Control-Flow Analysis of Dynamic Languages via Pointer Analysis

Steven Lyde William E. Byrd Matthew Might
University of Utah
{lyde,webyrd,might}@cs.utah.edu

Abstract

We demonstrate how to map a control-flow analysis for a higher-order language (dynamic languages are typically higher-order) into a pointer analysis for a first-order language, such as C. This allows us to use existing pointer analysis tools to perform a control-flow analysis, exploiting their technical advancements and the engineering effort that went into developing them. We compare the results of two recent parallel pointer analysis tools with a parallel control-flow analysis tool. While it has been known that a control-flow analysis of higher-order languages and a pointer analysis of first-order languages are very similar, we demonstrate that these two analyses are actually more similar than previously thought. We present the first mapping between a high-order control-flow analysis and a pointer analysis.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors—Optimization

General Terms Languages, Theory

Keywords program analysis, high-order languages, pointer analysis

1. Introduction

In dynamic languages, because they are typically higher-order, it is not always clear from the syntax which functions are being called at which call sites. Take for example the simple Racket program that implements Turner's tautology checker and invokes it two times [6]. The details of the example are not important, just the illustration it gives. The function `taut` takes a function `f` that represents a boolean expression. The function `taut` then checks if all possible assignments of either `#t` or `#f` to the parameters of `f` result in `f` evaluating to `#t`.

```
(define (taut f n)
  (if (= n 0) f
      (and (taut (f #t) (- n 1))
           (taut (f #f) (- n 1)))))

(define (g x)
  (lambda (y)
    (or (and x (not y))
        (or (not x) y))))

(define (h z) z)

(taut g 2)

(taut h 1)
```

Because functions are passed around as arguments, it might not be immediately clear which functions can flow to `f` and thus what functions we are invoking when we call `f`. There are two call sites which recursively invoke `taut`, but the argument is the result of invoking `f` itself. In order to answer the question of which functions can flow to `f`, we need a control-flow analysis. A higher-order control-flow analysis conservatively bounds the set of functions for a higher-order call site.

Imagine we have an analysis or compiler optimization that needs a control-flow analysis, but we don't have a control-flow analysis immediately available to us. Our analysis might need to know the control flow of the program in order to prove some security properties or prove the absence of errors. Our compiler optimization might need to know the control flow of the program to speed up the program by inlining functions.

There are a large number of tools that can perform a related but different analysis: pointer analysis. A pointer analysis tells us which objects variables point to in a program. Wouldn't it be nice if we could just use these tools? We could spend a significant amount of time developing a new modern control-flow analysis tool, pulling in the latest features

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm...$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

from these similar but different tools, or we could find ways to reuse these existing tools to solve the problem we have at hand. Using existing tools saves us the effort of developing our own tool and allows us to rely on mature and well tested tools.

1.1 Overview

Precise control-flow analysis is expensive. For example, 0CFA as described by Palsberg, which is the analysis we explore in this paper, is cubic [11]. This work was initially motivated by trying to identify ways we could speed up control-flow analysis. With the advancement of GPUs being used for general process computing and more cores being available on commodity machines, one way to speed up the analysis is to parallelize it. The only work we know of that parallelizes control-flow analysis of higher-order languages is EigenCFA [13]. We also know of two recent tools that parallelize pointer analysis developed by Méndez-Lojo et al. [8, 9].

One option we considered to speed up control-flow analysis was to deeply understand these two tools for pointer analysis and port over any ideas that would improve EigenCFA. However, we took the option to use those tools directly. But in order to do this, we needed to map a control-flow analysis into a pointer analysis. In this paper, we demonstrate how to do this. To our knowledge, even though the similarities between these analyses have been known and there has been a general feeling in the community that they are the same, this mapping has not been explicitly laid out [10].

With this paper we plan to show that we can successfully take the state of the art in pointer analysis and improve upon the state of the art in control-flow analysis in terms of performance. Our goal is to leverage existing tools available to us from the pointer analysis community. We wish to exploit the efficiency of these static analysis tools for control-flow analysis of dynamic languages.

The contributions of this paper are as follows.

- We show that there exists a direct mapping between a control-flow analysis of higher-order languages and a pointer analysis for first-order languages. In fact, we show three different mappings, each serving a slightly different purpose. The first mapping, in section 3, is to help us demonstrate the connection between a control-flow analysis and a pointer analysis. The second mapping, in section 5.1, allows us to use the benchmarks that were used by the control-flow analysis tool EigenCFA. The final mapping, in 5.4, makes a different trade-off by containing more inference rules, but results in fewer variables. Because of this mapping, we end up with one of the fastest tools for 0CFA.
- We show in section 4 that the constraints generated by a traditional control-flow analysis are equivalent to the constraints generated by a pointer analysis after going through our mapping. This is important because it means that the answer we get back from a pointer analysis tool will be the same answer we would get from a control-flow analysis tool.
- In section 6, we then demonstrate the benefit of this mapping by comparing two recent parallel pointer analysis tools with a recent parallel control-flow analysis tool called EigenCFA [13]. We compare a control-flow analysis tool that runs on the GPU with a pointer analysis tool that also runs on the GPU [9]. We also compare these tools to one that runs on a single CPU with multiple cores [8]. Both of theses pointer analysis tools were written by Méndez-Lojo et al. For the benchmarks we used, this multithreaded implementation performs the best. We saw that the CPU multicore pointer analysis tool is able to run up to 35 times faster than the GPU control-flow analysis tool. With the mapping of this paper, we beat the fastest known GPU version of CFA with a pointer analysis tool that runs on the CPU.

The implementation details of a static analysis tool matter. With the wide range of work that has been done for pointer analysis, applying these techniques directly to control-flow analysis of dynamic languages is advantageous. Due to the mapping in this paper, for the chosen benchmarks, we now have the fastest way we know of to do higher-order control-flow analysis that has the precision of a 0CFA as formulated by Palsberg [11].

2. Background

In this section, we give a brief description of a traditional pointer analysis and a traditional control-flow analysis. Control-flow analysis of higher-order programs and pointer analysis share much in common with each other and often the pointer analysis and control-flow analysis communities use similar techniques [7]. However, the relationship between the techniques is often obscured by differing terminology and presentation styles.

The brief overview of the two analyses given here illustrates their differences. But the mapping given in the next section illustrates their similarities. By showing that the gap between pointer analysis of first-order languages and control-flow analysis of higher-order languages is even narrower than once thought, this allows for further applications of the large amount of research that has gone into pointer analysis to be applied to control-flow analyses.

2.1 Pointer Analysis

Pointer analysis is one of the most fundamental static analyses with a broad range of applications. Is is used by traditional optimizing compilers and by applications involved with program verification, bug finding, refactoring, and security.

Pointer analyses can change based on the desired precision. There always exists a trade-off between the speed, scalability, and precision of any analysis. There also exists extensions for handling specific language features. In this paper, we will stick to using a very basic pointer analysis, though extended to handle very basic pointer arithmetic in order to handle fields of structures, whose importance will be demonstrated later.

We give a brief overview of a pointer analysis, describing the statements that are supported and the constraints that are generated from those statements.

2.1.1 Pointer Statements

The pointer analysis tools we used were developed to analyze C programs. These tools only consider pointer statements, disregarding the other statements of the program. There are five pointer statements that are supported: assigning the address of a variable, copying a pointer, dereferencing a pointer, assigning to dereferenced pointer, and simple pointer arithmetic.

$$\begin{aligned} x, y \in \mathsf{Var} &::= \text{a finite set of variables} \\ o \in \mathsf{Int} &::= \text{a finite set of integers} \end{aligned}$$

$$\begin{aligned} x &= \&y \\ x &= y \\ x &= *y \\ *x &= y \\ x &= y + o \end{aligned}$$

The basic pointer arithmetic allows us to handle structures. Some pointer analysis algorithms "collapse" a structure into a single variable, but this comes at the cost of too much precision [16]. Other algorithms treat each field as a separate field based on offset and size. While this is not portable because the memory layout of structures is implementation dependent, the analysis is still correct as long as pointer arithmetic is used strictly for accessing fields of structures, and not used in other parts of the program to access arbitrary parts of memory. This is sufficient for our needs since the only pointer arithmetic used in our encoding is to dereference fields.

2.1.2 Pointer Set Constraints

Now that we know what pointer statements we can handle, we will answer the basic question of how can we figure out which pointers point to what. A pointer analysis is usually formulated as a set-constraint problem. An analysis will iterate over the statements of the program, generating set constraints for each statement. These set constraints define the points-to sets $pts(x)$ for each variable x in the program.

The following constraints are those generated by Andersen in his style of analysis [1]. In these constraints, $loc(v)$ represents the memory location denoted by v.

$$\begin{aligned} x &= \&y & & loc(y) \in pts(x) \\ x &= y & & pts(y) \subseteq pts(x) \\ x &= *y & & \forall v \in pts(y) : pts(v) \subseteq pts(x) \\ *x &= y & & \forall v \in pts(x) : pts(y) \subseteq pts(v) \\ x &= y + o & & \{v + o : v \in pts(y)\} \subseteq pts(x) \end{aligned}$$

All the rules are generally straightforward. For a pointer dereference, we are stating that everything we could possibly point to is also pointed to by the variable we are assigning. For assigning to a pointer dereference, we are stating that everything that we could point to also points to what is pointed to on the right hand side.

A pointer analysis can be flow-sensitive or flow-insensitive. A flow-sensitive analysis takes into account the order of statements in a program. A pointer analysis can also be context-sensitive or context-insensitive. A context-sensitive analysis takes into account the calling context (where the function was called) of the function that contains the statements we are analyzing. In practice, context-sensitivity and flow-sensitivity are too expensive and as such the tools we used in our evaluation are context-insensitive and flow-insensitive.

2.2 Control-Flow Analysis

One way to perform a control-flow analysis is with constraints. In describing the constraints, we will operate over a simple language, the lambda calculus. We will first briefly describe the lambda calculus. Then we will briefly describe how to generate the constraints.

2.2.1 Lambda Calculus

The lambda calculus only has three language forms: variable reference, lambda terms, and function application. This is the core of many functional programming languages, such as Racket, and thus also of dynamic languages.

$$\begin{aligned} e \in \mathsf{Exp} &::= v \mid (\lambda\ (v)\ e_b) \mid (e_1\ e_2) \\ v \in \mathsf{Var} &\text{ is a set of identifiers} \end{aligned}$$

2.2.2 Palsberg Constraints for Solving 0CFA

As stated earlier, one way to solve a control-flow analysis is with constraints. We iterate over the expressions of the program, generating constraints for each expression. These constraints describe a set of lambda terms, $\mathit{flows}[\![e]\!]$, for each subexpression e in our program. These are the constraints that are generated by Palsberg [11]. For expression $e \in \mathsf{Exp}$ under analysis, let E be the set of all expressions in e.

$$\frac{(\lambda\ (v)\ e_b) \in E}{\{(\lambda\ (v)\ e_b)\} \subseteq \mathit{flows}[\![(\lambda\ (v)\ e_b)]\!]}$$

$$\frac{(e_1\ e_2) \in E \qquad (\lambda\ (v)\ e_b) \in \mathit{flows}[\![e_1]\!]}{\mathit{flows}[\![e_2]\!] \subseteq \mathit{flows}[\![v]\!]}$$

$$\frac{(e_1\ e_2) \in E \qquad (\lambda\ (v)\ e_b) \in \mathit{flows}[\![e_1]\!]}{\mathit{flows}[\![e_b]\!] \subseteq \mathit{flows}[\![(e_1\ e_2)]\!]}$$

The first rule states that a lambda term is in its own flow set. The second rules says that at a function application, for every lambda term that flows to the function being applied, the flow set of the formal parameter includes everything that is in the flow set of the argument. The third rule states that also at a function application, for every lambda term that flows to the function being applied, whatever is in the flow set of the body of the lambda term is also in the flow set of the function application.

3. Encoding

Looking at the inference rules for pointer analysis and the inference rules for a control-flow analysis, we can already see the similarities between them. This section further explores these similarities.

This section describes how we can take a lambda calculus expression and encode it into pointer statements that can then be analyzed using a points-to analysis. Taking into account how we map lambda calculus expressions into pointer variables, and being able to reverse this mapping, allows us to use the results of the pointer analysis and convert them into a result for our control-flow analysis.

3.1 Analysis Compilation

Here we describe how to take a program written in lambda calculus and encode it into a C program which will compile and on which you could perform a points-to analysis, but which does not preserve the meaning of the program. The results of a points-to analysis run on this program, given sufficient support for structures, and the results of a control-flow analysis would be equivalent.

This conversion is more for illustrative purposes in order to get a better intuition on how the inference rules given afterwards work.

The ability of the analysis to handle pointer arithmetic, and thus structures precisely, allows us to create a relationship between variables. This allows the correlation that is needed between the variable that represents a given lambda term's parameter and the variable that represents its body.

Create a struct to enable deconstructing a lambda term.

```
struct lambda {
  struct lamdba *var;
  struct lambda *body;
};
```

Create a variable of type **struct** lambda for every lambda term that appears in the lambda calculus program, giving each lambda term a unique variable name.

```
struct lambda lam;
```

Create a variable of type **struct** lambda $*$ for every function application and lambda term that appears in the lambda calculus program. Each expression needs a unique variable name.

```
struct lambda *exp;
```

We do not need to create a pointer for variables because they will be handled by the var pointer inside the structure for the lambda term that binds the variable. We will use this variable whenever we have a variable references that appears in another expression.

For every lambda term in the program, we need to assign the pointer for that lambda term to point to the structure for that same lambda term.

```
exp = &lam;
```

For every call site in the program there are three subexpressions and thus three pointer variables in our translated program: the pointer for the call itself, the pointer for the function, and the pointer for the argument. We need an assignment to state that the variable of the lambda term that is pointed to is bound to point to the same things as the argument. We also need an assignment to state that whatever the body of the lambda term being applied points to is also pointed to by the pointer representing the call site.

```
e1->var = e2;
exp = e1->body;
```

Example

To make the previous transformations more explicit, we will convert the following program which is both a valid lambda calculus expression and valid Racket program.

$$((\lambda\ (x)\ (x\ x))\ (\lambda\ (y)\ (y\ y)))$$

The resulting C code would be the following. The above example has two lambda terms, so we create lamx and lamy and pointers lam1 and lam2. We assign these pointers to point to the structures. There are three call expressions, so we create pointers call1, call2, call3. We create the six assignment statements associated with these calls.

```
struct lambda lamx;
struct lambda lamy;

struct lambda *call1;
struct lambda *call2;
struct lambda *call3;

struct lambda *lam1;
struct lambda *lam2;

lam1 = &lamx;
```

```
lam2 = &lamy;

lam1->var = lam2;
call1 = lam1->body;

lamx.var->var = lamx.var;
call2 = lamx.var->body;

lamy.var->var = lamy.var;
call3 = lamy.var->body;
```

These statements can be converted into the simple pointer statements referenced earlier by using both intermediate variables and pointer arithmetic to access the fields of structures.

3.2 Inference Rules

We will now slightly simplify the above conversion by going directly to the simple pointer statements. The following rules describe how to encode a lambda calculus expression into pointer statements. The function $\mathcal{L} : \mathsf{Exp} \to \mathsf{Var}$ maps expressions to a unique variable. The variables need to be layed out in memory such that for a lambda term $(\lambda\ (v)\ e)$, where $a = \mathcal{L}(v)$ and $b = \mathcal{L}(e)$, $loc(b) = loc(a) + 1$. For references to the same variable in different parts of the program $\mathcal{L}$ will map it to the same variable.

$$\frac{(\lambda\ (v)\ e) \in E \qquad x = \mathcal{L}((\lambda\ (v)\ e)) \qquad y = \mathcal{L}(v)}{x = \&y}$$

$$\frac{(e_1\ e_2) \in E \qquad x = \mathcal{L}(e_1) \qquad y = \mathcal{L}(e_2)}{*x = y}$$

$$\frac{(e_1\ e_2) \in E \qquad x = \mathcal{L}((e_1\ e_2)) \qquad y = \mathcal{L}(e_1)}{x = *y + 1}$$

Some of these forms are not one of the five pointer statements described as being supported by the tools we evaluated. However, it is easy to see how we can construct them using intermediate variables. For example, we can change the single statement $x = *y + 1$ into the two statements $y_p = *y$ and $x = y_p + 1$.

Going back to our earlier example program. Assume we have the following mapping from expressions to variable names.

$$\begin{aligned}
l_1 &= \mathcal{L}((\lambda\ (x)\ (x\ x)))\\
l_2 &= \mathcal{L}((\lambda\ (y)\ (y\ y)))\\
x &= \mathcal{L}(x)\\
y &= \mathcal{L}(y)\\
c_1 &= \mathcal{L}(((\lambda\ (x)\ (x\ x))\ (\lambda\ (y)\ (y\ y))))\\
c_2 &= \mathcal{L}((x\ x))\\
c_3 &= \mathcal{L}((y\ y))
\end{aligned}$$

We would then generate the following pointer statements.

$$\begin{aligned}
l_1 &= \&x\\
l_2 &= \&y\\
*l_1 &= l_2\\
c_1 &= *l_1 + 1\\
*x &= x\\
c_2 &= *x + 1\\
*y &= y\\
c_3 &= *y + 1
\end{aligned}$$

4. Equivalence of Constraints

Recall that we are trying take a lambda calculus expression, convert it into pointer statements, run a pointer analysis on these statements, and then use those results as a solution to a control-flow analysis of our original lambda calculus expression. We will now go through these steps and demonstrate that the solution generated is equivalent if we were to use the original constraint based formulation of Palsberg.

In the Palsberg constraints, there are three inference rules. We will examine each of these rules. We will take the lambda calculus expression and convert it into the equivalent pointer statements. We will then generate the Andersen constraints from those pointer statements and show how those constraints are equivalent to the ones generated by Palsberg.

For the control-flow analysis we generate constraints and find the least fixed point that satisfies the constraints, building up the set $\mathit{flows}[\![e]\!]$ for each expression e. For the points-to analysis, we also generate constraints and find the least fixed point that satisfies the constraints, building up the set $pts(x)$ for each variable x.

We need a way to deconstruct the results of the pointer analysis and convert them into useful results for our control-flow analysis. The key to this mapping is the labeling function $\mathcal{L} : \mathsf{Exp} \to \mathsf{Var}$ which we need to maintain certain properties in its mapping.

The labeling function assigns each expression a unique variable. In the mapping, given $x = \mathcal{L}((\lambda\ (v)\ e))$, the result $loc(x)$ will represent the lambda term $(\lambda\ (v)\ e))$. This means that if x is in the set of objects pointed to by a pointer, the lambda term is in the flow set of the expression that maps to that pointer.

Theorem 1. *Given* $x = \mathcal{L}((\lambda\ (v)\ e))$ *we have*

$$pts(x) = pts(\mathcal{L}((\lambda\ (v)\ e))) = \mathit{flows}[\![(\lambda\ (v)\ e)]\!]$$

We also require the property of the labeling function that if $x = \mathcal{L}((\lambda\ (v)\ e))$ and $y = \mathcal{L}(e)$, that the address of x be one greater than the address of x.

Case $(\lambda\ (v)\ e)$:

Given the labeling $x = \mathcal{L}((\lambda\ (v)\ e))$ and $y = \mathcal{L}(v)$ we generate the pointer statement $x = \&y$. This generates the constraint $loc(y) \in pts(x)$. The location of y is equal to the lambda term in our representation. The points-to set of x is equal to the flow set of the lambda term.

$$\begin{aligned} loc(y) &\in pts(x) \\ (\lambda\ (v)\ e) &\in pts(x) \\ (\lambda\ (v)\ e) &\in pts(\mathcal{L}((\lambda\ (v)\ e))) \\ (\lambda\ (v)\ e) &\in \mathit{flows}[\![(\lambda\ (v)\ e)]\!] \end{aligned}$$

This generates the desired constraint.

$$(\lambda\ (v)\ e) \in \mathit{flows}[\![(\lambda\ (v)\ e)]\!]$$

Case $(e_1\ e_2)$:
Given $x = \mathcal{L}(e_1)$ and $y = \mathcal{L}(e_2)$ we would generate the pointer statement $*x = y$. This generates the constraint $\forall v \in pts(x) : pts(y) \subseteq pts(v)$. Because $\mathit{flows}[\![e_1]\!] = pts(x)$ this generates the desired constraint.

$$\forall (\lambda\ (v)\ e) \in \mathit{flows}[\![e_1]\!] : \mathit{flows}[\![e_2]\!] \subseteq \mathit{flows}[\![v]\!]$$

Case $(e_1\ e_2)$:
Given $x = \mathcal{L}(e_1)$ and $y = \mathcal{L}(e_2)$ we would generate the pointer statement $x = *x + 1$. Which we would split into the pointer statements $p = *y$ and $x = p + 1$. The first statement generates the following constraint.

$$\forall v \in pts(y) : pts(v) \subseteq pts(p)$$

This gives us all the lambda terms pointed to by y because $\forall v \in pts(y)$. The points-to set of p is going to contain at least the values pointed to by y by this constraint, but since this is the only location where p is assigned, $pts(v) = pts(p)$.

The second statement generates the following constraint.

$$\{v + 1 : v \in pts(p) \subseteq pts(x)\}$$

The expression $v + 1$ gives the body of a lambda term. Because $pts(x) = pts(\mathcal{L}(e_1\ e_2)) = \mathit{flows}[\![(e_1\ e_2)]\!]$ we generate the following original constraint

$$\forall (\lambda\ (v)\ e_b) \in \mathit{flows}[\![e_1]\!] : \mathit{flows}[\![e_b]\!] \subseteq \mathit{flows}[\![(e_1\ e_2)]\!]$$

5. EigenCFA: A Point for Comparison

One possible intermediate representation for compilers of dynamic languages is continuation-passing style [2]. Given a language in continuation-passing style we will demonstrate how the encoding changes. We do this because this is the language form that is accepted by EigenCFA, as used by our benchmarks in section 6.

5.1 The Lambda Calculus in Continuation Passing Style

This language differs from the lambda calculus in that lambda terms now have multiple arguments and the body of a lambda term is now restricted to be only a call site.

$$\begin{aligned} \mathit{call} \in \mathsf{Call} &::= (f\ æ_1 \ldots æ_n) \\ f, æ \in \mathsf{AExp} &::= \mathit{lam} \mid v \\ \mathit{lam} \in \mathsf{Lam} &::= (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \\ v \in \mathsf{Var} &\text{ is a set of identifiers} \end{aligned}$$

5.2 Palsberg Style Inference Rules for CPS

We will now explore how the constraints for a control-flow analysis of a continuation-passing style language change. The constraints for continuation-passing style are simpler because we no longer have to worry about the flow sets of the body of lambda terms. This is because we never return, but rather invoke the passed explicit continuation. This causes us to go from three inference rules to two.

The inference rules are as follows. For expression $e \in \mathsf{Exp}$ under analysis, let E be the set of all expressions in e.

$$\frac{(\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \in E}{\{(\lambda\ (v_1 \ldots v_n)\ \mathit{call})\} \subseteq \mathit{flows}[\![(\lambda\ (v_1 \ldots v_n)\ \mathit{call})]\!]}$$

$$\frac{(f\ æ_1 \ldots æ_n) \in E \qquad (\lambda\ (v_1 \ldots v_n)\ \mathit{call}) \in \mathit{flows}[\![f]\!]}{\mathit{flows}[\![æ_1]\!] \subseteq \mathit{flows}[\![v_1]\!] \ldots \mathit{flows}[\![æ_n]\!] \subseteq \mathit{flows}[\![v_n]\!]}$$

5.3 Pointer Statement Encoding of CPS

Encoding a program in continuation-passing style into pointer statements uses the following inference rules.

$$\frac{e = (\lambda\ (v_0 \ldots v_n)\ \mathit{call}) \in E \qquad x = \mathcal{L}(e) \qquad y = \mathcal{L}(v_0)}{x = \&y}$$

$$\frac{(f\ æ_0 \ldots æ_n) \in E \qquad x = \mathcal{L}(f) \qquad y = \mathcal{L}(æ_i)}{*x + i = y}$$

We will now demonstrate how this encoding results in fewer statements and variables. Luckily our example program from before is already in continuation passing style.

$$((\lambda\ (x)\ (x\ x))\ (\lambda\ (y)\ (y\ y)))$$

For this program, we generate the following pointer statements:

$$\begin{aligned} l_1 &= \&x \\ l_2 &= \&y \\ *l_1 + 0 &= l_2 \\ *x + 0 &= x \\ *y + 0 &= y \end{aligned}$$

terms	EigenCFA	Pointer GPU	CPU-1	CPU-12
297	0.4	21	94	90
545	0.7	28	111	94
1,041	1.2	39	135	103
2,033	3	62	180	126
4,017	9	148	256	175
7,985	37	291	350	232
15,921	143	531	580	449
31,793	6367	1,317	784	836
63,537	3,709	4,030	1,578	1,452
127,025	31,228	12,175	3,819	2,557
190,513	142,162	40,881	13,615	4,349

Figure 1. The running time in milliseconds for each each of the implementations explored. The first column is the number of terms found in the benchmark. We show the running times of running the CPU pointer analysis with one thread (CPU-1) and with 12 threads (CPU-12).

Generating the pointer statements is quite simple, and since we do not need to worry about the body of lambda terms, results in fewer statements.

5.4 Alternative Encoding

We can forgo creating a variable for each lambda term if we deconstruct directly when we we have a lambda term in function position. A lambda term in function position is a let form and we know directly which variables we are binding, so the dereference to the lambda term is unnecessary. This results in more inference rules but results in fewer variables in the encoding.

$$\frac{((\lambda\ (x_0 \ldots x_n)\ call)\ æ_0 \ldots æ_n) \qquad æ_i = (\lambda\ (y \ldots)\ call_y)}{x_i = \&y}$$

$$\frac{((\lambda\ (x_0 \ldots x_n)\ call)\ æ_0 \ldots æ_n) \qquad æ_i = y}{x_i = y}$$

$$\frac{(f\ æ_0 \ldots æ_n) \qquad æ_i = (\lambda\ (y \ldots)\ call)}{*f + i = \&y}$$

$$\frac{(f\ æ_0 \ldots æ_n) \qquad æ_i = y}{*f + i = y}$$

6. Implementation

In this section, we explore how much we can improve upon the state of the art of higher-order control-flow analysis with this mapping. It turns out that a constraint based pointer analysis tool runs a lot faster than EigenCFA, a state of art control-flow analysis tool. EigenCFA is a lot faster than traditional control-flow analysis tools, but even it is outperformed by an optimized pointer analysis tool.

We evaluate EigenCFA as well as two recent parallel pointer analysis tools for C. We compare the following three tools.

EigenCFA
A GPU implementation that accelerates a control-flow analysis for higher-order languages, operating on the simple binary CPS language [13]. It encodes the analysis as matrix operations on sparse matrices.

GPU Inclusion-based Points-to Analysis
A GPU implementation that accelerates an inclusion-based points-to analysis [9]. It is based on a graph algorithm that monotonically grows the graph based on the constraints generated by the pointer statements.

CPU Inclusion-based Points-to Analysis
An inclusion-based points-to analysis that runs in parallel using multiple threads on the CPU [8]. It uses the same graph algorithm as the previous tool.

For the GPU tools, we ran them under Ubuntu on a Nvidia GTX-480 "Fermi" GPU with 1.5 GB of memory and the latest Nvidia drivers. We ran the parallel CPU tool on an machine running Mac OS X 10.8 with two Intel Xeon 3.07 Ghz processors, each having 6 cores, and 64 GB of memory.

We ran each tool on the benchmarks from the EigenCFA paper. To run the pointer analysis tools, we first ran the programs through our encoding and changed the input to be compatible with their tools.

The running times of the tools can be found in Figure 1. For EigenCFA the results are similar to those as from the original paper, though slightly slower. It is interesting to note that as the programs get large, the points-to analysis tools actually scales better than the original analysis.

To demonstrate how well the CPU scales with more threads we ran a various number of threads. In Figure 2 we see the running times for each of the values. The top row is the number of terms in the program. The columns are the run times in milliseconds. As we go down the column the number of threads increase.

From this we observe that there is actually a large improvement in run time for running an analysis on the CPU rather than on the GPU. This likely means that for the given benchmarks there is not parallelism that can be effectively exploited on a GPU. The GPU implementation visits every call site on every iteration, while the CPU implementation is able to more intelligently visit constraints. It will only visit constraints if they will add new values to the points-to sets of variables.

7. Future Work

There exist several avenues where this work could be extended.

	terms										
threads	297	545	1,041	2,033	4,017	7,985	15,921	31,793	63,537	127,025	190,513
1	94	111	135	180	256	350	580	784	1,578	3,819	13,615
2	91	107	129	167	247	322	495	767	1,796	3,812	12,216
4	89	99	115	140	192	280	475	692	1,315	2,501	6,997
8	86	93	107	127	180	230	449	820	927	2,848	5,293
10	85	93	103	129	170	229	467	823	1,176	2,765	4,833
12	90	94	103	126	175	232	449	836	1,452	2,557	4,349

Figure 2. The running time in milliseconds for each of the benchmarks on the multithreaded CPU implementation. This is to demonstrate how well the running time scales with the number of threads for the given benchmarks.

7.1 Pushdown Analysis

A major recent development in control-flow analysis has been pushdown analyses [4, 15]. This allows calls and returns to be precisely matched and has shown gains in precision. We believe it is possible to do a form of these analyses using our approach.

7.2 Flow and Context Sensitivity

The pointer analysis tools we explored are flow- and context insensitive. Because of this, we do not preserve any flow or context information in our transformation. However, if we had a tool that took advantage of these features, it would be ideal if our transformation could preserve this information. It has been shown though that flow sensitivity does not add much precision for Racket and other Scheme-like languages because there is not much mutation [3]. However, if this approach was applied to languages that use mutation more commonly (such as Javascript) preserving flow sensitivity would likely be beneficial.

7.3 Language Compilation

In is common to compile languages into other languages. If tools exists that analyze the target language but not the source language that we are working in, a flavor of this technique also applies. We could perform the desired analysis on the compiled language. Whether this would be useful or not would be highly dependent on our needs. If we need the analysis to provide information about our language in its original form, before we compiled it, we would need to ensure that the compilation process allows for decompilation and that the properties we are hoping to discover are not lost in the compilation process.

7.4 Additional Language Features

We have demonstrated how our technique works for the simple lambda calculus. However, mapping dynamic languages to the lambda calculus is nontrivial [5, 12]. In fact, we would recommend against performing this mapping solely to use our technique, as it is likely not to produce useful information. An alternative interesting approach worth investigating would be to see how this technique could be adapted in the presence of additional language features. How the additional language features are handled would be dependent on the information we wish our analysis to produce.

8. Related Work

One of the earliest works of a constraint-based formulation of control-flow analysis is Henglein's simple closure analysis [6]. It is based on unification and runs in almost linear time. Subsequent work applied a similar strategy to a points-to analysis, citing Henglein as an inspiration. This type of analysis is known as Steensgaard points-to analysis [14].

Control-flow analysis was also later developed in constraint form with the development of Palsberg [11]. This framework is based on subsets, rather than unification, and as such is more precise but is now cubic. At the same time a similar formulation was developed for pointer analysis of C programs by Andersen [1].

9. Conclusion

This paper demonstrates to static analysts, analyzing dynamic languages, a way to use existing tools by mapping their problem to ones that have already been solved with significant engineering effort behind them.

In this paper, we have demonstrated that a pointer analysis of a first-order language can be used to solve a control-flow analysis of a higher-order language, leveraging the significant effort that has gone into the state of the art of pointer analysis. We provided the inference rules to do this and demonstrated how these result in the same constraints as the control-flow analysis. We then demonstrated that we can effectively take advantage of existing pointer analysis tools for significant speed-ups.

Acknowledgments

This material is partially based on research sponsored by DARPA under agreements number AFRL FA8750-15-2-0092 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[3] J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for Higher-Order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, 1998.

[4] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 177–188, New York, NY, USA, 2012. ACM.

[5] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In T. D'Hondt, editor, *ECOOP 2010 Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer Berlin Heidelberg, 2010.

[6] F. Henglein. Simple closure analysis. Technical report, Department of Computer Science, University of Copenhagen (DIKU), Mar. 1992.

[7] O. Lhotak, Y. Smaragdakis, and M. Sridharan. Pointer analysis (dagstuhl seminar 13162). *Dagstuhl Reports*, 3(4):91–113, 2013.

[8] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 428–443, New York, NY, USA, 2010. ACM.

[9] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM.

[10] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. Object-Oriented program analysis. In *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 305–315, Toronto, Canada, June 2010.

[11] J. Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, Jan. 1995.

[12] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 217–232, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1.

[13] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 511–522, New York, NY, USA, 2011. ACM.

[14] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[15] D. Vardoulakis and O. Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming*, pages 570–589, 2010.

[16] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 91–103, New York, NY, USA, 1999. ACM.