# Control Flow Analysis with SAT Solvers

Steven Lyde, Matthew Might

University of Utah, Salt Lake City, Utah, USA

**Abstract.** Control flow analyses statically determine the control flow of programs. This is a nontrivial problem for higher-order programming languages. This work attempts to leverage the power of SAT solvers to answer questions regarding control flow. A brief overview of a traditional control flow analysis is presented. Then an encoding is given which has the property that any satisfying assignment will give a conservative approximation of the true control flow, along with additional ideas to improve the precision and efficiency of the encoding. The results of the encodings are then compared to those of a traditional implementation on several example programs. This approach is competitive in some instances with hand-optimized implementations. Finally, the paper concludes with a discussion of the implications of these results and work that can build upon them.

## 1 Introduction

A control flow analysis determines the control flow of a program. This is a difficult problem in higher order languages, because data flow affects control flow and control flow affects data flow. To address this issue, much work has been done. The first major effort was $k$-CFA as created by Shivers [8]. It is a family of algorithms where the chosen value of $k$ determines the precision of the analysis. A higher value of $k$ gives greater precision but at the cost of a greater run time. When $k = 0$, the algorithm, more commonly known as 0CFA, has been shown to be PTIME complete [9]. For $k \geq 1$, it has been shown that the algorithm is complete for EXPTIME [10].

We present an alternative approach to the problem by encoding a control flow analysis into SAT. The results are more similar to 0CFA than $k$-CFA as SAT is a NP-hard problem, while $k$-CFA is EXPTIME-hard. Similar work that took the idea of encoding $k$-CFA into another problem for performance reasons was done by Prabhu et al. [7]. The algorithm is ported to run on a GPU by encoding the problem into matrix operations. Another work that will feel similar is constraint based 0CFA analysis as summarized by Nielson [6]. They formulate 0CFA using constraints on sets and then provide an algorithm for solving these constraints. This work differs in that the constraints are not not encoded using matrices or sets, but propositional logic.

### 1.1 Motivation

Many problems are readily encoded into SAT problems and even though the problem is NP-complete, fast implementations are available. Control flow anal-

ysis is not trivial, and being able to leverage the effort that placed every year into improving SAT solvers would be a great benefit.

### 1.2 Accomplishments

This work attempts to leverage the power of SAT solvers to answer questions regarding control flow. It presents an encoding and compares its results to two traditional 0CFA implementations.

## 2 Preliminaries

In order to understand this work, you will need a passing understanding of continuation-passing style (CPS) lambda calculus and $k$-CFA. Brief descriptions of both will be given. The original formulation of $k$-CFA operates on CPS lambda calculus and this work also operates on the same language.

CPS is similar to the untyped lambda calculus but with additional constraints: functions never return, all calls are tail calls; where a function would normally return, the current continuation is invoked on the return value; and when calling a function, the caller must supply a continuation procedure. There are three types of terms: applications, anonymous functions, and variables. The grammar for CPS lambda calculus follows.

$$
\begin{aligned}
\mathit{call} \in \mathsf{Call} &::= (f\ e\ \ldots) \\
f, e \in \mathsf{Exp} &= \mathsf{Var} + \mathsf{Lam} \\
v &\in \mathsf{Var} \text{ is a set of identifiers} \\
\mathit{lam} \in \mathsf{Lam} &::= (\lambda\ (v\ \ldots)\ \mathit{call})
\end{aligned}
$$

The abstract state space and the abstract semantics of $k$-CFA reformulated as an operational semantics are easily accessible [3]. The basic idea is to take a CES machine and abstract it by making the number of addresses finite. Successor states are then generated, starting at the initial state of the program, until all the states have been visited. Because the number of addresses is finite the abstract state space is finite and the exploration will terminate.

## 3 Encodings

This section describes the devised encoding scheme. Here is a simple program we will work with in describing the encodings. In the following explanation, each lambda term will be identified by its line number.

```
((lambda (x)
   ((lambda (y)
      (y (lambda (z) (x z)))) x))
 (lambda (a) (a a)))
```

For the encoding, we introduce a variable for every variable lambda pair in the program. The variable will be true if the lambda flows to the variable, and false if it doesn't. We will assume that the program has been alphatised, meaning that each variable is only bound by a single lambda. In the example we have four variables and four lambda terms, resulting in sixteen variables. Lambdas use their line number as their subscript.

| | $\lambda_1$ | $\lambda_2$ | $\lambda_3$ | $\lambda_4$ |
|---|---|---|---|---|
| $a$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
| $x$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ |
| $y$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ |
| $z$ | $z_1$ | $z_2$ | $z_3$ | $z_4$ |

To generate the clauses of our encoding we look at each point where binding occurs in lambda calculus, at application sites. From the grammar of CPS lambda calculus we can see that there are four cases which need to be considered. The function and the arguments at an application can either be a lambda term or a variable.

**Case 1: Lambda Lambda** The first case to consider is the simplest, when there is a lambda term in both function and argument position. The top level application of the sample program is an example of this.

```
((lambda (x) call) (lambda (a) (a a)))
```

We know that the lambda in argument position flows to the parameter of the lambda in function position. For this call site, we would add the clause $x_4$.

**Case 2: Lambda Variable** The second case to consider is when there is still a lambda in function position but a variable in argument position. Observe the following call site from the example.

```
((lambda (y) call) x)
```

If we know a lambda flows to $x$, then we know that it must flow to $y$. We must assume that any lambda can flow to x, so we must create a clause for each lambda. This results in the following clauses: $x_1 \rightarrow y_1$, $x_2 \rightarrow y_2$, $x_3 \rightarrow y_3$, $x_4 \rightarrow y_4$.

**Case 3: Variable Lambda** The third case to consider is having a variable in function position and a lambda term in argument position.

```
(y (lambda (z) call))
```

We must assume that any variable can flow to $y$. Thus we need to create a clause for each lambda in the program. We infer that if a lambda term flows to $y$, then $\lambda_4$ will flow to the parameter of that lambda. This results in the following clauses: $y_1 \rightarrow x_3$, $y_2 \rightarrow y_3$, $y_3 \rightarrow z_3$, $y_4 \rightarrow a_3$.

**Case 4: Variable Variable** The most complicated case is when we have a variable in both function and argument position.

```
(x z)
```

We must assume that any lambda can flow to $x$ and any lambda can flow to $z$. If we know that two flows are true for $x$ and $z$, we can infer a third flow. For example, if we know $\lambda_2$ flows to $x$ and $\lambda_4$ flows to $z$, we can infer that $\lambda_4$ flows to $y$, the parameter of $\lambda_2$. Thus we create the clause $x_2 \wedge z_4 \rightarrow y_4$. Since there are four lambda terms, there are 16 total such clauses that need to be generated.

## 4 Additional Encoding Details

The generated clauses described above are necessary but not sufficient. The problem is that every variable can be set to true and the equation is still satisfied. What we really want is the lowest possible number of flows set to true that still satisfy all the generated clauses. However, the SAT solver is free to give any satisfying solution. In the end, we have constraints that will never give us false negatives, but we need constraints that will ideally never give us false positives, or at least limit them.

### 4.1 Additional Encodings

For each case we will show additional clauses that can be added which will limit the number of false positives.

**Case 1: Lambda Lambda** Since the program is alphatised we not only know that the given flow must be true, but we know that all other flows to that variable must be false. For the above example we add the clauses: $\neg x_1$, $\neg x_2$, $\neg x_3$.

**Case 2: Lambda Variable** In the description found above, we said you could infer an additional flow if a given lambda flows to the variable in argument position. But more can be inferred since the program is alphatised. The clauses are not just implications because the call site is the only place where the binding of the variable can occur. Thus we can change the clauses to equivalences: $x_1 \leftrightarrow y_1$, $x_2 \leftrightarrow y_2$, $x_3 \leftrightarrow y_3$, $x_4 \leftrightarrow y_4$.

**Case 3: Variable Lambda**

Unlike the previous case, we cannot turn the inference described in the previous section for case 3 into an equivalence. The issue is that because the lambda which flows to the variable in function position can flow to other application sites where there is a variable in function position, this is not the only place where a binding can occur. However, we can infer the disjoin of all the call sites where the binding could occur. An example will be be given below.

**Case 4: Variable Variable**

Much like the previous case, we cannot infer equivalences because bindings can happen at any call site where there is variable in function position. However, like the above case, additional clauses can still be created; we can infer the disjoin of all the call sites where the binding could occur. For example, if $\lambda_3$ flows $z$ it would mean that either $\lambda_3$ flows to $y$, $\lambda_3$ flows to $a$, or that $\lambda_3$ flows to $x$ and $\lambda_3$ flows to $z$. Thus we would add the following clause: $z_3 \rightarrow y_3 \vee a_3 \vee (x_3 \wedge z_3)$.

### 4.2 Enhancements

The encodings presented above give way to some enhancements that can be used to make the encoding more efficient.

- Not all lambdas can flow. Lambdas that appear in function position cannot be bound to variables, thus we do not need to create a variable for pairs involving lambdas in function position.
- Not all lambdas are compatible. Although the example shows lambda terms with only one parameter, the lambda terms can have any number of parameters. When there is a variable in function position, only lambdas with the same number of parameters as there are arguments at the application site need to be considered.
- Some clauses will be trivially true. While iterating through every lambda, when faced with a variable at an application site, some of the implications will involve the same pairs on both sides, thus they are trivially true and can be omitted.

In the implementation, the first two enhancements were used, but the third was omitted.

### 4.3 Complexity

In the described encoding, many clauses can be generated. However, it is bounded by a polynomial of the size of the program. The worst case to consider is when you have a variable in both function and argument position. You must consider each lambda flowing to each variable. If there are $n$ terms in the program, there are at most $n$ call sites and $n$ lambda terms. Thus the number of generated clauses will be bound by $n^3$. This seems logical as one of the simplest formulations of 0CFA is "nearly" cubic: $O(n^3/\log n)$ [2].

## 5 Implementation and Evaluation

We implemented the encoding in Scala using the back end of the analyzer written by Might et al. for parsing and pre-process transformations [5]. We compared its run times to those of that same analyzer, which closely follows the formal semantics, as well as a fast Racket implementation, which employs abstract

Church encodings and binary CPS lambda calculus [7]. MiniSat was used for solving the constructed encodings. All experiments were run on a 2.7 GHz Intel Core i7 on Mac OS X.

The first experiments were run on synthetic programs, which in a *constructive* complexity proof are shown to be the worst case for $k$-CFA when $k \geq 1$ and difficult for 0CFA [9, 10]. The results can be found in the following table. The first column is the number of terms in the program. The second column is the run time of the optimized Racket implementation. The Scala column is the run time of the traditional Scala implementation. The SAT column is the time taken to encode and solve the problem using SAT. This column is broken down into its two components in the last two columns. The Encode column is the time taken to create the encoding. The Solve column is the time taken by MiniSat to solve the encoding.

| **Terms** | **Racket** | **Scala** | **SAT** | **Encode** | **Solve** |
|---|---|---|---|---|---|
| 37 | 0.005s | 0.613s | 0.521s | 0.518s | 0.003s |
| 63 | 0.012s | 0.742s | 0.575s | 0.571s | 0.004s |
| 115 | 0.047s | 1.072s | 757s | 0.751s | 0.006s |
| 219 | 0.260s | 1.751s | 1.087s | 1.065s | 0.022s |
| 427 | 1.483s | 5.034s | 2.724s | 2.567s | 0.157s |
| 843 | 9.265s | 51.411s | 13.153 | 11.872s | 1.281s |
| 1675 | 53.361s | 16m4.607s | 1m34.829 | 1m24.611s | 10.218s |
| 3339 | 5m3.110s | >6h | 13m18.232 | 11m28.179s | 1m50.053s |

From the experiments, we see encoding the problem and solving it with MiniSat takes about the same amount of time as the fast implementation. However, this is not always the case. Experiments were also run on more traditional benchmarks. To run these, the language on which the encoding operates had to be enriched. Additional constructs were added (*e.g.*, `if` and `set!`) as well as support for Scheme primitives. The fast Racket implementation could not be run on these examples, as it only supports pure binary CPS lambda calculus.

| **Program** | **Terms** | **Scala** | **SAT** | **Encode** | **Solve** |
|---|---|---|---|---|---|
| eta | 79 | 0.639s | 0.528s | 0.525s | 0.003s |
| map | 182 | 0.802s | 0.689s | 0.682s | 0.007s |
| sat | 250 | 0.914s | 0.913s | 0.895s | 0.018s |
| rsa | 609 | 1.263s | 1.061s | 1.041s | 0.020s |
| prime | 891 | 1.663s | 7.664s | 4.695s | 2.969s |
| scm2java | 2505 | 3.054s | 1m29.605s | 1m15.181s | 14.424s |
| interp | 4484 | 3.973s | 7m46.352s | 5m29.850s | 2m16.502s |

The first two benchmarks test common functional patterns; sat is a simple SAT solver; rsa is a RSA implementation; prime is a Solovay-Strassen primality tester; scm2java is a Scheme to Java compiler; interp is a Scheme interpreter.

These benchmarks provide a stark contrast to the previous examples in performance. Further investigation is needed to find the source of this large difference in performance. One possible explanation is that the Scheme primitives are

not well modelled. Also, the traditional small step abstract interpreter is able to use widening to converge to the minimum fixed point faster. In addition, since its analysis is directed by the syntax of the program more closely, it can explore less spurious flows.

For the first set of benchmarks, the results returned by the encoding are exactly the same as those provided by the traditional implementations. However, running #SAT on the encodings, revealed that there are multiple valid interpretations. Thus the encoding does not exactly encode traditional 0CFA, which has a unique minimum fixed point.

### 5.1 Alternative Approach Using BDDs

Another approach attempted was to use a binary decision diagram (BDD) instead of a SAT solver to solve the constraints. The constraints are encoded in the same way, but the approach has the benefit that the minimum prime implicant is readily available from the structure of the BDD. The minimum prime implicant provides an equivalent solution as 0CFA. However, in practice, using a BDD requires large amounts of memory and time for even simple examples.

## 6 Conclusion

This work has presented an encoding for control flow analysis of CPS lambda calculus. It has shown that in some cases, the approach can be as fast as a highly optimized solution. While the soundness of the encoding was not proven, empirical results showed it to be accurate.

This work also provides a solid basis for additional work. Many avenues exist which can build upon it. Better encoding schemes can be developed, which possibly could be even more precise than 0CFA, given the extra power provided by SAT solvers being able to solve NP-complete problems. Van Horn and Mairson give a reduction from SAT to $k$-CFA, effectively showing how to do SAT solving with $k > 1$ CFA, which merits further investigation. Also, while this work operates on CPS lambda calculus, the encoding could easily be adapted to work on a more direct style language, such as ANF lambda calculus [1], as analyzed by Might and Prabhu [4].

This work was supported by the DARPA programs APAC and CRASH.

## References

1. Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (New York, NY, USA, June 1993), ACM, pp. 237–247.
2. Midtgaard, J., and Van Horn, D. Subcubic control flow analysis algorithms. Tech. rep., Roskilde Unversitet, 2009.

3. Might, M. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.
4. Might, M., and Prabhu, T. Interprocedural dependence analysis of higher-order programs via stack reachability. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming* (Boston, Massachussetts, USA, 2009).
5. Might, M., Smaragdakis, Y., and Van Horn, D. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2010), PLDI '10, ACM Press, pp. 305–315.
6. Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis*, corrected ed. Springer, Dec. 2004.
7. Prabhu, T., Ramalingam, S., Might, M., and Hall, M. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, Jan. 2011), vol. 38, ACM Press, pp. 511–522.
8. Shivers, O. G. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.
9. Van Horn, D., and Mairson, H. G. Relating complexity and precision in control flow analysis. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2007), ACM, pp. 85–96.
10. Van Horn, D., and Mairson, H. G. Deciding k-CFA is complete for EXPTIME. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (2008), ACM Press, pp. 275–282.