Lecture Notes: Control Flow Analysis for Functional Languages

17-355/17-665: Program Analysis *(Spring 2017)*
Jonathan Aldrich
aldrich@cs.cmu.edu

1 Analysis of Functional Programs

Analyzing functional programs challenges the framework we've discussed so far. Understanding and solving those problems illustrates constraint based analyses and is also closely related to call graph construction in object-oriented languages, as we discussed in the previous lecture. Consider an idealized functional language based on the lambda calculus, similar to the core of Scheme or ML, defined as follows:

$$\begin{array}{rcl} e & ::= & \lambda x.e \\ & | & x \\ & | & e_1\ e_2 \\ & | & \text{let } x = e_1 \text{ in } e_2 \\ & | & \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \\ & | & n \mid\ e_1 + e_2 \mid\ ... \end{array}$$

The grammar includes a definition of an anonymous function $\lambda x.e$, where x is the function argument and e is the function body.[1] The function can include any of the other types of expressions, such as variables x or function calls $e_1 e_2$, where e_1 is the function to be invoked and e_2 is passed to that function as an argument. (In an imperative language this would more typically be written $e_1(e_2)$ but we follow the functional convention here, with parenthesis included when helpful syntactically). We evaluate a function call $(\lambda x.e)(v)$ by substituting the argument v for all occurrences of x in e. For example, $(\lambda x.x + 1)(3)$ evaluates to $3 + 1$, which of course evaluates to 4.

A more interesting example is $(\lambda f.f\ 3)(\lambda x.x + 1)$, which first substitutes the argument for f, yielding $(\lambda x.x + 1)\ 3$. Then we invoke the function, getting $3 + 1$ which again evaluates to 4.

1.1 0-CFA

Static analysis be just as useful in this type of language as in imperative languages, but immediate complexities arise. For example: what is a *program point* in a language without obvious predecessors or successors? Computation is intrinsically nested. Second, because functions are first-class entities that can be passed around as variables, it's not obvious which function is being applied where. Although it is not obvious, we still need some way to figure it out, because the value a function returns (which we may hope to track, such as through constant propagation analysis)

[1] The formulation in PPA also includes a syntactic construct for explicitly recursive functions. The ideas extend naturally, but we'll follow the simpler syntax for expository purposes.

will inevitably depend on which function is called, as well as its arguments. *Control flow analysis*[2] seeks to statically determine which functions could be associated with which variables. Further, because functional languages are not based on statements but rather expressions, it is appropriate to consider both the values of variables and the values expressions evaluate to.

We thus consider each expression to be labeled with a label $l \in \mathcal{L}$. Our analysis information σ maps each variable *and* label to a lattice value. This first analysis is only concerned with possible functions associated with each location or variable, and so the abstract domain is as follows:

$$\sigma \quad \in \quad \mathit{Var} \cup \mathcal{L} \rightarrow \mathcal{P}(\lambda x.e)$$

The analysis information at any given program point, or for any program variable, is a set of functions that could be stored in the variable or computed at that program point. *Question: what is the $\sqsubseteq$ relation on this dataflow state?*

We define the analysis by via inference rules that generate constraints over the possible dataflow values for each variable or labeled location; those constraints are then solved. We use the $\hookrightarrow$ to denote a relation such that $[\![e]\!]^l \hookrightarrow C$ can be read as "The analysis of expression e with label l generates constraints C over dataflow state σ." For our first CFA, we can define inference rules for this relation as follows:

$$\frac{}{[\![n]\!]^l \hookrightarrow \varnothing}\ \mathit{const} \qquad\qquad \frac{}{[\![x]\!]^l \hookrightarrow \sigma(x) \sqsubseteq \sigma(l)}\ \mathit{var}$$

In the rules above, the constant or variable value flows to the program location l. *Question: what might the rules for the if-then-else or arithmetic operator expressions look like?* The rule for function calls is a bit more complex. We define rules for lambda and application as follows:

$$\frac{[\![e]\!]^{l_0} \hookrightarrow C}{[\![\lambda x.e^{l_0}]\!]^l \hookrightarrow \{\lambda x.e\} \sqsubseteq \sigma(l) \cup C}\ \mathit{lambda}$$

$$\frac{[\![e_1]\!]^{l_1} \hookrightarrow C_1 \quad [\![e_2]\!]^{l_2} \hookrightarrow C_2}{[\![e_1^{l_1}\ e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2 \cup \forall \lambda x.e_0^{l_0} \in \sigma(l_1) : \sigma(l_2) \sqsubseteq \sigma(x) \wedge \sigma(l_0) \sqsubseteq \sigma(l)}\ \mathit{apply}$$

The first rule just states that if a literal function is declared at a program location l, that function is part of the lattice value $\sigma(l)$ computed by the analysis for that location. Because we want to analyze the data flow inside the function, we also generate a set of constraints C from the function body and return those constraints as well.

The rule for application first analyzes the function and the argument to extract two sets of constraints C_1 and C_2. We then generate a conditional constraint, saying that for every literal function $\lambda x.e_0$ that the analysis (eventually) determines the function may evaluate to, we must generate additional constraints capture value flow from the formal function argument to the actual argument variable, and from the function result to the calling expression.

Consider the first example program given above, properly labelled as: $((\lambda x.(x^a + 1^b)^c)^d(3)^e)^g$ one by one to analyze it. The first rule to use is *apply* (because that's the top-level program construct). We will work this out together, but the generated constraints could look like:

$$(\sigma(x) \sqsubseteq \sigma(a)) \cup (\{\lambda x.x + 1\} \sqsubseteq \sigma(d)) \cup (\sigma(e) \sqsubseteq \sigma(x)) \wedge (\sigma(c) \sqsubseteq \sigma(g))$$

[2] This nomenclature is confusing because it is also used to refer to analyses of control flow graphs in imperative languages; We usually abbreviate to CFA when discussing the analysis of functional languages.

There are many possible valid solutions to this constraint set; clearly we want a precise solution that does not overapproximate. We will elide a formal definition and instead assert that a σ that maps all variables and locations except d to $\varnothing$ and d to $\{\lambda x.x + 1\}$ satisfies this set of constraints.

1.2 0-CFA with dataflow information

The analysis in the previous subsection is interesting if all you're interested in is which functions can be called where, but doesn't solve the general problem of dataflow analysis of functional programs. Fortunately, extending that approach to a more general analysis space is straightforward: we simply add the abstract information we're tracking to the abstract domain defined above. For constant propagation, for example, we can extend the dataflow state as follows:

$$\sigma \quad \in \quad \mathit{Var} \cup \mathit{Lab} \rightarrow L \qquad\qquad L \quad = \quad (\bot + \mathbb{Z} + \top) \times \mathcal{P}(\lambda x.e)$$

Now, the analysis information at any program point, or for any variable, may be an integer n, or $\top$, or a set of functions that could be stored in the variable or computed at that program point. This requires that we modify our inference rules slightly, but not as much as you might expect. Indeed, the rules mostly change for arithmetic operators (which we omitted above) and constants. We simply need to provide an abstraction over concrete values that captures the dataflow information in question. The rule for constants thus becomes:

$$\frac{}{[\![n]\!]^l \hookrightarrow \beta(n) \sqsubseteq \sigma(l)}\ \mathit{const}$$

Where β is defined as we discussed in abstract interpretation. We must also provide an abstraction over arithmetic operators, as before; we omit this rule for brevity.

Consider the second example, above, properly labeled: $(((\lambda f.(f^a\ 3^b)^c)^e(\lambda x.(x^g + 1^h)^i)^j)^k$ A constant propagation analysis could produce the following results:

$\mathit{Var} \cup \mathit{Lab}$	L	by rule
e	$\lambda f.f\ 3$	lambda
j	$\lambda x.x + 1$	lambda
f	$\lambda x.x + 1$	apply
a	$\lambda x.x + 1$	var
b	3	const
x	3	apply
g	3	var
h	1	const
i	4	add
c	4	apply
k	4	apply

1.3 m-Calling Context Sensitive Control Flow Analysis (m-CFA)

The control flow analysis described above—known as 0-CFA, where CFA stands for Control Flow Analysis and the 0 indicates context insensitivity—works well for simple programs like the example above, but it quickly becomes imprecise in more interesting programs that reuse functions in several calling contexts. The following code illustrates the problem:

let $\mathit{add} = \lambda x.\ \lambda y.\ x + y$
let $\mathit{add5} = (\mathit{add}\ 5)^{a5}$
let $\mathit{add6} = (\mathit{add}\ 6)^{a6}$
let $\mathit{main} = (\mathit{add}5\ 2)^{m}$

This example illustrates *currying*, in which a function such as *add* that takes two arguments x and y in sequence can be called with only one argument (e.g. 5 in the call labeled $a5$), resulting in a function that can later be called with the second argument (in this case, 2 at the call labeled m). The value 5 for the first argument in this example is stored with the function in the *closure add5*. Thus when the second argument is passed to *add5*, the closure holds the value of x so that the sum $x + y = 5 + 2 = 7$ can be computed.

The use of closures complicates program analysis. In this case, we create two closures, *add5* and *add6*, within the program, binding 5 and 6 and the respective values for x. But unfortunately the program analysis cannot distinguish these two closures, because it only computes one value for x, and since two different values are passed in, we learn only that x has the value $\top$. This is illustrated in the following analysis. The trace below has been shortened to focus only on the variables (the actual analysis, of course, would compute information for each program point too):

Var ∪ *Lab*	L	notes
add	$\lambda x.\ \lambda y.\ x + y$	
x	5	when analyzing first call
$add5$	$\lambda y.\ x + y$	
x	$\top$	when analyzing second call
$add6$	$\lambda y.\ x + y$	
$main$	$\top$	

We can add precision using a context-sensitive analysis. One could, in principle, use either the functional or call-string approach, as described earlier. In practice the call-string approach seems to be used for control-flow analysis in functional programming languages, perhaps because in the functional approach there could be many, many contexts for each function, and it is easier to place a bound on the analysis in the call-string approach.

We add context sensitivity by making our analysis information σ track information separately for different call strings, denoted by Δ. Here a call string is a sequence of labels, each one denoting a function call site, where the sequence can be of any length between 0 and some bound m (in practice m will be in the range 0-2 for scalability reasons):

$$\sigma \in (\mathit{Var} \cup \mathit{Lab}) \times \Delta \to L \qquad \delta \in \Delta = \mathit{Lab}^{n \leqslant m} \qquad L = (\bot + \mathbb{Z} + \top) \times \mathcal{P}((\lambda x.e, \delta))$$

When a lambda expression is analyzed, we now consider as part of the lattice the call string context δ in which its free variables were captured. We can then define a set of rules that generate constraints which, when solved, provide an answer to control-flow analysis, as well as (in this case) constant propagation:

$$\frac{}{\delta \vdash [\![n]\!]^{l} \hookrightarrow \alpha(n) \sqsubseteq \sigma(l,\delta)}\ \mathit{const} \qquad \frac{}{\delta \vdash [\![x]\!]^{l} \hookrightarrow \sigma(x,\delta) \sqsubseteq \sigma(l,\delta)}\ \mathit{var}$$

$$\frac{}{\delta \vdash [\![\lambda x.e^{l_0}]\!]^l \hookrightarrow \{(\lambda x.e, \delta)\} \sqsubseteq \sigma(l, \delta)} \; \textit{lambda}$$

$$\frac{\begin{array}{l} \delta \vdash [\![e_1]\!]^{l_1} \hookrightarrow C_1 \qquad \delta \vdash [\![e_2]\!]^{l_2} \hookrightarrow C_2 \qquad \delta' = \textit{suffix}(\delta{+}{+}l, m) \\ C_3 = \bigcup_{(\lambda x.e_0^{l_0}, \delta_0) \in \sigma(l_1, \delta)} \sigma(l_2, \delta) \sqsubseteq \sigma(x, \delta') \wedge \sigma(l_0, \delta') \sqsubseteq \sigma(l, \delta) \\ \qquad\qquad\qquad\qquad\qquad \wedge\, \forall y \in FV(\lambda x.e_0) : \sigma(y, \delta_0) \sqsubseteq \sigma(y, \delta') \\ C_4 = \bigcup_{(\lambda x.e_0^{l_0}, \delta_0) \in \sigma(l_1, \delta)} \textit{analyze}(\delta' \vdash [\![e_0]\!]^{l_0}) \end{array}}{\delta \vdash [\![e_1^{l_1}\ e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2 \cup C_3 \cup C_4} \; \textit{apply}$$

These rules contain a call string context δ in which the analysis of each line of code is done. The rules *const* and *var* are unchanged except for indexing σ by the current context δ. The *lambda* rule now captures the context δ along with the lambda expression, so that when the lambda expression is called the analysis knows in which context to look up the free variables.

The apply rule has gotten more complicated. A new context δ is formed by appending the current call site l to the old call string, then taking the suffix of length m (or less). We now consider all functions that may be called, as eventually determined by the analysis (our notation is slightly loose, because the quantifier must be evaluated continuously for more matches as the analysis goes along). For each, we produce constraints capturing the flow of values from the formal to actual arguments, and from the result to the calling expression. We also produce constraints that bind the free variables in the new context: all free variables in the called function flow from the point δ_0 at which the closure was captured. Finally, in C_4 we collect the constraints that we get from analyzing each of the potentially called functions in the new context δ'.

We can now reanalyze the earlier example, observing the benefit of context sensitivity. In the table below, • denotes the empty calling context (e.g. when analyzing the *main* procedure):

Var / Lab, δ	L	notes
add, •	$(\lambda x.\ \lambda y.\ x + y,\ \bullet)$	
x, a5	5	
add5, •	$(\lambda y.\ x + y,\ a5)$	
x, a6	6	
add6, •	$(\lambda y.\ x + y,\ a6)$	
main, •	7	

Note three points about this analysis. First, we can distinguish the values of x in the two calling contexts: x is 5 in the context a5 but it is 6 in the context a6. Second, the closures returned to the variables *add5* and *add6* record the scope in which the free variable x was bound when the closure was captured. This means, third, that when we invoke the closure *add5* at program point m, we will know that x was captured in calling context a5, and so when the analysis analyzes the addition, it knows that x holds the constant 5 in this context. This enables constant propagation to compute a precise answer, learning that the variable *main* holds the value 7.

1.4 Optional: Uniform k-Calling Context Sensitive Control Flow Analysis (k-CFA)

m-CFA was proposed recently by Might, Smaragdakis, and Van Horn as a more scalable version of the original k-CFA analysis developed by Shivers for Scheme. While m-CFA now seems to be a better tradeoff between scalability and precision, k-CFA is interesting both for historical reasons and because it illustrates a more precise approach to tracking the values of variables in a closure. The following example illustrates a situation in which m-CFA may be too imprecise:

$$
\begin{array}{lll}
\textbf{let}\ \mathit{adde} & = \lambda x. & \\
 & \quad \textbf{let}\ h = \lambda y.\ \lambda z.\ x + y + z & \\
 & \quad \textbf{let}\ r = (h\ 8)^{r} & \\
 & \quad \textbf{in}\ r & \\
\textbf{let}\ \mathit{t} & = (adde\ 2)^{t} & \\
\textbf{let}\ \mathit{f} & = (adde\ 4)^{f} & \\
\textbf{let}\ \mathit{e} & = (t\ 1)^{e} &
\end{array}
$$

When we analyze it with m-CFA (for $m = 1$), we get the following results:

Var / Lab, δ	L	notes
adde, •	$(\lambda x...,\ \bullet)$	
x, t	2	
y, r	8	
x, r	2	when analyzing first call
t, •	$(\lambda z.\ x + y + z,\ r)$	
x, f	4	
x, r	$\top$	when analyzing second call
f, •	$(\lambda z.\ x + y + z,\ r)$	
e, •	$\top$	

The k-CFA analysis is like m-CFA, except that rather than keeping track of the scope in which a closure was captured, the analysis keeps track of the scope in which each variable captured in the closure was defined. We use an environment η to track this. Note that since η can represent a separately calling context for each variable, rather than merely a single context for all variables, it has the potential to be more accurate, but also much more expensive. We can represent the analysis information as follows:

$$
\begin{array}{lllllll}
\sigma & \in & (\mathit{Var} \cup \mathit{Lab}) \times \Delta \to L & \qquad & \Delta & = & \mathit{Lab}^{n \leqslant k} \\
L & = & \mathbb{Z} + \top + \mathcal{P}(\lambda x.e, \eta) & \qquad & \eta & \in & \mathit{Var} \to \Delta
\end{array}
$$

Let us briefly analyze the complexity of this analysis. In the worst case, if a closure captures n different variables, we may have a different call string for each of them. There are $O(n^k)$ different call strings for a program of size n, so if we keep track of one for each of n variables, we have $O(n^{n*k})$ different representations of the contexts for the variables captured in each closure. This exponential blowup is why k-CFA scales so badly. m-CFA is comparatively cheap—there are "only" $O(n^k)$ different contexts for the variables captured in each closure—still exponential in k, but polynomial in n for a fixed (and generally small) k.

We can now define the rules for k-CFA. They are similar to the rules for m-CFA, except that we now have two contexts: the calling context δ, and the environment context η tracking the context in which each variable is bound. When we analyze a variable x, we look it up not in the current context δ, but the context $\eta(x)$ in which it was bound. When a lambda is analyzed, we track the current environment η with the lambda, as this is the information necessary to determine where captured variables are bound. The application rule is actually somewhat simpler, because we do not copy bound variables into the context of the called procedure:

$$\frac{}{\delta, \eta \vdash \llbracket n \rrbracket^l \hookrightarrow \alpha(n) \sqsubseteq \sigma(l, \delta)}\ \textit{const} \qquad \frac{}{\delta, \eta \vdash \llbracket x \rrbracket^l \hookrightarrow \sigma(x, \eta(x)) \sqsubseteq \sigma(l, \delta)}\ \textit{var}$$

$$\frac{}{\delta, \eta \vdash \llbracket \lambda x.e^{l_0} \rrbracket^l \hookrightarrow \{(\lambda x.e, \eta)\} \sqsubseteq \sigma(l, \delta)}\ \textit{lambda}$$

$$\frac{\begin{array}{l} \delta, \eta \vdash \llbracket e_1 \rrbracket^{l_1} \hookrightarrow C_1 \qquad \delta, \eta \vdash \llbracket e_2 \rrbracket^{l_2} \hookrightarrow C_2 \qquad \delta' = \textit{suffix}(\delta{+}{+}l, k) \\ C_3 = \bigcup_{(\lambda x.e_0^{l_0}, \eta_0) \in \sigma(l_1, \delta)} \sigma(l_2, \delta) \sqsubseteq \sigma(x, \delta') \wedge \sigma(l_0, \delta') \sqsubseteq \sigma(l, \delta) \\ C_4 = \bigcup_{(\lambda x.e_0^{l_0}, \eta_0) \in \sigma(l_1, \delta)} C \textit{ where } \delta', \eta_0 \vdash \llbracket e_0 \rrbracket^{l_0} \hookrightarrow C \end{array}}{\delta, \eta \vdash \llbracket e_1^{l_1}\ e_2^{l_2} \rrbracket^l \hookrightarrow C_1 \cup C_2 \cup C_3 \cup C_4}\ \textit{apply}$$

Now we can see how k-CFA analysis can more precisely analyze the latest example program. In the simulation below, we give two tables: one showing the order in which the functions are analyzed, along with the calling context δ and the environment η for each analysis, and the other as usual showing the analysis information computed for the variables in the program:

function	δ	η
main	$\bullet$	$\varnothing$
adde	t	$\{x \mapsto t\}$
h	r	$\{x \mapsto t,\ y \mapsto r\}$
adde	f	$\{x \mapsto f\}$
h	r	$\{x \mapsto f,\ y \mapsto r\}$
$\lambda z....$	e	$\{x \mapsto t,\ y \mapsto r,\ z \mapsto e\}$

Var / Lab, δ	*L*	notes
adde, $\bullet$	$(\lambda x...,\ \bullet)$	
x, t	2	
y, r	8	
t, $\bullet$	$(\lambda z.\ x + y + z,\ \{x \mapsto t,\ y \mapsto r\})$	
x, f	4	
f, $\bullet$	$(\lambda z.\ x + y + z,\ \{x \mapsto f,\ y \mapsto r\})$	
z, e	1	
t, $\bullet$	11	

Tracking the definition point of each variable separately is enough to restore precision in this program. However, programs with this structure—in which analysis of the program depends on different calling contexts for bound variables even when the context is the same for the function eventually called—appear to be rare in practice. Might et al. observed no examples among the real programs they tested in which k-CFA was more accurate than m-CFA—but k-CFA was often far more costly. Thus at this point the m-CFA analysis seems to be a better tradeoff between efficiency and precision, compared to k-CFA.

Acknowledgements

I thank Claire Le Goues for greatly appreciated extensions and refinements to these notes.