Linear-time Subtransitive Control Flow Analysis

Nevin Heintze* David McAllester†

Abstract

We present a linear-time algorithm for bounded-type programs that builds a directed graph whose transitive closure gives exactly the results of the standard (cubic-time) Control-Flow Analysis (CFA) algorithm. Our algorithm can be used to list all functions calls from all call sites in (optimal) quadratic time. More importantly, it can be used to give linear-time algorithms for CFA-consuming applications such as:

- effects analysis: find the side-effecting expressions in a program.
- k-limited CFA: for each call-site, list the functions if there are only a few of them ($\leq$ k) and otherwise output "many".
- called-once analysis: identify all functions called from only one call-site.

1 Introduction

The control-flow graph of a program plays a central role in compilation – it identifies the block and loop structure in a program, a prerequisite for many code optimizations. For first-order languages, this graph can be directly constructed from a program because information about flow of control from one point to another is explicit in the program (we remark that in the context of tail-call optimization, some extra work may be needed; see [3]).

For languages with higher-order functions, the situation is very different: the flow of control from one point to another is not readily apparent from program text because a function can be passed around as data and subsequently called from anywhere in the program. This limits compiler optimization. One way to address this problem is to perform *control-flow analysis* (CFA) to determine (an approximation) of the functions that may be called from each call site in a program [8, 9, 17, 16]. (In fact, some form of CFA is used in most forms of analyses for higher-order languages.)

Many different control-flow analyses have been developed (often independently), with variations in:

1. *the treatment of context:* does the analysis take into account the calling context of a function (polyvariant[1] treatment) or does it fold all activations of a function together (monovariant[2] treatment)?
2. *the treatment of dead-code:* does the analysis take into account which pieces of a program can actually be called? Does it take into account reduction order?
3. *the treatment of data-constructors:* what happens when a function is stored in a list and then later extracted from the list – is the identity of a function traced through recursive data-structures?

Despite these variations, a fundamental notion of CFA has emerged – the monovariant form of CFA defined over the pure lambda calculus. We call this analysis *standard CFA* (see Section 2 for a definition). Most other forms of CFA can be viewed as modification or extensions of standard CFA. Moreover, a number of connections have been established between standard CFA and a variety of type inference problems [15, 5].

*Bell Labs, 600 Mountain Ave, Murray Hill, NJ 07974, nch@bell-labs.com.

†AT&T Labs, 600 Mountain Ave, Murray Hill, NJ 07974, dmac@research.att.com.

[1] A number of different terminologies have developed for this concept: a polyvariant analyses is often called "context-sensitive", and sometimes "polymorphic".

[2] A monovariant analysis is often called "context-insensitive", and sometimes "monomorphic".

The usual algorithm for standard CFA has $O(n^3)$ time complexity and $O(n^2)$ space complexity, where n measures the length of the input program. The conventional wisdom is that this algorithm cannot be improved because the algorithm is essentially performing a dynamic transitive closure and the best known algorithm for *dynamic* transitive closure is $O(n^3)$. Dynamic transitive closure differs from the usual transitive closure problem because new basic edges may be added during the algorithm's execution; as a result the usual techniques for obtaining sub-cubic algorithms for transitive closure cannot be applied in the dynamic case.

The apparent cubic complexity of standard CFA has been a barrier to the use of CFA in compilers. In fact, although a number of prototypes analyzers have been built, standard CFA has yet to find its way into a widely used compiler. Many implementors have instead used simpler but less accurate algorithms. For example, Bondorf and Jorgensen [2] employ an equality-based algorithm for CFA because "the equality-based flow analysis can be done in almost-linear time whereas an inclusion-based analysis is expected to be at least cubic." Another common choice is to use very simple approximations of the control-flow graph based on known function applications. We remark that, in practice, the standard CFA algorithm (and its derivatives) rarely exhibit cubic behavior, but they are often non-linear.

Recent work [6] indicates that it is unlikely we can improve on the cubic-time complexity of standard control-flow analysis, because it is as hard as the 2-NPDA acceptability problem. (Melski and Reps [12] have recently shown a similar kind of "cubic-hardness" result for first-order set-based analysis with data-constructors.) In this paper we focus on bounded-type programs. For monotyped (or simply typed) programs[3], we simply bound the tree-size of a program's types by some constant. Equivalently, we could bound a program's order and arity, where arity is defined so that currying increases argument count rather than order; for example, the usual curried version of integer map with type $(Int \to Int) \to Int\ list \to Int\ list$ has arity 2 and order 2.

Bounded-type programs capture the intuition that functions rarely have more than, say, 20 arguments or have order greater than 3, and almost never at the same time (while this holds for most hand-written programs, the case is rather less clear for automatically generated programs). This class of programs has been particularly useful for understanding the observed linear behavior of type inference for ML [13]; see [7, 10]. However, it cannot be used to control the complexity of the standard CFA algorithm, which is cubic even when type size is bounded. Section 10 illustrates this with an example.

The main result of this paper is a linear-time algorithm for bounded-type programs that builds a directed graph whose transitive closure gives exactly the results of the standard (cubic-time) CFA algorithm. Our algorithm can be used to list all functions calls from all call sites in (optimal) quadratic time. More importantly, for many applications that consume (standard) control-flow information, we can adapt our algorithm to perform the necessary control-flow analysis and post-processing of the control-flow information all in linear-time. We illustrate this by giving linear-time algorithms for:

- effects analysis: find the side-effecting expressions in a program.
- k-limited CFA: for each call-site, list the functions if there are only a few of them ($\leq$ k) and otherwise output "many".
- called-once analysis: identify all functions called from only one call-site.

Our algorithm is simple, incremental, demand-driven and easily adapted to polyvariant usage. Early experimental evidence suggests that this new algorithm is significantly faster than the standard algorithm.

Section 2 gives the definition of the standard control-flow problem. In Section 3, we present the linear-time algorithm. The main insight of our algorithm is a decoupling of the transitive closure and edge addition aspects of the standard CFA algorithm (the standard CFA algorithm can be viewed as transitive closure intertwined with edge addition due to newly discovered function applications). Section 4 uses types to show termination. Sections 5 shows termination for polymorphically typed programs. Section 6 extends the basic algorithm to records and recursive datatypes (the treatment of recursive datatypes is somewhat less accurate than in other approaches such as set-based analysis [4]). Section 7 considers polyvariance. Sections 8 and 9 show how our algorithm can be used to provide linear-time algorithms for some CFA-consuming applications. In Section 10, we provide some empirical evidence that our new algorithm has practical significance. Preliminary results suggest that the new algorithm is significantly faster than the standard algorithm for ML programs. We also provide a comparison on some programs that exhibit worst-case cubic behavior. Perhaps

[3] We discuss the notion of bounded-type for polymorphically typed programs (in the sense of ML) in Section 5.

most important, the overhead of the new algorithm (i.e. the "size of the constant") appears to be small.

In general terms, what we establish in this paper is a connection between control-flow analysis and graph reachability. Recent work by Melski and Reps [12] has show a similar kind of connection between (a limited class of) set constraints and context-free reachability. While the differences are significant (our work deals with reachability in a standard graph, whereas Melski/Reps deals with S-path reachability in a *labeled* graph), the use of graphical constructs in analysis is becoming increasingly common. We also note that very recent (and independent) work by Mossin [11] investigates ideas similar to those in this paper (in particular, the use of types to bound control-flow graph construction).

2 Control-Flow Analysis

We define standard control-flow analysis on a variant of the λ-calculus with labeled abstractions. Expressions e are defined by[4]:

$$e ::= x \mid \lambda^l x.e \mid (e_1\ e_2)$$

where x is a variable and l is a label. Such labels allow us to trace a program's control flow. A *program* is a closed term in which each abstraction has a unique label. Program evaluation is β-reduction, appropriately adapted to labeled expressions. Note that reduction preserves labels on abstractions: e.g. a single step of β-reduction rewrites $(\lambda^l x.(x\ x))\ (\lambda^{l'} y.y))$ into $(\lambda^{l'} y.y)\ (\lambda^{l'} y.y)$. There are no restrictions on the reduction order: the β-reduction can be applied at any subterm at any time.

The purpose of control-flow analysis is to associate a set of labels $L(e)$ with each sub-expression e of the program[5] such that if e reduces to an abstraction labeled l during execution of the program, then $l \in L(e)$. In other words, control-flow analysis gives a conservative approximation of the abstractions that can be encountered at each expression during execution of a program.

We can now define standard control-flow analysis as the least such association of label sets that satisfies the following two conditions:

- for any abstraction $\lambda^l x.e$, we have $l \in L(\lambda^l x.e)$, and
- for any application $(e_1\ e_2)$, if $l \in L(e_1)$ and l labels the abstraction $\lambda^l x.e$, then
  - $L(x) \supseteq L(e_2)$ for each occurrence of x bound by the abstraction l, and
  - $L((e_1\ e_2)) \supseteq L(e)$.

The standard algorithm for computing this analysis is essentially a least fixed-point computation: each label set is initially the empty set, and then the algorithm repeatedly picks a sub-expression at which one of the conditions is not satisfied, and minimally adds new labels to label sets to locally satisfy the condition. This process is guaranteed to terminate because there are only a finite number of labels in a program, and the label sets are monotonically increasing. The complexity of this algorithm is $O(n^3)$ where n is the size (number of syntax nodes) of the input program: informally, at most n labels may be added to the n label sets, and each of these n^2 possible additions may involve up to $O(n)$ work.

To give some intuition about how this kind of behavior can arise in practice, consider the following program fragment:

```
fun f x = ....

... (f x1) ....

...... (f x2) ....
```

Using the above algorithm, the label set collected for `x` is the union of the label sets collected for `x1` and `x2`. Since the number of calls to function `f` can linearly increase with program size, the information collected for `x` can grow linearly – in effect, `x` acts like a "join" point, combining information from diverse parts of a program. If `x` is applied in the body of `f`, then we must perform work proportional to the information collected for `x` at this application site. Worse, if `x` is returned then all of the information joined by `x` can flow back to the call sites of the function `f`. This join-point behavior is independent of type size and is observed in practice, particularly for extensions of standard CFA that deal with data constructors (for which library functions such as append and map become common join-points). Although it rarely leads to cubic behavior, it can have a significant impact on analysis running time and space.

Note that the standard CFA algorithm computes the label sets for each program expression

[4] When discussing control-flow of ML typable programs, we shall assume the addition of a fix construct and a let construct. However, we describe the core of our algorithm using just this simple version of the λ-calculus.

[5] Strictly speaking, the association is with each occurrence of a sub-expression: different occurrences can have different label sets.

all at once. Since each label set contains up to n labels, this represents $O(n^2)$ information, and takes $O(n^2)$ time just to output. So a linear time algorithm for "compute all label sets!" is out of the question. However, in compiler applications we rarely want to know the control-flow information for every node in the program. Instead we usually only want to know the functions that can be called from a (relatively few) specific call sites. Alternatively, we may want to know whether only one function can be called from a particular site (e.g. for inlining or specialization applications). More generally, we may not be interested in specific control-flow information, but rather we may need to know about control-flow to answer other questions such as "is this expression side-effect free?". We address the complexity of these such questions in Sections 8 and 9; for now, we consider four basic control-flow questions:

- Given a label l and an expression occurrence e, is $l \in L(e)$?
- Given an expression occurrence e, compute $L(e)$.
- Given a label l, compute all expression occurrences e such that $l \in L(e)$.
- Compute all label sets.

The following table compares the complexity of our algorithm with the standard algorithm on these questions, for bounded-type programs.

Problem	Std Alg.	New Alg.
Is $l \in L(e)$?	$O(n^3)$	$O(n)$
$L(e)$	$O(n^3)$	$O(n)$
$\{e : l \in L(e)\}$	$O(n^3)$	$O(n)$
All Label Sets	$O(n^3)$	$O(n^2)$

3 The Algorithm

We begin by reformulating the definition of standard control-flow information as a transition system between program nodes. This reformulation makes explicit the connection with transitive closure. For convenience we assume that programs are renamed to ensure that bound variables are distinct. Now, for a program P, construct the following deduction rules:

$$\frac{}{\lambda^l x.e \longrightarrow \lambda^l x.e} \qquad (\textsc{abs})$$

$$\frac{e_1 \longrightarrow \lambda^l x.e}{x \longrightarrow e_2} \quad (\text{if } (e_1\ e_2) \text{ in } P) \qquad (\textsc{app-1})$$

$$\frac{e_1 \longrightarrow \lambda^l x.e}{(e_1\ e_2) \longrightarrow e} \quad (\text{if } (e_1\ e_2) \text{ in } P) \qquad (\textsc{app-2})$$

$$\frac{e_1 \longrightarrow e_2 \qquad e_2 \longrightarrow e_3}{e_1 \longrightarrow e_3} \qquad (\textsc{trans})$$

where the condition on the second and third rules "if $(e_1\ e_2)$ in P" indicates that there is instance of the respective rule for each occurrence of a term of the form $(e_1\ e_2)$ in P. Intuitively, an edge $e_1 \longrightarrow e_2$ indicates that anything (e.g. an abstraction) we can derive from e_2 is also derivable from e_1. In the case where e_2 is itself an abstraction, this edge says that e_2 is one possible "value" for e_1. The first rule is a boot-strapping rule that says that any abstraction leads to itself. The second and third rules deal with application and respectively say that if there is a transition from the operator of an application to an abstraction, then add a transition from the bound variable of the abstraction to the operand of the application (rule APP-1) and also add a transition from the entire application to the body of the abstraction (rule APP-2). The final rule is transitivity (we note that it is sufficient to restrict this rule to the case where e_3 is an abstraction). Standard control-flow analysis can now be redefined as follows: given a program expression e, find all abstractions $\lambda^l x.e$ such that $e \rightarrow \lambda^l x.e$ is derivable from the above rules.

For example, consider $(\lambda^l x.(x\ x)\ (\lambda^{l'} x'.x'))$. The above rules lead to:

$$\begin{array}{rcll}
\lambda^l x.(x\ x) & \longrightarrow & \lambda^l x.(x\ x) & (\textsc{abs}) \qquad (1)\\
\lambda^{l'} x'.x' & \longrightarrow & \lambda^{l'} x'.x' & (\textsc{abs}) \qquad (2)\\
x & \longrightarrow & \lambda^{l'} x'.x' & (\textsc{app-1}) \qquad (3)\\
(\lambda^l x.(x\ x)\ \lambda^{l'} x'.x') & \longrightarrow & (x\ x) & (\textsc{app-2}) \qquad (4)\\
x' & \longrightarrow & \lambda^{l'} x'.x' & (\textsc{app-1}) \qquad (5)\\
(x\ x) & \longrightarrow & x' & (\textsc{app-2}) \qquad (6)\\
(\lambda^l x.(x\ x)\ \lambda^{l'} x'.x') & \longrightarrow & \lambda^{l'} x'.x' & (\textsc{trans}{:}4,5,6)
\end{array}$$

The last rule follows from two applications of transitivity on (4), (6) and (5) (in that order). In effect, the four deduction rules ABS, APP-1, APP-2 and TRANS define a dynamic transitive closure problem: ABS sets up some initial edges, TRANS is transitive closure, and APP-1 and APP-2 add new basic edges as the transitive closure proceeds. We refer to this system as DTC because of its close connection to dynamic transitive closure. Clearly we can use DTC as the basis of an iterative fixed-point algorithm for control-flow analysis: we start with the empty set of transitions and add a transition $e \longrightarrow e'$ if it follows from one of the above rules in the context of the set of transitions obtained thus far. This algorithm performs essentially the same steps as the algorithm based on the previous control-flow definition and is also $O(n^3)$ (in fact, the close correspondence between the two algorithms can be used to provide an easy proof of equivalence of the two definitions).

Observe that in this algorithm, the transitive closure computation in intertwined with the addition of new edges. However, this does not need to be the case; this is a key insight of our algorithm. To show this, we define a new transition system. First, define a set of nodes n by the following grammar:

$$n ::= e \mid dom(n) \mid ran(n)$$

Intuitively, $dom(n)$ represents the "domain" of the node n, and $ran(n)$ represents the "range" of the node n. If n corresponds to an abstraction, then $dom(n)$ is simply the argument (bound variable) of the abstraction; otherwise, $dom(n)$ denotes the collection of arguments of the abstractions represented by n. Similarly, if n corresponds to an abstraction, then $ran(n)$ is simply the result (body) of the abstraction; otherwise, $dom(n)$ denotes the collection of results of abstractions represented by n.

Now, define a transition system between nodes n as follows:

$$\frac{}{x \longrightarrow dom(\lambda^l x.e)} \quad (\text{if } \lambda^l x.e \text{ in } P) \qquad (\text{ABS-1})$$

$$\frac{}{ran(\lambda^l x.e) \longrightarrow e} \quad (\text{if } \lambda^l x.e \text{ in } P) \qquad (\text{ABS-2})$$

$$\frac{}{dom(e_1) \longrightarrow e_2} \quad (\text{if } (e_1\ e_2) \text{ in } P) \qquad (\text{APP-1})$$

$$\frac{}{(e_1\ e_2) \longrightarrow ran(e_1)} \quad (\text{if } (e_1\ e_2) \text{ in } P) \qquad (\text{APP-2})$$

$$\frac{n_1 \longrightarrow n_2}{dom(n_2) \longrightarrow dom(n_1)} \qquad (\text{CLOSE-DOM})$$

$$\frac{n_1 \longrightarrow n_2}{ran(n_1) \longrightarrow ran(n_2)} \qquad (\text{CLOSE-RAN})$$

Call this system LC (for "linear closure"); it forms the core part of our linear-time CFA algorithm. To illustrate LC, consider the program $(\lambda^l x.(x\ x)\ (\lambda^{l'} x'.x'))$ used earlier in this section. Applying the first four rules leads to:

$$\begin{aligned}
x &\xrightarrow{\text{ABS-1}} dom(\lambda^l x.(x\ x)) && (1)\\
ran(\lambda^l x.(x\ x)) &\xrightarrow{\text{ABS-2}} (x\ x) && (2)\\
x' &\xrightarrow{\text{ABS-1}} dom(\lambda^{l'} x'.x') && (3)\\
ran(\lambda^{l'} x'.x') &\xrightarrow{\text{ABS-2}} x' && (4)\\
dom(\lambda^l x.(x\ x)) &\xrightarrow{\text{APP-1}} \lambda^{l'} x'.x' && (5)\\
(\lambda^l x.(x\ x)\ (\lambda^{l'} x'.x')) &\xrightarrow{\text{APP-2}} ran(\lambda^l x.(x\ x)) && (6)\\
dom(x) &\xrightarrow{\text{APP-1}} x && (7)\\
(x\ x) &\xrightarrow{\text{APP-2}} ran(x) && (8)
\end{aligned}$$

where the subscripts on the arrows indicate which rule is employed. Applying the last two rules leads to the following transitions (among others).

$$\begin{aligned}
dom(\lambda^{l'} x'.x') &\longrightarrow dom(dom(\lambda^l x.(x\ x))) && (9)\\
ran(dom(\lambda^l x.(x\ x))) &\longrightarrow ran(\lambda^{l'} x'.x') && (10)\\
dom(dom(\lambda^l x.(x\ x))) &\longrightarrow dom(x) && (11)\\
ran(x) &\longrightarrow ran(dom(\lambda^l x.(x\ x))) && (12)
\end{aligned}$$

(9) and (11) are by CLOSE-DOM; (10) and (12) are by CLOSE-RAN. Combining these transitions, we can derive $\lambda^{l'} x'.x'$ from $(\lambda^l x.(x\ x)\ (\lambda^{l'} x'.x'))$:

$$\begin{array}{lll}
(\lambda^l x.(x\ x)\ (\lambda^{l'} x'.x')) & & \\
\quad \longrightarrow ran(\lambda^l x.(x\ x)) & \text{by} & (6)\\
\quad \longrightarrow (x\ x) & \text{by} & (2)\\
\quad \longrightarrow ran(x) & \text{by} & (8)\\
\quad \longrightarrow ran(dom(\lambda^l x.(x\ x))) & \text{by} & (12)\\
\quad \longrightarrow ran(\lambda^{l'} x'.x') & \text{by} & (10)\\
\quad \longrightarrow x' & \text{by} & (4)\\
\quad \longrightarrow dom(\lambda^{l'} x'.x') & \text{by} & (3)\\
\quad \longrightarrow dom(dom(\lambda^l x.(x\ x))) & \text{by} & (9)\\
\quad \longrightarrow dom(x) & \text{by} & (11)\\
\quad \longrightarrow x & \text{by} & (7)\\
\quad \longrightarrow dom(\lambda^l x.(x\ x)) & \text{by} & (1)\\
\quad \longrightarrow \lambda^{l'} x'.x' & \text{by} & (5)
\end{array}$$

Compare this to DTC in which the transition $\lambda^{l'} x'.x' \longrightarrow (\lambda^l x.(x\ x)\ (\lambda^{l'} x'.x'))$ was added by the deduction rules. In other words, what was a single transition step in DTC has become a multi-step transition in LC. This relationship holds in general: the transitive closure of LC corresponds to DTC in the following sense:

Proposition 1 *For all programs P, and expressions e and $\lambda^l x.e$ appearing in P,*
$e \longrightarrow \lambda^l x.e$ in DTC iff for some nodes n_i, $e \longrightarrow n_1 \longrightarrow \cdots \longrightarrow n_k \longrightarrow \lambda^l x.e$ in LC.

What we have achieved, then, is a factorization of the algorithm into two separate parts: an edge-adding phase (which adds basic edges) and a transitive closure phase. However, note that the CLOSE-DOM and CLOSE-RAN rules are open ended: given any edge $n \to n'$ (added by one of the other rules), the CLOSE-RAN rule says that we can add all edges of the form $ran^k(n) \to ran^k(n')$ for all $k \geq 1$, and similarly for the CLOSE-DOM rule.

To control the application of the CLOSE-DOM and CLOSE-RAN rules, we make them demand driven, as follows:

$$\frac{n_1 \longrightarrow n_2 \qquad n \to dom(n_2)}{dom(n_2) \longrightarrow dom(n_1)} \; (\text{CLOSE-DOM}')$$

$$\frac{n_1 \longrightarrow n_2 \qquad n \to ran(n_1)}{ran(n_1) \longrightarrow ran(n_2)} \; (\text{CLOSE-RAN}')$$

This means that CLOSE-DOM$'$ can only applied if there is a transition whose right-hand-side could immediately match with the left-hand-side of the added transition i.e. if it is "needed". Similarly for CLOSE-RAN$'$. Call this new system LC$'$ (that is, LC$'$ consists of ABS-1, ABS-2, APP-1, APP-2, CLOSE-DOM$'$ and CLOSE-RAN$'$). LC$'$ is equivalent to LC in the following sense:

Proposition 2 *For all programs P, and expressions e and $\lambda^l x.e$ appearing in P:*

- *There exist nodes n_i such that $e \longrightarrow n_1 \longrightarrow \cdots \longrightarrow n_k \longrightarrow \lambda^l x.e$ in LC iff*
- *there exist nodes n'_i such that $e \longrightarrow n'_1 \longrightarrow \cdots \longrightarrow n'_{k'} \longrightarrow \lambda^l x.e$ in LC$'$.*

This modification of LC into LC$'$ improves the termination properties of the system (discussed further in the next section). It also introduces an element of demand-driven/incremental behavior. For example, suppose that we have a function $\lambda^l x.x$. The rules introduce edges $ran(\lambda^l x.x) \to x$ and $x \to dom(\lambda^l x.x)$. At this stage, these are the only edges involving these nodes. Eventually, if and when the function is used, we may invoke the CLOSE-DOM$'$ and CLOSE-RAN$'$ rules. For example, if the entire program is $((\lambda^l x.x\ \lambda^{l'} y.y)\ e)$, then the CLOSE-DOM$'$ and CLOSE-RAN$'$ rules will add $ran(ran(\lambda^l x.x)) \to ran(x)$ and $ran(x) \to ran(dom(\lambda^l x.x))$, amongst others. In other words, we only explore the parts of the "type" of an expression that are actually needed.

4 Termination

The LC$'$ system can be viewed as an algorithm: given a program P, add all of the edges specified by the rules (this is best represented as a graph). However, this procedure does not terminate in general. We now show that the algorithm:

- terminates for typed programs (either simply typed or polymorphically typed).
- is fast (linear) for bounded-type programs (either simply typed or polymorphically typed).

In essence, we shall use the types as a template for the nodes that need to be considered. To illustrate this, suppose e is a program expression with (non-polymorphic) type $(\tau_1 \to \tau_2) \to \tau_3 \to \tau_4$, then we need to consider six new nodes, one for each proper subexpression of the type: $dom(e)$, $ran(e)$, $dom(dom(e))$, $ran(dom(e))$, $dom(ran(e))$ and $ran(ran(e))$. In general, the number of new nodes that must be added corresponds to the size of the type trees of program nodes. We show how this idea can be applied to programs with polymorphic types in the next section (Section 5); for the moment we shall consider programs with monotypes.

Bounding the number of nodes guarantees termination, because the inference rules ABS-1, ABS-2, APP-1, APP-2 generate a fixed (program-dependent) number of rules, and CLOSE-DOM$'$ and CLOSE-RAN$'$ can add at most one new rule for each rule/node pair. In the case of programs with k-bounded types, the size of these type trees is bounded by k, and hence we can obtain a linear bound on the number of rules that must be added. (Note that, in general, the tree-size of a program can be exponential in program size — for programs that exhibit this behavior, our proposed algorithm would be a poor choice compared with the standard cubic-time algorithm. For untyped (or recursively typed programs), there is no bound, and our algorithm may not terminate.)

To make the behavior of our algorithm on bounded-type program more precise, fix on some constant k, and define $\mathcal{P}_k$ to be the class of monotyped programs whose whose types are bounded by k. In system LC$'$ we have:

Proposition 3 *There exists a constant c_k such that when LC$'$ is applied to a program P in $\mathcal{P}_k$, LC$'$ constructs at most $c_k \cdot |P|$ edges where $|P|$ denotes the size of program P.*

Hence, we obtain the following linear-time algorithms for bounded-type monotyped programs:

Algorithm 1
Input: A program P, label l and a program sub-expression e.
Output: Is $l \in L(e)$?
1. Apply LC$'$ to P.
2. Use graph reachability to determine whether l is reachable from e.

Algorithm 2
Input: A program P and a program sub-expression e.
Output: $L(e)$
1. Apply LC$'$ to P.
2. Use graph reachability to find all nodes reachable from e.
3. Output the labels of abstractions in these reachable nodes.

We can also obtain an $O(n^2)$ algorithm for computing all label sets (i.e. complete CFA information) by repeatedly applying Algorithm 2 to all program sub-expressions.

The key part of our new algorithm is the use of the type tree at each program node to limit the number of edges that must be added during the analysis. Proposition 4 bounds the number of nodes using the maximum size type trees that can appear at any program node. This provides a rather loose bound. We could obtain tighter bounds by observing that the work done by the algorithm at each node is proportional to the type tree *at that node*. Hence the total work done by the algorithm is proportional to the sum of the type tree sizes at each node. In other words, it is proportional to $k_{ave} \cdot |P|$, where k_{ave} is the average size of the type trees at each node in the program. One of the principal concerns of our implementation was the size of this constant: would it be prohibitive? Early results indicate that this is not the case: the constant is quite small, typically around 2 or 3.

Note that the algorithm itself does not actually look at the types during its execution. Rather, the types are used only to establish termination (and the linear-time complexity bounds in the bounded-type case). In other words, our algorithm only needs to know that the (appropriate) types exist – it does not need to know what they are. This simplifies implementation — we do not need to transmit the types from type inference to our algorithm.

5 Polymorphic Types

Thus far we have considered monomorphic programs and we have assumed a bound k on the size of the monotypes in a program. Now consider polymorphically typed programs (in the sense of ML) and suppose our expression language is extended with an appropriate let construct. There are at least two notions of bounded size polymorphically typed programs. McAllester [10] defines that a polymorphically typed program P has bounded type size if there is some constant k such that the tree-size of the monotypes of each expression in the let-expansion of P all have size $\leq k$. Alternatively, motivated by Henglein's "ML programs with small types" [7], we can define that a polymorphically typed program P has bounded type size if there is some constant k such that the types (including polytypes) of expressions in P all have tree-size bounded by $\leq k$.

Unfortunately, the two definitions are not equivalent[6]. We shall use McAllester's definition. Suppose that we have a polymorphically typed programs (according to McAllester's definition). For monotyped program, we bound the running of the algorithm by using the monotypes of expressions to provide a template for the nodes that need to be considered during the edge-adding phase. The situation is similar for polymorphically typed programs, except that we use the induced collection of monotypes in the let-expansion of a program to provide a collection of templates for bounding the behavior of our algorithm. Note, again, that our algorithm does not actually need to have the types (and in particular, our algorithm does not need to construct the let-expansion of the program!); we just use their existence to prove termination. For example, in the program

```
fun id x = x
val y = ((id id) id) 1
```

the induced monotypes for `id` are $int \rightarrow int$, $(int \rightarrow int) \rightarrow (int \rightarrow int)$ and $((int \rightarrow int) \rightarrow (int \rightarrow int)) \rightarrow ((int \rightarrow int) \rightarrow (int \rightarrow int))$. In essence, we can set up a correspondence between each node n added during the edge-adding step and a position in some type tree for the based node of n (recall that nodes in the edges-adding phase are built from basic nodes by applying $dom(\cdot)$ and $ran(\cdot)$).

To show that our algorithm adds only a

[6] Consider the program consisting of n functions where the first function f_0 is just the identity function, and f_{i+1} is defined to be $\lambda x.(f_i\ f_i)\ x$. This program has bounded type using Henglein's definition, but the monotypes in the let-expansion of the program have exponential tree size.

bounded number of edges for bounded type size polymorphically typed programs, it therefore suffices to show that the number of distinct positions in the type trees of the monotypes in the let-expanded version of the program. This is immediate, since the sizes of the monotypes in the let-expanded program are bounded by some constant k, and hence there is at most a total of 2^k positions in these types.

We have thus established that our algorithm runs in linear-time on bounded-type (in the sense of McAllester) polymorphic programs. Note that although we have addressed programs with polymorphic types, the algorithm itself is still monovariant (context insensitive). Making the algorithm polyvariant is a separate issue, and is considered in Section 7.

6 CFA for ML

Thus far, we have worked in the context of a simple version of the lambda calculus (with just abstraction and application). We now extend the algorithm to recursion, records and (recursive) datatypes. First consider fix: since we have worked with simply typed lambda terms with only abstraction and application, there is no recursion. To address this, consider adding a construct $\mathit{letrec}\ f = \lambda^l x.e_1\ \mathit{in}\ e_2$. It is simple to extend the linear-time algorithm for this construct: for each instance of this construct, we add transitions:

$$\begin{aligned} \mathit{letrec}\ f = \lambda^l x.e_1\ \mathit{in}\ e_2 &\longrightarrow e_2 \\ f &\longrightarrow \lambda^l x.e_1 \end{aligned}$$

Next consider records. Suppose we add constructs $(e_1, \ldots, e_n)$ and proj_j, $j = 1..n$, for record creation and accessing. We extend the algorithm by adding proj_j as a node operator (i.e. it has the same status as *dom* and *ran* in the last section, and has its own closure rule, similar to CLOSE-RAN). Then, for each expression $(e_1, \ldots, e_n)$, we add transitions $\mathit{proj}_j((e_1, \ldots, e_n)) \longrightarrow e_j$, $j = 1..n$, and each program expression $\mathit{proj}_j(e)$ is treated as a node. If the program has bounded types, then only a bounded number of nodes need be considered, and so the extended algorithm is linear-time. (Note, however, that for programs with records, bounded-order and bounded-arity no longer implies bounded type size.)

Next consider recursive data types. One possibility is to ignore recursive data types: whenever a function is put in a recursive data structure and then extracted, we lose all information about the function (i.e. we obtain the set of all abstraction labels). The rational for this is that functions are rarely put in recursive data structures, and so we can obtain most of the important information about control-flow in a program by ignoring recursive data types.

However, many generalized forms of CFA track data-constructors in the same way as they track functions: the advantage here is that not only do they give better control-flow information, but they also give information about the shape of first-order values. We consider a similar approach. First, we extend the node operators *dom* and *ran* with additional operators just as we did for records. Specifically, we add one operator ("de-constructor") for each argument of each data-constructor[7]: for an n-ary constructor c, we would add $c^{-1}_{(1)}, \ldots, c^{-1}_{(n)}$. Then we add appropriate (demand-driven) closures rules for deconstructors. Finally, for each expression e of form $c(e_1, \ldots, e_n)$, we add transitions $c^{-1}_{(j)}(e) \longrightarrow e_j$, $j = 1..n$.

Unfortunately, as formulated, we have no way of bounding the size of the nodes we must consider. In fact, the monadic monotone closure problem[8] can be mapped into this analysis problem, and Neal [14] has shown that monadic monotone closure is essentially as hard as the 2NPDA acceptability problem, a well-studied problem for which the best known algorithm is $O(n^3)$ and has not been improved since Aho, Hopcroft and Ullmann's early work [1] in 1968. Melski and Reps [12] have recently obtained a similar result in their work on set-based analysis and context-free reachability.

To reduce this complexity, we consider two alternatives, both of which reduce the accuracy of the analysis so that it is less accurate than, for example, mono-variant set-based analysis. The basic idea is to use a finite node congruence that bounds the number of nodes that are considered (the algorithm considers only one node from each congruence class) at the expense of reducing analysis accuracy. First, note that each node can be associated with a type. In particular, each node of the form $c^{-1}_{(i)}(n)$ can be associated with the type of the i^{th} argument of the constructor c. Let $\tau(n)$ denote the type thus associated with n (we

[7] For simplicity, we view an ML datatype declaration as a definition of a collection of of multi-arity data-constructors.

[8] This is a generalization of transitive closure that includes two (or more) monotone node functions f and g such that if n is a node, then $f(n)$ and $g(n)$ are nodes, and if $n \rightarrow n'$ then $f(n) \rightarrow f(n')$ and $g(n) \rightarrow g(n')$. Given a set of edges between nodes, and two nodes n and n' appearing in this set, does $n \rightarrow n'$ follow from the standard transitive closure rule and the additional f and g rules?

shall only be interested in the case where $\tau(n)$ is a datatype).

We now define two node congruences (by congruence, we mean that if $n_1 \equiv n_2$ then $dom(n_1) \equiv dom(n_2)$, etc.). The first congruence, $\equiv_1$, is defined to be the least congruence such that $n_1 \equiv_1 n_2$ whenever $\tau(n_1) = \tau(n_2)$ and both are datatypes. The second congruence differs from the first in that only nodes that have the same base node and involve a de-constructor are considered equivalent. To this end, observe that any node can be written in the form $\alpha(n)$ where n is a basic node and α is a sequence of *dom*, *ran*, de-constructors, etc. We say that n is the base node of $\alpha(n)$. Now, define $\equiv_2$, to be the least congruence such that $n_1 \equiv_2 n_2$ whenever (a) $\tau(n_1) = \tau(n_2)$ and both are datatypes and (b) n_1 and n_2 both have the same base node and involve a de-constructor. This second congruence is finer than the first, and it leads to a strictly more accurate analysis. To illustrate these constructions, suppose that we have the program fragment `cons(2, cons(1, nil))`. Let e denote this expression, and let e' denote the sub-expression `cons(1, nil)`. Using $\equiv_2$, we generate the following edges (we include some of the rules that may be generated by the closure rules):

$$
\begin{array}{rcll}
e & \longrightarrow & cons(2, e') & \\
e' & \longrightarrow & cons(1, nil) & \\
car(e) & \longrightarrow & 2 & \\
cdr(e) & \longrightarrow & e' & \\
car(e') & \longrightarrow & 1 & \\
cdr(e') & \longrightarrow & nil & \\
car(cdr(e)) & \longrightarrow & car(e') & \text{(CAR-CLOSE)} \\
cdr(e) & \longrightarrow & cdr(e') & \text{(CDR-CLOSE)}
\end{array}
$$

noting that in the last line, $cdr(e) \equiv_2 cdr(cdr(e))$. Now, if we use $\equiv_1$ instead of $\equiv_2$, then e, e', $cdr(e)$, $cdr(e')$ would all be in the same equivalence class, and so, for example, there would be edges to both 1 and 2 from $car(e)$.

For bounded type programs, $\equiv_1$ generates $O(n)$ congruence classes, and this leads to a linear-time analysis algorithm. In contrast, for bounded type programs, $\equiv_2$ generates up to $O(n^2)$ congruence classes, and hence leads to a quadratic-time analysis algorithm. However, if in addition to bounded types, we assume that nesting levels of datatypes are bounded, then $\equiv_2$ gives a linear-time algorithm. We define nesting levels of datatypes as follows: label a datatype definition that does not mention other datatypes with 0, and label any other datatype definition with the maximum of the labels of all datatypes it uses, plus 1.

We are currently investigating the tradeoffs between these two approaches. In particular, how much more accurate is the second approach? Note that the first approach is akin to statically fixing the set of allowed functions that can appear in a particular slot in a data-constructor; the second approach allows more dynamic behavior. We also plan to investigate whether the bounded nesting-level assumption for datatypes is realistic.

7 Polyvariance

So far, we have described a monovariant algorithm. We now describe polyvariant extensions to our algorithm that are analogous to let-polymorphism. Consider a program P. At a very naive level, consider just let-expanding P and analyzing the resulting (probably very large) program. Our intent is to develop an analysis whose end result is equivalent to doing a monomorphic analysis of the let-expanded P, without doing the explicit let-expansion. Instead, we analyze the function once, and build a summary of the analysis of its code body. The resulting parameterized and simplified graph can then be instantiated (copied) at the points of the function where it is mentioned, much like polymorphic type inference in ML.

One of the strengths of our algorithm is that the simplification/parameterization steps can be easily carried out — they correspond to graph reachability and simplification steps. To illustrate the basic issues, let e be the code fragment $\lambda^l x.((\lambda^{l'} y.x)\ nil)$. If we look at e in isolation, the LC$'$ rules introduce edges:

$$
\begin{array}{rcl}
ran(\lambda^{l'} y.x) & \longrightarrow & x \\
y & \longrightarrow & dom(\lambda^{l'} y.x) \\
dom(\lambda^{l'} y.x) & \longrightarrow & nil \\
((\lambda^l y.x)\ nil) & \longrightarrow & ran(\lambda^{l'} y.x) \\
ran(e) & \longrightarrow & ((\lambda^l y.x)\ nil) \\
x & \longrightarrow & dom(e)
\end{array}
$$

To simplify/parameterize this graph fragment, we first isolate the critical nodes, which are those nodes that may be used by surrounding program text, in this case $ran(\lambda^{l'} x.((\lambda^l y.x)\ nil))$ and $dom(\lambda^{l'} x.((\lambda^l y.x)\ nil))$. Next, we do a graph reachability from these two nodes (and here we must generalize reachable so that if n is reachable, then so is $dom(n)$ and $ran(n)$). Any non-reachable nodes (such as *nil*) can now be removed. Finally, we can compress the graph and remove intermediate nodes (such as x). In this case, we are left with just $ran(e) \rightarrow dom(e)$.

For a let-bound function, we can first analyze the function using the above technique to obtain a simplified graph fragment representing the analysis of the function; then, we make copies of this graph fragment for each place the function is used. In practice, we rarely want to blindly duplicate graph fragments for all functions in this way (in general, this duplication could lead to an exponential control-flow analysis). Rather, we focus on functions where polyvariance pays off. For example, we could look at the types of the program and determine which functions are polymorphic. Alternatively, we could first perform a simple monovariant analysis, and then use that information to control a subsequent polyvariant analysis (see e.g. [4]).

Note that we could force our polyvariant algorithm to be linear-time by restricting polyvariance so that there is some global bound on the number of times each graph fragment is effectively duplicated (if one graph fragment is duplicated inside another graph fragment, then any duplication of the enclosing graph fragment must be counted as duplication of the enclosed fragment).

8 Linear-time Effects Analysis

Suppose that we want to use CFA information to drive an effects analysis. We could, for example, run the standard CFA algorithm, build the list of functions that can be called from each call-site, and then iterate over this information to determine which expressions have side-effects. For simplicity, assume that all side-effecting primitives are fully applied and that the language consists only of applications and abstractions. We start by marking applications of side-effecting primitives as side-effecting. Then, we mark an application $(e_1\ e_2)$ as side-effecting if either e_1 or e_2 are marked side-effecting or if the CFA says that e_1 could be a side-effecting function. Note that this analysis has complexity at least quadratic in the program size, because it uses a representation of control-flow information that is quadratic in program size.

Using our algorithm, we can obtain a procedure that computes exactly the same effects information as the process just described, but runs in linear-time. The basic idea is that we color all applications that involve side-effecting operations with red, and then propagate coloring as follows: (a) a node $(e_1\ e_2)$ is colored red if either e_1, e_2 or $ran(e_1)$ are red; (b) a node $ran(e)$ is colored red if there is an edge $ran(e) \rightarrow e'$ and e' is red. Rule (a) corresponds to the condition used in the previous paragraph; rule (b) is a (limited) form of transitive closure for coloring. This is clearly a linear-time procedure (it is just a graph reachability problem).

9 Linear-time k-limited CFA

In many applications of CFA, we are only interested in knowing information about call sites where a small number of functions can be called (e.g. one, two or three functions). If more functions than that may be called, then we might not be interested in knowing the exact details of the functions, because the optimization we have in mind may be intractable or inappropriate. Examples of these kinds of applications include inlining and specialization.

We can use our algorithm to obtain a linear-time procedure for computing this information as follows. First, we annotate with each node with a value that is either a small set or the token "many". We start by annotating nodes corresponding to functions with the singleton set containing just that function, and all other nodes with the empty set. Then, we propagate information back along edges: if a node n has edges to nodes with sets $S_1, \ldots, S_j$, then we update the annotation at n with $S_1 \cup \cdots \cup S_j$ if this is a small set (size $\leq k$), and "many" otherwise. Each update can be done in constant time, each node can be updated at most a constant number of times, and hence if we only propagate changes, we can obtain a linear-time algorithm for computing k-limited CFA.

10 Experimental Results

We present some preliminary results from a prototype implementation of the linear-time algorithm. This prototype implements the basic linear-time CFA algorithm, with extensions for datatypes and records; however certain aspects of ML have not yet been implemented, and as a result the benchmarks we have been able to run are limited.

Our implementation is essentially a naive implementation of the algorithm described in this paper. A number of improvements could be made (such as taking advantage of the many nodes that have only one outgoing edge). We expect considerable improvement in the performance of the prototype as we better understand how the algorithm behaves in practice. It is also likely that we can exploit a number of graph implementation techniques.

We compare the performance of the linear-time algorithm with an implementation of set-based analysis (SBA) [4], run in monovariant mode (a generalization of the standard CFA algorithm). All results are for a 150 MHz MIPS R4400 processor with 512 MBytes; all timings are user time in seconds and represent the fastest of 10 runs of the benchmark. Each benchmark consists of analyzing the example program and writing out the control flow information for all non-trivial applications (i.e. applications of the form $(e_1\ e_2)$ where e_1 is not a function identifier or an abstraction).

The first set of results are for a parameterized benchmark that illustrates the cubic behavior of the standard CFA algorithm. The benchmark of size 1 consists of:

```
fun fs x = x
fun bs x = x
fun f1 x = x
fun b1 x = x
val x1 = b1(fs f1)
val y1 = (bs b1) f1
```

and the benchmark of size n consists of the first two lines of the above code and n copies of the last four lines, with `f1`, `b1`, `x1` and `y1` appropriately renamed. The following table presents results for a variety of sizes of this benchmark. The cubic-time behavior of SBA is clear (the table not only give runtimes, but also a measure of the units of work involved, since cache effects and optimizations in the implementation of SBA itself mean that timings are somewhat misleading). The last three columns describe the behavior of our new algorithm: the first two of them give the results for the linear-time LC algorithm, and the last one shows the quadratic cost of querying all non-trivial applications (there are $O(n)$ of them and each has cost $O(n)$). Note that the graph building phase (which consists of a simple linear-time pass over the program text) appears to be slightly non-linear (e.g. $0.029 \times 2 \neq 0.082$). This may be due to cache effects, timing inaccuracies, or to inefficiencies introduced in the implementation of the prototype (e.g. lists are currently used to represent some aspects of the graph's structure).

Next, we give the results from two standard SML benchmarks, the life program, and the lexer generator. These results indicate that our preliminary implementation of the linear-time algorithm is 2.5 – 3 times faster than SBA. Perhaps more significant is the number of nodes generated by the linear-time algorithm. The number of nodes in the "build" phase of the analysis is essentially the same as the number of syntax nodes in the program. The key quantity is the number of nodes added during the "close" phase of the algorithm: this gives a measure of the number of times rules such as CLOSE-DOM and CLOSE-RAN are applied. The results suggest that the number of nodes added in the close phase is typically no more than the number of nodes in the build phase, although more benchmarks are clearly needed.

We remark that the timing results probably overstate the cost of the linear-time algorithm. Additional measurements have shown that the cost of the analysis time for the linear-algorithm is now dominated by the cost of just traversing the intermediate representation: for the lexgen example, this cost accounted for up to 198 ms out of the total 368 ms for the benchmark, and for life it was 65 ms out of 83 ms.

11 Conclusion

We have introduced the notion of a sub-transitive control-flow graph: this is a graph whose transitive closure represents control-flow information. The key advantage of this graph is that we can develop $O(n)$ algorithms (where n measures the size of the graph) for many control-flow consuming applications.

Our main result is a linear-time algorithm for bounded-type programs that builds a sub-transitive control-flow graph whose transitive closure gives exactly the results of the standard (cubic-time) CFA algorithm. Our algorithm can be used to list all functions calls from all call sites in (optimal) quadratic time. However, we argue that the "all calls from all call-sites" view of control-flow analysis is unsuitable for investigating the complexity of analyses. In particular, we show that by directly using the sub-transitive graph (instead of using the quadratic sized "all calls from all call-sites" representation), we can develop linear-time algorithms for many CFA-consuming applications. Examples include effects analysis and k-limited CFA. We leave the question of the generality of this approach to future work. In particular, are there "natural" CFA-consuming applications that require the entire "all calls from all call-sites" information, or can we adapt our techniques to all CFA-consuming applications?

We note that linear-time algorithms for *other* forms of control-flow analysis have previously been proposed. In effect, these algorithms replace containment by unification in the definition of control-flow information, and as a result compute information that is strictly less accurate that standard CFA. Our paper shows that this loss of information is not necessary to obtain linear-time algorithms.

benchmark parameter	prog. size (lines)	SBA work units	SBA time (sec)	new alg. ("all" sites) build	new alg. ("all" sites) close	new alg. ("all" sites) querying
10	42	1594	0.028	0.008	0.001	0.000
20	82	9964	0.100	0.016	0.001	0.001
40	162	71104	0.422	0.029	0.002	0.003
80	322	538984	3.494	0.082	0.003	0.012
160	642	4201144	27.115	0.184	0.006	0.062

Table 1:

benchmark	prog. size (lines)	SBA (time)	linear-time algorithm total time	build time	build nodes	close time	close nodes
life	150	0.201	0.083	0.069	1429	0.013	564
lexgen	1180	1.090	0.368	0.217	3624	0.150	2651

Table 2:

Our algorithm could potentially be combined with the standard cubic-time CFA algorithm to obtain a hybrid algorithm that terminates for arbitrary programs but is linear for bounded-type programs. Another area of future work involves extending the analysis to make use of evaluation-order information.

References

[1] A. Aho, J. Hopcroft, and J. Ullmann, "Time and tape complexity of pushdown automaton languages", *Information and Control*, vol. 13, no. 3, pp. 186-206, 1968.

[2] A. Bondorf and J. Jorgensen, "Efficient analysis for realistic off-line partial evaluation", *Journal of Functional Programming*, Vol. 3, No. 3, July 1993.

[3] S. Debray and T. Proebsting, "Interprocedural Control Flow Analysis of First Order Programs with Tail Call Optimization", draft, May 1996. (http://www.cs.arizona.edu/people/debray/papers/cfa.ps)

[4] N. Heintze, "Set-Based Analysis of ML Programs", *ACM Conference on Lisp and Functional Programming*, pp 306–317, 1994.

[5] N. Heintze, "Control-Flow Analysis and Type Systems", *Static Analysis Symposium*, 1995, pp 189–206.

[6] N. Heintze and D. McAllester, "On the Cubic-Bottleneck of Subtyping and Flow Analysis" *IEEE Symposium on Logic in Computer Science*, 1997, to appear.

[7] F. Henglein, "Type Inference with Polymorphic Recursion", *Transactions on Programming Languages and Systems*, Vol. 15, No. 2, pp. 253-289, 1993.

[8] N. Jones, "Flow Analysis of Lambda Expressions", *Symp. on Functional Languages and Computer Architecture*, pp. 66-74, 1981.

[9] N. Jones, "Flow Analysis of Lazy Higher-Order Functional Programs", in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, 1987.

[10] D. McAllester, "Inferring Recursive Data Types", draft manuscript, July 1996.

[11] C. Mossin, "Control Flow Analysis for Higher-Order Typed Programs", draft Ph.D. thesis, DIKU, University of Copenhagen, December, 1996.

[12] E. Melski and T. Reps, "Interconvertibility of Set Constraints and Context-Free Language Reachability", PEPM'97, to appear.

[13] R. Milner, M. Tofte and R. Harper, "The Definition of Standard ML", MIT Press, 1990.

[14] R. Neal, "The computational complexity of taxonomic inference", unpublished manuscript, 18 pages, 1989, (ftp://ftp.cs.utoronto.ca/pub/radford/taxc.ps.Z).

[15] J. Palsberg and P. O'Keefe, "A Type System Equivalent to Flow Analysis", *POPL-95*, pp. 367–378, 1995.

[16] J. Palsberg and M. Schwartzbach, "Safety Analysis versus Type Inference" *Information and Computation*, Vol. 118, No. 1, pp. 128-141, April 1995.

[17] O. Shivers, "Control Flow Analysis in Scheme", *Proc. 1988 ACM Conf. on Programming Language Design and Implementation*, Atlanta, pp. 164–174, June 1988.