Higher-order Symbolic Execution via Contracts

Sam Tobin-Hochstadt
David Van Horn
HOW CAN WE...

DO SYMBOLIC EXECUTION OF HIGHER-ORDER PROGRAMS?
How can we...

Make program analysis modular?
How can we...

Verify sophisticated contracts at compile-time?
Problems

How can we...

* do Symbolic Execution of H.O. programs?
* make program analysis modular?
* verify sophisticated contracts at compile-time?

Solution
PROBLEMS

How can we...

* do Symbolic Execution of H.O. programs?
* make program analysis modular?
* verify sophisticated contracts at compile-time?

SOLUTION

Abstract reduction Semantics
Higher-order Symbolic Execution via Contracts
Higher-order Symbolic Execution via Contracts
PCF
(quotient 10 (if0 7 2 0))

(quotient 10 0)

(err "Divide by zero")
\(((\lambda ((x : \text{nat})) \ (\text{quotient} \ 10 \ x))) \ (\text{if}0 \ 7 \ 2 \ 3))\)
(((\lambda ((f : (\text{nat} \to \text{nat})))) (f 3))
(\lambda ((x : \text{nat})) (\text{quotient} 10 x))))

(((\lambda ((x : \text{nat})) (\text{quotient} 10 x)) 3)

(\text{quotient} 10 3)

3
SYMBOLIC PCF
PCF

\{ \lambda ([f : \text{nat} \to \text{nat}]) : \text{nat} \to \text{nat} \}
  (f \ 3)
(\lambda ([f : (nat -> nat)]) (f 3))
\[ \text{Abstraction} \]

\[
(\bullet \ ((\text{nat} \to \text{nat}) \to \text{nat}))
\]
PCF

\[
\lambda \left[ \begin{array}{l} x : \text{nat} \\ \end{array} \right] (\text{if } 0 \leq x^2)
\]

\[
\lambda \left[ \begin{array}{l} f : (\text{nat} \rightarrow \text{nat}) \\ \end{array} \right] (f 3)
\]

\[
\lambda \left[ \begin{array}{l} x : \text{nat} \\ \end{array} \right] \left( \text{quotient } 10^x \right)
\]

\[
\text{quotient } 10^\left( \text{if } 0 \leq 723 \right)
\]

\[
\text{quotient } 10^\left( \text{if } 0 \leq 720 \right)
\]
Abstraction
Concretization
Concretization

(• ((nat → nat) → nat))

'PCF

PCF

Concretization
Concretization

\( \lambda ([g : (\text{nat} \to \text{nat})]) \)

\( (g \ (g \ 0)) \)
Concretization

\( \lambda (\lambda [x : \text{nat}] : (\text{nat} \rightarrow \text{nat})) \)

(\text{quotient} 10 (\text{if} 0 7 2 3))

(\text{quotient} 10 (\text{if} 0 7 2 0))
Concretization

PCF

((\x: \(g : \text{nat} \rightarrow \text{nat})\):
  (g (g 0)))
Concretization
Soundness: All concretizations are approximated by 'PCF
(quotient 10 (if0 7 2 3))

(quotient 10 3)

3
(quotient (• nat) (if0 7 2 3))
(quotient (• nat) (if0 7 2 3))

(if0 -1

(quotient (• nat) 3)

(• nat)

N != 0
\[
\text{(quotient (\text{\textbullet\ nat}) (if0 7 2 3))}
\]

\[
\text{(quotient (\text{\textbullet\ nat}) 3)}
\]

\[
(\text{\textbullet\ nat})
\]

\[
\text{(quotient (\text{\textbullet\ nat}) N) \delta (\text{\textbullet\ nat})}
\]

\[
N \neq 0
\]
(quotient 10 (if0 (• nat) 2 3))
\((\text{quotient } 10 (\text{if0 } (\bullet \text{ nat}) 2 3))\)
(quotient 10 (if0 (• nat) 2 3))

(quotient 10 2)

(if0 (• nat) M_0 M_1) if\• M_0

(quotient 10 3)

(if0 (• nat) M_0 M_1) if\• M_1
\[(\text{quotient} \ 10 \ (\text{if0} \ 7 \ 2 \ (\cdot \ \text{nat})))\]
(quotient 10 (if0 7 2 (∙ nat)))

(err "Divide by zero")

(quotient 10 (∙ nat))

(∙ nat)
((• (nat -> nat)) 7)
\[ (((\bullet (\text{nat} \rightarrow \text{nat})) ~ 7) \beta \rightarrow \bullet \text{nat}) \]
\[ (((\cdot (\text{nat} \to \text{nat})) \ 7) \]

\[ (((\cdot (\text{nat} \to \text{nat})) \ \cdot \text{nat}) \]

\[ (((\cdot (\text{T}_0 \ldots \text{1} \to \text{T})) \ \text{V} \ \ldots \text{1}) \ \text{\beta} (\cdot \ \text{T}) \]

\text{SPCF}
(define-extended-language SPCF PCF
  ;; Values
  (V .... (• T)))

(define s
  (reduction-relation SPCF
    (--> ((• (T ... -> T_0)) V ...) (• T_0) β•)
    (--> (if0 (• nat) M_0 M_1) M_0 if•-t)
    (--> (if0 (• nat) M_0 M_1) M_1 if•-f)))

(define-judgment-form SPCF
  #:mode (δ I I O)
  [(δ quotient (any (• nat)) (• nat))]
  [(δ quotient (any (• nat)) (err "Divide by zero"))]
  [(δ quotient ((• nat) 0) (err "Divide by zero"))]
  [(δ quotient ((• nat) N) (• nat))
    (side-condition (not-zero? N))]
  [(δ O (any_0 ... (• nat) any_1 ...) (• nat))
    (side-condition (not-quotient? O))]
  [(δ O (N_0 ...) M)
    (where M (δf O (N_0 ...)))]
(define-extended-language SPCF PCF)

;; Values
(V .... (• T)))

(define s
  (reduction-relation SPCF
    (--> ((• (T .... 0)) V ....) (• T_0) β•)
    (--> (if0 (• nat) M_0 M_1) M_0 if•-t)
    (--> (if0 (• nat) M_0 M_1) M_1 if•-t)))

(define-judgment s (PCF)
  #:mode (I)
  [(δ quotient (any (• nat))) (• nat)]
  [(δ quotient (any (• nat))) (err "Divide by zero"))]
  [(δ quotient (• nat) 0) (err "Divide by zero")]
  [(δ quotient (• nat) N) (• nat)]
  (side-condition (not-zero? N))]
  [(δ O (any_0 ... (• nat) any_1 ...) (• nat)]
  (side-condition (not-quotient? O))]
  [(δ O (N_0 ...) M)
   (where M (δf O (N_0 ...)))]

(define-judgment s (PCF)
  #:mode (I)
  [(δ quotient (any (• nat))) (• nat)]
  [(δ quotient (any (• nat))) (err "Divide by zero"))]
  [(δ quotient (• nat) 0) (err "Divide by zero")]
  [(δ quotient (• nat) N) (• nat)]
  (side-condition (not-zero? N))]
  [(δ O (any_0 ... (• nat) any_1 ...) (• nat)]
  (side-condition (not-quotient? O))]
  [(δ O (N_0 ...) M)
   (where M (δf O (N_0 ...)))]

Unsound
\((\bullet ((\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat})) \)

\((\lambda ((x : \text{nat})) \ (\text{quotient} \ 10 \ x)))\)
((• ((nat -> nat) -> nat))
(λ ((x : nat)) (quotient 10 x)))

(• ((nat -> nat) -> nat))
\[
\left(\left(\cdot \ (\text{nat} \to \text{nat}) \to \text{nat}\right)\ (\lambda \ ((x : \text{nat})) \ (\text{quotient} \ 10 \ x))\right)
\]

\[
\left(\cdot \ \text{nat}\right)
\]

\[
\left(\lambda \ ((\ f : \ (\text{nat} \to \text{nat})) ) \ (f \ 3)\right)
\]
((• ((nat -> nat) -> nat))
 (λ ((x : nat)) (quotient 10 x)))

(• nat)

(λ (((g : (nat -> nat)))) (g 0))
\[(\cdot (\text{nat} \to \text{nat}) \to \text{nat})\]  
\[(\lambda ((x : \text{nat})) (\text{quotient} \ 10 \ x)))\]  
\[\beta\]  
\[((\lambda ((g : (\text{nat} \to \text{nat}))) \ (g \ 0))\]  
\[(\lambda ((x : \text{nat})) (\text{quotient} \ 10 \ x)))\]  
\[\beta\]  
\[((\lambda ((x : \text{nat})) (\text{quotient} \ 10 \ x)) \ 0)\]  
\[\beta\]  
\[(\text{quotient} \ 10 \ 0)\]  
\[\delta\]  
\[(\text{err} \ "\text{Divide by zero}"))\]
\(((\cdot ((\text{nat} \to \text{nat}) \to \text{nat}))
(\lambda ((x : \text{nat})) (\text{quotient} \ 10 \ x)))\)
\[((\bullet ((\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}))
(\lambda ((x : \text{nat})) (\text{quotient} \ 10 \ x))\)\]
\((\lambda ((x : \text{nat})) \ (\text{quotient} \ 10 \ x))\)
((\((x : \text{nat}\))\) (quotient 10 x))
7)
((\( \ (x : \text{nat}) \) \ (\text{quotient} \ 10 \ x))\ 8)
\((\lambda (\lambda (x : \text{nat}) \ (\text{quotient} 10 \ x)) \ 625)\)
\[
((\lambda \ ((x : \text{nat})) (\text{quotient} \ 10 \ x)) \ 0)
\]
(((\ (x : nat)) (quotient 10 x))
• nat))
((\lambda ((x : nat)) (quotient 10 x))
  (\cdot nat))

(((\cdot (T_0 \ldots _1 T T_1 \ldots \to T_0))
  V_0 \ldots _1 V V_1 \ldots )
(havoc T V)
(((λ ((x : nat)) (quotient 10 x)))
  • nat))

(((• (T_0 ..._1 T T_1 ... -> T_0))
  V_0 ..._1 V V_1 ...)
  (havoc T V))

(havoc (nat -> nat)
  (λ ([x : nat]) (quotient 10 x))))
(((λ ((x : nat)) (quotient 10 x)))
 (• nat))

(((• (T_0 ... 1 T T_1 ... → T_0))
 V_0 ... 1 V V_1 ...)
 (havoc T V))

(havoc (nat → nat)
 (λ ([x : nat]) (quotient 10 x)))
(((\text{nat} \to \text{nat}) \to \text{nat})
(\lambda ((x : \text{nat})) (\text{quotient} \ 10 \ x)))

(((\lambda ((x : \text{nat})) (\text{quotient} \ 10 \ x))
(\text{nat}))

(\text{quotient} \ 10 \ (∗ \text{nat}))

(err "Divide by zero")

(∗ \text{nat})
(((\cdot (\text{nat -> nat) -> nat}))
 (\lambda (\text{x : nat}) (\text{quotient 10 x})))

(((\lambda (\text{x : nat}) (\text{quotient 10 x}))
 (\cdot \text{nat}))

(\text{quotient 10 (\cdot \text{nat}))}

(\text{err "Divide by zero"})

(\cdot \text{nat})
(define-extended-language SPCF PCF

;; Values
(V .... (• T)))

(define s
(reduction-relation SPCF
  (--> (((• (T ... -> T_0)) V ...) (• T_0) β•)
  (--> (if0 (• nat) M_0 M_1) M_0 if•-t)
  (--> (if0 (• nat) M_0 M_1) M_1 if•-f)
  (--> (((• (T_0 ...1 T T_1 ... -> T_o))
      V_0 ...1 V V_1 ...)
    (havoc T V)
    havoc)))

(define-judgment-form SPCF
  #:mode (δ I I O)
  [(δ quotient (any (• nat)) (• nat))]
  [(δ quotient (any (• nat)) (err "Divide by zero"))]
  [(δ quotient (((• nat) 0) (err "Divide by zero"))]
  [(δ quotient (((• nat) N) (• nat))
    (side-condition (not-zero? N)))]
  [(δ O (any_0 ... (• nat) any_1 ...) (• nat))
    (side-condition (not-quotient? O))]
  [(δ O (N_0 ...)) M]
  (where M (δf O (N_0 ...))))]

(define-metafunction SPCF
  [(havoc nat M) M]
  [(havoc (T_0 ... -> T_1) M)
    (havoc T_1 (M (• T_0) ...))])
(define-extended-language SPCF PCF

;; Values
(V .... (• T)))

(define s
(reduction-relation
SPCF
(--> (((• (T ... -> T_0)) V ...) (• T_0) β•)
(--> (if0 (• nat) M_0 M_1) M_0 if•-t)
(--> (• (• nat) M_0 M_1) M_1 if•-f)
(--> (((• (T_0 ... -> T_1 ... -> T_o))
V_0 ... _1 V V_1 ...)
(havoc T V)
(havoc)))

(define-judgment A SPCF
#:mode (• : I O)
[(δ quotient (((• nat)) (• nat)))]
[(δ quotient (any (• nat)) (err "Divide by zero"))]
[(δ quotient (((• nat) 0) (err "Divide by zero")))]
[(δ quotient (((• nat) N) (• nat))
(side-condition (not-quotient? 0)))]
[(δ O (any_0 ... (• nat) any_1 ...)) (• nat)]
(side-condition (not-quotient? 0))]
[(δ O (N_0 ...)) M]
(where M (δf O (N_0 ...)))]

(define-metafunction SPCF
[(havoc nat M) M]
[(havoc (T_0 ... -> T_1) M)
(havoc T_1 (M (• T_0) ...))])
Soundness:
All concretizations are approximated by 'PCF
Soundness:
All concretizations are approximated by 'PCF

Verification:
Error free 'PCF programs are error free PCF programs for all concretizations
PROBLEMS

How can we...

* do Symbolic Execution of H.O. programs?
* make program analysis modular?
* verify sophisticated contracts at compile-time?

SOLUTION

abstract reduction semantics
Problems

How can we...

* do Symbolic Execution of H.O. programs?
* make program analysis modular?
* verify sophisticated contracts at compile-time?

Solution

abstract reduction semantics
ANALYSIS
Analysis

Think hard about modularity
Think hard about modularity.
Abstracting abstract machines, CACM'11

Think hard about modularity

Analysis

Semantics
Think hard about modularity.
Think hard about modularity

Semantics

Analysis
(define-extended-language SPCF PCF

;; Values
(V ... (• T)))

(define s
  (reduction-relation SPCF
    (--> (((• (T ... -> T_0)) V ...) (• T_0) β*)
    (--> (if0 (• nat) M_0 M_1) M_0 if•-t)
    (--> (if0 (• nat) M_0 M_1) M_1 if•-f)
    (--> (((• (T_0 ... _1 T T_1 ... -> T_o))
        V_0 ... _1 V V_1 ...)
        (havoc T V)
        havoc)))

(define-judgment-form SPCF
  #:mode (δ I I O)
  [(δ quotient (any (• nat)) (• nat))]
  [(δ quotient (any (• nat)) (err "Divide by zero")))
  [(δ quotient ((• nat) 0) (err "Divide by zero"))]
  [(δ quotient ((• nat) N) (• nat))
    (side-condition (not-zero? N))]
  [(δ O (any_0 ...) (• nat) any_1 ...) (• nat))
    (side-condition (not-quotient? O))]
  [(δ O (N_0 ...) M)
    (where M (δf O (N_0 ...)))]

(define-metafunction SPCF
  [(havoc nat M) M]
  [(havoc (T_0 ... -> T_1) M)
       (havoc T_1 (M (• T_0) ...))])
Semantics Analysis
Whole-Program Analysis
Semantics Analysis

Whole-Program Analysis

Sound & computable, modular program analyzer for PCF
Problems

How can we...

* do symbolic execution of H.O. programs?
* make program analysis modular?
* verify sophisticated contracts at compile-time?

Solution

Abstract Reduction Semantics
PROBLEMS

How can we...
* do Symbolic Execution of H.O. programs?
* make program analysis modular?
* Verify sophisticated contracts at compile-time?

SOLUTION

Abstract Reduction Semantics
Contract PCF
CPCF
;; Terms
M ::= .... (C \iff M) blame

;; Contracts
C ::= = M (C ... \rightarrow C)
\[(\text{pos?} \, \text{?”} \, 7)\]

\[(\text{if0} \, (\text{pos?} \, 7) \, 7 \, \text{blame})\]

\[(\text{if0} \, 0 \, 7 \, \text{blame})\]

\[7\]
\[
(p \rightarrow \bot \ 0)
\]
\[
(if0\ (p \rightarrow 0)\ 0\ \text{blame})
\]
\[
(if0\ 1\ 0\ \text{blame})
\]
\[
\text{blame}
\]
\(((\text{prime?} \to \text{even?}) \uparrow \downarrow (\lambda \ (x) \ ...))\)

\((\lambda \ (x) \ (\text{even?} \downarrow \downarrow (\lambda \ (x) \ ...) \ (\text{prime?} \downarrow \downarrow x)))\)
\[
\lambda \left( [f \colon (\text{nat} \to \text{nat})] \right)
\left( f \ 3 \right)
\]
\( (\cdot ((\text{nat} \to \text{nat}) \to \text{nat})) \)

\text{Abstraction}

\text{CPCF}
CPCF

\[
\bullet (\{(\text{nat} \to \text{nat}) \to \text{nat}\}
\{(\text{pos}? \to \text{prime}? \to \text{prime}?)\})
\]
Symbolic values are sets of contracts.

\[
\text{CPCF} \\
\text{• } ((\text{nat} \to \text{nat}) \to \text{nat}) \\
((\text{pos?} \to \text{prime?}) \to \text{prime?}))
\]
(pos? False (\cdot \text{nat}))

(if0 (pos? (\cdot \text{nat})) (\cdot \text{nat pos?}) blame)

(if0 (\cdot \text{nat}) (\cdot \text{nat pos?}) blame)

\text{blame} \quad (\cdot \text{nat pos?})
Symbolic values remember contracts they've satisfied
(pos? \(\#\) (• nat pos?))

• nat pos?
Contracts influence Computation

\[ (\text{pos?} \; \Rightarrow \; (\bullet \; \text{nat pos?})) \]

\[ (\bullet \; \text{nat pos?}) \]
Contracts influence computation

\[(\text{quotient}\ 10\ (\bullet\ \text{nat}\ \text{pos}?))\]
((· ((nat -> nat) -> nat))
  ((pos? -> any?) ⊧ (λ ((x : nat)) (quotient 10 x))))

((· ((nat -> nat) -> nat))
  (λ ((x : nat))
   ((λ ((x : nat)) (quotient 10 x))
    (pos? ⊧ x)))
  )

((λ ((x : nat)) (quotient 10 x))
  (· nat pos?))

(quotient 10 (· nat pos?))
'CPCF

\[ ((\cdot ((\text{nat} \to \text{nat}) \to \text{nat}))
    ((\text{pos}? \to \text{any}?)) \triangleq (\lambda ((x : \text{nat})) (\text{quotient} 10 x))) \]

\[ ((\cdot ((\text{nat} \to \text{nat}) \to \text{nat}))
    (\lambda ((x : \text{nat}))
     ((\lambda ((x : \text{nat})) (\text{quotient} 10 x))
      (\text{pos}? \not\equiv x))) \]

\[ ((\lambda ((x : \text{nat})) (\text{quotient} 10 x))
   (\cdot \text{nat} \text{pos}?)) \]

\[ (\text{quotient} 10 (\cdot \text{nat} \text{pos}?) ) \]

\[ (\cdot \text{nat}) \]

HAVOC RESPECTS CONTRACTS
Soundness:
All concretizations Are Approximated by 'CPCF
Soundness:
All concretizations are approximated by ‘CPCF

Verification:
Error free ‘CPCF programs are error free CPCF programs for all concretizations
'RACKET
* Rich Language
* Rich Contracts
* Interactive Verification Environment

'Racket
Racket

Rich Language:
* Module system
* Untyped
* Data Structures
* Many Base Types
* Many Primitive Operations
Rich Contracts:
* Dependent Functions
* Data Structures
* Conjunction
* Disjunction
* Recursive Contracts
We show that our symbolic execution strategy soundly scales soundness, addressed with order values. These values present new complications to symbols, including precise specifications for abstract higher-order programs are omitted, represented only by their specifications, producing a core model of symbolic executiontracts in the setting of Contract PCF. We then extend this core calculus to a model of programs with contracts. Verifying contracts holds their dynamic semantics. As we shall see, we are able to approximate to our semantics from our reduction system. We then turn our semantics into a tool for program verification. Next, we extend the semantics which is integrated into the Racket toolchain and can serve as the basis for automated program verification, optimization, and static analysis. This semantics allows sound reasoning about the behavior of programs, with the behavior of language, for our language of higher-order contracts, which we dub preserves its advantages in higher-order reasoning. Moreover, the technique of describing symbolic values with contracts, and to verify rich properties of programs by expression of higher-order programs, in existing Racket code, contract checks take more time the semantics of contract systems into tools for verification. As the modular semantics is uncomputable, this verification strategy is necessarily incomplete. To address this, we implement an uncomputable approximation. Finally, we discuss prior work in abstracting abstract machines.

Our contributions:

1. We propose our semantics allows us to use contracts for verification
2. We then turn our semantics into a tool for program verification
3. We then scale over the technique of describing symbolic values with contracts, and to verify rich properties of programs by expression of higher-order programs, in existing Racket code, contract checks take more time the semantics of contract systems into tools for verification. As the modular semantics is uncomputable, this verification strategy is necessarily incomplete. To address this, we implement an uncomputable approximation. Finally, we discuss prior work in abstracting abstract machines.

We make the following contributions:

1. We propose our semantics allows us to use contracts for verification
2. We then turn our semantics into a tool for program verification
3. We then scale over the technique of describing symbolic values with contracts, and to verify rich properties of programs by expression of higher-order programs, in existing Racket code, contract checks take more time
4. We then turn our semantics into a tool for program verification
5. We then scale over the technique of describing symbolic values with contracts, and to verify rich properties of programs by expression of higher-order programs, in existing Racket code, contract checks take more time
We show that our symbolic execution strategy soundly scales with programs 
[sort nums]). Users can click a button and explore the behavior of symbolic values in the Racket environment.

As the modular semantics is uncomputable, this verification works despite the omitted portions. This technique is surprisingly effective, particularly in verifying specifications and analyzing programs.

We propose extending Contract PCF with abstract values described by specifications, which provide a rich domain for higher-order symbolic execution. This is a variant of using contracts as abstract values, enabling sound reasoning about function behavior.

Our plan is as follows: we begin with a review of contract systems, addressing the necessity of describing symbolic values with contracts that integrate into the Racket toolchain and support automated program verification, optimization, and static analysis. This process is computable for any instantiation of the opaque components.

We then scale contracts in existing Racket code, turning the semantics of contract systems into tools for verifying specifications and analyzing programs. As a result, contracts promise both for ensuring correctness and improving performance, compared to previous approaches.

In summary, our contributions include:

1. A novel approach to symbolic execution for higher-order programs.
2. A rich domain of abstract values for specifying contracts.
3. A computable approximation for verifying specifications.
4. Integration with the Racket toolchain for automated program verification.

Figure 1.

We discuss prior work in the context of contract systems, analyzing their performance and limitations. This allows us to give a comprehensive overview of our plan and contributions, providing insights into the benefits and challenges of using contracts in higher-order reasoning.

### Example Code

```racket
(define-contract list/c
  (rec/c X (or/c empty? (cons/c nat? X))))

(module opaque
  (provide
    [insert (nat? (and/c list/c sorted?)
               -> (and/c list/c sorted?))]
    [nums list/c])))

(module insertion-sort
  (require opaque)
  (define (foldl f l b)
    (if (empty? l) b
        (foldl f (cdr l) (f (car l) b))))
  (define (sort l) (foldl insert l empty))
  (provide
    [sort
     (list/c -> (and/c list/c sorted?))]))
```

This code demonstrates the use of contracts in higher-order reasoning, showing how symbolic execution can be applied to verify properties of programs. The example involves sorting a list of natural numbers, using contracts to specify the behavior of the sorting function.
(define-contract list/c
  (rec/c X (or/c empty? (cons/c nat? X)))))

(module opaque
  (provide
    [insert (nat? (and/c list/c sorted?)
             -> (and/c list/c sorted?)])
    [nums list/c]])

(module insertion-sort
  (require opaque)
  (define (foldl f l b)
    (if (empty? l) b
        (foldl f (cdr l) (f (car l) b))))
  (define (sort l) (foldl insert l empty))
  (provide
    [sort
      (list/c -> (and/c list/c sorted?)])])
(define-contract list/c
  (rec/c X (or/c empty? (cons/c nat? X))))

(module opaque
  (provide
    [insert (nat? (and/c list/c sorted?)
               -> (and/c list/c sorted?))]
    [nums list/c]]))

(module insertion-sort
  (require opaque)
  (define (foldl f l b)
    (if (empty? l) b
        (foldl f (cdr l) (f (car l) b)))
  (define (sort l) (foldl insert l empty))
  (provide
    [sort
      (list/c -> (and/c list/c sorted?))])))
> (sort nums)

(• (and/c list/c sorted?))
Higher-Order Symbolic Execution via Contracts

Sam Tobin-Hochstadt    David Van Horn
Northeastern University {samth,dvanhorn}@ccs.neu.edu

Abstract
We present a new approach to automated reasoning about higher-order programs by extending symbolic execution to use behavioral contracts as symbolic values, thus enabling symbolic approximation of higher-order behavior.

Our approach is based on the idea of an abstract reduction semantics that gives an operational semantics to programs with both concrete and symbolic components. Symbolic components are approximated by their contract and our semantics gives an operational interpretation of contracts as values. The result is an executable semantics that soundly predicts program behavior, including contract failures, for all possible instantiations of symbolic components. We show that our approach scales to an expressive language of contracts including arbitrary programs embedded as predicates, dependent function contracts, and recursive contracts. Supporting this rich language of specifications leads to powerful symbolic reasoning using existing program constructs.

We then apply our approach to produce a verifier for contract correctness of components, including a sound and computable approximation to our semantics that facilitates fully automated contract verification. Our implementation is capable of verifying contracts expressed in existing programs, and of justifying contract-elimination optimizations.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Languages, Theory, Verification

Keywords Higher-order contracts, symbolic execution, reduction semantics

1. Behavioral contracts as symbolic values

Whether in the context of dynamically loaded JavaScript programs, low-level native C code, widely-distributed libraries, or simply intractably large code bases, automated reasoning tools must cope with access to only part of the program. To handle missing components, the omitted portions are often assumed to have arbitrary behavior, greatly limiting the precision and effectiveness of the tool.

Of course, programmers using external components do not make such conservative assumptions. Instead, they attach specifications to these components, often with dynamic enforcement. These specifications increase their ability to reason about programs that are only partially known. But reasoning solely at the level of specification can also make verification and analysis challenging as well as requiring substantial effort to write sufficient specifications.

The problem of program analysis and verification in the presence of missing data has been widely studied, producing many effective tools that apply symbolic execution to nondeterministically consider many or all possible inputs. These tools typically determine constraints on the missing data, and reason using these constraints. Since the central lesson of higher-order programming is that computation is data, we propose symbolic execution of higher-order programs for reasoning about systems with omitted components, taking specifications to be our constraints.

Our approach to higher-order symbolic execution therefore combines specification-based symbolic reasoning about opaque components with semantics-based concrete reasoning about available components; we characterize this technique as specifications as values. As specifications, we adopt higher-order behavioral software contracts [17]. Contracts have two crucial advantages for our strategy. First, they provide benefit to programmers outside of verification, since they automatically and dynamically enforce their described invariants. Because of this, modern languages such as C#, Haskell, and Racket come with rich contract libraries that programmers already use [15, 17, 22]. Rather than requiring programmers to annotate code with assertions, we leverage the large body of code that already attaches contracts at code boundaries. For example, the Racket standard library features more than 4000 uses of contracts [21]. Second, the meaning of contracts as specifications is neatly captured by
Higher-Order Symbolic Execution via Contracts

Sam Tobin-Hochstadt  David Van Horn
Northeastern University
{samth,dvanhorn}@ccs.neu.edu

Abstract
We present a new approach to automated reasoning about higher-order programs by extending symbolic execution to use behavioral contracts as symbolic values, thus enabling symbolic approximation of higher-order behavior.

Our approach is based on the idea of an abstract reduction semantics that gives an operational semantics to programs with both concrete and symbolic components. Symbolic components are approximated by their contract and our semantics gives an operational interpretation of contracts-as-values. The result is an executable semantics that soundly predicts program behavior, including contract failures, for all possible instantiations of symbolic components. We show that our approach scales to an expressive language of contracts including arbitrary programs embedded as predicates, dependent function contracts, and recursive contracts. Supporting this rich language of specifications leads to powerful symbolic reasoning using existing program constructs.

We then apply our approach to produce a verifier for contract correctness of components, including a sound and computable approximation to our specifications that facilitates fully automated contract verification. Our implementation is capable of verifying contracts expressed in existing programs, and of justifying contract-elimination optimizations.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Languages, Theory, Verification

Keywords Higher-order contracts, symbolic execution, reduction semantics

* Semantics for ‘Racket
* Soundness for ‘Racket
* Analysis for ‘Racket
* Interactive Verification Environment
#lang racket/load

;; 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? string
    exact-nonnegative-integer?)
  (empty-scene (exact-nonnegative-integer? exact-nonnegative-integer?)
  (place-image (image? exact-nonnegative-integer? exact-nonnegative-integer?)

;; -- Source
(module data racket
  (require 2htdp/unfold)
  (require 'scenes)
  (run (snake world food))

;; Contracts
(define direction/c
  (one-of/c 'up 'down 'left 'right))
(define posn/c
  (struct/c posn
    exact-nonnegative-integer?
    exact-nonnegative-integer?))
(define snake/c
  (struct/c snake
direction/c
  (non-empty-listof posn/c)))
(define world/c
  (struct/c world
    snake/c
    posn/c))

;; posn=? : Posn Posn -> Boolean
;; Are the posns the same?
#lang var

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

;; -- Source
(module data racket
  (struct posn (x y))
  (struct snake (dir segs))
  (struct world (snake food))

;; Contracts
(define direction/c
  (one-of/c 'up 'down 'left 'right))
(define posn/c
  (struct/c posn
    exact-nonnegative-integer?
    exact-nonnegative-integer?))
(define snake/c
  (struct/c snake
    direction/c
    (non-empty-listof posn/c)))

Welcome to DrRacket, version 5.3.1.1--2012-10-13(2b902d0e/d) [3m].
Language: var; memory limit: 128 MB.
#lang var

(module image racket
  :(require 2htdp/image)

  (provide contract
    [image? (any/c . -> . boolean?)
    [empty-scene (exact-nonnegative-integer? exact-nonnegative-integer?
    [place-image (image? exact-nonnegative-integer?

---

(module data racket
  (struct posn (x y))
  (struct snake (dir segs))
  (struct world (snake food))

---

;; Contracts
(define direction/c
  (one-of/c 'up 'down 'left 'right))
(define posn/c
  (struct/c posn
    exact-nonnegative-integer?
    exact-nonnegative-integer?))
(define snake/c
  (struct/c snake
    direction/c
    (non-empty-listof posn/c)))
CONCLUSION

Abstract Reduction semantics Enables
* Higher-Order Symbolic Execution,
* Modular Program Analysis, and
* Contract Verification

https://github.com/samth/var/
https://github.com/dvanhorn/pcf/
Thank you
CONCLUSION

Abstract Reduction semantics Enables
* Higher-Order Symbolic Execution,
* Modular Program Analysis, and
* Contract Verification

https://github.com/samth/var/
https://github.com/dvanhorn/pcf/