Scalable Abstractions for Trustworthy Software

David Van Horn
Matthew Might
Purpose

Demonstrate our approach can be *efficient*

Demonstrate our approach can be *modular*

Sketch how we can verify *rich properties*
Semantics → Analysis
Analysis

Semantics

Manifest.xml

Analysis

Manifest.xml
Analysis

Semantics

Compile-time manifest checking

Analysis

Manifest.xml
Analysis

Semantics

Compile-time specification checking

Specification

Analysis

Specification
Analysis

Semantics

Compile-time specification checking

Has to be:
- precise
- fast
- scalable to rich specifications
Efficiency
Analysis

Semantics

Analysis
But is it fast?
Good news: it’s *blazingly* fast...
Good news: it’s *blazingly* fast... ...to implement.
Good news: it’s *blazingly* fast... ...*to implement.*

Good news (for people who love bad news): it’s *dog* slow *to run.*
Good news: it’s *blazingly* fast... ...*to implement.*

Good news (for people who love bad news): it’s *dog slow* *to run.*

Good news (for people who love good news): we can make it faster, systematically.
Expressions

\[ e = \text{var} (x) \]
| \[ \text{lit} (l) \]  
| \[ \text{lam} (x,e) \]  
| \[ \text{app} (e,e) \]  
| \[ \text{if} (e,e,e) \]  

Variables

\[ x = x | y | \ldots \]

Literals

\[ l = z | b | o \]

Integers

\[ z = 0 | 1 | -1 | \ldots \]

Booleans

\[ b = \text{tt} | \text{ff} \]

Operations

\[ o = \text{zero?} | \text{add1} | \text{sub1} | \ldots \]
Expressions

\[ e = \text{var}(x) \]
| \[ \text{lit}(l) \]
| \[ \text{lam}(x,e) \]
| \[ \text{app}(e,e) \]
| \[ \text{if}(e,e,e) \]

Variables

\[ x = x \mid y \mid \ldots \]

Literals

\[ l = z \mid b \mid o \]

Integers

\[ z = 0 \mid 1 \]

Booleans

\[ b = \text{tt} \mid \text{ff} \]

Operations

\[ o = \text{zero?} \mid \text{add} \]

It’s just Core Java
\[
eval(e) = \{ \varsigma \mid \text{ev } (e, \emptyset, \emptyset, \text{mt}) \mapsto \varsigma \}
\] where

\[
\text{ev } (\text{var } (x), \rho, \sigma, \kappa) \mapsto \text{co } (\kappa, v, \sigma) \text{ where } v \in \sigma(\rho(x))
\]

\[
\text{ev } (\text{lit } (l), \rho, \sigma, \kappa) \mapsto \text{co } (\kappa, l, \sigma)
\]

\[
\text{ev } (\text{lam } (x, e), \rho, \sigma, \kappa) \mapsto \text{co } (\kappa, \text{clos } (x, e, \rho), \sigma)
\]

\[
\text{ev } (\text{app } (e_0, e_1), \rho, \sigma, \kappa) \mapsto \text{ev } (e_0, \rho, \sigma', \text{ar } (e_1, \rho, a)) \text{ where } a, \sigma' = \text{push } (\sigma, \kappa)
\]

\[
\text{ev } (\text{if } (e_0, e_1, e_2), \rho, \sigma, \kappa) \mapsto \text{ev } (e_0, \rho, \sigma', \text{fi } (e_1, e_2, \rho, a)) \text{ where } a, \sigma' = \text{push } (\sigma, \kappa)
\]

\[
\text{co } (\text{mt}, v, \sigma) \mapsto \text{ans } (\sigma, v)
\]

\[
\text{co } (\text{ar } (e, \rho, a), v, \sigma) \mapsto \text{ev } (e, \rho, \sigma, \text{fn } (v, a))
\]

\[
\text{co } (\text{fn } (u, a), v, \sigma) \mapsto \text{ap } (v, u, \kappa, \sigma) \text{ where } \kappa \in \sigma(a)
\]

\[
\text{co } (\text{fi } (e_0, e_1, \rho, a), \text{tt}, \sigma) \mapsto \text{ev } (e_0, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a)
\]

\[
\text{co } (\text{fi } (e_0, e_1, \rho, a), \text{ff}, \sigma) \mapsto \text{ev } (e_1, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a)
\]

\[
\text{ap } (\text{clos } (x, e, \rho), v, \sigma, \kappa) \mapsto \text{ev'} (e, \rho', \sigma', \kappa) \text{ where } \rho', \sigma', ' = \text{bind } (\sigma, x, v)
\]

\[
\text{ap } (o, v, \sigma, \kappa) \mapsto \text{co } (\kappa, v', \sigma) \text{ where } \kappa \in \sigma(a) \text{ and } v' \in \Delta(o, v)
\]
Arbiter of context sensitivity

Arbiter of polyvariance
Interpreter:

\[
push(\ell, \sigma, \kappa) = a, \sigma \sqcup [a \mapsto \{\kappa\}] \text{ where } a \notin \sigma
\]
\[
bind(\sigma, x, v) = \rho[x \mapsto a], \sigma \sqcup [a \mapsto \{v\}] \text{ where } a \notin \sigma
\]

Abstract interpreter (0CFA):

\[
push(\ell, \sigma, \kappa) = \ell, \sigma \sqcup [\ell \mapsto \{\kappa\}]
\]
\[
bind(\sigma, x, v) = \rho[x \mapsto x], \sigma \sqcup [x \mapsto \{v\}]
\]
\( \text{eval}(e) = \{ \zeta \mid \text{ev}(e, \emptyset, \emptyset, \text{mt}) \mapsto \zeta \} \) where

\[
\begin{align*}
\text{ev}(\text{var}(x), \rho, \sigma, \kappa) & \mapsto \text{co}(\kappa, v, \sigma) \text{ where } v \in \sigma(\rho(x)) \\
\text{ev}(\text{lit}(l), \rho, \sigma, \kappa) & \mapsto \text{co}(\kappa, l, \sigma) \\
\text{ev}(\text{lam}(x, e), \rho, \sigma, \kappa) & \mapsto \text{co}(\kappa, \text{clos}(x, e, \rho), \sigma) \\
\text{ev}(\text{app}(e_0, e_1), \rho, \sigma, \kappa) & \mapsto \text{ev}(e_0, \rho, \sigma', \text{ar}(e_1, \rho, a)) \text{ where } a, \sigma' = \text{push}(\sigma, \kappa) \\
\text{ev}(\text{if}(e_0, e_1, e_2), \rho, \sigma, \kappa) & \mapsto \text{ev}(e_0, \rho, \sigma', \text{fi}(e_1, e_2, \rho, a)) \text{ where } a, \sigma' = \text{push}(\sigma, \kappa) \\
\text{co}(\text{mt}, v, \sigma) & \mapsto \text{ans}(\sigma, v) \\
\text{co}(\text{ar}(e, \rho, a), v, \sigma) & \mapsto \text{ev}(e, \rho, \sigma, \text{fn}(v, a)) \\
\text{co}(\text{fn}(u, a), v, \sigma) & \mapsto \text{ap}(v, u, \kappa, \sigma) \text{ where } \kappa \in \sigma(a) \\
\text{co}(\text{fi}(e_0, e_1, \rho, a), \text{tt}, \sigma) & \mapsto \text{ev}(e_0, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a) \\
\text{co}(\text{fi}(e_0, e_1, \rho, a), \text{ff}, \sigma) & \mapsto \text{ev}(e_1, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a) \\
\text{ap}(\text{clos}(x, e, \rho), v, \sigma, \kappa) & \mapsto \text{ev}'(e, \rho', \sigma', \kappa) \text{ where } \rho', \sigma', ' = \text{bind}(\sigma, x, v) \\
\text{ap}(\text{o}, v, \sigma, \kappa) & \mapsto \text{co}(\kappa, v', \sigma) \text{ where } \kappa \in \sigma(a) \text{ and } v' \in \Delta(o, v)
\end{align*}
\]
;; State → Set of State
(define (step state)
  (match state
    [(ev σ e ρ k)
      (match e
        [(var x) (for/set ((v (lookup ρ σ x))) (co σ k (v)))]
        [(lit l) (set (co σ k n))]
        [(lam x e) (set (co σ k (clos x e ρ)))]
        [(app f e)
         (define-values (σ* a) (push state))
         (set (ev σ* f ρ (ar e ρ a)))]
        [(ife e0 e1 e2)
         (define-values (σ* a) (push state))
         (set (ev σ* e0 ρ (ifk e1 e2 ρ a))))]]
    [(co σ k v)
      (match k
        ['mt (set (ans σ v))]
        [(ar e ρ) (set (ev σ e ρ (fn v l)))]
        [(fn f) (for/set ((k (get-cont σ l))) (ap σ f v k))]
        [(fi c a ρ)
         (for/set ((k (get-cont σ l)))
               (ev σ (if v c a) ρ k))]]
    [(ap σ fun a k)
      (match fun
        [(clos l x e ρ)
         (define-values (ρ* σ*) (bind state))
         (set (ev σ* e ρ* k))]
        [(? op? o)
         (for*/set ((k (get-cont σ l))
                    (v (Δ o (list v)))
                    (co σ k v))
         [_ (set)]))])]}
Generic fixpoint calculator

;; appl : (∀ (X) ((X -> (Setof X)) -> ((Setof X) -> (Setof X))))
(define ((appl f) s)
  (for/fold ([i (set)])
    ([x (in-set s)])
    (set-union i (f x))))

;; Calculate fixpoint of (appl f).
;; fix : (∀ (X) ((X -> (Setof X)) (Setof X) -> (Setof X)))
(define (fix f s)
  (let loop ((accum (set)) (front s))
    (if (set-empty? front)
        accum
        (let ((new-front ((appl f) front)))
          (loop (set-union accum front)
                (set-subtract new-front accum))))))
\[\text{Expr} \rightarrow \text{Setof State} \]
\[
(\text{define (eval e)}
\quad (\text{fix step (set (inj e))))
)\]
;; multiplication distributes over addition

((church=? ((mult \[2\]) ((plus \[1\] \[3\])))
  ((plus ((mult \[2\] \[1\])) ((mult \[2\] \[3\]))))))
(define `2\( \lambda \) (f) (\lambda (x) (f (f x))))
;; multiplication distributes over addition
((church=\((\text{mult}\ 2\ (\text{plus}\ 1\ 3))\))
((\text{plus}\ (\text{mult}\ 2\ 1))\ (\text{mult}\ 2\ 3))))

\[
(\text{define}\ \overline{2}^\backslash (\lambda\ (f)\ (\lambda\ (x)\ (f\ (f\ x))))))
\]

interface Function<X,Y> { Y apply(X x); }

class Two<X,X> implements Function<Function<X,X>,Function<X,X>> {
    apply(final Function<X,Y> f) = {
        return new Function<X,X>() {
            X apply(X x) {
                return f.apply(f.apply(x));
            }
        };
    }
}

// (\lambda\ (f)\ (\lambda\ (x)\ (f\ (f\ x))))
;; multiplication distributes over addition
((church=? ((mult `2`) ((plus `1`) `3`)))
 ((plus ((mult `2`) `1`) ((mult `2`) `3`))))

(define pred
  (λ (n)
    (λ (rf)
      (λ (rx)
        (((n (λ (g) (λ (h) (h (g rf))))))
         (λ (i) rx))
       (λ (id) id))))))
;; multiplication distributes over addition
((church=? ((mult [2]) ((plus [1] [3]))))
((plus ((mult [2] [1])) ((mult [2] [3]))))))
;; multiplication distributes over addition
((church=? ((mult [2]) ((plus [1] [3])))
((plus ((mult [2] [1])) ((mult [2] [3]))))))

Time: ∞
\{<E_1, R_1, K_1>, <E_1, R_1, K_1>, <E_2, R_2, K_2>\},
[0 \mapsto \{8\}, 1 \mapsto \{3, 6\}]
Generic store widening

;; State^ = (cons (Set Conf) Store)

;; (State -> Setof State) -> State^ -> { State^ }
(define ((wide-step step) state)
  (match state
    [(cons cs σ)
      (define ss ((appl step)
        (for/set ([c cs]) (c->s c σ)))]
      (set (cons (for/set ([s ss]) (s->c s))
        (join-stores ss))))))
Generic store widening

;;; State\(^{\wedge}\) = (cons (Set Conf) Store)

;;; (State -> Setof State) -> State\(^{\wedge}\) -> \{ State\(^{\wedge}\} 
(define ((wide-step step) state)
  (match state
    [(cons cs σ)
     (define ss ((appl step)
       (for/set ([c cs]) (c->s c σ))))
     (set (cons (for/set ([s ss]) (s->c s))
       (join-stores ss))))]))

Time: 551571ms (≈9.2m)
Precision Preserving Recipe
Lazy non-determinism

\[\text{eval}(e) = \{\varsigma \mid \text{ev}(e, \emptyset, \emptyset, \text{mt}) \Rightarrow \varsigma\} \text{ where }\]

\[
\begin{align*}
\text{ev}(\text{var}(x), \rho, \sigma, \kappa) &\Rightarrow \text{co}(\kappa, v, \sigma) \text{ where } v \in \sigma(\rho(x)) \\
\text{ev}(\text{lit}(l), \rho, \sigma, \kappa) &\Rightarrow \text{co}(\kappa, l, \sigma) \\
\text{ev}(\text{lam}(x, e), \rho, \sigma, \kappa) &\Rightarrow \text{co}(\kappa, \text{clos}(x, e, \rho), \sigma) \\
\text{ev}(\text{app}(e_0, e_1), \rho, \sigma, \kappa) &\Rightarrow \text{ev}(e_0, \rho, \sigma', \text{ar}(e_1, \rho, a)) \text{ where } a, \sigma' = \text{push}(\sigma, \kappa) \\
\text{ev}(\text{if}(e_0, e_1, e_2), \rho, \sigma, \kappa) &\Rightarrow \text{ev}(e_0, \rho, \sigma', \text{fi}(e_1, e_2, \rho, a)) \text{ where } a, \sigma' = \text{push}(\sigma, \kappa) \\
\text{co}(\text{mt}, v, \sigma) &\Rightarrow \text{ans}(\sigma, v) \\
\text{co}(\text{ar}(e, \rho, a), v, \sigma) &\Rightarrow \text{ev}(e, \rho, \sigma, \text{fn}(v, a)) \\
\text{co}(\text{fn}(u, a), v, \sigma) &\Rightarrow \text{ap}(v, u, \kappa, \sigma) \text{ where } \kappa \in \sigma(a) \\
\text{co}(\text{fi}(e_0, e_1, \rho, a), \text{tt}, \sigma) &\Rightarrow \text{ev}(e_0, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a) \\
\text{co}(\text{fi}(e_0, e_1, \rho, a), \text{ff}, \sigma) &\Rightarrow \text{ev}(e_1, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a) \\
\text{ap}(\text{clos}(x, e, \rho), v, \sigma, \kappa) &\Rightarrow \text{ev}'(e, \rho', \sigma', \kappa) \text{ where } \rho', \sigma', '=' = \text{bind}(\sigma, x, v) \\
\text{ap}(o, v, \sigma, \kappa) &\Rightarrow \text{co}(\kappa, v', \sigma) \text{ where } \kappa \in \sigma(a) \text{ and } v' \in \Delta(o, v)
\end{align*}
\]
Lazy non-determinism

\[
\text{eval}(e) = \{ \varsigma \mid \text{ev } (e, \emptyset, \emptyset, \text{mt}) \mapsto \varsigma \}\text{ where}\]

\[
\text{ev } (\text{var } (x), \rho, \sigma, \kappa) \mapsto \text{co } (\kappa, v, \sigma) \text{ where } v \in \sigma(\rho(x))
\]

\[
\text{ev } (\text{lit } (l), \rho, \sigma, \kappa) \mapsto \text{co } (\kappa, l, \sigma)
\]

\[
\text{ev } (\text{lam } (x, e), \rho, \sigma, \kappa) \mapsto \text{co } (\kappa, \text{clos } (x, e, \rho), \sigma)
\]

\[
\text{ev } (\text{app } (e_0, e_1), \rho, \sigma, \kappa) \mapsto \text{ev } (e_0, \rho, \sigma', \text{ar } (e_1, \rho, a)) \text{ where } a, \sigma' = \text{push } (\sigma, \kappa)
\]

\[
\text{ev } (\text{if } (e_0, e_1, e_2), \rho, \sigma, \kappa) \mapsto \text{ev } (e_0, \rho, \sigma', \text{fi } (e_1, e_2, \rho, a)) \text{ where } a, \sigma' = \text{push } (\sigma, \kappa)
\]

\[
\text{co } (\text{mt}, v, \sigma) \mapsto \text{ans } (\sigma, v)
\]

\[
\text{co } (\text{ar } (e, \rho, a), v, \sigma) \mapsto \text{ev } (e, \rho, \sigma, \text{fn } (v, a))
\]

\[
\text{co } (\text{fn } (u, a), v, \sigma) \mapsto \text{ap } (v, u, \kappa, \sigma) \text{ where } \kappa \in \sigma(a)
\]

\[
\text{co } (\text{fi } (e_0, e_1, \rho, a), \text{tt}, \sigma) \mapsto \text{ev } (e_0, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a)
\]

\[
\text{co } (\text{fi } (e_0, e_1, \rho, a), \text{ff}, \sigma) \mapsto \text{ev } (e_1, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a)
\]

\[
\text{ap } (\text{clos } (x, e, \rho), v, \sigma, \kappa) \mapsto \text{ev ' } (e, \rho', \sigma', \kappa) \text{ where } \rho', \sigma', ' = \text{bind } (\sigma, x, v)
\]

\[
\text{ap } (o, v, \sigma, \kappa) \mapsto \text{co } (\kappa, v', \sigma) \text{ where } \kappa \in \sigma(a) \text{ and } v' \in \Delta(o, v)
\]
Lazy non-determinism

\[ \text{eval}(e) = \{ \varsigma \mid \text{ev } (e, \emptyset, \emptyset, \text{mt}) \to\!	o \varsigma \} \text{ where} \]

\[ \text{ev } (\text{var } (x), \rho, \sigma, \kappa) \to\!	o \text{co } (\kappa, v, \sigma) \text{ where } v \in \sigma(\rho(x)) \]

\[ \text{ev } (\text{lit } (l), \rho, \sigma, \kappa) \to\!	o \text{co } (\kappa, l, \sigma) \]

\[ \text{ev } (\text{lam } (x, e), \rho, \sigma, \kappa) \to\!	o \text{co } (\kappa, \text{clos } (x, e, \rho), \sigma) \]

\[ \text{ev } (\text{app } (e_0, e_1), \rho, \sigma, \kappa) \to\!	o \text{ev } (e_0, \rho, \sigma', \text{ar } (e_1, \rho, a)) \text{ where } a, \sigma' = \text{push } (\sigma, \kappa) \]

\[ \text{ev } (\text{if } (e_0, e_1, e_2), \rho, \sigma, \kappa) \to\!	o \text{ev } (e_0, \rho, \sigma', \text{fi } (e_1, e_2, \rho, a)) \text{ where } a, \sigma' = \text{push } (\sigma, \kappa) \]

\[ \text{co } (\text{mt}, v, \sigma) \to\!	o \text{ans } (\sigma, v) \]

\[ \text{co } (\text{ar } (e, \rho, a), v, \sigma) \to\!	o \text{ev } (e, \rho, \sigma, \text{fn } (v, a)) \]

\[ \text{co } (\text{fn } (u, a), v, \sigma) \to\!	o \text{ap } (v, u, \kappa, \sigma) \text{ where } \kappa \in \sigma(a) \]

\[ \text{co } (\text{fi } (e_0, e_1, \rho, a), \text{tt}, \sigma) \to\!	o \text{ev } (e_0, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a) \]

\[ \text{co } (\text{fi } (e_0, e_1, \rho, a), \text{ff}, \sigma) \to\!	o \text{ev } (e_1, \rho, \sigma, \kappa) \text{ where } \kappa \in \sigma(a) \]

\[ \text{ap } (\text{clos } (x, e, \rho), v, \sigma, \kappa) \to\!	o \text{ev } ' (e, \rho', \sigma', \kappa) \text{ where } \rho', \sigma', ' = \text{bind } (\sigma, x, v) \]

\[ \text{ap } (o, v, \sigma, \kappa) \to\!	o \text{co } (\kappa, v', \sigma) \text{ where } \kappa \in \sigma(a) \text{ and } v' \in \Delta(o, v) \]
;; multiplication distributes over addition
((church=? ((mult 2) ((plus 1 3)))
((plus ((mult 2 1)) ((mult 2 3))))))

Time: 551571ms (≈9.2m)
;; multiplication distributes over addition
((church=? ((mult 2) ((plus 1) 3)))
 ((plus ((mult 2) 1) ((mult 2) 3))))

Time: 551571ms (≈9.2m)

Time: 255397ms (≈4.3m)
\[
ev (\text{app} (\text{app} (\text{app} (x, e_1), e_2), e_3), \rho, \kappa, \sigma_0)
\]
\[
\quad \rightarrow \ev (\text{app} (\text{app} (x, e_1), e_2), \rho, \text{ar} (e_3, \rho, a_1), \sigma_1)
\]
\[
\quad \rightarrow \ev (\text{app} (x, e_1), \rho, \text{ar} (e_2, \rho, a_2), \sigma_2)
\]
\[
\quad \rightarrow \ev (x, \rho, \text{ar} (e_1, \rho, a_3), \sigma_3)
\]
\[
\quad \rightarrow \text{co} (\text{ar} (e_1, \rho), v, \sigma_4) \text{ where } v \in \sigma (\rho (a))
\]
\[
\text{ev (app (app (app } x, e_1, e_2), e_3), \rho, \kappa, \sigma_0)
\]
\[
\quad\rightarrow \text{ev (app (app } x, e_1, e_2), \rho, \text{ar } (e_3, \rho, a_1), \sigma_1)
\]
\[
\quad\rightarrow \text{ev (app } x, e_1, \rho, \text{ar } (e_2, \rho, a_2), \sigma_2)
\]
\[
\quad\rightarrow \text{ev } (x, \rho, \text{ar } (e_1, \rho, a_3), \sigma_3)
\]
\[
\quad\rightarrow \text{co } (\text{ar } (e_1, \rho), v, \sigma_4) \text{ where } v \in \sigma(\rho(a))
\]

\[
\llbracket \text{var } (x) \rrbracket = \lambda(\rho, \sigma, \kappa).\text{co } (\kappa, v, \sigma) \text{ where } v \in \sigma(\rho(x))
\]
\[
\llbracket \text{lit } (l) \rrbracket = \lambda(\rho, \sigma, \kappa).\text{co } (\kappa, l, \sigma)
\]
\[
\llbracket \text{lam } (x, e) \rrbracket = \lambda(\rho, \sigma, \kappa).\text{co } (\kappa, \text{clos } (x, \llbracket e \rrbracket, \rho), \sigma)
\]
\[
\llbracket \text{app } (e_0, e_1) \rrbracket = \lambda (\rho, \sigma, \kappa).\llbracket e_0 \rrbracket (\rho, \sigma’, \text{ar } (\llbracket e_1 \rrbracket, \rho, a)) \text{ where } a, \sigma’ = \text{push } (\sigma, \kappa)
\]
\[
\llbracket \text{if } (e_0, e_1, e_2) \rrbracket = \lambda (\rho, \sigma, \kappa).\llbracket e_0 \rrbracket^\delta (\rho, \sigma’, \text{fi } (\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \rho, a)) \text{ where } a, \sigma’ = \text{push } (\sigma, \kappa)
\]
;; multiplication distributes over addition
((church=? ((mult 2) ((plus 1 3)))
  ((plus ((mult 2 1)) ((mult 2 3)))))

Time: 551571ms (≈9.2m)
Time: 255397ms (≈4.3m)
;; multiplication distributes over addition
((church=? ((mult [2]) ((plus [1] [3])))
 ((plus ((mult [2] [1])) ((mult [2] [3]))))))

Time: 551571ms (≈9.2m)
Time: 255397ms (≈4.3m)
Time: 31173ms (≈.5m)
Specialized fixpoint computation

;; State^ -> State^  
;; Specialized from wide-step : State^ -> State^  ≈ State^ -> State^  
(define (wide-step-specialized state)  
  (match state  
    [(cons σ cs)  
      (define-values (cs* σ*)  
        (for/fold ([cs* (set)] [σ* σ])  
          ([c cs])  
          (match (step-compiled^ (cons σ c))  
            [(cons σ** cs**)  
              (values (set-union cs* cs**) (join-store σ* σ**)))])))]))

Time: 551571ms (≈9.2m)  
Time: 255397ms (≈4.3m)  
Time: 31173ms (≈.5m)
Specialized fixpoint computation

;;; State^ -> State^  
;;; Specialized from wide-step : State^ -> State^ ≈ State^ -> State^  
(define (wide-step-specialized state)  
  (match state  
    [(cons σ cs)  
      (define-values (cs* σ*)  
        (for/fold ([cs* (set)] [σ* σ])  
          ([c cs])  
            (match (step-compiled^ (cons σ c))  
              [(cons σ** cs**)  
                (values (set-union cs* cs**) (join-store σ* σ**))])))  
      (cons σ* (set-union cs cs*))]))

Time: 551571ms (≈9.2m)  
Time: 255397ms (≈4.3m)  
Time: 31173ms (≈.5m)  
Time: 14212ms
Specialized fixpoint computation

;;; State^ -> State^  
;;; Specialized from wide-step : State^ -> State^ ≈ State^ -> State^  
(define (wide-step-specialized state)  
  (match state  
    [(cons σ cs)  
      (define-values (cs* σ*)  
        (for/fold ([cs* (set)] [σ* σ])  
          ([c cs])  
          (match (step-compiled^ (cons σ c))  
            [(cons σ** cs**)  
              (values (set-union cs* cs**) (join-store σ* σ**)))]))])  
Time: 551571ms (≈9.2m)  
Time: 255397ms (≈4.3m)  
Time: 31173ms (≈.5m)  
Time: 14212ms
Computing with store diffs

;; State^ -> State^ 
;; Specialized from wide-step : State^ -> State^ ~ State^ -> State^ 
(define (wide-step-specialized state)
  (match state
    [(cons σ cs)
      (define-values (cs* Δ)
        (for/fold ([cs* (set)] [Δ* '()])
          ([c cs])
          (match (step-compiled^ (cons σ c))
            [(cons Δ** cs**) (values (set-union cs* cs**) (append Δ** Δ*))]))
        (cons (update Δ σ) (set-union cs cs*))])]

Time: 14212ms
Computing with store diffs

;;; State^ -> State^ 
;;; Specialized from wide-step : State^ -> State^ ≈ State^ -> State^ 
(define (wide-step-specialized state)
  (match state
    [(cons σ cs)
      (define-values (cs* Δ)
        (for/fold ([cs* (set)] [Δ* '()])
          ([c cs])
            (match (step-compiled^ (cons σ c))
              [(cons Δ** cs**)]
                (values (set-union cs* cs**) (append Δ** Δ*)))))
      (cons (update Δ σ) (set-union cs cs*))))))
\[ \{ \hat{v}_3, \hat{v}_5, \hat{v}_6 \} = 0001011 \]
\( \{ \hat{v}_3, \hat{v}_5, \hat{v}_6 \} = 00010111 \)
\{\hat{v}_3, \hat{v}_5, \hat{v}_6\} = 0001011

\sigma = \#(0_0, \ldots, 0_{|P|})
\[ \{ \hat{\nu}_3, \hat{\nu}_5, \hat{\nu}_6 \} = 0001011 \]

\[ \sigma = \#(\theta_0, \ldots, \theta_{|P|}) \]
Precision Preserving Recipe

- Lazy non-determinism
- Abstract compilation
- Specialized fixpoint
- Store diffs
- Finite sets as bit vectors
- Pre-allocation
Precision Preserving Recipe

- Lazy non-determinism
- Abstract compilation
- Specialized fixpoint
- Store diffs
- Finite sets as bit vectors
- Pre-allocation

≈ 5000x improvement
Modularity

- Some programs are open
- Good components in bad languages
- Programs are big; analysis is hard
- Libraries matter
Analysis
Think hard about modularity

Analysis
Think hard about modularity
Think hard about modularity
PCF
as shorthand for paper, we treat the non-dependent function contract between a function’s input and result. In the remainder of the

CPCF is equipped with a standard type system for PCF

plus the addition of a contract type

Component labels play an important role in case a con-

f,g

h

\( \lambda X : T.E \) \( V \) \( \mu X : T.E \) \( O(\vec{V}) \)

\( \text{if } \text{tt } E_1 E_2 \) \( \text{if } \text{ff } E_1 E_2 \)

\( \text{if } \text{tt } E_1 E_2 \) \( \text{if } \text{ff } E_1 E_2 \)

\( (\lambda X : T.E) V \) \( \mu X : T.E \)

\( O(\vec{V}) \)

\( \text{if } \text{tt } E_1 E_2 \) \( \text{if } \text{ff } E_1 E_2 \)

\( (\lambda X : T.E) V \) \( \mu X : T.E \)

\( O(\vec{V}) \)

\( [\mu X : T.E / X] E \)

\( [V / X] E \)

\( A \text{ if } \delta(O, \vec{V}) = A \)
Symbolic PCF
Typically, the interpretation of operations is defined by equipping the operational semantics with an interpretation of the abstracting away that value to an unknown should produce known values. Pre-values are refined by a set of contracts to form unknown values have arbitrary behavior, but we refine unsignal with appropriate blame. It does, the value is returned. Otherwise, a contract error is the monitor of a flat contract reduces to the so-called cut. The semantics given below replace that of section 3.2 Branching on symbolic values. The high-level goal of the following semantics is to extend the interpretation of conditionals, e.g., from the set. We write mon that maps an operation and argument values to required, and these cases would largely mimic the existing rule; our dependent contract rule; our dependent contract rule.

For simplicity, we present the so-called C-expression which tests whether the predicate holds. If δ ∈ N, then C[δ] produces an answer X. From here, all that remains is adding appropriate lax/contract labels.
if $\bullet T$ $E_1$ $E_2 \iff E_1$

if $\bullet T$ $E_1$ $E_2 \iff E_2$

$(\bullet T \rightarrow T') V \iff \bullet T'$

$(\bullet T \rightarrow T') V \iff \text{havoc}_T V$
if $\inl{T} E_1 E_2 \leftrightarrow E_1$

if $\inl{T} E_1 E_2 \leftrightarrow E_2$

$(\inl{T \rightarrow T'}) V \leftrightarrow \inl{T'}$

$(\inl{T \rightarrow T'}) V \leftrightarrow \text{havoc}_T V$

$havoc_B = \mu x.x$

$havoc_{T \rightarrow T'} = \lambda x : T \rightarrow T'. \text{havoc}_{T'}(x \inl{T})$
if \( tt \, E_1 \, E_2 \) \( \mapsto \) \( E_1 \)
if \( ff \, E_1 \, E_2 \) \( \mapsto \) \( E_2 \)
\((\lambda X : T. E) \, V \) \( \mapsto \) \([V/X]E\)
\(\mu X : T. E \) \( \mapsto \) \([\mu X : T. E/X]E\)
\(O(V) \) \( \mapsto \) \( A \) if \( \delta(O, \tilde{V}) = A \)
if \( \cdot^T \, E_1 \, E_2 \) \( \mapsto \) \( E_1 \)
if \( \cdot^T \, E_1 \, E_2 \) \( \mapsto \) \( E_2 \)
\((\cdot^{T \to T'}) \, V \) \( \mapsto \) \( T' \)
\((\cdot^{T \to T'}) \, V \) \( \mapsto \) \( \text{havoc}_T \, V \)
Contracts \( C ::= \text{flat}(E) | C \mapsto C | C \mapsto \lambda X : T. C \)

\[
\begin{align*}
\text{mon}_{h}^{f,g}(C_1 \mapsto \lambda X : T. C_2, V) & \mapsto \\
& \lambda X : T. \text{mon}_{h}^{f,g}(C_2, (V \text{mon}_{h}^{g,f}(C_1, X))) \\
\text{mon}_{h}^{f,g}(\text{flat}(E), V) & \mapsto \text{if } (E \ V) \ V \text{ blame}_{h}^{f}
\end{align*}
\]
Symbolic Contract PCF
The main requirement is that the results of running such refinements can guide an operational characterization of a knowns by attaching a set of contracts that specify an agreement to take the values of CPCF as "pre"-values for handling symbolic values.

As an example, the definition includes: the shift from the semantics of section 3. Symbolic PCF with Contracts and (2) applying a symbolic function.

The revised reduction relation reduces an operation, non-deterministically, to any answer in the result when applied to symbolic values, and if it does, the value is returned. Otherwise, a contract error is signaled with appropriate blame.

The high-level goal of the following semantics is to enrich the language with abstracts that map an operation and argument values to allowable value. More precisely, if a program involves some value that approximates the one-step reduction relation must be extended to interpret operations \( C \mapsto \bigcup_{X} V \) as "approximates" that maps an operation and argument values to an answer, e.g.:

\[
\begin{align*}
\text{Values} & \quad V ::= U/\{C, \ldots \} \\
\text{Prevalues} & \quad U ::= \bullet^{T} | \lambda X : T.E | 0 | 1
\end{align*}
\]

We write \( \text{if } C \text{ then } tt \text{ else } ff \) and add a no.

without this slight refactoring for handling symbolic values. Without this slight refactoring for handling symbolic values.

Without this slight refactoring for handling symbolic values.
\[
\begin{align*}
\text{mon}_{h}^{f,g}(C, V) & \quad \iff V \text{ if } \vdash V : C \checkmark \\
\text{mon}_{h}^{f,g}(\text{flat}(E), V) & \quad \iff \\
\text{if } (E V) (V \cdot \text{flat}(E)) & \text{ blame}_{g}^{f} \text{ if } \not\vdash V : \text{flat}(E) \checkmark \\
\text{mon}_{h}^{f,g}(C_1 \mapsto \lambda X : T.C_2, V) & \quad \iff \\
\lambda X : T.\text{mon}_{h}^{f,g}(C_2, V \text{ mon}_{h}^{g,f}(C_1, X)) & \quad \text{if } \not\vdash V : C_1 \mapsto \lambda X : T.C_2 \checkmark
\end{align*}
\]
\[ \text{mon}^{f,g}_{h}(C, V) \iff V \text{ if } \vdash V : C \checkmark \]

\[ \text{mon}^{f,g}_{h}(\text{flat}(E), V) \iff \text{if } (E \cdot V) (V \cdot \text{flat}(E)) \text{ blame}^f_g \text{ if } \not\vdash V : \text{flat}(E) \checkmark \]

\[ \text{mon}^{f,g}_{h}(C_1 \mapsto \lambda X : T.C_2, V) \iff \lambda X : T.\text{mon}^{f,g}_{h}(C_2, V \text{ mon}^{g,f}_{h}(C_1, X)) \text{ if } \not\vdash V : C_1 \mapsto \lambda X : T.C_2 \checkmark \]

\[ C \in C \]

\[ \vdash V/C : C \checkmark \]
Symbolic Core Racket
The definition of $h$ is

$$
\begin{align*}
\forall V. & \quad h(V) \\
\iff & \quad \delta(CD, V) \Rightarrow \text{blame}^f
\end{align*}
$$

The implementation is naive but effective for structural decomposition of contracts and values. The first two are for pair contracts. If the value is determined to be a pair by $\text{cons}$, it is blamed when primitive operations are misused, the violated contract is on $C \times D$, $V$. If $\delta(\text{cons}, V) \Rightarrow \text{tt}$ and $V' = V \cdot \text{flat} \left(\text{cons}\right)$, $\text{blame}^f$. In other cases, $\delta \left(\text{cons}, V\right) \Rightarrow \text{tt}$.

The last set of rules decompose combinations of higher-order contract reductions.

$\Rightarrow \left(\text{flat} \left(\text{cons}\right)\right)\left(\text{cons}(\text{card}, V')\right)$. When abstract functions are applied, there is no abstract function that is always present, and the rules for $\text{cons}$ are given in figure 3.4. If the $\text{cons}$ is on $\left(\text{cons}(\text{card}, V')\right)$ we can blame the $\text{card}$, $\text{V}'$ if $\delta(\text{cons}, V) \Rightarrow \text{tt}$ and $V' \Rightarrow \text{flat} \left(\text{cons}\right)$.

$\text{mon}^f(C \times D, V) \Rightarrow \text{havoc} \left(\text{cons}(\text{card}, V')\right)$. If $\delta(\text{cons}, V) \Rightarrow \text{tt}$ and $V' \Rightarrow \text{flat} \left(\text{cons}\right)$, $\text{mon}^f(C \times D, V) \Rightarrow \text{blame}^f$. Otherwise, $\delta(\text{cons}, V) \Rightarrow \text{tt}$.

$\Rightarrow \left(\text{flat} \left(\text{cons}\right)\right)\left(\text{cons}(\text{card}, V')\right)$. When abstract functions are applied, there is no abstract function that is always present, and the rules for $\text{cons}$ are given in figure 3.4. If the $\text{cons}$ is on $\left(\text{cons}(\text{card}, V')\right)$ we can blame the $\text{card}$, $\text{V}'$ if $\delta(\text{cons}, V) \Rightarrow \text{tt}$ and $V' \Rightarrow \text{flat} \left(\text{cons}\right)$.

$\text{mon}^f(C \times D, V) \Rightarrow \text{havoc} \left(\text{cons}(\text{card}, V')\right)$. If $\delta(\text{cons}, V) \Rightarrow \text{tt}$ and $V' \Rightarrow \text{flat} \left(\text{cons}\right)$, $\text{mon}^f(C \times D, V) \Rightarrow \text{blame}^f$. Otherwise, $\delta(\text{cons}, V) \Rightarrow \text{tt}$.

$\Rightarrow \left(\text{flat} \left(\text{cons}\right)\right)\left(\text{cons}(\text{card}, V')\right)$. When abstract functions are applied, there is no abstract function that is always present, and the rules for $\text{cons}$ are given in figure 3.4. If the $\text{cons}$ is on $\left(\text{cons}(\text{card}, V')\right)$ we can blame the $\text{card}$, $\text{V}'$ if $\delta(\text{cons}, V) \Rightarrow \text{tt}$ and $V' \Rightarrow \text{flat} \left(\text{cons}\right)$.

$\Rightarrow \left(\text{flat} \left(\text{cons}\right)\right)\left(\text{cons}(\text{card}, V')\right)$. When abstract functions are applied, there is no abstract function that is always present, and the rules for $\text{cons}$ are given in figure 3.4. If the $\text{cons}$ is on $\left(\text{cons}(\text{card}, V')\right)$ we can blame the $\text{card}$, $\text{V}'$ if $\delta(\text{cons}, V) \Rightarrow \text{tt}$ and $V' \Rightarrow \text{flat} \left(\text{cons}\right)$.
Interactive verification environment

Interactive step-by-step exploration. The model mode selects completion with a read-eval-print loop, visualizing a directed explored. The choices are simply running the program to compression.

Header, where programs are written in a subset of Racket, consisting of a series of module definitions and a top-level expression. There is a sound approximation mode: for the main expression.

To validate our approach, we have implemented a prototype for contract conjunction; expressing equality with atomic values; one-of/c space explored in practice, including abstract garbage collection. Following rules to model non-recursive functions and non-dependent contracted recursive functions to their contracts on recursion. Let there exist.

Theorem 3

The proofs of these theorems closely follow those given for expressing equality with atomic values; one-of/c space explored in practice, including abstract garbage collection. Following rules to model non-recursive functions and non-dependent contracted recursive functions to their contracts on recursion. Let there exist.

Lemma 7

We have now established any instantiation of the machine to its exact counterpart. Let.

We now relate any approximating variant of the machine to its exact counterpart. Let.

We now establish the correspondence between the previous reduction semantics and the machine model when no approximation occurs. Let.

The key case is on stores: state-space of the exact machine to its approximate counter.

We define the relation that maps a closure and store to the closed term it represents.

We can take the example from section 42. We de-
#lang racket/load

([(string=? ke "s") (world-change-dir w 'down)]
 [(string=? ke "a") (world-change-dir w 'left)]
 [(string=? ke "d") (world-change-dir w 'right)]
 [else w]))

;;; game-over? : World -> Boolean
(define (game-over? w)
  (or (snake-wall-collide? (world-snake w))
      (snake-self-collide? (world-snake w))))

(provide/contract [handle-key (world/c string? . -> . world/c)]
  [game-over? (world/c . -> . boolean?)])

(module snake racket
  (require 2htdp/universe)
  (require 'scenes 'handlers 'motion)
  ;; RUN PROGRAM RUN
  
  ;; World -> World
  (define (start w)
    (big-bang w
      (to-draw world->scene)
      (on-tick world->world 1/2)
      (on-key handle-key)
      (stop-when game-over?)))

  (provide start))

(require 'snake 'const)
(start (WORLD))
#lang racket/load

;; -- Primitive modules
(module image racket
  (require 2htdp/image)
  (provide/contract
   [image? (any/c . -> . boolean?)])
  [circle (exact-nonnegative-integer? string? exact-nonnegative-integer? exact-nonnegative-integer?]
  [empty-scene (exact-nonnegative-integer? exact-nonnegative-integer?)
  [place-image (image? exact-nonnegative-integer?)

;; -- Source
(module snake racket
  (require 2htdp/universe)
  (require 'scenes 'handy)
  (require 'game 'to-draw)
  (require 'key input)
  (require 'game 'self-color)
  (require 'game 'animation)
  (require 'game 'game-over)

;; -- RUN PROGRAM RUN

;; World -> World
(define (start w)
  (big-bang w
    (to-draw
      (on-key (on-tick 'self-color)
        (stop-when
          (define/c direction/c
            (one-of/c 'up 'down 'left 'right))
          (define/c posn/c
            (struct/c posn
              exact-nonnegative-integer?
              exact-nonnegative-integer?)
            
          (define/c snake/c
            (struct/c snake
              direction/c
              (non-empty-listof posn/c))

          (define/c world/c
            (struct/c world
              snake/c
              posn/c))

          (define/c posn=? : Posn Posn -> Boolean
            (define/c (are the posns the same?)

Welcome to DrRacket, version 5.3.1.1--2012-10-13(2b902d0a/d) [3m].
Language: racket/load [custom]; memory limit: 1024 MB.
Behavioral contracts

Specify pre- & post-conditions as predicates

@SafeSocket(url)
Socket openURL(@OnWhiteList(wl) URL url)

class OnWhiteList extends Contract<List<URL>> {
   bool checkContract(List<URL> wl, URL u) {...}
}
Analysis

Semantics

Specification

Compile-time specification checking

Analysis

Specification
Fast design & development times
Fast analysis times
Modular
Handles libraries
Verifies rich properties

Semantics

Compile-time specification checking

Analysis
Thank you

★ Fast design & development times
★ Fast analysis times
★ Modular
★ Handles libraries
★ Verifies rich properties