Static Program Analysis

Part 9 – control flow analysis

https://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University

Agenda

- **Control flow analysis for TIP with first-class functions**
- Control flow analysis for the λ-calculus
- The cubic framework
- Control flow analysis for object-oriented languages

TIP with first-class functions

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = f(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

Control flow complications

- First-class functions in TIP complicate CFG construction:
  - several functions may be invoked at a call site
  - this depends on the dataflow
  - but dataflow analysis first requires a CFG

- Same situation for other features:
  - function values with free variables (closures)
  - a class hierarchy with objects and methods
  - prototype objects with dynamic properties

Control flow analysis

- A control flow analysis approximates the call graph
  - conservatively computes possible functions at call sites
  - the trivial answer: *all* functions
- Control flow analysis is usually flow-*insensitive*:
  - based on the AST
  - the call graph can be used for an interprocedural CFG
  - a subsequent dataflow analysis may use the CFG
- Alternative: use flow-sensitive analysis
  - potentially on-the-fly, during dataflow analysis

CFA for TIP with first-class functions

- For a computed function call

  $E(E_1, \ldots, E_n)$

  we cannot immediately see which function is called

- A coarse but sound approximation:
  - assume any function with right number of arguments

- Use CFA to get a much better result!

CFA constraints (1/2)

- Tokens are all functions $\{f_1, f_2, \ldots, f_k\}$
- For every AST node, v, we introduce the variable $[\![v]\!]$ denoting the set of functions to which v may evaluate

- For function definitions $f(\ldots)\{\ldots\}$:

  $f \in [\![f]\!]$
- For assignments $x = E$:

  $[\![E]\!] \subseteq [\![x]\!]$

CFA constraints (2/2)

- For **direct** function calls $f(E_1, \ldots, E_n)$:

  $[\![E_i]\!] \subseteq [\![a_i]\!]$ for i=1,...,$n \wedge [\![E']\!] \subseteq [\![f(E_1, \ldots, E_n)]\!]$

  where f is a function with arguments $a_1, \ldots, a_n$ and return expression E'

- For **computed** function calls $E(E_1, \ldots, E_n)$:

  $f \in [\![E]\!] \Rightarrow ([\![E_i]\!] \subseteq [\![a_i]\!]$ for i=1,...,$n \wedge [\![E']\!] \subseteq [\![(E)(E_1, \ldots, E_n)]\!])$

  for every function f with arguments $a_1, \ldots, a_n$ and return expression E'

  – If we consider typable programs only:

    only generate constraints for those functions f for which the call would be type correct

Generated constraints

$$
\begin{aligned}
&\texttt{inc} \in [\![\texttt{inc}]\!] \\
&\texttt{dec} \in [\![\texttt{dec}]\!] \\
&\texttt{ide} \in [\![\texttt{ide}]\!] \\
&[\![\texttt{ide}]\!] \subseteq [\![\texttt{f}]\!] \\
&[\![\texttt{f(n)}]\!] \subseteq [\![\texttt{r}]\!] \\
&\texttt{inc} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{i}]\!] \wedge [\![\texttt{i+1}]\!] \subseteq [\![\texttt{f(n)}]\!] \\
&\texttt{dec} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{j}]\!] \wedge [\![\texttt{j-1}]\!] \subseteq [\![\texttt{f(n)}]\!] \\
&\texttt{ide} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{k}]\!] \wedge [\![\texttt{k}]\!] \subseteq [\![\texttt{f(n)}]\!] \\
&[\![\texttt{input}]\!] \subseteq [\![\texttt{x}]\!] \\
&[\![\texttt{foo(x,inc)}]\!] \subseteq [\![\texttt{y}]\!] \\
&[\![\texttt{foo(x,dec)}]\!] \subseteq [\![\texttt{y}]\!] \\
&\texttt{foo} \in [\![\texttt{foo}]\!] \\
&\texttt{foo} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \wedge [\![\texttt{inc}]\!] \subseteq [\![\texttt{f}]\!] \wedge [\![\texttt{r}]\!] \subseteq [\![\texttt{foo(x,inc)}]\!] \\
&\texttt{foo} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \wedge [\![\texttt{dec}]\!] \subseteq [\![\texttt{f}]\!] \wedge [\![\texttt{r}]\!] \subseteq [\![\texttt{foo(x,dec)}]\!] \\
&\texttt{main} \in [\![\texttt{main}]\!]
\end{aligned}
$$

assuming we do not use the special rule for direct calls

(At each call we only consider functions with matching number of parameters)

Least solution

$[\![\texttt{inc}]\!] = \{\texttt{inc}\}$

$[\![\texttt{dec}]\!] = \{\texttt{dec}\}$

$[\![\texttt{ide}]\!] = \{\texttt{ide}\}$

$[\![\texttt{f}]\!] = \{\texttt{inc}, \texttt{dec}, \texttt{ide}\}$

$[\![\texttt{foo}]\!] = \{\texttt{foo}\}$

$[\![\texttt{main}]\!] = \{\texttt{main}\}$

(the solution is the empty set for the remaining constraint variables)

With this information, we can construct the call edges and return edges in the interprocedural CFG

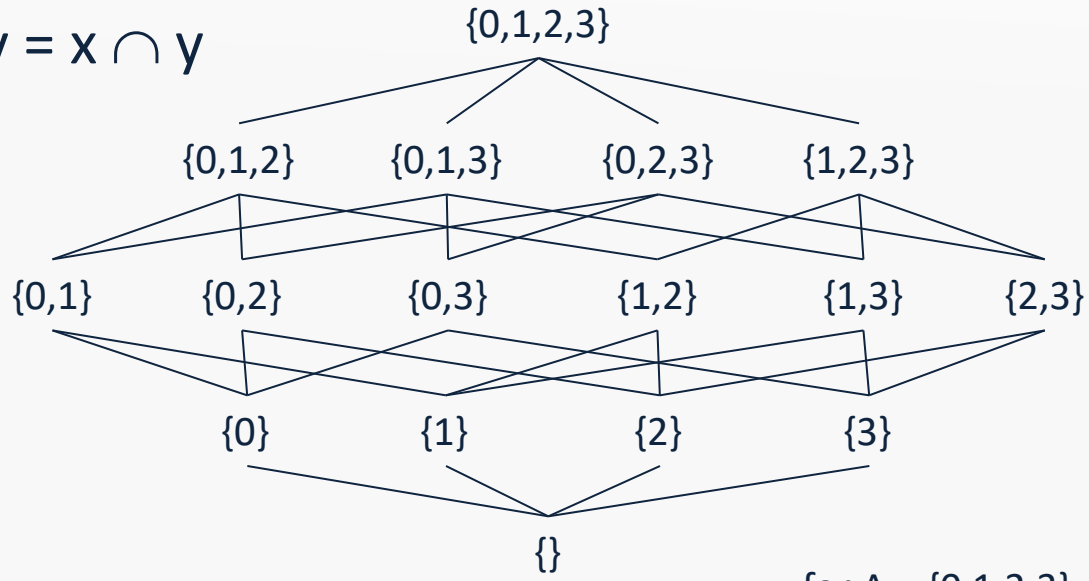
Analysis lattice

$$(\mathcal{P}(A))^n$$

where A is the set of function definitions
and n is the number of AST nodes in the program

The powerset lattice

- Every finite set A defines a complete lattice $(\mathcal{P}(A), \subseteq)$, where
  - $\bot = \varnothing$
  - $\top = A$
  - $x \sqcup y = x \cup y$
  - $x \sqcap y = x \cap y$



for A = {0,1,2,3}

Agenda

- Control flow analysis for TIP with first-class functions
- **Control flow analysis for the λ-calculus**
- The cubic framework
- Control flow analysis for object-oriented languages

CFA for the lambda calculus

- The pure lambda calculus

$Exp \rightarrow \lambda Id.Exp$		(function definition)
	$\mid Exp_1\ Exp_2$	(function application)
	$\mid Id$	(variable reference)

- Assume all λ-bound variables are distinct
- An *abstract closure* λx abstracts the function $\lambda x.E$ in all contexts (values of free variables)
- Goal: for each call site E_1E_2 determine the possible functions for E_1 from the set $\{\lambda x_1, \lambda x_2, \ldots, \lambda x_n\}$

Closure analysis

A flow-insensitive analysis that tracks function values:

- For every AST node, v, we introduce a variable $[\![v]\!]$ ranging over subsets of abstract closures
- For $\lambda x.E$ we have the constraint

$$\lambda x \in [\![\lambda x.E]\!]$$

- For $E_1 E_2$ we have the *conditional* constraint

$$\lambda x \in [\![E_1]\!] \Rightarrow \left([\![E_2]\!] \subseteq [\![x]\!] \wedge [\![E]\!] \subseteq [\![E_1 E_2]\!]\right)$$

for every function $\lambda x.E$

Agenda

- Control flow analysis for TIP with first-class functions
- Control flow analysis for the λ-calculus
- **The cubic framework**
- Control flow analysis for object-oriented languages

The cubic framework

- We have a set of tokens T=$\{t_1, t_2, \ldots, t_k\}$
- We have a collection of constraint variables V=$\{x_1, \ldots, x_n\}$ ranging over subsets of tokens
- A collection of constraints of these forms:
  - $t \in x$
  - $x \subseteq y$
  - $t \in x \Rightarrow y \subseteq z$
- Compute the unique minimal solution
  - this exists since solutions are closed under intersection
- A cubic time algorithm exists!

The solver data structure

- Each variable is mapped to a node in a directed graph
- Each node has a bitvector in $\{0,1\}^k$
  - initially set to all 0's
- Each bit has a list of pairs of variables
  - used to model conditional constraints
- The edges model inclusion constraints
- The bitvectors will at all times directly represent the minimal solution to the constraints seen so far

Implementation: `SimpleCubicSolver`

The solver data structure

- $x.sol \subseteq T$: the set of tokens for x (the bitvectors)
- $x.succ \subseteq V$: the successors of x (the edges)
- $x.cond(t) \subseteq V \times V$: the conditional constraints for x and t
- $W \subseteq T \times V$: a worklist (initially empty)

Adding constraints

- $t \in x$

  addToken(t, x)
  propagate()

- $x \subseteq y$

  addEdge(x, y)
  propagate()

- $t \in x \Rightarrow y \subseteq z$

  if t $\in$ x.sol
    addEdge(y, z)
    propagate()
  else
    add (y, z) to x.cond(t)

```
addToken(t, x):
  if t ∉ x.sol
    add t to x.sol
    add (t, x) to W

addEdge(x, y):
  if x ≠ y ∧ y ∉ x.succ
    add y to x.succ
    for each t in x.sol
      addToken(t, y)

propagate():
  while W ≠ ∅
    pick and remove (t, x) from W
    for each (y, z) in x.cond(t)
      addEdge(y, z)
    for each y in x.succ
      addToken(t, y)
```

Time complexity

- O(n) functions and O(n) applications, with program size n
- O(n) singleton constraints, O(n) subset constraints, O(n^2) conditional constraints
- O(n) nodes, O(n^2) edges, O(n) bits per node
- addToken takes time O(1)
- addEdge takes time O(n)
- Each pair (t, x) is processed at most once by propagate
- O(n^2) calls to addEdge (either immediately or via propagate)
- O(n^3) calls to addToken

Time complexity

- Adding it all up, the upper bound is $O(n^3)$
- This is known as the *cubic time bottleneck*:
  - occurs in many different scenarios
  - but $O(n^3/\log n)$ is possible…

Implementation tricks

- Cycle elimination (collapse nodes if there is a cycle of inclusion constraints)
- Process worklist in topological order
- Interleaving solution propagation and constraint processing
- Shared bit vector representation
- Type filtering
- On-demand processing
- Difference propagation
- Subsumed node compaction
- …

Cycle elimination

1998

Partial Online Cycle Elimination in Inclusion Constraint Graphs

Manuel Fähndrich* Jeffrey S. Foster* Zhendong Su* Alexander Aiken*

EECS Department
University of California, Berkeley
387 Soda Hall #1776
Berkeley, CA 94720-1776
{manuel,jfoster,zhendong,aiken}@cs.berkeley.edu

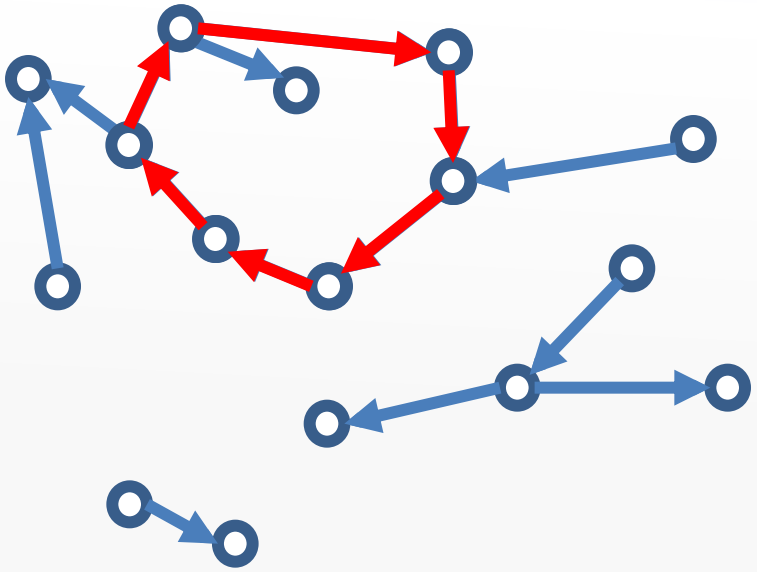
Abstract

Many program
plemented usin
sults on the sca
on two insights:
straints yields
sis time for larg
straint represen
online cycle eli
that explains o
predictions ma
periment.

ing inclusion constraint analyses scale to large problems with good performance. Cyclic constraints have the form $X_1 \subseteq X_2 \subseteq X_3 \ldots \subseteq X_n \subseteq X_1$ where the X_i are set variables. All variables on such a cycle are equal in all solutions of the constraints, and thus the cycle can be collapsed to a single variable.

by performing cycle detection and elimination online, i.e., at every update of the constraint graph. At first glance, this approach seems overly expensive, since the best known

Cycle elimination

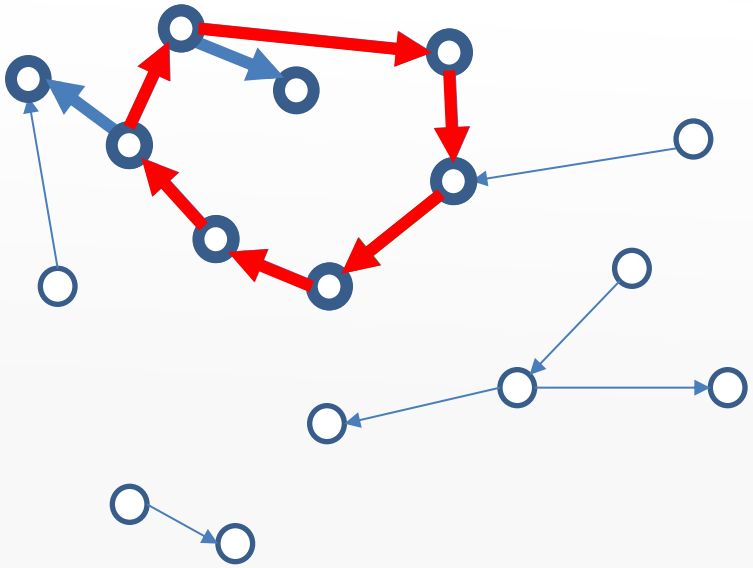


Using Tarjan’s algorithm for computing strongly connected components

Cycle elimination

- If applied naively, more expensive than the actual analysis ☹
- Pragmatic approaches:
  - “partial online cycle detection”
  - “lazy cycle detection”
  - “wave propagation”
  - ...
- Recommended:
  - apply a variant of Tarjan’s algorithm, only when a batch of edges have been added
  - **search for cycles only from nodes where new edges have been added**

Cycle elimination



Remaining nodes are not visited!

Agenda

- Control flow analysis for TIP with first-class functions
- Control flow analysis for the λ-calculus
- The cubic framework
- **Control flow analysis for object-oriented languages**

Simple CFA for OO (1/3)

- CFA in an object-oriented language:

```
x.m(a,b,c)
```

- Which method implementations may be invoked?

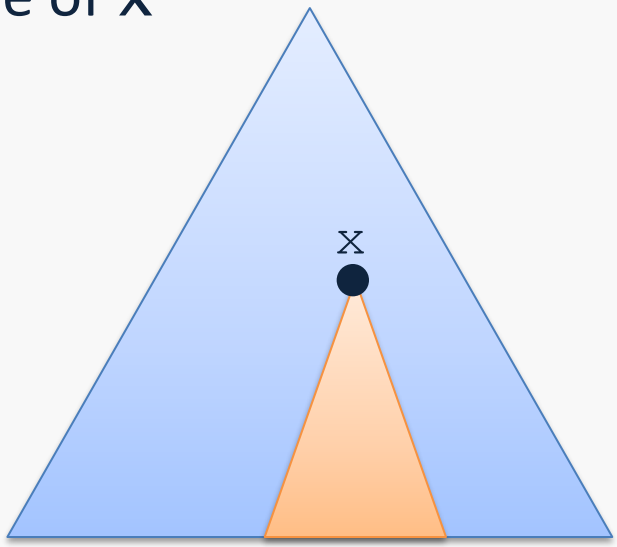
- Full CFA is a possibility...
- But the type information enables simpler solutions

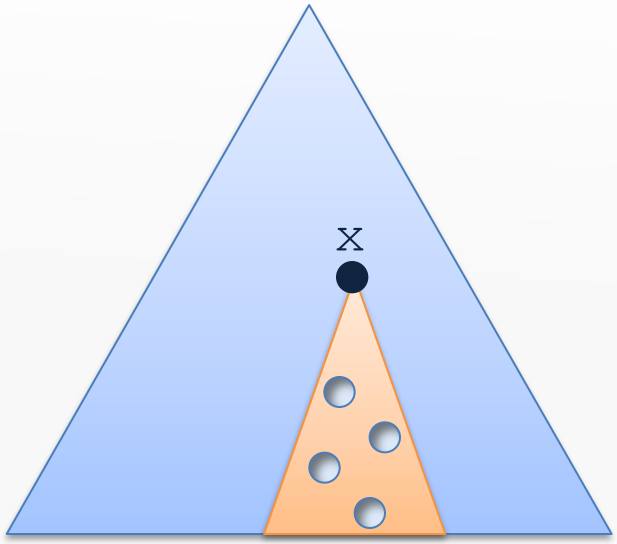
Simple CFA for OO (2/3)

- Simplest solution:
  - select all methods named `m` with three arguments

- Class Hierarchy Analysis (CHA):
  - consider only the part of the class hierarchy rooted by the declared type of `x`

```
Collection<T> c = ...
c.add(e)
```



Simple CFA for OO (3/3)

- Rapid Type Analysis (RTA):
  - restrict to those classes that are actually used in the program in `new` expressions
  - start from `main`, iteratively find reachable methods