Introduction

(9 min)

[board 1, left]
Control Flow Analysis in Scheme
Olin Shivers, PLDI 1988

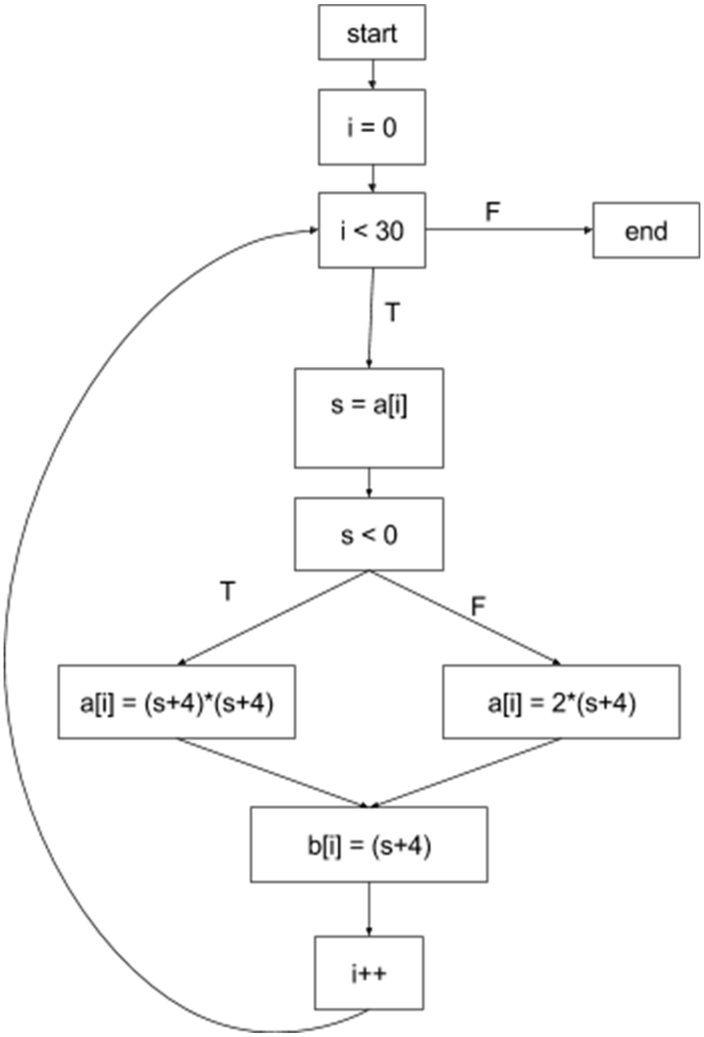
Olin Shivers is a prof in CCIS.
To set up the problem, let’s look at C first.

[board 1, left]

```
for (int i = 0; i < 30; i++) {
    int s = a[i];
    // empty space
    if (s < 0) {
        a[i] = (s+4) * (s+4);
    } else {
        a[i] = 2 * (s+4)
    }
    b[i] = s+4;
}
```

Note that s+4 is computed multiple times, so we could probably optimize that.
How would GCC figure this out?
First, it constructs a control flow graph, to determine the execution paths in the program.

[board 1, right]



From the control flow graph, we can see that `s+4` can be cached because s is never redefined. We introduce a temporary and assign `t=s+4`, so that it is computed only once. You could imagine a different program where something very expensive is cached.

This is an example of a **data flow** problem. What is true at a given program point *p*, independent of the execution path taken to reach *p*? To solve this, we need a **control flow** graph.

[board 1, right]
Control flow ---> Data flow

This kind of technique is very well known for C, Fortran, etc, descendents of the Algol family. You can find it in any compilers textbook, and it dates back to the 1960s.

What about the Lisp family of languages, e.g. Scheme and Racket?
Let’s try analyzing a Lisp program.

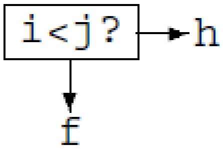
[board 2, left]

```
(let ((f (foo 7 g k))
      (h (bar a7 i j)))
  (if (< i j) (h 30) (f h)))
```

We call `foo` and its result is bound to `h`.
We call `bar` and its result is bound to `g`.
Then in the body of the let, we have a conditional, with two call sites: `h` or `f`.

Let’s try to do a data flow analysis. First we need a control flow graph.

[board 2, left]



What are `f` and `h`? They are computed at run time.
We need a data flow analysis to determine `f` and `h`.
But then we wanted a control flow graph to do data flow analysis.
Circularity!

[board 2, left]
Control flow <---> Data flow

What exactly makes this problem hard?
In C, there was a distinction between data flow and control flow.
The Lisp family is higher-order: functions are values.
Data flow and control flow are intertwined.
The paper presents CFA, a technique for solving this problem.

[board 2, right]
CFA: **What functions can be invoked at a given call site?**
Related question: **What lambdas can a given variable evaluate to?**

Overview of CFA

(7 min)

Let me present some of the rules (or constraints) of CFA.
For this example, I'll use the untyped lambda calculus.
The paper defines a helper function called `defs`.

[board 1, left]

```
defs : value -> {lambda}

defs  x       = { lambdas bound to x }
defs (λx . e) = (λx . e)
```

$$(e_1\ e_2)$$

Suppose we have an application $(e_1\ e_2)$.
Whatever lambda e_1 can evaluate to is the function that can be invoked at this call site.

Suppose e_1 evaluates to a set of lambdas $\{(\lambda x_1\ .\ e'_1),\ (\lambda x_2\ .\ e'_2),\ \dots\ \}$.
Then we generate the following constraints.

[board 1, left]

$$\text{defs}\ e_1 = \{(\lambda x_a\ .\ e_a),\ (\lambda x_b\ .\ e_b),\ \dots\ \}$$

$$\text{defs}\ e_2 \subset \text{defs}\ x_a$$
$$\text{defs}\ e_2 \subset \text{defs}\ x_b$$

Whatever lambda the argument can evaluate to is a subset of whatever lambda the parameter can evaluate to.
When analyzing the expression e_a, we know what x_a can evaluate to. (Variable rule.)

This can get pretty complicated. What if e_2 contains nested applications?

[board 1, right]

```
(f (foo (bar 1)))
```

To analyze the argument, we need to analyze `foo`.
To analyze `foo`, we need to analyze `bar`.
This is just two levels of nesting.
And this is just the untyped lambda calculus. What if we add more features?
The paper analyzes Scheme.

To deal with this complexity, the paper uses CPS as an intermediate representation.

CPS makes the terms more complicated and harder to read.
But it simplifies the cases that the analysis needs to handle.

Consider function application, which could be nested.
We've seen 1-arg application in class, let's look at 2-arg as an example.

[board 1, right]

$$[e_f\ e_1\ e_2]k =$$
$$\quad [e_f](\lambda f\ .\ [e_1](\lambda x_1\ .\ [e_2](\lambda x_2\ .\ f\ k\ x_1\ x_2)))$$

Note that application only occurs inside the body of a lambda.
Furthermore, x_1 and x_2 are variables, not arbitrary expressions.

Another example is `if`.

[board 1, right]

$$[\mathtt{if}\ e_0\ e_1\ e_2]k = [e_0](\lambda x_0\ .\ \mathtt{if}\ x_0\ (\lambda.\ [e_1]k)\ (\lambda.\ [e_2]k))$$

Note that e_1 and e_2 are wrapped in lambdas, because they need to be continuations for if.

Our questions

(2 min)

After reading the paper, we had two main questions.

[board 2, left]
1. How to actually implement CFA?

Paper presented the analysis as a set of constraints.
Considered each language feature separately.
We were not sure how everything fit together.
We did not know how to solve the constraints.
We could not run the analysis "by hand" on the examples in the paper.
So we wanted to do this project to implement and understand CFA.

[board 2, left]
2. Why use CPS?

We saw that CPS greatly simplifies the control flow.
But the translated terms are very verbose and difficult to understand, see example.
This was also why we could not understand the examples in the paper.
Another problem: CFA returns the lambdas that could be invoked at a given call site.
But these lambdas will be in CPS, so you have to map them back to the source language.

Approach

(2 min)

We implemented CFA as a Redex model.
First, we had to implement our source language and a CPS translation.

[board 2, right]

```
e  ::= (v₀ v₁ ... vₙ) | (op v₁ v₂ v₃) | (if v₀ v₁ v₂)
v  ::= x | n | b | (λx₁ ... xₙ . e)
n  ::= ... | -1 | 0 | 1 | …
b  ::= true | false
op ::= + | - | *
```

Note that op takes two values and a continuation, and if takes a value and two continuations.

Then we implemented CFA.

Redex helped us organize our thoughts.
Interacting with the model and writing tests helped us discover bugs.
Also helped us identify gaps in our understanding.
We could then focus on parts of the paper and other sources.

Now Aviral will take over and tell you what we learned from our project.

Insight

(19 min)

Questions: how do we implement CFA? Why CPS?
As Ming-Ho pointed out, when we started, we couldn't even run the analysis "by hand". We didn't know how to represent the constraints and to propagate them correctly. We got an insight from the application rule of untyped lambda calculus. Instead of substitution, we could create and environment, just like in the case of abstract machines.
In the environment each variable maps to a set of lambda functions.
Intuitively, it is the functions that a variable can evaluate to.
This environment was implicit in the paper.

```
env: [Var ↦ {lambda}]
```

Example -

```
env = [ f ↦ {(λa b. a b), (λa b. b a) …}
      , x ↦ {(λa . a) …}
      , y ↦ {(λa. 23) …}
      … ]
```

Now, constraints are bindings in the environment and we have to correctly flow and "merge" the environment. So, let's give ourselves a merge function.

```
merge : env₁, env₂ -> env₃

merge : env₁ env₂ =
  env₃ = clone(env₁)
  for [var ↦ {lambda}] in env₂:
      env₃[var] = env₃[var] ∪ {lambda}
env₃
```

Show an example of merge.

Now, the key question we are asking is, which functions can be called at a given call site. A call site in CPS can either be a variable or an actual lambda. So Let's look at the different cases and how they are handled with the environment.

[board 1]

```
defs: value, env -> {lambda}
```

We need a function to tell us what a value might evaluate to. (This is "defs".)
A lambda evaluates to a lambda.

A variable must be looked up.
Other values don’t evaluate to lambdas.

```
defs (λ$x_1$ ... $x_n$ . e) env = (λ$x_1$ ... $x_n$ . e)
defs x env = env(x)
defs v env = {}
```

Now that we know how to merge and lookup environments, we need to have a mechanism to build environments as we descend into expressions. This building will result in an environment which will map all variables to the lambdas that variable can evaluate to and the defs function will just look it up. The function to build this environment is appropriately named “build”.

```
build : expression, env -> env
```

OP

Due to CPS, the two operands and the continuation are variables.
Again, determine all the lambdas the continuation could evaluate to.
For each lambda, we analyze its body.

```
build (op $x_1$ $x_2$ k) env =
  env* = Ø
  for (λx . e) in (defs k env) <-- Looking up environment is CFA
    env* = merge env* (build e env)
  env*
```

IF

Due to CPS, the condition is a variable, and the two continuations are lambdas.
Recall that the continuations take no arguments.

How do we update the environment for this? We need to go to descend into both continuations, build up their environments and merge them.

```
build (if $x_0$ (λ. $e_1$) (λ. $e_2$)) env =
  merge (build $e_1$ env) (build $e_2$ env)
```

APP

Due to CPS, we know that everything in an application is a variable.
We have to determine all the lambdas that `f` could evaluate to.
For each lambda, we bind the argument to the corresponding variable.
Then we analyze the bodies of each of the lambdas.
Finally, we combine the results.

```
build (f $a_1$ ... $a_n$) env =
  env* = Ø
  for (λ$x_1$ ... $x_n$ . e) in (defs f env) <-- Looking up environment is CFA
    env’ = env[$x_i$ ↦ (defs $a_i$ env)]
    env* = merge env* (build  e env’)
  env*

build v env = env
```

Review: if the analysis encounters a variable, it needs to look it up in the environment.
Bindings are only introduced by function applications.

Conclusion

(2 min)

So, in conclusion, CFA is a technique for analyzing higher-order languages. Determines which lambda functions could be invoked at a given call site
Tri-faceted nature of lambda expressions - Control, data and environment
Since higher order functions were absent from Algol family of languages, the algorithms of that time couldn’t handle such a construct. CFA handled this construct by tracking its flow through the program in an abstract environment. So lambda is the core essence of the problem.
Its still an active area of research. People (Matthew Might, David Van Horn and collaborators) are still working on improving the precision of this algorithm.
Hopefully our presentation gave you an understanding of how the algorithm works.