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 runtime. When $k = 0$, the algorithm, more commonly known as 0CFA, has been shown to be cubic [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]. They run the analysis on a GPU by encoding the problem into matrix operations. Another work that will feel similar to the work presented in this paper 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 and even though satisfiability is NP-complete, fast implementations are available. Every year there is consider-

able work being done to create efficient SAT solvers A CFA implementation based on satisfiability could benefit directly from that work.

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 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 idea is to take an abstract 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 label.

```
((lambda¹ (x)
   ((lambda² (y)
      (y (lambda³ (z) (x z)))) x))
 (lambda⁴ (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 label as their subscript.

	λ_1	λ_2	λ_3	λ_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[1] (x) call) (lambda[4] (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[2] (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. Observe the following call site from the example.

```
(y (lambda[3] (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 λ_3 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. Observe the following call site from the example program.

```
(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 λ_2 flows to x and λ_4 flows to z, we can infer that λ_4 flows to y, the parameter of λ_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 formula 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. Note that in an anlysis, having false positives is still sound; only in having false negatives does the analysis become unsound.

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 λ_3 flows to z it would mean that either λ_3 flows to y, λ_3 flows to a, or that λ_3 flows to x and λ_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, by generating less clauses.

- 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 preprocess transformations [5]. We compared its runtimes 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 Ubuntu.

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 Table 1. The first column is the number of terms in the program. The second column is the runtime of the optimized Racket implementation. The Scala column is the runtime 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.

Table 1. Runtime comparison of a control-flow analysis using a fast Racket implementation, a Scala implementation and using MiniSAT.

Terms	**Racket**	**Scala**	**SAT**	**Encode**	**Solve**
37	0.008s	1.059s	0.730s	0.725s	0.005s
63	0.016s	1.056s	0.796s	0.792s	0.004s
115	0.046s	1.454s	1.025s	1.017s	0.008s
219	0.222s	2.338s	1.418s	1.387s	0.031s
427	1.374s	5.337s	2.759s	2.642s	0.117s
843	8.396s	44.873s	11.337s	10.481s	0.856s
1675	49.029s	12m34.301s	1m15.984s	1m9.222s	6.762s
3339	4m46.726s	>6h	8m50.671s	8m43.103s	7.568s

We also looked into the sensitivity of the encoding to different SAT solvers, using SAT solvers that were some of the best performers from the 2011 international SAT competition. See Table 2. We report the time taken to solve the encoding, the number of flows that agree with the Scala implementation and the number of flows that disagree. When there is a disagreement, the encoding says that the flow does occur but the traditional 0CFA reports that it does not.

From the results in Table 1, we see encoding the problem and solving it with MiniSat takes about the same amount of time as the fast Racket 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 without using Church encodings, as it only supports pure binary CPS lambda calculus. See Table 3.

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.

Table 2. Runtime and precision results from some of the best performers from the 2011 international SAT competitions.

Solver	Results	$n = 37$	$n = 63$	$n = 155$	$n = 219$	$n = 237$	$n = 843$	$n = 1675$
minisat	Time	0.005s	0.007s	0.012s	0.039s	0.133s	0.848s	6.714s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
3S	Time	2.548s	2.570s	2.554s	2.777s	5.952s	1m15.335s	>2m
	Agree	96	280	936	3400	12936	50440	-
	Disagree	0	0	0	0	0	0	-
cirminisat	Time	0.004s	0.005s	0.009s	0.031s	0.152s	1.312s	11.299s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
clasp	Time	0.004s	0.005s	0.010s	0.029s	0.152s	1.055s	7.959s
	Agree	54	150	486	1734	6534	25350	165378
	Disagree	42	130	450	1666	6402	25090	33798
cryptominisat //	Time	0.007s	0.011s	0.026s	0.086s	0.413s	3.598s	>2m
	Agree	96	280	936	3400	12936	50440	-
	Disagree	0	0	0	0	0	0	-
csls //	Time	0.006s	0.006s	0.034s	0.579s	32.656s	>2m	>2m
	Agree	60	216	711	2754	12936	-	-
	Disagree	36	64	225	646	0	-	-
eagleup	Time	0.004s	0.006s	0.017s	0.063s	0.541s	18.500s	>2m
	Agree	70	192	674	2566	9479	36639	-
	Disagree	26	88	262	834	3457	13801	-
glucose	Time	0.011s	0.012s	0.020s	0.050s	0.198s	1.415s	4.805s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
glueminisat	Time	0.004s	0.005s	0.011s	0.036s	0.164s	1.363s	11.640s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
lingeling	Time	0.006s	0.010s	0.024s	0.078s	0.454s	1.980s	10.365s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
march_rw	Time	0.007s	0.010s	0.033s	0.466s	18.936s	>2m	>2m
	Agree	54	150	486	1734	6534	-	-
	Disagree	42	130	450	1666	6402	-	-
plingeling	Time	0.009s	0.012s	0.028s	0.096s	0.477s	2.851s	19.695s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
plingeling //	Time	0.010s	0.017s	0.037s	0.083s	0.465s	3.551s	30.653s
	Agree	96	280	574	1992	7813	30463	120112
	Disagree	0	0	362	1408	5123	19977	79064
ppfolio	Time	0.006s	0.009s	0.010s	0.051s	0.341s	2.453s	14.588s
	Agree	78	280	936	3400	12936	50440	165378
	Disagree	18	0	0	0	0	0	33798
ppfolio //	Time	0.007s	0.007s	0.012s	0.028s	0.178s	0.989s	7.409s
	Agree	92	280	936	3400	12936	50440	165378
	Disagree	4	0	0	0	0	0	33798
qutersat	Time	0.035s	0.042s	0.061s	0.134s	0.638s	4.786s	41.886s
	Agree	96	280	936	3400	12936	50440	199176
	Disagree	0	0	0	0	0	0	0
sattime2011	Time	0.005s	0.008s	0.021s	0.093s	0.796s	10.857s	0.020s
	Agree	83	230	805	2919	11275	44001	-
	Disagree	13	50	131	481	1661	6439	-
sparrow2011	Time	0.013s	0.007s	0.029s	0.100s	3.034s	>2m	>2m
	Agree	72	266	936	3400	10846	-	-
	Disagree	24	14	0	0	2090	-	-

Table 3. Runtime comparison between a traditional abstract interpreter and determining the control-flow usining MiniSAT.

Program	Terms	Scala	SAT	Encode	Solve
eta	79	0.879s	0.805s	0.801s	0.004s
map	182	0.879s	0.805s	0.801s	0.004s
sat	250	1.311s	1.216s	1.198s	0.018s
rsa	609	1.805s	1.427s	1.396s	0.031s
prime	891	2.258s	4.584s	4.269s	0.315s
scm2java	2505	3.845s	1m6.550s	1m0.090s	6.460s
interp	4484	6.314s	5m6.519s	4m26.078s	40.441s

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.

5.2 Alternative Approach Using MaxSAT

Another approach that could be promising is to use a MaxSAT solver instead of a traditional SAT solver. The additional clauses from Section 4 could be elided and only the clauses from Section 3 would be needed. The partial maximum satisfiability problem has two types of clauses, hard and soft. The hard clauses must be satisfied, while the soft clauses can be relaxed. The solver finds the assignment with maximum number of soft clauses satisfied. All the clauses from Section 3 would be hard clauses and then for each variable, its negation would be added as a soft clause. A satisfying assignment from this formulation would be equivalent to 0CFA.

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 [9]. 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.