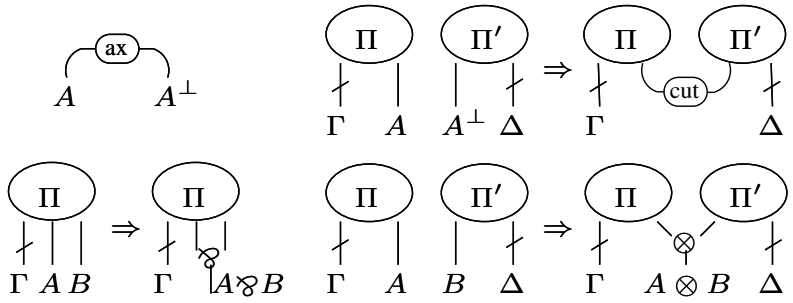


Figure 4.5: MLL proofnets.

[4]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 $\otimes$.

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.[5] 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 4.6.

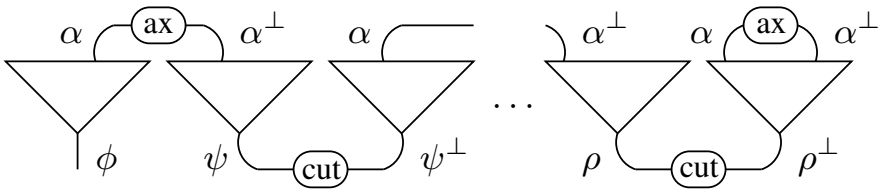


Figure 4.6: MLL proofnet with atomic axioms.

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

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.*

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

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 ($\parr$).

4.4.3 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,b):

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.

4.4.4 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 and 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.*

4.4.5 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.[6] 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))
```

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

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.)

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.[7]

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

4.5 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 and Mairson 2000) and our graphical formulation of control flow analysis carries over without modification.

Lawall and 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, $\lambda_{\mathcal{K}}$). 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 and 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 4.7 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 ($\lambda v \ldots$) 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 4.7 gives the CFA graph for the program:

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

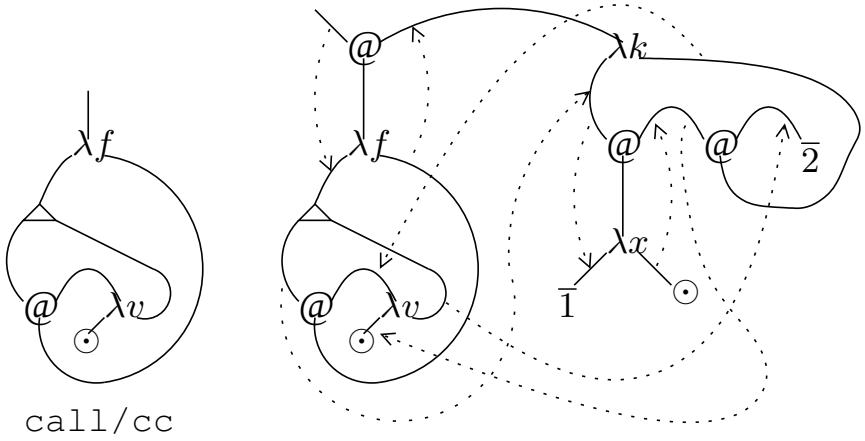


Figure 4.7: Graph coding of `call/cc` and example CFA graph.

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 4.7, 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 $\lambda_{\mathcal{K}}$, 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 λ_K. The type-based expansion algorithm of Figure 4.4 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 and Turbak 2000; Palsberg and Pavlopoulou 2001; Wells et al. 2002; Banerjee and 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 Mossin (1997b, page 78) 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 and Laneve (1995) as an approximation to "legal paths" (Lévy 1978) and results on proof normalization in linear logic (Mairson and Terui 2003) informed the novel flow analysis algorithms presented here.

Chapter 5

kCFA and EXPTIME

In this chapter, 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.

5.1 Shivers' kCFA

As noted in section 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 (chapter 3).

As described in chapter 3, 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 section 3.2,

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

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, 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 kCFA 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 kCFA 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 kCFA 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 kCFA work (2004):

> It did not take long to discover that the basic analysis, for any $k > 0$, was intractably slow for large programs. In the ensuing years, researchers have expended a great deal of effort deriving clever ways to tame the cost of the analysis.

A fairly straightforward calculation—see, for example, Nielson et al. (1999)—shows that 0CFA can be computed in polynomial time, and for any $k > 0$, kCFA can be computed in exponential time.

These naive upper bounds suggest that the kCFA hierarchy is essentially *flat*; re-

searchers subsequently "expended a great deal of effort" trying to improve them.[1] 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.[2]

  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.

[1] 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.

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

> 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.

We proceed by first bounding the complexity of kCFA from above, showing that kCFA can be solved in exponential time (section 5.2). This is easy to calculate and is known (Nielson et al. 1999). Next, we bound the complexity from below by using kCFA as a SAT-solver. This shows kCFA is at least NPTIME-hard (section 5.3). 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 section 5.4. These ideas are then scaled up and applied in section 5.5 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 kCFA, thus showing kCFA to be complete for EXPTIME.

5.2 kCFA is in EXPTIME

Recall the definition of kCFA from section 2.3. 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 kCFA is contained in* EXPTIME.

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

5.3 kCFA is NPTIME-hard

Because kCFA 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

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

Figure 5.1: NPTIME-hard construction for kCFA.

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 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 non-deterministic algorithm might compute using the "collection semantics" $\mathcal{E}[\![t^\ell]\!]^\rho_\delta$, 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.

5.4 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, $\texttt{Implies}\ \texttt{True}\ \texttt{False}$ causes `False` to flow out. Unlike in normal evaluation, where within a given scope we know that multiple occurrences of the same variable refer to the same value, in the approximation of analysis, multiple occurrences of the same variable range over *all* values that they are possible bound to *independent of each other*.

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

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

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.

5.5 kCFA is EXPTIME-hard

5.5.1 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 section 5.3.

5.5.2 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:

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

Figure 5.2: 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.)

5.5.3 Transition Function

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

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

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

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

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

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

(Note that when $H' = C'$, we have already written the salient tuple using the transition rule.) This rule *communicates* state and head position (for the first tuple

computed with time stamp $T + 1$, where the head and cell address coincided) to all the other tuples coding the rest of the Turing machine tape.

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

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

Clearly, these three rules can be coded by a single Boolean circuit, and we have all the required Boolean logic at our disposal from section 3.5.

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 5.3. The `Copy` functions just copy enough of the input for the separate cal-

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

Figure 5.3: Turing machine transition function construction.

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

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

Figure 5.4: EXPTIME-hard construction for kCFA.

5.5.4 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 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 kCFA, which can be assumed to be either $\mathbf{Y}$, or an exponential function composer. There just has to be enough iteration of the transition function to produce a fixed point for the flow analysis.

To make the coding easy, we just assume 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 kCFA analysis of e ($k > 0$),* `True` *flows into ℓ iff M accepts x.*

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

5.6 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 kCFA 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 kCFA 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 kCFA, 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 $\mathcal{E}$ of section section 2.2 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 and Jagannathan 1998). Might and 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` $x\,x$, 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.

5.7 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 $\mathbf{Y}$ combinator. Exponential time is the "limit" because with a polynomial-length tuple (as constrained by a logspace reduction), you can only code an exponential number of closures.

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 and 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.

5.8 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.

Chapter 6

Related Work

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.

6.1 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 chapter 3.

In the setting of higher-order programming languages, Melski and 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 6.3 for further discussion.

The impact of pattern matching on analysis complexity is further examined by Heintze and McAllester (1997b), which shows how deep pattern matching affects

monovariant analysis, making it complete for EXPTIME.

6.2 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 and Wand 1997). This must analysis is a simple instance of tracking linearity information in order to increase the precision of the analysis. Might and 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 chapter 3.

Damian and 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 and Mairson 1991; Mairson 2004).[1]

The observation translates to flow analysis, as shown in chapter 3, but in a typed

[1]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.

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 and Mairson 1991) and rank-bound intersection type inference (Møller Neergaard and Mairson 2004). See section 6.10 for further discussion.

6.3 Context-Free-Language Reachability

Melski and 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 and 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-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, Melski and Reps (2000, section 5) 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, Melski and Reps (2000, section 6) 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. Heintze (1994, page 314) 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," (Reps 1996, section 2).

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*.

6.4 2NPDA and the Cubic Bottleneck

The class 2NPDA contains all languages that are recognizable by a two-way non-deterministic push-down automaton.[2] 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 and McAllester (1997c) proved deciding a monovariant flow analysis problem to be at least as hard as 2NPDA, and argued this provided evidence the "cubic bottleneck" of flow analysis was unlikely to be overcome since the best known algorithm for 2NPDA 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 and McAllester 1997c,a; Melski and Reps 2000; McAllester 2002; Van Horn and Mairson 2008b). 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 and 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 2NPDA. Moreover, refined analyses similar to Heintze and McAllester (1997b) that incorporate notions of reachability to improve precision remain subcubic. See Midtgaard and Van Horn (2009) for details.

0CFA is complete for both 2NPDA (Heintze and McAllester 1997c) and PTIME (chapter 3). Yet, it is not clear what relation these class have to each other.

[2] This section is derived from material in Midtgaard and Van Horn (2009).

The 2NPDA 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 2NPDA."

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

6.5 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 approxi-

mation that results from nonlinearity.

6.6 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 and Ungar 1990; Palsberg and Schwartzbach 1991; Bacon and 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 and 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, Shivers (1991, page 6)[3] had written essentially the same:

[3]It is a testament to Shivers' power as a writer that his original story has been told over and over

> 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 and Jensen (2003) give a reformulation of several class analyses, including that of Palsberg and Schwartzbach (1991); Bacon and 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 inclu-

again in so many places, usually with half the style.

sion, equality, and optimistic based class analyses with close analogies to 0CFA, simple closure analysis, and rapid type analysis (Bacon and 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.[4]

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 and Chambers 2001), or partially flow sensitivity (Rinetzky et al. 2008).

Grove and Chambers (2001) provide a framework for a functional and object-oriented hybrid language that can be instantiated to obtain a kCFA analysis and an object-oriented analogue called *k-l-CFA*. There is a discussion and references in Section 9.1. In this discussion, Grove and 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 kCFA to object-oriented programs," and although this paper actually has analogies to kCFA 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' kCFA work or any functional flow analysis work.

The construction of Figure 5.2 can be translated in an object-oriented language such as Java, as given in Figure 6.1.[5] Functions are simulated as objects with an apply method. The crucial subterm in Figure 6.1 is the construction of the list

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

[5] 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`.

```
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};}}
```

Figure 6.1: Translation of kCFA EXPTIME-construction into an object-oriented language.

`{x1,x2,..xn}`, where 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 x_i to the calling context in which it was bound to either true or false for all possible combinations. Grove and 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 section 2.5 and the broader applicability of the approach to proving lower bounds employed in chapter 3. 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 section 2.5 can be applied to prove lower bounds about this analysis since it has nothing to do with approximating evaluation.

6.7 Pointer Analysis

Just as flow analysis plays a fundamental role in the analysis of higher-order functional programs, *pointer analysis*[6] 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

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

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 (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 and 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.[7]

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

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

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

6.8 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 and 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 and 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 section 5.5 has a conceptual basis in Datalog complexity research (Hillebrand et al. 1995; Gaifman et al. 1993). See section 6.5 for a discussion.

6.9 Termination Analysis

Termination analysis of higher-order programs (Jones and Bohr 2008; Sereni and 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 and Jones 2005; Sereni 2007). Once a call-graph is constructed, the so-called "size-change" principle[8]

[8]The size-change principle has enjoyed a complexity investigation in its own right (Lee et al. 2001;

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. 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.

6.10 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 and 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.[9] This observation motivates the analysis of type

Ben-Amram and Lee 2007).

[9] An aberrant case of this phenomenon is examined by Møller Neergaard and Mairson (2004), which analyzed a type system where normalization and type inference are synonymous in *every*

inference described by Mairson (1990, 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 and 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.

> **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

case. The tractability of type inference thus implied a certain inexpressiveness of the language.

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.[10] 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, 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.[11]

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:[12]

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

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, 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.

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

[11] See section 6.7 for more on the relation of pointer analysis and flow analysis.

[12] In the survey, *Type systems for programming languages*, Mitchell (1990) attributes this observation to Albert Meyer. Henglein and Mairson (1991, page 122) 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.

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 $\wedge$:[13]

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

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 and 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 typ-

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

ing polymorphic recursion, local definitions, conditionals and pattern matching (Damiani and Prost 1998; Damiani 2003; Banerjee and 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-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 and 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 and 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 and 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 and 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.

Chapter 7

Conclusions and Perspective

7.1 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.

7.2 Future Work

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

7.2.1 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 and Ryder (1991); Landi (1992a,b); Choi et al. (1993); Horwitz (1997); Muth and 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.

7.2.2 Polyvariant, Polynomial Flow Analyses

To echo the remark of Bravenboer and 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 and Weeks 1995; Nielson and Nielson 1997), but the landscape of tractable, context-sensitive flow analyses is mostly open and in need of development.

The ingredients, detailed in chapter 5, 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).

7.2.3 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 section 5.2). 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 and 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.

7.2.4 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 and 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 chapter 4 can be adapted to exploit these breakthroughs.

7.2.5 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 and 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 and 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."

Bibliography

Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, USA, 1996. ISBN 0262011530. Cited on pages ix, 10, and 27.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullmana. Time and tape complexity of pushdown automaton languages. *Information and Control*, **13**(3):186–206, 1968. Cited on pages 47 and 92.

María Alpuente and Germán Vidal, editors. *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*. Springer, 2008. ISBN 978-3-540-69163-1. Cited on pages 116, 126, and 134.

Torben Amtoft and Franklyn A. Turbak. Faithful translations between polyvariant flows and polymorphic types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 26–40. Springer-Verlag, London, UK, 2000. ISBN 3-540-67262-1. Cited on page 70.

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. Ph.D. thesis, DIKU, University of Copenhagen, 1994. Cited on page 99.

J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Trans. Program. Lang. Syst.*, **20**(4):845–868, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/291891.291898. Cited on pages 6, 29, 48, and 100.

Andrea Asperti and Cosimo Laneve. Paths, computations and labels in the λ-calculus. *Theor. Comput. Sci.*, **142**(2):277–297, 1995. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(94)00279-7. Cited on page 70.

Andrea Asperti and Harry G. Mairson. Parallel beta reduction is not elementary recursive. In POPL 1998, pages 303–315. doi: http://doi.acm.org/10.1145/268946.268971. Cited on page 65.

Andrew Edward Ayers. *Abstract analysis and optimization of Scheme*. Ph.D. thesis, Cambridge, Massachusetts, USA, 1993. Cited on page 55.

David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In OOPSLA 1996, pages 324–341. doi: http://doi.acm.org/10.1145/236337.236371. Cited on pages 94, 95, and 96.

Anindya Banerjee. A modular, polyvariant and type-based closure analysis. In Berman (1997), pages 1–10. doi: http://doi.acm.org/10.1145/258948.258951. Cited on pages 24 and 100.

Anindya Banerjee and Thomas Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathematical. Structures in Comp. Sci.*, **13**(1):87–124, 2003. ISSN 0960-1295. doi: http://dx.doi.org/10.1017/S0960129502003845. Cited on pages 70 and 106.

Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984. ISBN 0-444-87508-5. Cited on page ix.

Henk P. Barendregt. Functional programming and lambda calculus. In van Leeuwen (1990), pages 321–363. Cited on page ix.

Julien Basch, Sanjeev Khanna, and Rajeev Motwani. On diameter verification and Boolean matrix multiplication. Technical report, Stanford University, Stanford, California, USA, 1995. Cited on page 92.

Amir M. Ben-Amram and Chin Soon Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst.*, **29**(1):5, 2007. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1180475.1180480. Cited on page 102.

A. Michael Berman, editor. *ICFP '97: Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, New York, USA, 1997. ISBN 0-89791-918-1. Cited on pages 114 and 120.

Sandip K. Biswas. A demand-driven set-based analysis. In POPL 1997, pages 372–385. doi: http://doi.acm.org/10.1145/263699.263753. Cited on page 55.

Hans-J. Boehm. Partial polymorphic type inference is undecidable. In *SFCS '85: Proceedings of the 26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*, pages 339–345. IEEE Computer Society, Washington, DC, USA, 1985. ISBN 0-8186-0844-4. doi: http://dx.doi.org/10.1109/SFCS.1985.44. Cited on page 106.

Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2009. To appear. Cited on pages 100, 101, and 110.

G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In H. Ganzinger and N. Jones, editors, *Programs as data objects*, pages 42–62. Springer-Verlag, New York, New York, USA, 1985. ISBN 0-387-16446-4. Cited on pages 85 and 110.

Felice Cardone and J. Roger Hindley. History of lambda-calculus and combinatory logic. Technical Report MRRS-05-06, Swansea University Mathematics Department Research Report, 2006. To appear in *Handbook of the History of Logic, Volume 5*, D. M. Gabbay and J. Woods, editors. Cited on pages ix and 104.

Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 115–125. ACM, New York, New York, USA, 2003. ISBN 1-58113-628-5. doi: http://doi.acm.org/10.1145/604131.604142. Cited on pages 99, 100, and 109.

Venkatesan T. Chakaravarthy and Susan Horwitz. On the non-approximability of points-to analysis. *Acta Informatica*, **38**(8):587–598, 2002. doi: http://dx.doi.org/10.1007/s00236-002-0081-8. Cited on page 100.

Craig Chambers and David Ungar. Interactive type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 150–164. ACM, New York, New York, USA, 1990. ISBN 0-89791-364-7. doi: http://doi.acm.org/10.1145/93542.93562. Cited on page 94.

Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *STOC '07: Proceedings of the thirty-ninth annual ACM Symposium on Theory of Computing*, pages 590–598. ACM, New York, New York, USA, 2007. ISBN 978-1-59593-631-8. doi: http://doi.acm.org/10.1145/1250790.1250877. Cited on page 92.

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Complexity of points-to analysis of Java in the presence of exceptions. *IEEE Trans. Softw. Eng.*, **27**(6):481–512, 2001. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.926173. Cited on pages 100 and 109.

Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In POPL 2008, pages 159–169. doi: http://doi.acm.org/10.1145/1328438.1328460. Cited on pages 48 and 92.

Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In POPL 1993, pages 232–245. doi: http://doi.acm.org/10.1145/158511.158639. Cited on pages 100 and 109.

Marco Comini, Ferruccio Damiani, and Samuel Vrech. On polymorphic recursion, type systems, and abstract interpretation. In Alpuente and Vidal (2008), pages 144–158. doi: http://dx.doi.org/10.1007/978-3-540-69166-2_10. Cited on page 106.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, New York, New York, USA, 1977. doi: http://doi.acm.org/10.1145/512950.512973. Cited on pages 2 and 17.

Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, **2**(4):511–547, 1992. Cited on page 2.

Haskell B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, **23**(2):83–92, 1969. doi: http://dx.doi.org/10.1111/j.1746-8361.1969.tb01183.x. Cited on page 104.

Luis Damas. *Type assignment in programming languages*. Ph.D. thesis, University of Edinburgh, Department of Computer Science, 1985. Technical Report CST- 33-85. Cited on page 104.

Daniel Damian and Olivier Danvy. CPS transformation of flow information, Part II: administrative reductions. *J. Funct. Program.*, **13**(5):925–933, 2003. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796803004702. Cited on page 89.

Ferruccio Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. Program. Lang. Syst.*, **25**(4):401–451, 2003. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/778559.778560. Cited on page 106.

Ferruccio Damiani. Rank 2 intersection for recursive definitions. *Fundam. Inf.*, **77**(4):451–488, 2007. ISSN 0169-2968. Cited on page 106.

Ferruccio Damiani and Frédéric Prost. Detecting and removing dead-code using rank 2 intersection. In *TYPES '96: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 66–87. Springer-Verlag, London, UK, 1998. ISBN 3-540-65137-3. Cited on page 106.

Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and Functional Programming*, pages 151–160. ACM Press, New York, New York, USA, 1990. ISBN 0-89791-368-X. doi: http://doi.acm.org/10.1145/91556.91622. Cited on page 68.

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, **18**(6):730–751, 1996. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/236114.236119. Cited on page 67.

Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In POPL 1998, pages 222–236. doi: http://doi.acm.org/10.1145/268946.268965. Cited on page 95.

Roberto Di Cosmo, Delia Kesner, and Emmanuel Polonovski. Proof nets and explicit substitutions. *Mathematical. Structures in Comp. Sci.*, **13**(3):409–450, 2003. ISSN 0960-1295. doi: http://dx.doi.org/10.1017/S0960129502003791. Cited on page 59.

Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In OOPSLA 1996, pages 292–305. doi: http://doi.acm.org/10.1145/236337.236367. Cited on pages 95 and 98.

Cynthia Dwork, Paris C. Kanellakis, and John C. Mitchell. On the sequential nature of unification. *J. Log. Program.*, **1**(1):35–50, 1984. ISSN 0743-1066. doi: http://dx.doi.org/10.1016/0743-1066(84)90022-0. Cited on page 104.

R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, third edition, 2002. Published online: http://www.scheme.com/tspl3/. Cited on page ix.

Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 242–256. ACM, New York, New York, USA, 1994. ISBN 0-89791-662-X. doi: http://doi.acm.org/10.1145/178243.178264. Cited on page 100.

Jon Erickson. *Hacking: The Art of Exploitation*. No Starch Press, 2 edition, 2008. ISBN 1-59327-144-1. Cited on page 26.

Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In Mycroft (1995), pages 136–153. Cited on page 24.

Karl-Filip Faxén. Polyvariance, polymorphism and flow analysis. In *Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, pages 260–278. Springer-Verlag, London, UK, 1997. ISBN 3-540-62503-8. Cited on pages 24 and 100.

Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. 2009. Soon to be published manuscript, in development since 1989. Cited on page 12.

Andrzej Filinski. Declarative continuations: an investigation of duality in programming language semantics. In *Category Theory and Computer Science*, pages 224–249. Springer-Verlag, London, UK, 1989. ISBN 3-540-51662-X. Cited on page 68.

Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. MIT Press, third edition, 2008. ISBN 0-262-06279-8. Cited on page ix.

Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, first edition, 1992. ISBN 0-262-06145-7. Cited on page ix.

Haim Gaifman, Harry Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. *J. ACM*, **40**(3):683–713, 1993. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/174130.174142. Cited on pages 93 and 101.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, New York, USA, 1979. ISBN 0716710447. Cited on page 25.

Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2006. ISBN 3-540-36834-5. Cited on page 101.

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, **50**(1):1–102, 1987. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(87)90045-4. Cited on page 58.

Jean-Yves Girard. Geometry of interaction I: Interpretation of System F. In C. Bonotto, editor, *Logic Colloquium '88*, pages 221–260. North Holland, 1989. Cited on pages 63 and 86.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, New York, USA, 1989. ISBN 0-521-37181-3. Reprinted with corrections 1990. Cited on pages x, 59, and 106.

Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26. ACM Press, New York, New York, USA, 1992. ISBN 0-89791-453-8. doi: http://doi.acm.org/10.1145/143165.143172. Cited on pages 63 and 86.

Timothy G. Griffin. A formulae-as-type notion of control. In POPL 1990, pages 47–58. doi: http://doi.acm.org/10.1145/96709.96714. Cited on page 70.

David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, **23**(6):685–746, 2001. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/506315.506316. Cited on pages 94, 96, and 98.

Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College, 2004. ISBN 0-9543006-5-3. Cited on page ix.

Chris Hankin, Rajagopal Nagarajan, and Prahladavaradan Sampath. *The essence of computation: complexity, analysis, transformation*, chapter Flow analysis: games and nets, pages 135–156. Springer-Verlag New York, Inc., New York, New York, USA, 2002. ISBN 3-540-00326-6. Cited on page 49.

Robert Harper. *Programming in Standard ML*. 2005. Published online only: http://www.cs.cmu.edu/~rwh/smlbook/, Working draft of December 6, 2007. Cited on page ix.

Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, New York, USA, 1977. ISBN 0444002162. Cited on page 88.

Nevin Heintze. Set-based analysis of ML programs. In *LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–317. ACM Press, New York, New York, USA, 1994. ISBN 0-89791-643-3. doi: http://doi.acm.org/10.1145/182409.182495. Cited on pages 24, 90, 91, and 99.

Nevin Heintze. Control-flow analysis and type systems. In Mycroft (1995), pages 189–206. Cited on page 24.

Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 261–272. ACM Press, New York, New York, USA, 1997a. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258939. Cited on pages 29, 49, and 92.

Nevin Heintze and David McAllester. On the complexity of set-based analysis. In Berman (1997), pages 150–163. doi: http://doi.acm.org/10.1145/258948.258963. Cited on pages 55, 88, and 92.

Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *LICS '97: Proceedings of the 12th Annual IEEE Symposium*

on Logic in Computer Science, page 342. IEEE Computer Society, Washington, DC, USA, 1997c. ISBN 0-8186-7925-5. Cited on pages 47, 48, and 92.

Fritz Henglein. Type invariant simulation: A lower bound technique for type inference. 1990. Unpublished manuscript. Cited on pages 90 and 102.

Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 448–472. Springer-Verlag, London, UK, 1991. ISBN 3-540-54396-1. Cited on pages 99 and 104.

Fritz Henglein. Simple closure analysis. Technical report, 1992. DIKU Semantics Report D-193. Cited on pages 6, 29, 34, 71, 99, and 104.

Fritz Henglein and Harry G. Mairson. The complexity of type inference for higher-order lambda calculi. In POPL 1991, pages 119–130. doi: http://doi.acm.org/10.1145/99583.99602. Cited on pages 89, 90, 102, 103, 104, 106, and 111.

Jacques Herbrand. *Investigations in proof theory: The properties of true propositions*, pages 525–581. Harvard University Press, 1930. Chapter 5 of Herbrand's Ph.D. dissertation, *Recherches sur la théorie de la deémonstration*. Cited on page 104.

Gerd G. Hillebrand, Paris C. Kanellakis, Harry G. Mairson, and Moshe Y. Vardi. Undecidable boundedness problems for datalog programs. *J. Logic Program.*, **25**(2):163–190, 1995. Cited on pages 93 and 101.

Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61. ACM, New York, New York, USA, 2001. ISBN 1-58113-413-4. doi: http://doi.acm.org/10.1145/379605.379665. Cited on pages 99 and 100.

J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, pages 29–60, 1969. Cited on page 104.

J. Roger Hindley. BCK-combinators and linear λ-terms have types. *Theor. Comput. Sci.*, **64**:97–105, 1989. Cited on pages 89 and 103.

Sachio Hirokawa. Principal type-schemes of BCI-lambda-terms. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 633–650. Springer-Verlag, London, UK, 1991. ISBN 3-540-54415-1. Cited on pages 89 and 103.

Martin Hofmann. *Type Systems for Polynomial-Time Computation*. Ph.D. thesis, TU Darmstadt, 1998. Habilitation Thesis. Cited on page 101.

Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, **183**(1):57–85, 2003. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/S0890-5401(03)00009-9. Cited on page 101.

Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, **19**(1):1–6, 1997. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/239912.239913. Cited on pages 100 and 109.

William A. Howard. The formulae-as-types notion of construction. In Seldin and Hindley (1980), pages 479–490. Cited on page 57.

ICFP'07. *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, Freiburg, Germany, October 1–3*. ACM, New York, New York, USA, 2007. ISBN 978-1-59593-815-2. Cited on pages 131 and 134.

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In POPL 1998, pages 329–341. doi: http://doi.acm.org/10.1145/268946.268973. Cited on pages 3 and 89.

Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407. ACM Press, New York, New York, USA, 1995. ISBN 0-89791-692-1. doi: http://doi.acm.org/10.1145/199448.199536. Cited on pages 48, 100, and 110.

Ralph E. Johnson, Justin O. Graver, and Laurance W. Zurawski. TS: An optimizing compiler for Smalltalk. In *OOPSLA '88: Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 18–26. ACM, New York, New York, USA, 1988. ISBN 0-89791-284-5. doi: http://doi.acm.org/10.1145/62083.62086. Cited on page 94.

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

Neil D. Jones. *Computability and Complexity: From a Programming Perspective*. MIT Press, Cambridge, Massachusetts, USA, 1997. ISBN 0-262-10064-9. Cited on page x.

Neil D. Jones and Nina Bohr. Call-by-value termination in the untyped λ-calculus. *Logical Methods in Computer Science*, **4**(1):1–39, 2008. Cited on page 101.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA, 1993. ISBN 0-13-020249-5. Cited on page 67.

Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. pages 527–636. Oxford University Press, Oxford, UK, 1995. ISBN 0-19-853780-8. Cited on page 17.

Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML-type reconstruction. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991. Cited on page 104.

Assaf J. Kfoury, Harry G. Mairson, Franklyn A. Turbak, and J. B. Wells. Relating typability and expressiveness in finite-rank intersection type systems (extended abstract). In *ICFP '99: Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*, pages 90–101. ACM, New York, New York, USA, 1999. ISBN 1-58113-111-9. doi: http://doi.acm.org/10.1145/317636.317788. Cited on page 105.

L. Kristiansen and K.-H. Niggl. On the computational complexity of imperative programming languages. *Theor. Comput. Sci.*, **318**(1-2):139–161, 2004. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2003.10.016. Cited on page 101.

George Kuan and David MacQueen. Efficient type inference using ranked type variables. In *ML '07: Proceedings of the 2007 Workshop on ML*, pages 3–14. ACM, New York, New York, USA, 2007. ISBN 978-1-59593-676-9. doi: http://doi.acm.org/10.1145/1292535.1292538. Cited on page 73.

Richard E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, **7**(1):18–20, 1975. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/990518.990519. Cited on pages 36 and 42.

Yves Lafont. From proof-nets to interaction nets. In *Proceedings of the Workshop on Advances in Linear Logic*, pages 225–247. Cambridge University Press, New York, New York, USA, 1995. ISBN 0-521-55961-8. Cited on page 70.

William Landi. *Interprocedural Aliasing in the Presence of Pointers*. Ph.D. thesis, Rutgers University, 1992a. Cited on pages 98, 100, and 109.

William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, **1**(4):323–337, 1992b. ISSN 1057-4514. doi: http://doi.acm.org/10.1145/161494.161501. Cited on pages 100 and 109.

William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In POPL 1991, pages 93–103. doi: http://doi.acm.org/10.1145/99583.99599. Cited on pages 100 and 109.

William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 235–248. ACM, New York, New York, USA, 1992. ISBN 0-89791-475-9. doi: http://doi.acm.org/10.1145/143095.143137. Cited on page 21.

William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In McKinley (2004), pages 473–489. doi: http://doi.acm.org/10.1145/989393.989440. Cited on page 21.

Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, **6**(4):308–320, 1964. Cited on pages ix and 9.

Julia L. Lawall and Harry G. Mairson. Sharing continuations: Proofnets for languages with explicit control. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 245–259. Springer-Verlag, London, UK, 2000. ISBN 3-540-67262-1. Cited on page 68.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

pages 81–92. ACM, New York, New York, USA, 2001. ISBN 1-58113-336-7. doi: http://doi.acm.org/10.1145/360204.360210. Cited on page 101.

Daniel Leivant. Stratified functional programs and computational complexity. In POPL 1993, pages 325–333. doi: http://doi.acm.org/10.1145/158511.158659. Cited on page 101.

Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul*. Ph.D. thesis, University of Paris VII, 1978. Thése d'Etat. Cited on pages 70 and 86.

Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In Seldin and Hindley (1980), pages 159–191. Cited on page 86.

Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In POPL 1990, pages 382–401. doi: http://doi.acm.org/10.1145/96709.96748. Cited on pages 73, 84, 90, 93, 102, and 103.

Harry G. Mairson. Quantifier elimination and parametric polymorphism in programming languages. *J. Functional Programming*, **2**:213–226, 1992a. Cited on page 104.

Harry G. Mairson. A simple proof of a theorem of Statman. *Theor. Comput. Sci.*, **103**(2):387–394, 1992b. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(92)90020-G. Cited on page 65.

Harry G. Mairson. From Hilbert spaces to Dilbert spaces: Context semantics made simple. In *FST TCS '02: Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science*, pages 2–17. Springer-Verlag, London, UK, 2002. ISBN 3-540-00225-1. Cited on page 63.

Harry G. Mairson. Linear lambda calculus and PTIME-completeness. *J. Functional Program.*, **14**(6):623–633, 2004. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796804005131. Cited on pages 44, 59, 62, 89, 102, 103, and 112.

Harry G. Mairson. Axiom-sensitive normalization bounds for multiplicative linear logic. 2006a. Unpublished manuscript. Cited on page 64.

Harry G. Mairson. MLL normalization and transitive closure: circuits, complexity, and Euler tours. In *GEOCAL (Geometry of Calculation): Implicit Computational Complexity. Institut de Mathematique de Luminy, Marseille.* 2006b. Cited on pages 44, 64, and 66.

Harry G. Mairson and Kazushige Terui. On the computational complexity of cut-elimination in linear logic. In Carlo Blundo and Cosimo Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003, Bertinoro, Italy, October 13-15, 2003, Proceedings*, volume 2841 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2003. ISBN 3-540-20216-1. Cited on page 70.

John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill Higher Education, 1997. ISBN 0070408459. Cited on page 92.

David McAllester. On the complexity analysis of static analyses. *J. ACM*, **49**(4):512–537, 2002. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/581771.581774. Cited on pages 92 and 101.

Kathryn S. McKinley, editor. *SIGPLAN Not., Special Issue: 20 Years of PLDI (1979 - 1999): A Selection*, volume 39. ACM, New York, New York, USA, 2004. Cited on pages 124 and 132.

David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theor. Comput. Sci.*, **248**(1-2):29–98, 2000. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(00)00049-9. Cited on pages 88, 90, and 92.

Jan Midtgaard. Control-flow analysis of functional programs. Technical Report BRICS RS-07-18, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 2007. To appear in revised form in ACM Computing Surveys. Cited on pages viii, 3, 34, and 99.

Jan Midtgaard and Thomas Jensen. A calculational approach to control-flow analysis by abstract interpretation. In Alpuente and Vidal (2008), pages 347–362. Cited on pages 24 and 55.

Jan Midtgaard and Thomas Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP '09: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. 2009. To appear. Extended version available as INRIA research report RR-6681. Cited on pages 24 and 55.

Jan Midtgaard and David Van Horn. Subcubic control-flow analysis algorithms. Technical Report 125, Roskilde University, Denmark, 2009. To appear in the Symposium in Honor of Mitchell Wand. Cited on pages 92 and 111.

Matthew Might. *Environment analysis of higher-order languages*. Ph.D. thesis, Georgia Institute of Technology, Atlanta, Georgia, 2007. Adviser-Olin G. Shivers. Cited on pages 24 and 100.

Matthew Might and Olin Shivers. Environment analysis via ΔCFA. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 127–140. ACM, New York, New York, USA, 2006a. ISBN 1-59593-027-2. doi: http://doi.acm.org/10.1145/1111037.1111049. Cited on pages 24 and 100.

Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*, pages 13–25. Portland, Oregon, 2006b. Cited on pages 3, 28, 84, 86, 89, and 111.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, **14**(1):1–41, 2005. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1044834.1044835. Cited on pages 96 and 100.

John C. Mitchell. Type systems for programming languages. In van Leeuwen (1990), pages 365–458. Cited on page 104.

Peter Møller Neergaard. *Complexity Aspects of Programming Language Design: From Logspace to Elementary Time via Proofnets and Intersection Types*. Ph.D. thesis, Brandeis University, Waltham, Massachusetts, USA, 2004. Cited on pages 70 and 105.

Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, pages 138–149. ACM Press, New York, New York, USA, 2004. ISBN 1-58113-905-5. doi: http://doi.acm.org/10.1145/1016850.1016871. Cited on pages 90, 102, and 105.

Christian Mossin. Exact flow analysis. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 250–264. Springer-Verlag, London, UK, 1997a. ISBN 3-540-63468-1. Cited on pages 50 and 85.

Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. Ph.D. thesis, DIKU, University of Copenhagen, 1997b. Cited on pages 24 and 70.

Christian Mossin. Higher-order value flow graphs. *Nordic J. of Computing*, **5**(3):214–234, 1998. ISSN 1236-6064. Cited on pages 29, 49, and 110.

Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981. Cited on pages 2 and 26.

Robert Muth and Saumya Debray. On the complexity of flow-sensitive dataflow analyses. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80. ACM, New York, New York, USA, 2000. ISBN 1-58113-125-9. doi: http://doi.acm.org/10.1145/325694.325704. Cited on pages 100 and 109.

Alan Mycroft, editor. *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 25-27, 1995, Proceedings*, volume 983 of *Lecture Notes in Computer Science*. Springer, 1995. ISBN 3-540-60360-3. Cited on pages 118 and 120.

Eugene M. Myers. A precise inter-procedural data flow algorithm. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–230. ACM, New York, New York, USA, 1981. ISBN 0-89791-029-X. doi: http://doi.acm.org/10.1145/567532.567556. Cited on page 100.

Radford Neal. The computational complexity of taxonomic inference. 1989. Unpublished manuscript. ftp://ftp.cs.utoronto.ca/pub/radford/taxc.ps. Cited on page 92.

Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In POPL 1997, pages 332–345. doi: http://doi.acm.org/10.1145/263699.263745. Cited on pages 24, 100, and 110.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, New Jersey, USA, 1999. ISBN 3540654100. Cited on pages x, 2, 8, 15, 17, 19, 20, 23, 32, 33, 72, and 74.

OOPSLA 1996. *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, New York, USA, 1996. ISBN 0-89791-788-X. Cited on pages 114 and 118.

Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 329–349. Springer-Verlag, London, UK, 1992. ISBN 3-540-55668-0. Cited on page 96.

Jens Palsberg. Closure analysis in constraint form. *ACM Trans. Program. Lang. Syst.*, **17**(1):47–62, 1995. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/200994.201001. Cited on page 99.

Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, **17**(4):576–599, 1995. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/210184.210187. Cited on page 24.

Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Program.*, **11**(3):263–317, 2001. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S095679680100394X. Cited on pages 24 and 70.

Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91: Conference proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161. ACM, New York, New York, USA, 1991. ISBN 0-201-55417-8. doi: http://doi.acm.org/10.1145/117954.117965. Cited on pages 94 and 95.

Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Inf. Comput.*, **118**(1):128–141, 1995. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1995.1058. Cited on page 55.

Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, USA, 1994. ISBN 0201530821. Cited on pages ix, 24, 25, and 50.

Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, vol-

ume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992. ISBN 3-540-55727-X. Cited on page 68.

Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996. ISBN 052156543X. Cited on page ix.

Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundam. Inf.*, **19**(1-2):185–199, 1993. ISSN 0169-2968. Cited on page 106.

POPL 1990. *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, New York, USA, 1990. ISBN 0-89791-343-4. Cited on pages 119 and 125.

POPL 1991. *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, New York, USA, 1991. ISBN 0-89791-419-8. Cited on pages 121 and 124.

POPL 1993. *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, New York, USA, 1993. ISBN 0-89791-560-7. Cited on pages 116 and 125.

POPL 1997. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, New York, USA, 1997. ISBN 0-89791-853-3. Cited on pages 114 and 128.

POPL 1998. *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, New York, USA, 1998. ISBN 0-89791-979-3. Cited on pages 114, 117, and 122.

POPL 2008. *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, New York, USA, 2008. ISBN 978-1-59593-689-9. Cited on pages 116 and 133.

G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, **16**(5):1467–1471, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/186025.186041. Cited on page 100.

Laurent Regnier. *Lambda calcul et réseaux*. Ph.D. thesis, University of Paris VII, 1992. Cited on page 70.

Thomas W. Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, **33**(8):739–757, 1996. Cited on pages 9, 88, and 91.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740. ACM, New York, New York, USA, 1972. doi: http://doi.acm.org/10.1145/800194.805852. Republished in (Reynolds 1998). Cited on pages ix, 8, and 131.

John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, London, UK, 1974. ISBN 3-540-06859-7. Cited on page 106.

John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, **11**(4):363–397, 1998. Originally published in (Reynolds 1972). Cited on pages ix and 131.

Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, **74**:358–366, 1953. Cited on page 2.

N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. On the complexity of partially-flow-sensitive alias analysis. *ACM Trans. Program. Lang. Syst.*, **30**(3):1–28, 2008. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1353445.1353447. Cited on pages 96 and 100.

J. Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, **12**(1):23–41, 1965. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321250.321253. Cited on pages 104 and 106.

Wojciech Rytter. Fast recognition of pushdown automaton and context-free languages. *Inf. Control*, **67**(1-3):12–22, 1985. ISSN 0019-9958. doi: http://dx.doi.org/10.1016/S0019-9958(85)80024-3. Cited on page 92.

Jonathan P. Seldin and J. Roger Hindley, editors. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, London, 1980. Cited on pages 122 and 125.

Damien Sereni. Termination analysis and call graph construction for higher-order functional programs. In ICFP'07, pages 71–84. doi: http://doi.acm.org/10.1145/1291151.1291165. Cited on pages 100, 101, and 102.

Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2005. ISBN 3-540-29735-9. Cited on page 101.

Peter Sestoft. *Replacing function parameters by global variables*. Master's thesis, DIKU, University of Copenhagen, Denmark, 1988. Master's thesis no. 254. Cited on pages 3 and 24.

Peter Sestoft. Replacing function parameters by global variables. In *FPCA '89: Proceedings of the fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 39–53. ACM Press, New York, New York, USA, 1989. ISBN 0-89791-328-0. doi: http://doi.acm.org/10.1145/99370.99374. Cited on pages 3 and 24.

Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, **27**, 1948. Cited on page 21.

Olin Shivers. Control flow analysis in Scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174. ACM, New York, New York, USA, 1988. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54007. Cited on pages 3, 29, 71, and 72.

Olin Shivers. *Control-flow analysis of higher-order languages*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991. Cited on pages 3, 24, 72, 94, and 100.

Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. In McKinley (2004), pages 257–269. doi: http://doi.acm.org/10.1145/989393.989421. Cited on pages 47, 68, 72, and 86.

Dorai Sitaram. *Teach Yourself Scheme in Fixnum Days*. 2004. Published online: http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html. Cited on page ix.

Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Inc., New York, New York, USA, 2006. ISBN 0444520775. Cited on pages ix, 57, and 59.

Fausto Spoto and Thomas Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, **25**(5):578–630, 2003. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/937563.937565. Cited on page 95.

Richard Statman. The typed λ-calculus is not elementary recursive. *Theor. Comput. Sci.*, **9**:73–81, 1979. Cited on pages 50, 65, and 111.

Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.*, **19**(1):48–86, 1997. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/239912.239915. Cited on page 89.

Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41. ACM, New York, New York, USA, 1996. ISBN 0-89791-769-3. doi: http://doi.acm.org/10.1145/237721.237727. Cited on pages 99 and 104.

Yan Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 224–243. Springer-Verlag, London, UK, 1994. ISBN 3-540-57887-0. Cited on page 24.

Kazushige Terui. On the complexity of cut-elimination in linear logic. 2002. Invited talk at LL2002 (LICS2002 affiliated workshop), Copenhagen. Cited on page 64.

Jerzy Tiuryn. Type inference problems: a survey. In *MFCS '90: Proceedings on Mathematical Foundations of Computer Science 1990*, pages 105–120. Springer-Verlag New York, Inc., New York, New York, USA, 1990. ISBN 0-387-52953-5. Cited on page 104.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In POPL 2008, pages 395–406. doi: http://doi.acm.org/10.1145/1328438.1328486. Cited on page 106.

Jeffrey D. Ullman and Allen van Gelder. Parallel complexity of logical query programs. In *SFCS '86: Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS 1986)*, pages 438–454. IEEE Computer Society, Washington, DC, USA, 1986. ISBN 0-8186-0740-8. doi: http://dx.doi.org/10.1109/SFCS.1986.40. Cited on page 90.

Paweł Urzyczyn. Type reconstruction in F_ω. *Mathematical. Structures in Comp. Sci.*, **7**(4):329–358, 1997. ISSN 0960-1295. doi: http://dx.doi.org/10.1017/S0960129597002302. Cited on page 106.

Steffen van Bakel. Complete restrictions of the intersection type discipline. *Theor. Comput. Sci.*, **102**(1):135–163, 1992. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(92)90297-S. Cited on page 105.

David Van Horn and Harry G. Mairson. Relating complexity and precision in control flow analysis. In ICFP'07, pages 85–96. doi: http://doi.acm.org/10.1145/1291151.1291166. Cited on page xi.

David Van Horn and Harry G. Mairson. Deciding kCFA is complete for EXPTIME. In *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming, Victoria, BC, Canada, September 22–24*, pages 275–282. 2008a. Cited on page xi.

David Van Horn and Harry G. Mairson. Flow analysis, linearity, and PTIME. In Alpuente and Vidal (2008), pages 255–269. Cited on pages xi and 92.

J. van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*. MIT Press, Cambridge, Massachusetts, USA, 1990. ISBN 0-444-88074-7. Cited on pages 114 and 127.

Virginia Vassilevska. Efficient algorithms for clique problems. *Inf. Process. Lett.*, **109**(4):254–257, 2009. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/j.ipl.2008.10.014. Cited on page 92.

Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-time analysis of object-oriented programs. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 236–250. Springer-Verlag, London, UK, 1992. ISBN 3-540-55984-1. Cited on page 96.

Mitchell Wand. A simple algorithm and proof for type inference. *Fundam. Inform.*, **10**:115–122, 1987. Cited on pages 104 and 106.

J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, **98**:111–156, 1999. Cited on page 106.

J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Program.*, **12**(3):183–227, 2002. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796801004245. Cited on page 70.

Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 1–12. ACM, New York, New York, USA, 1995. ISBN 0-89791-697-2. doi: http://doi.acm.org/10.1145/207110.207111. Cited on page 100.

Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst.*, **20**(1):166–207, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/271510.271523. Cited on pages 3, 28, 84, and 100.

Uri Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. *Algorithmica*, **46**(2):181–192, 2006. ISSN 0178-4617. doi: http://dx.doi.org/10.1007/s00453-005-1199-1. Cited on page 92.

Colophon

This dissertation was produced using Free Software on a digital computer. The document was composed in GNU Emacs and typeset using LaTeX and the *Brandeis dissertation* class by Peter Møller Neergaard. The Times font family is used for text and the Computer Modern font family, designed by Donald E. Knuth, is used for mathematics. Some figures were typeset using the METAPOST language by John D. Hobby and sequent calculus proofs were typeset using Didier Rémy's *Math Paragraph for Typesetting Inference Rules*.

Machine wash cold in commercial size, front loading machine, gentle cycle, mild powder detergent. Rinse thoroughly. Tumble dry low in large capacity dryer. Do not iron. Do not dry clean. Do not bleach. Secure all velcro closures.