Control-Flow Analysis for ML in Linear Time

Nevin Heintze* David McAllester†

November 8, 1996

Abstract

Standard control-flow analysis (often called 0-CFA) is thought to be a cubic time problem — the standard algorithm for it is cubic time and the conventional wisdom is that this algorithm cannot be improved because of its strong connections with dynamic transitive closure. This algorithm is even cubic if we restrict attention to ML programs whose types have bounded size.

We present a new algorithm for standard control-flow analysis that has linear time complexity for ML programs whose types have bounded size. This 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.

(This paper contains 4876 words (according to delatex).)

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 [7, 8, 14, 13]. (In fact some form of CFA is used in most forms of analyses for higher-order languages.)

*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.

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 (i.e. is it polyvariant[1]) or does it fold all activations of a function together (i.e. is it monovariant[2])?
2. *the treatment of dead-code:* does the analysis take into account which pieces of a program can actually be called?
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 [12, 5].

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 significantly less accurate algorithms. For example, [2] employs 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.

In practice, the standard CFA algorithm (and its derivatives) rarely exhibits cubic behavior – it often runs in close to linear time (e.g. see [4]). One way to try to reconcile this difference between observed and theoretic complexity is to try to codify the structural properties of "real programs". For example, in the context of ML programs, we could restrict attention to programs with bounded type size, or equivalently, to programs of bounded order and arity. (Note that arity is defined so that currying increases argument

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

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

count rather than order; for example, the usual curried version of map has type $(\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$ and is considered to have arity 2 and order 2). The rational for this approach is that people rarely write functions with more than, say, 20 arguments or of order greater than 3, and almost never at the same time (this is less clear for automatically generated programs). Such program classes have been particularly useful for understanding the observed linear behavior of type inference for ML [10]; see for example [6, 9]. Unfortunately, for the standard CFA algorithm, this approach fails: the algorithm is cubic even if we choose some constant k and consider only programs of k-bounded type size. (A program is said to have k-bounded type size if the size of the type tree of each program node is k or less.) Section 5 presents a parameterized example that illustrates this fact.

In this paper we present a new algorithm for standard CFA that has linear time complexity for bounded type ML programs. This algorithm is very simple. It is also easy to implement in an incremental, demand-driven and polyvariant manner. Moreover, its data structures can be adapted to provide linear-time algorithms for analyses that piggy-back on control-flow analysis, such as effect analysis.

Section 2 gives the definition of the standard control-flow problem, and addresses the question of complexity measures for control-flow analysis. There are a number of definitions and we discuss the choices; our claim of a linear-time algorithm is based on a definition that is particularly appropriate in the setting of compiler optimization. In Section 3, we present the linear-time algorithm. The main insights of the algorithm are (1) a decoupling 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), (2) the use of a new closure principle to provide appropriate flow of information from one expression to another and (3) the use of types to bound the number of edges added by this new closure principle. In Section 4, we extend this algorithm to records and recursive datatypes. In Section 5, 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 most important, the overhead of the new algorithm (i.e. the "size of the constant") appears to be small.

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[3]:

[3]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.

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

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[4] 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 labels 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) ....
```

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

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 significantly impact on analysis running time and space.

Note that the standard CFA algorithm computes the label sets for each program expressions 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?". To summarize, we can pose at least four general 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.

We argue the last question is the least important of the four. Perhaps the most important questions are those built on top of control-flow information (such as effect analysis). We conclude this section by comparing the complexities of the four questions for the standard algorithm and our new algorithm.

Problem	Standard Algorithm	New Algorithm
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 Linear-Time 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:

$$\frac{}{\lambda^l x.e \longrightarrow \lambda^l x.e} \qquad \text{(ABS)}$$

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

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

$$\frac{e_1 \longrightarrow e_2 \qquad e_2 \longrightarrow e_3}{e_1 \longrightarrow e_3} \qquad \text{(TRANS)}$$

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 re-defined 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 the program $(\lambda^l x.(x\ x)\ (\lambda^{l'} x'.x'))$. Applying the above rules, we obtain the following:

$$\begin{array}{rll}
\lambda^l x.(x\ x) \longrightarrow \lambda^l x.(x\ x) & & \text{by ABS} \\
\lambda^{l'} x'.x' \longrightarrow \lambda^{l'} x'.x' & & \text{by ABS} \\
x \longrightarrow \lambda^{l'} x'.x' & & \text{by APP-1} \\
(\lambda^l x.(x\ x)\ \lambda^{l'} x'.x') \longrightarrow (x\ x) & & \text{by APP-2} \\
x' \longrightarrow \lambda^{l'} x'.x' & & \text{by APP-1} \\
(x\ x) \longrightarrow x' & & \text{by APP-2} \\
(\lambda^l x.(x\ x)\ \lambda^{l'} x'.x') \longrightarrow x' & & \text{by TRANS} \\
(x\ x) \longrightarrow \lambda^{l'} x'.x' & & \text{by TRANS} \\
(\lambda^l x.(x\ x)\ \lambda^{l'} x'.x') \longrightarrow \lambda^{l'} x'.x' & & \text{by TRANS}
\end{array}$$

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:

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

Intuitively, $dom(n)$ represents the "domain" of the node n, and $ran(n)$ represents the "range" of the node n. Now, define a transition system between nodes n as follows:

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

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

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

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

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

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

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:

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

and applying the last rule leads to (among others):

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

Combining these transitions, we can derive $\lambda^{l'} x'.x'$ from $(\lambda^l x.(x\ x)\ (\lambda^{l'} x'.x'))$ as follows:

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

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. The problem is that the edge-adding phase does not terminate in general: given an edge $n \rightarrow n'$, we can add $ran^k(n) \rightarrow ran^k(n')$ for all $k \geq 1$.

However, in the case of simply typed programs, we can bound the number of edges that need to be added. In essence, we 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 $(\tau_1 \rightarrow \tau_2) \rightarrow \tau_3 \rightarrow \tau_4$, 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. This idea can be extended to polymorphically typed ML programs (we can extend our expression language with an appropriate let construct) by considering the collection of monotypes induced for each program expression.

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 added rules. (Strictly speaking, the bound here is on principal types; however given k, we can find a constant k' such that for all k-bounded programs, the sum of the distinct tree nodes in all of the monotypes for a program expression is each bounded in size by k'.)

To take advantage of this property, we modify the rules CLOSE-RAN and CLOSE-DOM so that they are demand driven. For example, CLOSE-DOM is modified as follows

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

so that it is only applied if there is a transition whose left-hand-side could immediately match with the right-hand-side of the added transition. Call this new system LC'. Importantly 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'.*

Now, fix on some constant k, and define ML_k to be the class of ML programs whose whose types are bounded by k. In system LC' we have:

Proposition 3 *There exists a constant c such that when LC' is applied to and program P in ML_k, LC' constructs at most $c.|P|$ edges where $|P|$ denotes the size of program P.*

Hence, we obtain the following linear-time algorithms:

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 nodes that must be added during the analysis. Specifically, Proposition 3 bounds the number of nodes by $c.|P|$ where c is the based on the maximum size type trees that can appear in the program. This could be very large. However this bound is rather loose and can be significantly improved. For example, for simply typed programs, we can bound the number of nodes needed by the sum of the sizes of the type trees at each node. In other words, we can bound the number of new nodes by $c_{ave}.|P|$, where c_{ave} is the average size of the type tree at each node. This constant could still be large. 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.

4 CFA for ML

To date, we have described our CFA algorithm for a simple version of the lambda calculus (with just abstraction and application). While this is sufficient for defining the core ideas of our algorithm, we must argue that these ideas can be extended to other language features without changing the linear-time complexity of the algorithm.

First consider fix: our algorithm thus far has been defined on simply typed lambda terms and so there is no recursion! To address this, consider adding a construct *letrec* $f = \lambda^l x.e_1$ *in* e_2. It is simple to extend the linear-time algorithm for this construct: for each instance of this construct, we add transitions

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

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 $proj_j((e_1, \ldots, e_n)) \longrightarrow e_j$, $j = 1..n$, and each program expression $proj_j(e)$ is treated as a node. Note that the bounded type assumption can still be applied to bound the nodes that need to be considered, and the extended algorithm is still linear-time.

Next consider recursive data types. On 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 (such as SBA [4]) 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 might consider adopting the same idea here. For example consider *cons*, *car* and *cdr*. As for records, we could introduce node operators *car* and *cdr* (with their own closure rules, similar to CLOSE-RAN), and for an expression $cons(e_1, e_2)$, write $car(cons(e_1, e_2)) \longrightarrow e_1$ and $cdr(cons(e_1, e_2)) \longrightarrow e_2$. Unfortunately, we have no way of bounding the size of the nodes we must consider. In fact, we can linearly embed the monadic monotone closure problem[5] in this analysis problem, and [11] has show there is an *nlogn* embedding of the acceptability problem for 2-way nondeterministic pushdown automata (2NPDA) into the monotone closure problem. Hence this analysis problem 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 [1] in 1968.

As an alternative, our implementation uses the type structure of recursive types to obtain a relatively accurate treatment of data structures. To illustrate this, consider again *cons*, *car* and *cdr*. The main change from what was outlined previously, is that when we encounter an expression $cons(e_1, e_2)$, we construct a new node x, and transitions:

$$\begin{aligned} x &\longrightarrow cons(e_1, e_2) \\ x &\longrightarrow e_2 \\ car(cons(e_1, e_2)) &\longrightarrow e_1 \\ cdr(cons(e_1, e_2)) &\longrightarrow x \end{aligned}$$

Importantly, we rewrite any node $cdr(cdr(e))$ into $cdr(e)$. In effect, we use the node x to capture all of the expressions obtainable by *cdr*'ing down the list. In principle, this is less accurate than, for example, SBA; however in practice it yields very similar results. This is because many functions over recursive data structures recurse over all of the cells (of the appropriate type) in the data structure (e.g. most list functions are applied to all – or almost all – of the cons cells in any argument to which they are applied, at least so far as the analysis can determine).

[5]Monadic monotone closure is a generalization of transitive closure that includes two (or more) monotone functions f_1 and f_2 such that if n is a node, then $f_i(n)$ is a node, and if $n \rightarrow n'$ then $f_i(n) \rightarrow f_i(n')$. Given a set of initial edges between nodes, and two nodes n and n' that appear in this set, does $n \rightarrow n'$ follow from the standard transitive closure rule and the additional f_i closure rules?

5 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 optimizes some applications of known functions. However there are many other additional improvements that 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 [4], run in monovariant mode (SBA is a generalization of the standard CFA algorithm). All results are for a 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. those applications whose operator is not immediately apparent from the program text).

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 make runtime measures somewhat unreliable). The linear-time behavior of the new algorithm is also clear.

benchmark parameter	prog. size (lines)	SBA work units	SBA time (sec)	linear-time algorithm
10	42	1594	0.028	0.009
20	82	9964	0.100	0.017
40	162	71104	0.422	0.032
80	322	538984	3.494	0.086
160	642	4201144	27.115	0.195

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.

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

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 198ms out of the total 368ms for the benchmark, and for life it was 65ms out of 83ms.

6 Conclusion

We have presented the first linear-time algorithm for standard control-flow analysis (0-CFA) of bounded-type programs. The previous best algorithm for this problem has cubic time complexity[6]. We have also presented experimental evidence to suggest that this is a good algorithm in practice.

Many analyses piggy-back on top of CFA, and one of the strengths of our algorithm is

[6]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 0-CFA. Our paper, in effect, shows that this considerable loss of information is not necessary to obtain a linear-time algorithm for control-flow.

that we can very directly lift it to these problems. For example, consider effect analysis: we want to determine whether an expression will have a side-effect. We can directly use the graph structure computed by the linear-time algorithm as follows. We first add side-effect edges between each function body and its subexpressions. We then annotate the know side effecting expressions (for simplicity, suppose that all side-effect primitives are fully applied). Finally, we back-propagate these annotations along the edges in the graph. Assuming function size is bounded, this can be done in linear-time. Alternatively, we could formulate this as a demand driven analysis, in which case it becomes a graph reachability problem.

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

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

[8] 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.

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

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

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

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

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

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