Research Topics in Software Quality

Detecting Mechanical Bugs using Static Analysis:
Abstract Interpretation of Higher-Order Programs
Acknowledgements

These notes combine slides from:

- Matthew Might
  “Control-flow analysis of order k (k-CFA)” at 2009 Summer School on Theory and Practice of Language Implementation.
  “Tutorial: Small-step CFA” NII Shonan Meeting on Higher-Order Program Analysis

Source code from:
- “k-CFA: Determining types and/or control-flow in languages like Python, Java and Scheme”
  http://matt.might.net/articles/implementation-of-kcfa-and-0cfa/
Motivation

Analyses seen so far propagate dataflow facts through a control flow graph.

What about higher-order languages for which there is no static control flow graph at compile-time?

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

Doing data flow analysis requires control flow graph!
Determining control flow requires doing data flow analysis!
Continuation-passing style as an intermediate language

CPS: every function \( f \) takes an extra argument \( k \) (= the continuation of \( f \)), representing what should be done with the result \( v \) that the function is calculating. Instead of returning \( v \), \( f \) calls \( k \) on \( v \).

**direct style:**
\[
(\text{print } (* (+ x y) (- z w)))
\]

**CPS style:**
\[
(+ x y (\lambda (\text{sum})
(- z w (\lambda (\text{diff})
(* sum diff (\lambda (\text{prod})
(print prod *c*)))))
\]

advantage: all control (e.g., sequencing, conditional branching, try-catch, call/cc, ...) represented uniformly by procedure calls
CPS lambda calculus: concrete syntax

\[ \nu \in \text{Var} \text{ is a set of variables} \]

\[ \text{lam} \in \text{Lam} ::= (\lambda (\nu_1 \ldots \nu_n) \text{ call}) \]

\[ f, \, \alpha \in \text{AExp} ::= \nu \mid \text{lam} \]

\[ \text{call} \in \text{Call} ::= (f \, \alpha_1 \ldots \alpha_n) \]

atomic expressions: variable reference or lambda exp always evaluate to a value, cannot cause side effects

complex expressions: may not terminate, may produce side effects

notation for prototypical element

syntactic category
CPS lambda calculus: abstract syntax

all constructs are labeled

;; exp ::= (ref  <label>  <var>)
;; | (lambda  <label>  (<var1>  ...  <varN>)  <call>)
;; call ::= (call  <label>  <exp0>  <exp1>  ...  <expN>)

;; label = integer

(define (var? exp) (symbol? exp))
(define (ref? exp) (and (pair? exp) (eq? (car exp) 'ref)))
(define (ref->var exp) (caddr exp))

(define (lambda? exp) (and (pair? exp) (eq? (car exp) 'lambda)))
(define (lambda->lab exp) (cadr exp))
(define (lambda->formals exp) (caddr exp))
(define (lambda->call exp) (cadddr exp))

(define (call? term) (and (pair? term) (eq? (car term) 'call)))
(define (call->lab call) (cadr call))
(define (call->fun call) (caddr call))
(define (call->args call) (cdddr call))

(define (explode-call call k)
  (k (call->lab call)
     (call->fun call)
     (call->args call)))

accessor for the name of the referenced variable (represented as a 'symbol)

accessors for the label, formal parameters and the body of a lambda expression (a single call exp)

convenience function that deconstructs a list representing a call into its label, function expression and concrete argument expressions
CPS lambda calculus: abstract syntax tree factory

(labels will be taken care off by constructor functions)

(define label-count 1)

(define (new-label)
  (set! label-count (+ 1 label-count))
  label-count)

(define (make-ref var)
  (list 'ref (new-label) var))

(define (make-lambda formals call)
  (list 'lambda (new-label) formals call))

(define (make-call fun args)
  (cons 'call
    (cons (new-label)
      (cons fun
        args))))

(define (make-let var exp call)
  (make-call (make-lambda (list var) call) (list exp)))

(let is equivalent to a lambda application)
CPS lambda calculus: example

(id function takes in CPS an additional continuation argument
invoked on the result of its evaluation

small deviation from formal: no call in body denotes halt?)

```
(define standard-example
  (make-let 'id (make-lambda '(x k) (make-call (make-ref 'k) (list (make-ref 'x))))
    (make-call (make-ref 'id)
      (list (make-lambda '(z) (make-ref 'z))
        (make-lambda '(a)
          (make-call (make-ref 'id)
            (list (make-lambda '(y) (make-ref 'y))
              (make-lambda '(b)
                (make-ref 'b)))))))))
```
Deriving an abstract interpreter from a machine-based concrete one

**Concrete semantics**

- Convert program \( e \) into initial machine state \( s_0 \)
- Transition from state \( s_n \) to state \( s_{n+1} \)
- Until end state reached (if program terminates)
- Value within end state is result of evaluating \( e \)

**Abstract semantics**

- Abstract states computing on abstract values
- State transitions can be non-deterministic (multiple successor states)
- Possible infinite state space
- Finite state space to ensure termination

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots \]

\[ \hat{s}_0 \rightarrow \hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_4 \rightarrow \hat{s}_3.1 \]
Relation between concrete & abstract
1st attempt: exact simulation...

values computed abstractly are the same as
the abstraction of concrete computation results
Relation between concrete & abstract
1st attempt: exact simulation... fails!

\[ p: Y := X+Y; \text{ goto } q \]

State \(\xrightarrow{\text{next}}\) State
\[ \alpha \]
\[ \alpha(\text{next}(p, [X \rightarrow 1, Y \rightarrow -2])) = \alpha(q, [X \rightarrow 1, Y \rightarrow -1]) = (q, [X \rightarrow +, Y \rightarrow -]) \neq \]

AbState \(\xrightarrow{\text{next}}\) AbState
\[ \alpha \]
\[ \text{next}(\beta(p, [X \rightarrow 1, Y \rightarrow -2])) = \text{next}(p, [X \rightarrow +, Y \rightarrow -]) = (q, [X \rightarrow +, Y \rightarrow ?]) \]

values computed abstractly are NOT the same as the abstraction of concrete computation results
Relation between concrete & abstract: values computed abstractly can be less precise than abstraction of computation results.

Values computed abstractly conservatively approximate the abstraction of concrete computation results.
Concrete CE machine

1st attempt: CE machine ala SICP-interpreter
concrete state consists of expression and environment in which it is to be evaluated

\[ \varsigma \in \Sigma = \text{Call} \times \text{Env} \]

\[ \rho \in \text{Env} = \text{Var} \rightarrow \text{Clo} \]

\[ \text{clo} \in \text{Clo} = \text{Lam} \times \text{Env} \]

BUT: environments are recursive structure,
implying state space remains infinite even after abstracting
Concrete CES machine: state space

CES-machine: indirect lookup from variables to addresses & from addresses to values
concrete state consists of expression, environment, and store

\[ \zeta \in \Sigma = \text{Call} \times \text{Env} \times \text{Store} \]
\[ \rho \in \text{Env} = \text{Var} \rightarrow \text{Addr} \]
\[ \text{clo} \in \text{Clo} = \text{Lam} \times \text{Env} \]
\[ \sigma \in \text{Store} = \text{Addr} \rightarrow \text{Clo} \]
\[ a \in \text{Addr} \text{ is an infinite set.} \]
Concrete CES machine: transitions

evaluation of atomic expressions

\[ A : \text{AExp} \times \text{Env} \times \text{Store} \rightarrow \text{Clo} \]

\[ A(v, \rho, \sigma) = \sigma(\rho(v)) \]

\[ A(\text{lam}, \rho, \sigma) = (\text{lam}, \rho) \]

actual state transitions for complex expressions

\[ \llbracket (f \ \alpha_1 \ldots \alpha_n) \rrbracket, \rho, \sigma \Rightarrow (\text{call}, \rho'', \sigma'), \text{ where} \]

\[ \llbracket (\lambda \ (v_1 \ldots v_n) \ \text{call}) \rrbracket, \rho' = A(f, \rho, \sigma) \]

\[ \rho'' = \rho'[v_i \mapsto a_i] \]

\[ \sigma' = \sigma[a_i \mapsto A(\alpha_i, \rho, \sigma)] \]

\[ a_i = \text{alloc}(v_i, \sigma) \]
1st Abstract CES machine: state space

\[ \eta \in \hat{\Sigma} = \text{Call} \times \hat{\text{Env}} \times \hat{\text{Store}} \]
\[ \hat{\rho} \in \hat{\text{Env}} = \text{Var} \rightarrow \hat{\text{Addr}} \]
\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \times \hat{\text{Env}} \]
\[ \hat{\sigma} \in \hat{\text{Store}} = \hat{\text{Addr}} \rightarrow \mathcal{P}(\hat{\text{Clo}}) \]
\[ \hat{a} \in \hat{\text{Addr}} \text{ is a finite set.} \]

SET of values, rather than value! consequence of guaranteeing termination through a finite set of abstract addresses: 1 abstract address represents multiple concrete addresses, have to track all their values

=> non-deterministic lookup, proc application, ...
1st Abstract CES machine: abstraction

component-wise lifting of atomic abstraction function

$$\alpha(call, \rho, \sigma) = (call, \alpha(\rho), \alpha(\sigma))$$

$$\alpha(\rho) = \lambda v. \alpha(\rho(v))$$

$$\alpha(\sigma) = \lambda \hat{a}. \bigsqcup \{ \alpha(\sigma(a)) \}$$

$$\alpha(lam, \rho) = \{(lam, \alpha(\rho))\}$$

evaluation of atomic expressions

$$\hat{A}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(v))$$

$$\hat{A}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\}$$
1st Abstract CES machine: transitions

$\left[\left(f \, \alpha_1 \ldots \alpha_n\right)\right], \hat{\rho}, \hat{\sigma} \leadsto \left(\text{call}, \hat{\rho}''', \hat{\sigma}'\right)$, where

$\left(\left(\lambda \, (v_1 \ldots v_n) \, \text{call}\right)\right], \hat{\rho}' \in \hat{A}(f, \hat{\rho}, \hat{\sigma})$

\[ \hat{\rho}''' = \hat{\rho}'[v_i \mapsto \hat{a}_i] \]

\[ \hat{\sigma}' = \hat{\sigma} \uplus [\hat{a}_i \mapsto \hat{A}(\alpha_i, \hat{\rho}, \hat{\sigma})] \]

\[ \hat{a}_i = \text{alloc}(v_i, \hat{\sigma}) \]

eval-to-apply has become non-deterministic (set membership)

store extension has become store join: abstract address in store is (re)used for the binding of a formal parameter to an argument value

which address is allocated for the value of the argument can determine whether the analysis is mono-variant or poly-variant: whether argument values from different invocations of the same function are merged or kept separate!

poly-variant analyses can prevent cross flow between different invocations of the same function:

BUT need a way to distinguish invocations of the same function in different contexts (see previous lecture, e.g., top-of-stack call strings)
Abstract CES machine: transitions

\[ ((f \, \alpha_1 \ldots \alpha_n) \] \hat{\xi}, \hat{\rho}, \hat{\sigma}, \hat{t} \) \rightsquigarrow (\text{call}, \hat{\rho}''', \hat{\sigma}'', \hat{t}'') \text{, where} \\
\hat{\text{clo}} \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \\
\hat{\text{clo}} = ((\lambda (v_1 \ldots v_n) \text{ call}) \] \hat{\rho}' \text{) = } \hat{\text{clo}} \\
\hat{\rho}''' = \hat{\rho}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(\alpha_i, \hat{\rho}, \hat{\sigma})] \\
\text{Context-sensitivity: } t' = \hat{\text{tick}}(\hat{\text{clo}}, \hat{\xi}) \\
\text{Polyvariance: } \hat{a}_i = \hat{\text{alloc}}(v_i, \hat{t}', \hat{\text{clo}}, \hat{\xi}) \\

\text{states now have an additional abstract time component: each state} \\
\text{transition has to augment the time through a tick function} \\
\text{addresses can represent bindings for variables at a particular time} \\
\text{(i.e., in a particular context denoted by the new arguments)}
Abstract CES machine: state space

\[ \xi \in \hat{\Sigma} = \text{Call} \times \hat{\text{Env}} \times \hat{\text{Store}} \times \hat{\text{Time}} \]

\[ \hat{\rho} \in \hat{\text{Env}} = \text{Var} \rightarrow \text{Addr} \]

\[ \hat{\text{clo}} \in \hat{\text{Clo}} = \text{Lam} \times \hat{\text{Env}} \]

\[ \hat{\sigma} \in \hat{\text{Store}} = \hat{\text{Addr}} \rightarrow \mathcal{P} (\hat{\text{Clo}}) \]

\[ \hat{a} \in \hat{\text{Addr}} \text{ is a finite set.} \]

\[ \hat{t} \in \hat{\text{Time}} \text{ is a finite set.} \]
Abstract CES machine: k-CFA

abstract:
  time is k-most recent call strings

\( \hat{\text{tick}}(\hat{\text{clo}}, (\text{call}, \ldots, \hat{t})) = [\text{call} : \hat{t}]_k \)

addresses represent binding for variable at particular time

\( \hat{\text{alloc}}(v_i, \hat{t}, \hat{\text{clo}}, \hat{\zeta}) = (v_i, \hat{t}) \)
R5RS Implementation: states

by Matthew Might: http://matt.might.net/articles/implementation-of-kcfa-and-0cfa/

;; Abstract state-space.

;; state ::= (<call> <benv> <store> <time>)

(define (make-state call benv store time)
  (list call benv store time))

(define (state->call state) (car state))
(define (state->benv state) (cadr state))
(define (state->store state) (caddr state))
(define (state->time state) (cadddr state))

(define (explode-state state k)
  (k (state->call state)
      (state->benv state)
      (state->store state)
      (state->time state)))
R5RS Implementation: environment

;; benv = alist[var,addr]
;; A binding environment maps variables to addresses.

; benv-lookup : benv var -> addr
(define (benv-lookup benv var)
  (let ((entry (assq var benv)))
    (if entry
      (cadr entry)
      (begin (display "No value for ")
              (display var)
              (display " in ")
              (display benv)
              (newline)
              (error "Couldn't look up variable!")))))

; benv-extend : benv var addr -> benv
(define (benv-extend benv var addr)
  (cond
   ((null? benv) (list (list var addr)))
   ((var<=? var (car (car benv))) (cons (car benv)
                                        (benv-extend (cdr benv) var addr)))
   ((var<=? (car (car benv)) var) (cons (list var addr)
                                        benv))
   (else (cons (list var addr) (cdr benv))))

; benv-extend* : benv list[var] list[addr] -> benv
(define (benv-extend* benv vars addrs)
  (if (and (pair? vars) (pair? addrs))
    (benv-extend* (benv-extend benv (car vars) (car addrs))
                   (cdr vars)
                   (cdr addrs))
    benv))
R5RS Implementation: store

;; store = alist[addr,d]
;; A store (or a heap/memory) maps address to denotable values.

; store-insert : store addr d -> store
(define (store-insert store addr d)
  (if (not (pair? store))
      (list (list addr d))
      (if (equal? (car (car store)) addr)
          (cons (list addr d) (cdr store))
          (cons (car store) (store-insert (cdr store) addr d))))

; store-lookup : store addr -> d
(define (store-lookup store addr)
  (let ((entry (assoc addr store)))
    (if entry (cadr entry) '()))

; store-update : store addr d -> store
(define (store-update store addr value)
  (let ((d (store-lookup store addr)))
    (store-insert store addr (d-join d value))))

; store-update*: store list[addr] list[d] -> store
(define (store-update* store addr\$s values)
  (if (or (not (pair? addr\$s)) (not (pair? values)))
      store
      (store-update* (store-update store (car addr\$s) (car values))
                      (cdr addr\$s)
                      (cdr values)))))

; store-join : store store -> store
(define (store-join store1 store2)
  (unzip-k store2 (lambda (addr\$s values)
                       (store-update* store1 addr\$s values)))))
R5RS Implementation: polyvariance and context-sensitivity

;; bind ::= (binding <var> <time>)
;; A binding is minted each time a variable gets bound to a value.

(define (binding? a)
  (and (pair? a) (eq? (car a) 'binding)))
(define (binding->var binding)
  (cadr binding))

;; time = lab^k
;; In k-CFA, time is a bounded memory of program history.
;; In particular, it is the last k call sites through which
;; the program has traversed.

;; k-CFA parameters

;; Change these to alter the behavior of the analysis.

; k : natural
(define k 1)

; tick : call time -> time
(define (tick call time)
  (take k (cons (call->lab call) time)))

; alloc : time -> var -> addr
(define (alloc time)
  (lambda (var)
    (list 'binding var time)))
R5RS Implementation: transitions

; k-CFA abstract interpreter

atom-eval : benv store -> exp -> d
define (atom-eval benv store)
  (lambda (exp)
    (cond
      ((ref? exp) (store-lookup store (benv-lookup benv (ref->var exp))))
      ((lambda? exp) (list (list 'closure exp benv)))
      (else (display exp) (error "unknown expression type: " exp)))))

next : state -> set[state]
define (next state)
  (explode-state state (lambda (call benv store time)
    (if (not (call? call)) '()
      (let* ((time* (tick call time)))
        (explode-call call (lambda (lab f args)
          (let* ((procs ((atom-eval benv store) f))
            (params (map (atom-eval benv store) args)))
            (map (lambda (proc)
              (cond
                ((closure? proc)
                  (let* ((lam (closure->lambda proc))
                    (benv* (closure->benv proc)))
                    (let* ((formals (lambda->formals lam))
                      (call* (lambda->call lam))
                      (bindings (map (alloc time*) formals))
                      (benv** (benv-extend* benv* formals bindings))
                      (store* (store-update* store bindings params)))
                      (make-state call* benv** store* time*)))))
            procs))))))))

reached end state
record call site in new time
evaluate proc and args in current env/store
update store with new bindings for formals at current time
non-determinism
R5RS Implementation: state crawling

;; State-space exploration.

; explore : set[state] list[state] -> set[state]
(define (explore seen todo)
  (cond
   ((null? todo)
    seen)
   ((set-member? (car todo) seen)
    (explore seen (cdr todo)))
   (else
    (let ((succs (next (car todo))))
      (explore (cons (car todo) seen)
               (append succs todo))))))

; summarize : set[state] -> store
(define (summarize states)
  (if (not (pair? states))
    '()
    (store-join (state->store (car states))
                (summarize (cdr states))))))

> summary
((binding id (18)) ((closure (lambda 5 (x k) (call 4 (ref 2 k) (ref 3 x))) ()))
 ((binding x (16)) ((closure (lambda 8 (z) (ref 7 z)) ((id (binding id (18)))))))
 ((binding k (16))
  ((closure
    (lambda 15 (a) (call 14 (ref 9 id) (lambda 11 (y) (ref 10 y)) (lambda 13 (b) (ref 12 b))))
     ((id (binding id (18)))))))
 ((binding a (4)) ((closure (lambda 8 (z) (ref 7 z)) ((id (binding id (18)))))))
 ((binding x (14)) ((closure (lambda 11 (y) (ref 10 y)) ((id (binding id (18)))) (a (binding a (4)))))
  ((binding k (14)) ((closure (lambda 13 (b) (ref 12 b)) ((id (binding id (18)))) (a (binding a (4)))))))
 ((binding b (4)) ((closure (lambda 11 (y) (ref 10 y)) ((id (binding id (18)))) (a (binding a (4)))))
  ()))
Results for Example Program

```scheme
(let ((id (lambda (x k) (k x))))
  (id (lambda (z) z)
      (lambda (a) (id (lambda (y) y)
                      (lambda (b) b)))))) 2)
```

2

monovariant summary of set of reachable states, printed as map from variables to lambda expressions that might flow to them

Welcome to DrRacket, version 6.1.1.1--2014-10-21(3b006df/a) [3m].
Language: R5RS; memory limit: 128 MB.

```
((x ((lambda 11 (y) (ref 10 y)) (lambda 8 (z) (ref 7 z))))
 (k
  ((lambda 15 (a) (call 14 (ref 9 id) (lambda 11 (y) (ref 10 y)) (lambda 13 (b) (ref 12 b)))
   (lambda 13 (b) (ref 12 b))))
 (id ((lambda 5 (x k) (call 4 (ref 2 k) (ref 3 x))))
 (b ((lambda 11 (y) (ref 10 y))))
 (a ((lambda 8 (z) (ref 7 z))))))
```

multiple lambda terms flow to k

but no cross-flow between results of different id invocations
Only the tip of the iceberg!

Applications: analysis for JavaScript (Jens), analysis for concurrent Scheme (Quentin), ...

Optimizations: abstract counting, abstract garbage collection, ...

See https://raw.githubusercontent.com/jensnicolay/aac/master/ds/ceske.rkt for educational K-CFA of direct-style Scheme with type lattice

=> abstract machine is written in CPS itself, using an additional continuation component for the abstract states (which, analogous to the problem with closures, has to be store-allocated to guarantee termination) ... however, exposes state transitions to more non-determinism.

(match s
  ((ev (?) symbol? x) ρ σ (cons φ κ) Ξ)
    (let ((v (store-lookup σ (env-lookup ρ x))))
      (set (ko φ v σ κ Ξ))))
  ((ev `(lambda ,x ,es ...) ρ σ (cons φ κ) Ξ)
    (set (ko φ (α (clo (lam x es) ρ)) σ κ Ξ))
  ((ev `(quote ,e) ρ σ (cons φ κ) Ξ)
    (set (ko φ e σ κ Ξ))
  ((ev (and `(if ,e0 ,e1 ,e2) e) ρ σ κ Ξ)
    (let ((τ (ctx e ρ σ))
      (set (ev e0 ρ σ (cons (ifk e1 e2 ρ) τ) (stack-alloc Ξ τ κ))))
  ((ev (and `(letrec ((,x ,e0)) ,es ...) e) ρ σ κ Ξ)
    (let* ((a (alloc x))
      (ρ* (env-bind ρ x a))
      (τ (ctx e ρ σ))
      (set (ev e0 ρ* σ (cons (letk a es ρ* τ) (stack-alloc Ξ τ κ))))
    ((ev (and `(set!, x ,e0) e) ρ σ κ Ξ))