Relating Complexity and Precision in Control Flow Analysis

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

Abstract

We analyze the computational complexity of kCFA, a hierarchy of control flow analyses that determine which functions may be applied at a given call-site. This hierarchy specifies related decision problems, quite apart from any algorithms that may implement their solutions. We identify a simple decision problem answered by this analysis and prove that in the 0CFA case, the problem is complete for polynomial time. The proof is based on a nonstandard, symmetric implementation of Boolean logic within multiplicative linear logic (MLL). We also identify a simpler version of 0CFA related to η-expansion, and prove that it is complete for LOGSPACE, using arguments based on computing paths and permutations.

For any fixed $k > 0$, it is known that kCFA (and the analogous decision problem) can be computed in time exponential in the program size. For $k = 1$, we show that the decision problem is NP-hard, and sketch why this remains true for larger fixed values of k. The proof technique depends on using the *approximation* of CFA as an essentially nondeterministic computing mechanism, as distinct from the exactness of normalization. When $k = n$, so that the "depth" of the control flow analysis grows linearly in the program length, we show that the decision problem is complete for EXPTIME.

In addition, we sketch how the analysis presented here may be extended naturally to languages with control operators. All of the insights presented give clear examples of how straightforward observations about linearity, and linear logic, may in turn be used to give a greater understanding of functional programming and program analysis.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Computability theory, Computational logic, Lambda calculus and related systems; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms Languages, Theory

Keywords Control flow analysis, static analysis, linear logic, eta expansion, continuation, geometry of interaction, proofnet, normalization, complexity

1. Introduction

We investigate the *precision* of static, compile-time analysis, and the necessary analytic *tradeoff* with the computational *resources* that go into the analysis.

Control flow analysis provides a fundamental and ubiquitous static analysis of higher-order programs. Heintze and McAllester (1997) point out that "in fact, some form of CFA is used in most forms of analyses for higher-order languages." Control flow analysis answers the following basic questions (Palsberg 1995):

1. For every application, which abstractions can be applied?
2. For every abstraction, to which arguments can it be applied?

These questions specify well-defined, significant *decision problems*, quite apart from any algorithm proposed to solve them. What is the inherent computational difficulty of solving these problems? What do they have to do with normalization (evaluation), as opposed to approximation? Where is the fulcrum located between exact computation and approximation to it, and how does that balance make this compile-time program analysis tractable?

To answer either of the enumerated questions above, we must be able to approximate the set of values to which any given subexpression may evaluate. This approximation can be formulated as a *flows to* relation between fragments of program text—the set of values a subexpression may evaluate to, then, is the set of appropriately defined (program text) values that flow into that subexpression. This analysis possibly includes "false positives" (values that *may* flow to a call site, but in fact do not), and thus a *flows to* analysis might relate every program fragment. We focus attention on an acceptably "least" analysis, that is, a solution to the constraints which has a minimum of false positives. More precision means fewer false positives.

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's left in a reasonable amount of time. A good example of what is lost during 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, and a substitution of a function is made for f, 0CFA will blur which copy of the function is applied to which argument.

Further refining the relatively coarse approximation of the above control flow analysis, a hierarchy of analyses was developed by Olin Shivers in his Ph.D. thesis (Shivers 1991), the so-called kCFA analyses, of which the above analysis marks the base—0CFA. A 1CFA, for example, uses a *contour* to distinguish a more specific and dynamic calling context in the analysis of each program point; kCFA, then, distinguishes between k levels of such contexts. Moving up the hierarchy increases the precision of this analysis, by constructing more elaborate contours. However, this increased precision is accompanied by an empirically observed

increase in cost. As Shivers observed in his "Best of PLDI" retrospective on the kCFA work:

> 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.

Technical contributions: We identify a simple decision problem answered by control flow analysis and prove that in the 0CFA case, the problem is complete for polynomial time. The proof is based on a nonstandard, symmetric implementation of Boolean logic within multiplicative linear logic (MLL). We also identify a simpler version of 0CFA related to η-expansion, and prove that it is complete for logarithmic space, using arguments based on computing paths and permutations.

Moreover, we show that the decision problem for 1CFA is NP-hard. Given that there is a naive exponential algorithm, we conjecture that it is in fact NP-complete, where prudent guessing in the computation of the naive algorithm can answer specific questions about flows. Moreover, the proof likely generalizes to an NP-hardness proof for all fixed $k > 1$; we will report on this in the final version of the paper.

Like all good proofs, this NP-hardness result has simple intuitions. 1CFA depends on *approximation*, where multiple values (here, closures) flow to the same program point, including false positives. The bottleneck of the naive algorithm is its handling of closures with many free variables. 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. Approximation allows us to bind each x_i, independently, to either closed λ-terms for "true" or "false". In turn, application to an n-ary Boolean function, as analyzed in CFA, must then evaluate the function on each of the possible environments. Asking whether "true" can flow out of the call site then becomes a way of asking if the Boolean function is satisfiable.

No such "pseudo parallelism" would be possible in an exact normalization—it is the existence of *approximation* that mashes these distinct closures together. Similar NP-hardness of kCFA for $k > 1$ results by suitably "padding" the construction for 1CFA so as to render the added precision useless. We remark also that the NP-hardness construction cannot be coded in 0CFA because in that case, there are no environments and hence no closures.

Despite being the fundamental analysis of higher-order programs, despite being the subject of investigation for over twenty-five years (Jones 1981), 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 control flow analysis and the kCFA hierarchy, the essence of how this hierarchy is computed, and where the sources of approximation occur that make the analysis work. This paper is intended to repair such lacunae in our understanding of program analysis.

2. Preliminaries

In this section, we describe the programming language that is the subject of our control flow analysis and provide the necessary mechanics of the graphical representation of programs employed in our algorithms. The language on which the control flow analysis will be performed is the untyped λ-calculus extended by a labelling scheme serving to index subexpressions of a program. Following Nielson et al. (1999), we define the following syntactic categories:

e	$\in$	$\mathbf{Exp}$	expressions (or labeled terms)
t	$\in$	$\mathbf{Term}$	terms (or unlabeled expressions)

A countably infinite set of labels ($\mathbf{Lab}$) is assumed and for simplicity we suppose the set of variable names are included in $\mathbf{Lab}$. The abstract syntax of the language is given by the following grammar:[1]

$e ::= t^l$	expressions (or labeled terms)
$t ::= x \mid (e\ e) \mid (\lambda x.e)$	terms (or unlabeled expressions)

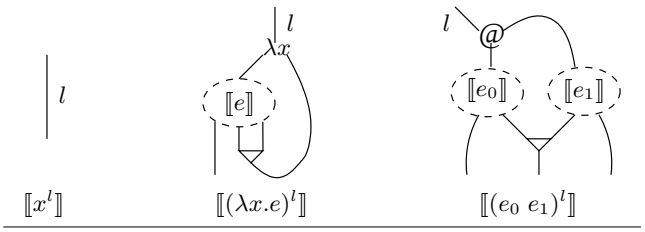
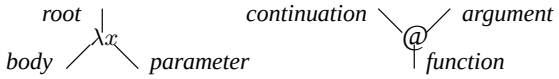


Figure 1. Graphical coding of expressions.

Expressions are represented graphically as folllows: a graph consists of ternary apply (@), abstraction (λ), and sharing nodes ($\triangledown$); unary weakening nodes ($\odot$); and wires between these nodes. Each node has a distinguished *principal port*; ternary node's other *auxilary ports* are distinguished as the left and right (or black and white) ports. We call the principal port of an @-node the *function*, the left port the *continuation*, and the right port the *argument*. Likewise, the principal port of a λ-node is the *root*, the left port is the *body*, and the right port is the *parameter*. That is:



Wires are labeled to correspond with the labels on terms, i.e. the continuation wire for the graph of $(e\ e)^l$ will be labeled l and the root wire for $(\lambda x.e)^l$ will be labeled l. Parameter wires will be labeled with the bound variable's name.

Figure 1 gives the graphical coding of expressions. In the $\lambda x.e$ case, the figure is drawn as though x appears twice in e, and thus the sharing node is used to duplicate the wire, one going to each occurrence. The wire between the λ-node and the sharing node is implicitly labeled x. The two wires attached to the auxiliary ports of the sharing node will be labeled with the distinct labels of each occurrence of x. If x occurred more than twice, more sharing nodes would be attached to fan out the binding the appropriate amount. If the bound variable occurred exactly once, the wire would connect the λ node directly to the variable occurrence; the label x and the label used at the occurrence of x would both refer to the same wire. If the bound variable x did not occur in e, the wire from the λ-node would attach to a weakening node. The "dangling wire" from the graph of the body of the function denotes a variable free in e. In the case of $(e_0\ e_1)$, the figure is drawn with e_0 and e_1 both having a free variable in common, i.e. there is an x in $\mathbf{fv}(e_0) \cap \mathbf{fv}(e_1)$, and both e_0 and e_1 have another free variable not occuring in the other.

3. 0CFA

Control flow analysis seeks to answer questions such as "what functions can be applied at a give application position?" Because both functions and applications may be copied, such questions become ambiguous. When one points to an application in the program text, running the program may duplicate this point several times, so which of the copies are we really asking about? Likewise, if a lambda term is said to flow into some application, which copy of the term is going to be applied? The answer 0CFA gives is: all of them—all copies of a term are identified. Later, we see how con-

[1] Unlike Nielson et al. (1999), constants, binary operators, and recursive- and let-binding syntax are omitted. These language features add nothing interesting to the computational complexity of the analysis.

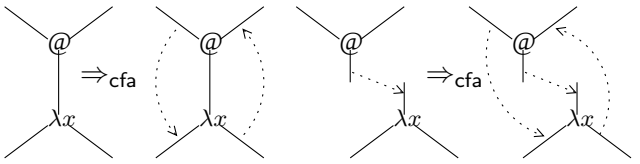


Figure 2. CFA virtual wire propagation rules.

textual information can be used to distinguish the copies and give more precise answers to these questions.

We follow Nielson et al. (1999) and say the result of 0CFA is an *abstract cache* $\widehat{\mathsf{C}}$ associating abstract values with each labeled program point. More precisely:

$$\begin{array}{rcccll} \widehat{v} & \in & \widehat{\mathbf{Val}} & = & \mathcal{P}(\mathbf{Term}) & \text{abstract values} \\ \widehat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}} & = & \mathbf{Lab} \to \widehat{\mathbf{Val}} & \text{abstract caches} \end{array}$$

An *abstract cache* maps a program label to an *abstract value* $\widehat{v}$, a set of lambda expressions, which represents the set of (textual) values that may *flow into* that label's subexpression during evaluation. Similarly, the *abstract enviroment* maps a variable to an abstract value, which represents the set of (textual) values that may be bound to that variable during evaluation.

An acceptable control flow analysis for an expression e is written $\widehat{\mathsf{C}} \models e$. Recalling again from Nielson et al. (1999), the acceptability relation is given by the greatest fixed point of the functional defined according to the following clauses:

$$\begin{array}{rcl} \widehat{\mathsf{C}} & \models & x^l \text{ iff } \widehat{\mathsf{C}}(x) \subseteq \widehat{\mathsf{C}}(l) \\ \widehat{\mathsf{C}} & \models & (\lambda x.e)^l \text{ iff } \lambda x.e \in \widehat{\mathsf{C}}(l) \\ \widehat{\mathsf{C}} & \models & (t_1^{l_1}\ t_2^{l_2})^l \text{ iff } \widehat{\mathsf{C}} \models t_1^{l_1} \wedge \widehat{\mathsf{C}} \models t_2^{l_2} \wedge \forall \lambda x.t_0^{l_0} \in \widehat{\mathsf{C}}(l_1) : \\ & & \quad \widehat{\mathsf{C}} \models t_0^{l_0} \wedge \widehat{\mathsf{C}}(l_2) \subseteq \widehat{\mathsf{C}}(x) \wedge \widehat{\mathsf{C}}(l_0) \subseteq \widehat{\mathsf{C}}(l) \end{array}$$

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 2 defines the virtual path propagation rules. 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 $l \rightsquigarrow l'$ in a CFA-graph iff: 1) $l \equiv l'$, 2) there is a virtual wire from l to l', 3) l connects to an auxiliary port and l' connects to the principal port of a sharing node, or 4) $l \rightsquigarrow l''$ and $l'' \rightsquigarrow l'$.

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. An apply node is reachable if it is on the spine of the program, i.e., if $e = (\ldots((e_0 e_1)^{l_1} e_2)^{l_2} \ldots e_n)^{l_n}$ then the apply nodes with continuation wires labeled $l_1, \ldots, l_n$ are reachable, or it is on the spine of an expression with a virtual path from a reachable apply node.

The graph-based analysis can now be performed in the following way: construct the CFA graph according to the rules in Figure 2, then define $\widehat{\mathsf{C}}(l)$ as $\{(\lambda x.e)^{l'} \mid l \rightsquigarrow l'\}$. It is easy to see that the algorithm constructs answers that satisfy the acceptability relation specifying the analysis. Moreover, this algorithm constructs least solutions according to the partial order $\widehat{\mathsf{C}} \sqsubseteq_e \widehat{\mathsf{C}}'$ iff $\forall l \in \mathbf{Lab}_e : \widehat{\mathsf{C}}(l) \subseteq \widehat{\mathsf{C}}'(l)$, where $\mathbf{Lab}_e$ denotes the set of labels restricted to those occurring in e, the program of interest.

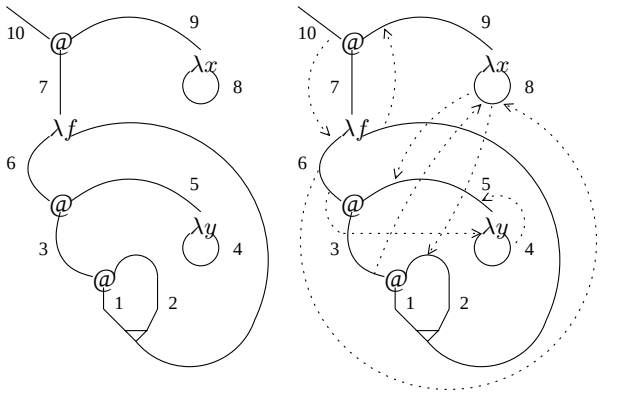


Figure 3. Graph coding and CFA graph.

Lemma 1. *$\widehat{\mathsf{C}}' \models e$ implies $\widehat{\mathsf{C}} \sqsubseteq_e \widehat{\mathsf{C}}'$ for $\widehat{\mathsf{C}}$ 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 3 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:

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

We now describe a natural decision problem answered by this control flow analysis. After describing the problem here, subsequent sections will consider variants of it for kCFA.

3.1 The 0CFA decision problem

A decision problem—a question that can be answered with a yes or a no—makes the analysis insensitive to the output size of any control flow analysis. Typically, this analysis computes the answer to questions like "what functions can be applied at a particular call site?" or "what arguments can a particular function be applied to?", so a natural decision problem based on these questions are "is this particular function applied at this particular call site?" or "does this function get applied to this argument?", where the function is denoted by some lambda expression or application in the program text. These questions provide ways of answering the more general "to what values can a subexpression evaluate?"

Control Flow Problem (0CFA): Given expressions e, $\lambda x.e_0$, and label l, is $\lambda x.e_0 \in \widehat{\mathsf{C}}(l)$ in a least analysis of e?

The graphical analogue of this problem point to a λ-node and an application node, and ask if there a *virtual path* (describing a β-redex from reductions to take place) from the function port of the

apply to the root port of the λ? Or likewise, point to two λ-nodes: is there a virtual path from the variable port of one to the root of the other? The graph-reduction characterization has the virtue of being free of a lot of notational clutter (like variable renaming).

We now prove that this decision problem is complete for polynomial time. A graph-based argument for containment in PTIME is straightforward:

Theorem 1. *0CFA is contained in* PTIME.

Proof. 0CFA computes a binary relation over a fixed structure (the graph description of a program). 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. □

We now turn to the more interesting PTIME-hardness property of the problem. Recall that a property ϕ is PTIME-hard if, given any PTIME Turing Machine M and input x, $\langle M, x\rangle$ can be compiled using $O(\ln |x|)$ space into a problem instance I, where I has the property ϕ iff M accepts x. The canonical PTIME-complete decision problem (Ladner 1975) is the following:

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?

As has been observed by Henglein and Mairson (1991); Mairson (2004); Neergaard and Mairson (2004), *linearity* is the key ingredient in understanding the lower-bound complexity of analyses. In looking at the control flow analysis algorithm of section 3, we may ask *what is the source of approximation?* The answer is that every copy of a function is identified, and every copy of an application site is identified. In other words, a program with copying engenders an approximate analysis. More concretely, notice that an application site has multiple virtual paths to a value *only* through virtual paths that pass through copying nodes. If there are no sharing nodes—as in a linear term—it is simple to see that there is only a single virtual path. By the conservativity of the analysis, it follows immediately that if there is a single path from an apply node to a lambda node, then these nodes will fuse during reduction. Thus, *linearity subverts approximation* and renders control flow analysis synonymous with normalization.

As a calculational tool, we use a linear fragment of Standard ML to illustrate the construction.[2] The Boolean values `True` and `False` are built out of the constants `TT` and `FF`:

```
- fun TT (x:'a,y:'a)= (x,y);
val TT = fn : 'a * 'a -> 'a * 'a
- fun FF (x:'a,y:'a)= (y,x);
val FF = fn : 'a * 'a -> 'a * 'a

- val True= (TT: ('a * 'a -> 'a * 'a),
             FF: ('a * 'a -> 'a * 'a));
val True = (fn,fn) : ('a * 'a -> 'a * 'a)
                   * ('a * 'a -> 'a * 'a)
- val False= (FF: ('a * 'a -> 'a * 'a),
              TT: ('a * 'a -> 'a * 'a));
val False = (fn,fn) : ('a * 'a -> 'a * 'a)
                    * ('a * 'a -> 'a * 'a)
```

This little hack will print out what Boolean we are talking about:

```
- fun Show (u,v)=
  (let val (x,y)= u(true,false) in x end,
   let val (x,y)= v(true,false) in x end);
val Show = fn : (bool * bool -> 'a * 'b)
              * (bool * bool -> 'c * 'd) -> 'a * 'c
- Show True;
val it = (true,false) : bool * bool
- Show False;
val it = (false,true) : bool * bool
```

The way we compute `And` is to use the famous 1930s-era coding of conjunction (hacked by Alonzo Church?) in the first component, and the disjunction in the second component. That way, we are guaranteed that the junk in `v` and `v'` is symmetric: one is `TT`, and the other is `FF`. Then function composition can be used to erase the symmetric garbage—the slogan is, "symmetric garbage is self-annihilating."

```
- fun And (p,p') (q,q')=
  let val ((u,v),(u',v')) = (p (q,FF), p' (TT,q'))
  in (u,Compose (Compose (u',v),Compose (v',FF)))
  end;
val And = fn
  : ('a * ('b * 'b -> 'b * 'b) -> 'c * ('d -> 'e))
    * (('f * 'f -> 'f * 'f) * 'g
       -> ('e -> 'h) * ('i * 'i -> 'd))
    -> 'a * 'g -> 'c * ('i * 'i -> 'h)

- Show (And True False);
val it = (false,true) : bool * bool
```

Notice that since `p'` is the complement of `p`, and `q'` that of `q`, we know `u'` is the complement of `u`. Composing `v`, `v'` and `FF` is always the *identity function* `TT`, which can then be composed with `u'` without changing the value of `u'`.

The construction of the `Or` term is symmetric to the `And` term, the `Not` term is just an inversion[3]—all this an obvious consequence of deMorgan duality and the construction here of the Booleans. The `Copy` gate uses the fact that either `p` or `p'` will invert its argument, so that either `((TT,FF),(TT,FF))` or `((FF,TT),(FF,TT))` is returned:

```
- fun Copy (p,p')= (p (TT,FF), p' (FF,TT));
val Copy = fn
  : (('a * 'a -> 'a * 'a) *
     ('b * 'b -> 'b * 'b) -> 'c)
  * (('d * 'd -> 'd * 'd) *
     ('e * 'e -> 'e * 'e) -> 'f)
  -> 'c * 'f

- let val (p,q)= Copy True in (Show p, Show q) end;
val it = ((true,false),(true,false)) : (bool * bool)
                                     * (bool * bool)
```

By writing logic gates in continuation-passing style, for example:

```
- fun Andgate p q k= k (And p q)
```

we can then write circuits that look like straight-line code:

```
- fun Circuit e1 e2 e3 e4 e5 e6=
  (Andgate e2 e3 (fn e7=>
  (Andgate e4 e5 (fn e8=>
  (Andgate e7 e8 (fn f=>
  (Copygate f (fn (e9,e10)=>
  (Orgate e1 e9 (fn e11=>
  (Orgate e10 e6 (fn e12=>
  (Orgate e11 e12 (fn Output=> Output))))))))))));
val Circuit = fn : < big type... >
```

The above code says: compute the `and` of `e2` and `e3`, putting the result in register `e7`,..., make two copies of register `f`, putting the values in registers `e9` and `e10`,..., compute the `or` of `e11` and `e12`, putting the result in the `output` register.

[2] Use of `let` binding is only for linear unpairing, not polymorphism.

[3] Both are omitted for space reasons.

```
- Circuit True False False False False True;
val it = (fn,fn) : ('a * 'a -> 'a * 'a)
                 * ('a * 'a -> 'a * 'a)
- Show (Circuit True False False False False True);
val it = (true,false) : bool * bool

- let val (u,u')=
    Circuit True False False False False True
  in
  let val ((x,y),(x',y'))= (u (f,g), u' (f',g')) in
      ((x a, y b),(x' a', y' b')) end end;
```

Now for the decision problem: the pair `(x,y)` is either the pair `(f,g)` or `(g,f)`, depending on the Boolean circuit computing output `u`. Thus, is `f` applied to `a` and `g` to `b`, or the other way around? Because the computation is entirely linear, the set of possible binders to `x` is—this must be emphasized—*not* {`f`, `g`}: it is precisely {`f`}, or {`g`}. The 0CFA "approximation" is in fact an exact normalization.

Theorem 2. *0CFA is complete for* PTIME.

4. kCFA

Increasing the precision of the coarse approximation given by the above control flow analysis, a hierarchy of more and more refined analyses was developed. These analyses are *context-sensitive* or *polyvariant* because they distinguish functions applied in distinct call sites to increase the precision of the analysis. A 1CFA analysis, for example, uses a *contour* to distinguish *one* level of dynamic calling context in the analysis of each program point.

In the example given in section 3, for instance, the analysis is able to distinguish each occurrence of f as distinct in the analysis. The increased precision allows us to conclude that the program is *approximated* by the (singleton) set $\{\lambda y.y\}$. We return to this example after specifying the analysis.

Contours δ are strings of labels of length at most k; they serve to record the last k dynamic call points and thus distinguish instances of variables and program points. Contour environments map variable names to contours.

$$\begin{array}{rcrcll} \delta & \in & \Delta & = & \mathbf{Lab}^{\leq k} & \text{contour information} \\ ce & \in & \mathbf{CEnv} & = & \mathbf{Var} \to \Delta & \text{contour environment} \end{array}$$

Abstract values are extended to pairs of (textual) program values and contour environments closing the term, i.e. abstract closures. The environment maps variable names to the contours in place at the definition point for the free variables. The abstract cache now maps a label and a contour to an abstract (closure) value. That is:

$$\begin{array}{rcrcll} v & \in & \widehat{\mathbf{Val}} & = & \mathcal{P}(\mathbf{Term} \times \mathbf{CEnv}) & \text{abstract values} \\ \widehat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}} & = & (\mathbf{Lab} \times \Delta) \to \widehat{\mathbf{Val}} & \text{abstract caches} \end{array}$$

An acceptable k-level control flow analysis for an expression e is written $\widehat{\mathsf{C}} \models^{ce}_{\delta} e$, which states that $\widehat{\mathsf{C}}$ is an acceptable analysis of e in the context of the current environment ce and current contour δ (for the top level analysis of a program, these will both be empty). Colloquially we understand these judgments as follows: when running the programing, fragments of the original program text may be duplicated, contours distinguish between these copies (for copies created via at most k application points, beyond this the distinction in copies is blurred), so when judging an analysis correct for a program fragment, we use δ to tell us which copy of the text is being analyzed, and ce to tell us which copies of the free variables in this copy of the program are being analyzed.

The acceptability relation is given by the greatest fixed point of the functional defined according to the following clauses (again we are concerned only with least solutions):[4]

$$\begin{array}{rcl} \widehat{\mathsf{C}} & \models^{ce}_{\delta} & x^l \text{ iff } \widehat{\mathsf{C}}(x, ce(x)) \subseteq \widehat{\mathsf{C}}(l, \delta) \\ \widehat{\mathsf{C}} & \models^{ce}_{\delta} & (\lambda x.e)^l \text{ iff } (\lambda x.e, ce_0) \in \widehat{\mathsf{C}}(l, \delta) \\ & & \quad \text{where } ce_0 = ce|\mathbf{fv}(\lambda x.e_0) \\ \widehat{\mathsf{C}} & \models^{ce}_{\delta} & (t_1^{l_1}\ t_2^{l_2})^l \text{ iff } \widehat{\mathsf{C}} \models^{ce}_{\delta} t_1^{l_1} \wedge \widehat{\mathsf{C}} \models^{ce}_{\delta} t_2^{l_2} \wedge \\ & & \quad \forall (\lambda x.t_0^{l_0}, ce_0) \in \widehat{\mathsf{C}}(l_1, \delta) : \widehat{\mathsf{C}} \models^{ce'_0}_{\delta_0} t_0^{l_0} \wedge \\ & & \quad\quad \widehat{\mathsf{C}}(l_2, \delta) \subseteq \widehat{\mathsf{C}}(x, \delta_0) \wedge \widehat{\mathsf{C}}(l_0, \delta_0) \subseteq \widehat{\mathsf{C}}(l, \delta) \\ & & \quad\quad \text{where } \delta_0 = \lceil \delta, l \rceil_k \\ & & \quad\quad \text{and } ce'_0 = ce_0[x \mapsto \delta_0] \end{array}$$

The notation $\lceil \delta, l \rceil_k$ denotes the string obtained by appending l to the end of δ and taking the rightmost k labels.

Let us consider an acceptable least analysis for the program in Figure 3. We write ϵ for the empty contour and $[\,]$ for the empty contour environment. Since every λ-term in the program is closed, the contour environments in the results will always be empty so we omit it from this table:

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

And the following holds:

$$\widehat{\mathsf{C}} \models^{[\,]}_{\epsilon} ((\lambda f.((f^1 f^2)^3 (\lambda y.y^4)^5)^6)^7 (\lambda x.x^8)^9)^{10}$$

4.1 The kCFA decision problem

As we did in subsection 3.1, we now formulate a decision problem naturally answered by the analysis and ask: What is the difficulty of computing within this hierarchy? What are the sources of approximation that render such analysis tractable?

Control Flow Problem (kCFA): Given an expression e, an abstract closure $(\lambda x.e_0, ce_0)$, and a label and contour pair (l, δ) with $|\delta| \leq k$, is $(\lambda x.e_0, ce_0) \in \widehat{\mathsf{C}}(l, \delta)$ in a least analysis of e?

The source of approximation in kCFA is the bounding of the length of contour strings. But suppose k is sufficiently large that δ is never truncated during the analysis. What can be said about the precision of the result? If the contour is never truncated, the analysis is just normalization. The acceptability relation above can be read as specifying a non-standard interpreter, which is given an expression and constructs a table from which the normalized program can be retrieved.

Let's rewrite the specification to make this clear. Evaluation is paramaterized by an initially empty table and inclusion constraints are interpreted as destructive updates to the table. $\mathcal{E}[\![t^\ell]\!]^{ce}_{\delta}$ evaluates

[4] To be precise, we take as our starting point *uniform* kCFA rather than a kCFA in which $\widehat{\mathbf{Cache}} = (\mathbf{Lab} \times \mathbf{CEnv}) \to \widehat{\mathbf{Val}}$. The differences are immaterial for our purposes.

t and writes the result into the table $\widehat{\mathsf{C}}$ at location (ℓ, δ).

$$\begin{array}{lcl}
\mathcal{E}[\![x^\ell]\!]^{ce}_\delta & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \widehat{\mathsf{C}}(x, ce(x)) \\
\mathcal{E}[\![(\lambda x.e_0)^\ell]\!]^{ce}_\delta & = & \widehat{\mathsf{C}}(\ell,\delta) \leftarrow (\lambda x.e_0, ce_0) \\
 & & \quad \text{where } ce_0 = ce|\mathbf{fv}(\lambda x.e_0) \\
\mathcal{E}[\![(t_1^{\ell_1} t_2^{\ell_2})^\ell]\!]^{ce}_\delta & = & \mathcal{E}[\![t_1^{\ell_1}]\!]^{ce}_\delta; \mathcal{E}[\![t_2^{\ell_2}]\!]^{ce}_\delta; \\
 & & \text{let } (\lambda x.t_0^{\ell_0}, ce_0) = \widehat{\mathsf{C}}(\ell_1,\delta) \text{ in} \\
 & & \quad \widehat{\mathsf{C}}(x,\delta,\ell) \leftarrow \widehat{\mathsf{C}}(\ell_2,\delta); \\
 & & \quad \mathcal{E}[\![t_0^{\ell_0}]\!]^{ce_0[x\mapsto\delta,\ell]}_{\delta,\ell} \\
 & & \quad \widehat{\mathsf{C}}(\ell,\delta) \leftarrow \widehat{\mathsf{C}}(\ell_0,\delta,\ell)
\end{array}$$

The contour environment plays much the same role as an environment in a typical interpreter, but rather than mapping a variable to its value, it maps a variable to a location in the table where its value is found. The contour δ is a history of call sites in place at the current point of evaluation and serves to keep locations in the table distict (a simple induction proof shows that (ℓ, δ) is unique).

In the application case, the operator and operand are evaluated, updating the table at positions (ℓ_1, δ) and (ℓ_2, δ), respectively. The closure in (ℓ_1, δ) is retrieved from the table and the variable is "bound" by writing the value of the argument (found at position (ℓ_2, δ)) into position (x, δ, ℓ). The body of the closure is evaluated in an extended environment that maps x to the location in the table where its value is stored. After evaluating the body, the table is updated to record the value of the application as being the value found in (ℓ_0, δ, ℓ).

This evaluation function can be seen as a variant of the exact collecting semantics from which the analysis was originally abstracted[5], in other words, if k is big enough to that it is not truncated, the analysis is simply normalization. A lower bound on the hardness of the analysis, then, is the expressivity of the language that can be evaluated in the above interpreter with a δ of length at most k.

When the contours are *truncated* to length at most k (remembering the *last* k labels added to the contour, the result is kCFA. The evaluator can be modified to perform kCFA by truncating contours and since the truncation destroys the uniqueness of locations in the table, the evaluator has to be iterated until a fixed point is reached. In this case, the table $\widehat{\mathsf{C}}(\ell, \delta)$ is finite and has n^{k+1} entries. Each entry contains a set of *values* and the only values are closures $(\lambda x.e_0, ce_0)$; the environment in a closure maps p free variables to any one of n^k contours. Because there are n possible $\lambda x.e_0$ and n^{kp} such environments, there are sets of size at most n^{1+kp} in any table entry.

Observe that the above evaluation is *monotonic*: each table entries is initialzed to the empty set, and built up incrementally. Thus in kCFA, there can be at most $n^{1+(k+1)p}$ updates to $\widehat{\mathsf{C}}$, and $\mathcal{E}[\![t^\ell]\!]^{ce}_\delta$ then has at most $n^{O(p)}$ program states during evaluation. Because $p \leq n$, we conclude with the well-known observation—see, for example, Nielson et al. (1999, page 193):

Theorem 3. *kCFA is contained in* EXPTIME.

4.1.1 kCFA is NP-hard

Because CFA makes approximations, many closures can flow to a single program point and contour. In 1CFA, for example, $\lambda w.wx_1x_2\cdots x_n$ has n free variables, with an exponential number of possible associated environments mapping these variables to program points (contours of length 1). Approximation allows us to bind each x_i, independently, to either of the closed λ-terms for "true" or "false" that we saw in the PTIME-completeness proof for 0CFA. In turn, application to an n-ary Boolean function necessitates computation of all 2^n such bindings in order to compute the flow out from the application site. The term for "true" can only flow out if the Boolean function is satisfiable by some truth valuation.

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

For an appropriately chosen program point (label) ℓ, the cache location $\widehat{\mathsf{C}}(v, \ell)$ will contain the set of all possible closures which are approximated to flow to v. This set is that of all closures

$$(\lambda w.wx_1x_2\cdots x_n, ce)$$

where ce 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 $(\lambda w.wx_1x_2\cdots x_n, ce)$ provides a satisfying truth valuation for ϕ. We have as a consequence:

Theorem 4. *1CFA is* NP*-hard.*

We observe that while the computation of the *entire* cache $\widehat{\mathsf{C}}$ requires exponential time, the existence of a *specific* flow in it may well be computable in NP. A nondeterministic polynomial might compute using the "collection semantics" $\mathcal{E}[\![t^\ell]\!]^{ce}_\delta$, but rather than compute entire sets, *choose* the element of the set that bears witness to the flow. There are some details to be worked out here, which seem straightforward enough, and which we hope to report in the final version of the paper.

Conjecture 1. *1CFA is* NP*-complete.*

Increasing the precision to kCFA with $k > 1$ undermines the approximation that allows for construction of the NP-hardness proof. We use the following program transformation to render a kCFA of the original synonymous with a $(k+1)$CFA of the transformed program. The NP-hardness of kCFA for $k > 0$ falls out by iterating the transformation $k-1$ times on the 1CFA construction.

Transform expressions as follows, where $l^\star$ is a distinguished label and k a distinguished variable not appearing in the source expression:

$$\begin{array}{rcl}
\langle x^l \rangle & = & x^l \\
\langle (e_1 e_2)^l \rangle & = & (\langle e_1 \rangle \langle e_2 \rangle)^l \\
\langle (\lambda x.e)^l \rangle & = & ((\lambda k.(\lambda x.\langle e \rangle)^{l^\star})(\lambda y.y))^l
\end{array}$$

[5] However it varies in flavor—being a big step semantics rather than the structured operational semantics of Nielson et al. (1999). The motivation for SOS in Nielson et al. (1999) was to prove correctness of the analysis for non-terminating programs. Our evaluator only works for finite programs, but since we are investigating the complexity of the analysis, this is agreeable.

The transformation works by nesting every λ-abstraction in a single application context (a K-redex), which consumes the added precision of a $(k+1)$CFA. The following lemma states that top-level flows are analogous in the transformed program under a more precise analysis:

Lemma 2. *If $\widehat{\mathsf{C}} \models_\epsilon^{[]} t^l$ is a least kCFA, and $\widehat{\mathsf{C}}' \models_\epsilon^{[]} \langle t^l \rangle$ is a least $(k+1)$CFA, then $(\lambda x.e, ce) \in \widehat{\mathsf{C}}(l)$ iff $(\lambda x.\langle e \rangle, ce') \in \widehat{\mathsf{C}}'(l)$, where ce and ce' are isomorphic upto placement of $\star$-labels.*

From this, we conclude:

Theorem 5. *kCFA is* NP*-hard, for any $k > 0$.*

Observe that this program transformation is exponential in k (since the $\lambda y.y$ terms, introduced by the transformation, must then be themselves transformed in an iterated composition of the technique, in order to reduce from kCFA to 1CFA) and linear in program size. However, for any fixed k, this constant-factor expansion can be computed by a logspace-computable reduction.

4.1.2 nCFA is complete for EXPTIME

We now examine the complexity of kCFA where k is allowed to vary linearly in the size of the program.

In the previous section, we saw that 0CFA was sufficient to evaluate linear λ-terms. Following the construction of Neergaard and Mairson (2004), we can code a Turing machine transition function and machine IDs with linear terms. Suppose ϕ is a linear transition function that takes a tuple $\langle q, L, R\rangle$ consisting of the current machine state, the tape to the left of the head in reverse order, and the tape to the right of the head and returns a tuple $\langle q', L', R'\rangle$ consisting of the new state and tape. Let I be the initial configuration, then ϕI simulates one step of the machine, $\phi(\phi I)$ two steps, and $\phi^n I$, n steps.

If 0CFA is sufficient to evaluate ϕI, then what is 1CFA sufficient for? By introducing non-linearity using the Church numeral $\overline{2}$, we can iterate the transition function twice, as follows:

$$(((\lambda s.(\lambda z.(s^1(s^2z^3)^4)^5)^6)^7\phi)^8 I)^9$$

A contour of length 1 is sufficient to distinguish between the application of ϕ in the calling context of 4 and that of 5, we are able to maintain an exact analysis. Scaling up, suppose we have:

$$\begin{array}{c}((\lambda s'.(\lambda z'.(s'^{11}(s'^{12}z'^{13})^{14})^{15})^{16})^{17}\\ ((\lambda s.(\lambda z.(s^1(s^2z^3)^4)^5)^6)^7\phi)^8 I)^9\end{array}$$

A contour of length 2 is sufficient to distinguish between the application of ϕ in the calling context 14,4; 14,5; 15,4; and 15,5. Now let Φ_n be $\overline{2}^n\phi$, then an nCFA is sufficient to distinguish all of the calling contexts in which ϕ is applied, thus nCFA is exact for $\Phi_n I$—it is synonymous with normalization—and the program normalizes to $\phi^{2^n} I$. So when k is linear in the size of the program n, we can simulate a Turing machine for an exponential number of steps.

Theorem 6. *nCFA is complete for* EXPTIME.

It should be remarked that in this case, the contours are large enough that the computation is essentially by *normalization*, without using the power of any approximation. Every location of the cache $\widehat{\mathsf{C}}$ contains at most one value.

Researchers have noted that computing a more precise analysis is often cheaper than performing a less precise one. A 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). Harnessing the computational power of this reevaluation is precisely what makes the NP-hardness construction work and relegates lower bounds using exact analyses to limit cases such as $k = n$; an analysis that is exact can only be polynomial in n^k. On the other hand, these limiting cases shed analytic light on the nature of kCFA. Even when the polyvariance of the analysis is taken to an extreme as in $k = n$, the expressivity of the analysis is still limited to EXPTIME. It seems likely that alternative control flow analyses can be designed to be more expressive in the exact case.

5. LOGSPACE and η-expansion

In this section, 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. This distinction is highlighted in the discussion below.

We remark that η-expansion changes control flow analysis. If 0CFA infers that a call site χ may call a function ϕ in a program Π, and we ask the same question of the residual χ and ϕ in the η-expanded version of Π, the answer may vary.

5.1 MLL and (linear) functional programming

The sequent rules of MLL are:

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

These rules have an easy functional programming interpretation as the types of a linear programming language, following the intuitions of the Curry-Howard correspondence (Girard et al. 1989, Chapter 3).

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 $A^\perp$" interchangably, along with similar dualisms. We also regard $(A^\perp)^\perp$ synonymous with A: for example, `Int` is an integer, and $\texttt{Int}^\perp$ is a request (need) for an integer, and if you need to need an integer—$(\texttt{Int}^\perp)^\perp$—then you have an integer.

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

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

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

5.2 Atomic versus non-atomic axioms: PTIME versus LOGSPACE

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.

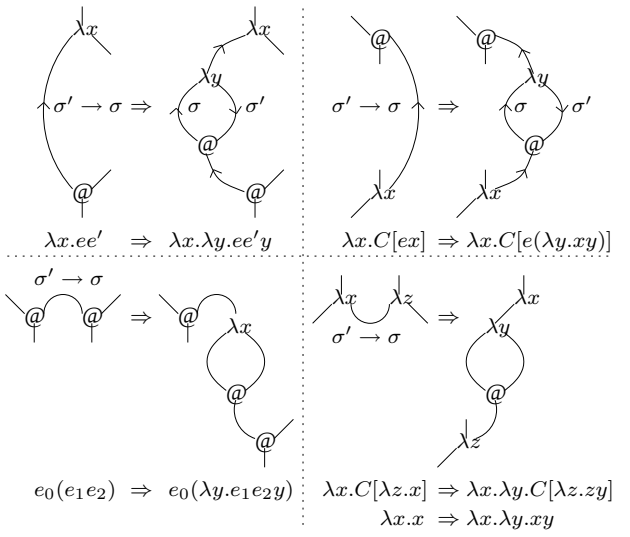


Figure 4. Expansion algorithm.

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

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

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

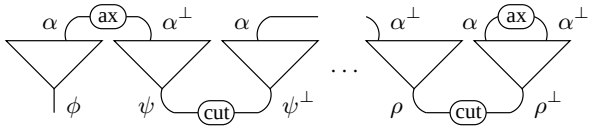
The program transformation corresponding to the above proof expansion is a version of η-expansion: see Figure 4. The left hand expansion rule is simply η, dualized in the unusual right hand rule. The right rule is written with the @ above the λ only to emphasis its duality with the left rule. Although not shown in the graphs, but implied by the term rewriting rules, an axiom may pass through any number of sharing nodes.

5.3 Normalization and 0CFA for linear programs in LOGSPACE

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. 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`.[7] 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`.

The geometry of interaction (GoI)—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. 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.



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

Lemma 3. *Begin at an axiom α and "descend" to a cut-link, saving in an (initially empty) stack whether nodes are encountered on their left or right auxiliary port. Once a cut is reached, "ascend" the accompanying structure, popping the stack and continuing left or right as specified by the stack token. Then (1) the stack empties exactly when the next axiom α' is reached, and (2) if the k-th node from the start traversed is a $\otimes$, the k-th node from the end traversed is a $\parr$, and vice versa.*

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

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 4. *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 2006):

Theorem 7. *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 1. *0CFA of linear, simply-typed, and fully η-expanded functional programs is contained in* LOGSPACE.

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

[7] 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 $\otimes$.

5.4 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 nonelementary recursive, (Statman 1979; Mairson 1992; Asperti and Mairson 1998). However, 0CFA remains contained in LOGSPACE, because it is now an *approximation*. This result follows from the following observation:

Lemma 5. *Suppose* $(\lambda x.e_0)^{l'}$ *and* $(t_1^l\ e_2)$ *occur in a simply typed, fully* η*-expanded program and* $\lambda x.e_0 \in \widehat{\mathsf{C}}(l)$*. 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 8. *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 9. *For fully* η*-expanded, simply-typed terms, 0CFA-normalization can be computed in* nondeterministic LOGSPACE.

Conjecture 2. *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 3. *An algorithm for 0CFA normalization can be realized by* optimal reduction*, where sharing nodes* always *duplicate, and never annihilate.*

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

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

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

Denote as ν the type of vector `V`.

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

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

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

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

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

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

Lemma 6. *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 10. *0CFA of a simply-typed, fully* η*-expanded program is complete for* LOGSPACE.

6. Languages with first-class 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, seperate contextual analyses and transforms must be also implemented—redundantly," in his view. Although our formulation of kCFA 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 Filinksi's symmetric λ-calculus (1989), Parigot's λ_μ calculus (1992), and most any language expressible in CPS.[8]

[8] 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

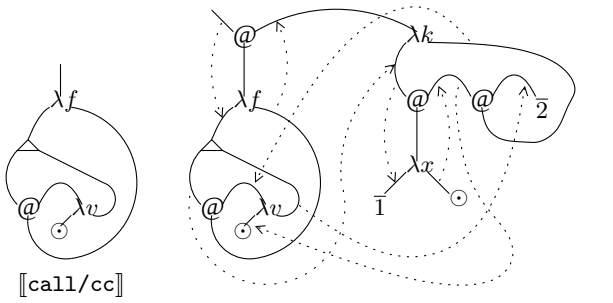


Figure 5. Graph coding of `call/cc` and example CFA graph.

The left side of Figure 5 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 5 gives the CFA graph for the program:

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

From the CFA graph we see that $\widehat{\mathsf{C}}(l) = \{\overline{1}, \overline{2}\}$, 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 non-linearity of bound variables, approximation can now be introduced via non-linear 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 5, 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—and likewise for the results on kCFA—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 $\lambda_{\mathcal{K}}$. The type-based expansion algorithm of Figure 4 applies without modification and Lemma 5 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.

functions or continuations be in tail position. Adapting this approach to such languages constitutes an active area of research for us.

7. Related work

The PTIME-completeness of 0CFA is most closely related to the PTIME-completeness of simply typing in the λ-caclculus (Mairson 2004). Both results use linearity to subvert the approximation of the analysis, and since both analyses rely on the same source of approximation, it is no suprise that they share the same lower bound on complexity.

ML typing can be viewed as a bounded running of a program (reducing all `let`-redexes) followed by a simple typing of the residual. The residual program can be exponentially larger, leading to EXPTIME-completeness results by using polymorphism to iterate a linear TM transition function (Mairson 1990). The EXPTIME-completeness of nCFA can be viewed in a similar light. Contours of length proportional to the program size provide a bounded "running" of the program by exact analysis of the non-linearity introduced by iterative doubling of the transition function.

The story is the same for k-rank bounded intesection typing—a program is run by computing k successive *minimal complete developments* and the residual is simply typed. The resulting hardness of typing is elementary in k (Neergaard and Mairson 2004), and thus the complexity class of each fixed k is seperated. On the other hand, for $(k > 0)$CFA, the complexity in the "hierarchy" remains the same as k grows. There should be a natural way of developing an alternative control flow hierarchy that relies on complete developments for its notion of bounded running that will be strictly more expressive than the kCFA examined in this paper. The result is likely to be similiar in spirit to that of Mossin (1997b), although Mossin's analysis is simply evaluation by virtue of its exactness. To remain useful, some information must be purposeless lost in order to compute an answer in less time than it takes to run the program.

It also seems likely that the linear logic based investigation into CFA presented here can be coupled with that of Neergaard and Mairson (2004) to provide the foundation for complexity results for the control flow analysis of rank-2 bounded intersection typed programs (Banerjee and Jensen 2003).

Static program analysis has been recast as various kinds of graph reachability problems, and parenthesis languages have been used to describe paths in these graphs; see Reps (2000) for example. Words in these languages are the *contexts* of the context semantics presentation (Mairson 2003) of the geometry of interaction (Girard 1989). The undecidability of decision problems for these specialized parenthesis languages corresponds naturally to versions of the halting problem.

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 graph codings, CFA graphs, and virtual wire propogation rules share a strong resemblance to the "pre-flow" graphs, flow graphs, and graph "closing rules", respectively, of Mossin (1997a). Casting the analysis in this light leads to insights from linear logic and optimal reduction. For example, as Mossin (1997a, page 78) notes, the CFA virtual paths computed by 0CFA are an approximation of the actual runtime 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 CFA algorithms presented here.

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.

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 typ-

ing and kCFA (Mossin 1997a; Palsberg and Pavlopoulou 1998). 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), 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. In the preface to their textbook, Nielson et al. (1999) make an analogy between their approach to the study of program analysis and that of complexity theory, whereby seemingly unrelated problems are shown to be the same through notions such as logspace reduction. We believe there is more than an analogy here—results on the complexity of static analyses are a useful way of understanding when two seemingly different program analyses are in fact computing the same thing.

8. Conclusions and perspective

The most obvious thing you can say about program analysis is this: if you want to know what a program is going to do, run it and find out. If you don't have time to run the program, run it for a while. If you don't have time to run it for a while, make a crude approximation. If you want to make sure you have all the answers in such an approximation, you'll end up computing "false positives". A common approximation idea is to merge whatever is known for all occurrences of a variable. Approximations are refined by gently distinguishing these variable occurrences.

Type inference follows this blueprint. Simple typing is the approximation: all occurrences of a variable have to have the same type. ML `let`-polymorphism computes a *finite development* before simple typing. Rank-bounded intersection systems iterate several minimal complete developments before simple typing (with $\wedge$-idempotency) or a linearity check (without $\wedge$-idempotency): see (Neergaard and Mairson 2004).

This is also the story of 0CFA: a truncated evaluation that merges all occurrences of a bound variable. The occurrences are then distinguished in kCFA ($k > 0$) by *contours* that annotate application sites, and an *environment* whose values are closures—and it is the free variables of these closures that are the bottleneck of kCFA. The NP-hardness lower bound, and likely NP-completeness given the known exponential upper bound, comes from computing with approximations rather than computing via exact normalization.

What other kind of static analysis could there possibly be? We would like to relate kCFA in a more precise way to the geometry of interaction, a complete static program analysis based on linear logic. It should be possible to construct an exact correspondence when k is unbounded. A further extension would be a generalization of kCFA to the precision of iterated exponentials, combining its technique with finite developments à la ML. Finally, we hope to use the techniques described here to provide a useful taxonomy of other program analyses, viewed from the perspective of the computational resources needed to realize them.

Acknowledgements: We are grateful to Olin Shivers and Matthew Might for patient deliberations on the nature of kCFA. We thank Julia Lawall for providing comments on this work.

References

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

Andrea Asperti and Harry G. Mairson. Parallel beta reduction is not elementary recursive. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 303–315. ACM Press, 1998.

Anindya Banerjee. A modular, polyvariant and type-based closure analysis. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 1–10. ACM Press, 1997.

Anindya Banerjee and Thomas Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathematical. Structures in Comp. Sci.*, 13(1):87–124, 2003.

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, 1990.

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, 18(6):730–751, 1996.

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, 1997.

Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 136–153. Springer-Verlag, 1995.

Andrzej Filinski. Declarative continuations: an investigation of duality in programming language semantics. In *Category Theory and Computer Science*, pages 224–249. Springer-Verlag, 1989.

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

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.

Timothy G. Griffin. A formulae-as-type notion of control. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–58. ACM Press, 1990.

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, 1994.

Nevin Heintze. Control-flow analysis and type systems. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 189–206. Springer-Verlag, 1995.

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, 1997.

Fritz Henglein and Harry G. Mairson. The complexity of type inference for higher-order lambda calculi. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–130. ACM Press, 1991.

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 Prin-*

ciples of programming languages, pages 393–407. ACM Press, 1995.

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

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

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

Yves Lafont. From proof-nets to interaction nets. In *Proceedings of the workshop on Advances in linear logic*, pages 225–247. Cambridge University Press, 1995.

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, 2000.

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

Harry G. Mairson. From Hilbert space to Dilbert space: context semantics as a language for games and flow analysis. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 125–125. ACM Press, 2003.

Harry G. Mairson. Linear lambda calculus and PTIME-completeness. *Journal of Functional Programming*, 14(6):623–633, 2004.

Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 382–401. ACM Press, 1990.

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

Harry G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103(2):387–394, 1992.

Harry G. Mairson and Kazushige Terui. On the computational complexity of cut-elimination in linear logic. In Carlo Blundo and Cosimo Laneve, editors, *ICTCS*, volume 2841 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2003.

Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, January (revised August) 1997a.

Christian Mossin. Exact flow analysis. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 250–264. Springer-Verlag, 1997b.

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, 2004.

Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 332–345. ACM Press, 1997.

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

Jens Palsberg. Closure analysis in constraint form. *ACM Trans. Program. Lang. Syst.*, 17(1):47–62, 1995.

Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, 17(4):576–599, 1995.

Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208. ACM Press, 1998.

Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

Thomas Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22 (1):162–186, 2000.

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, 1989.

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

Olin Shivers. *Control-flow analysis of higher-order languages of taming lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. *SIGPLAN Not.*, 39(4):257–269, 2004.

Richard Statman. The typed λ-calculus is not elementary recursive. *Theor. Comput. Sci.*, 9:73–81, 1979.

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, 1994.

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

Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst.*, 20(1):166–207, 1998.