Verifying Contracts of Dynamic Language Statically

Guannan Wei, Jian Lan

2016/4/21

Contract

- Contract in dynamic language describes the preconditions and postconditions for a function
- For example, in Racket, when you call a function, the contract system will check the argument satisfy the precondition or not, as well as the returned value of function.

But, in runtime.

We want to verify these contracts
before run the program.

Setting

- Dynamic typing (type of a variable may change in runtime)
- Higher-order language (function as value)
- A-Normal Form as intermediate representation of program

ANF

```
lam ::= (λ (var) exp)
aexp ::= lam |  var |  true  |  false
       | integer |  (prim aexp*)
cexp ::= (aexp0 aexp1)
       | (if aexp exp exp)
       | (letrec ((var aexp)) exp)
exp ::= aexp | cexp
       | (let ((var exp)) exp)
prim ::= +|-|*|=|>|and|or|not
```

Abstract Interpretation

- CESK* Abstracting Abstract Machine
  - Control: the current expression
  - Environment: variable → address
  - Store: address → D
  - Continuation: the next computation, allocated in store
- D is a set of abstract value
- Abstract value is a predicate of value

Abstract Interpretation

- Control Flow Analysis (k-CFA), currently using 1-CFA
- Each state in concrete machine has a corresponding abstract state,
- An abstract state may transit to one or many states,
- By tracing these state transition, we have a graph and know
  - the argument of call
  - the actually function being called
  - the returned abstract value from function

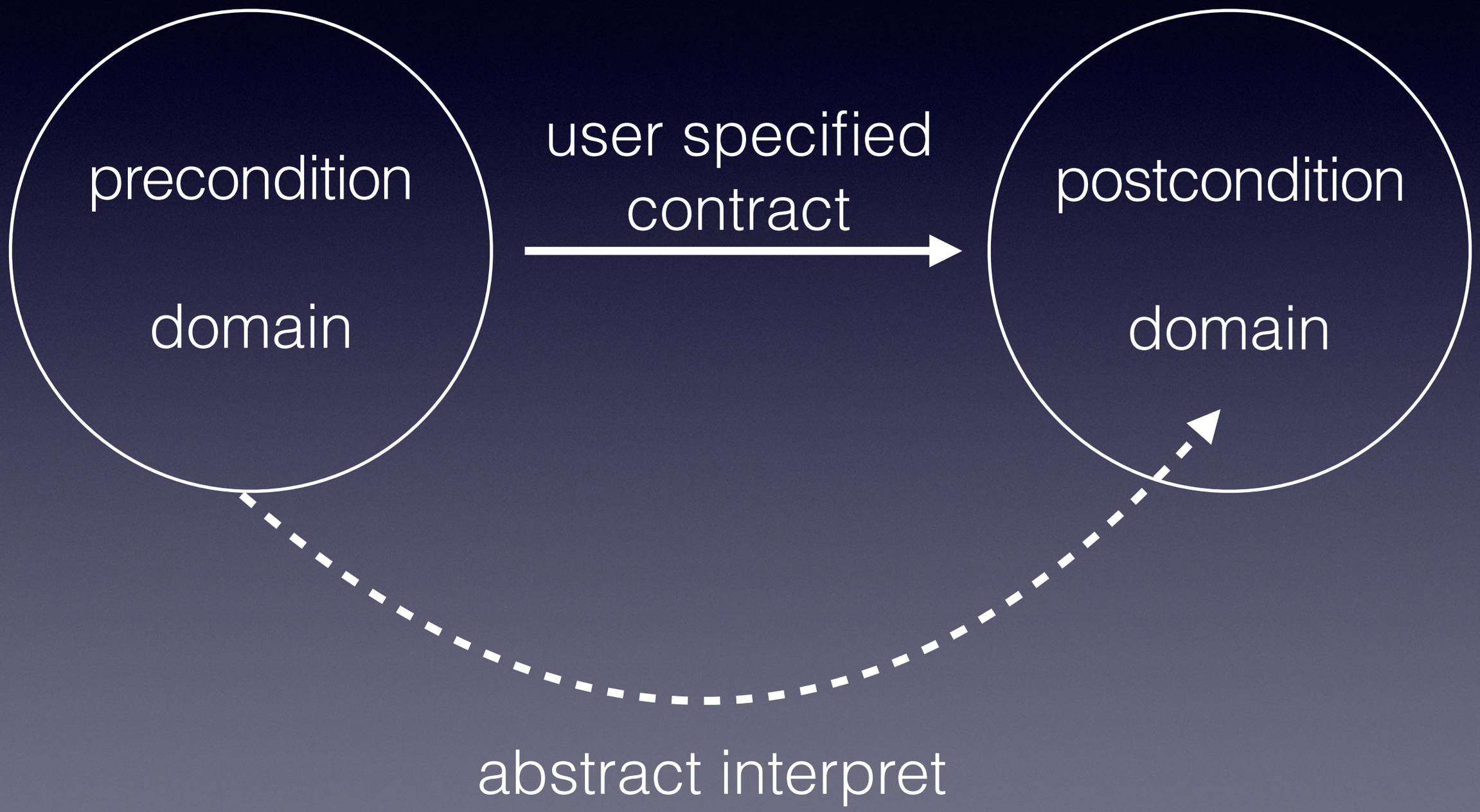
Verification

- Part 1: is the contract annotation correct?
- Part 2: is each call to function obey the contract?

Part 1: is the contract annotation correct?

- Given the precondition (specified by user) as argument, after abstractly interpret the body, is the result obey the postcondition (specified by user) ?

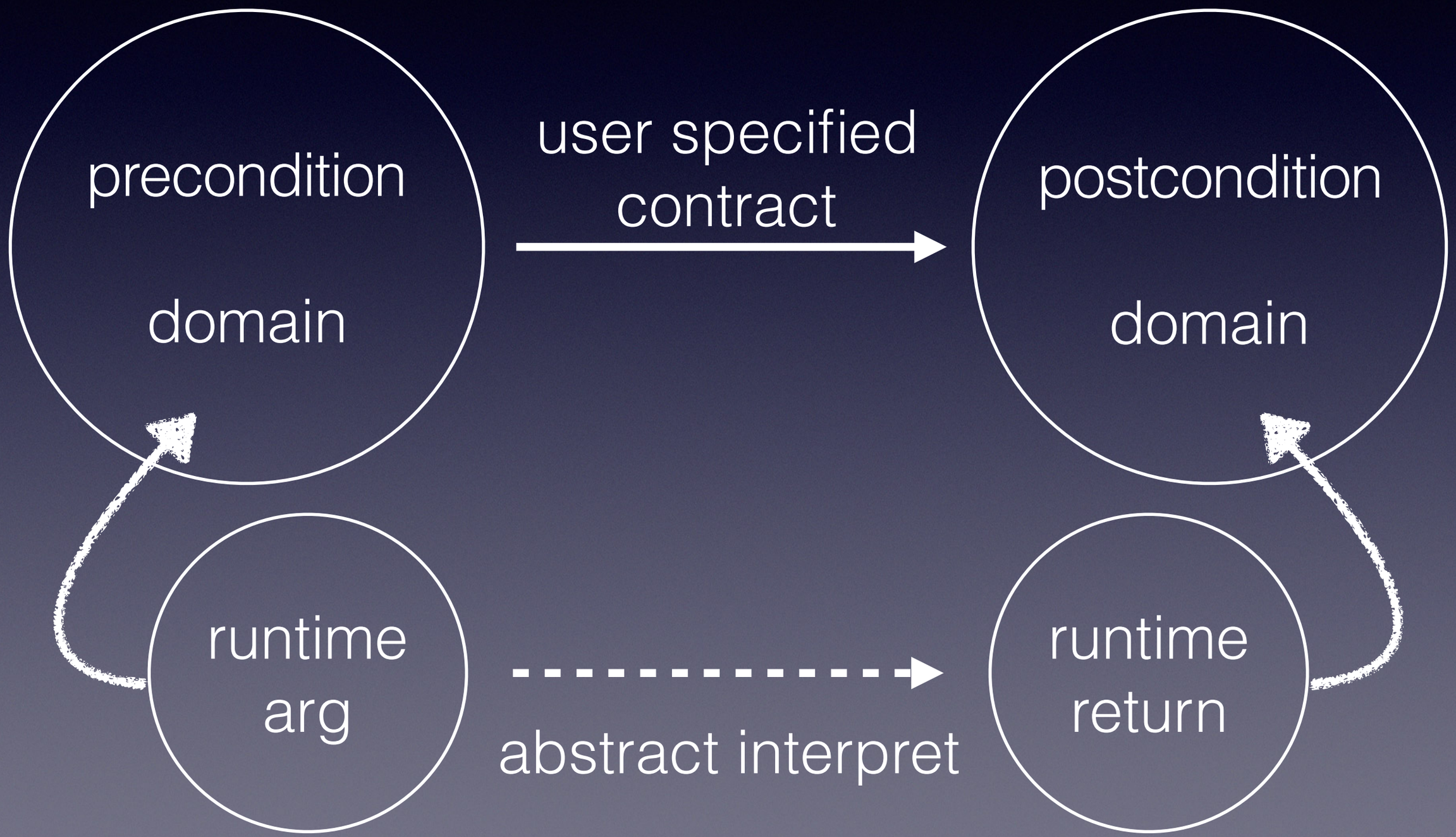
Part 1: is the contract annotation correct?



Part 2: is each call obey the contract?

- Is the real runtime argument of function obeys the precondition?
- Is the returned value from function obeys the postcondition?

Part 2: is each call obey the contract?



Demo

```
; Example 1
; id should be (Int -> Int), but user provide (Int -> Bool)
(define wrong-contract (define-types->hash '((: id (-> Int Bool)))))

(define id (parse '{lambda id {x} x}))

(verify-contract id wrong-contract) ;reject
```

Demo

```
; Example2: check runtime type without predicate
; id: int -> int
(define contract2 (define-types->hash '((: id (-> Int Int)))))

; id could has an intersection type (int -> int) & (bool -> bool)
(define contract2-bool (define-types->hash '((: id (-> Int Int))
                                             (: id (-> Bool Bool)))))

(define example2 (parse '{let {{id {lambda id {x} x}}}
                           {let {{one {id 1}}}
                             {let {{fls {id false}}}
                               one}}}))

(verify-runtime example2 contract2)       ; reject
(verify-runtime example2 contract2-bool)  ; accept
```

Demo

```
; Example3: higher-order function and contract with predicate
; add1: int[x>0] -> int[y>1]
; add2: int[x>3 && x<9] -> int[x>5 && x<11]
(define contract3 (define-types->hash
                    '((: add1 (-> (Int (> _ 0)) (Int (> _ 1))))
                      (: add2 (-> (Int (and (> _ 3) (< _ 9)))
                                  (Int (and (> _ 5) (< _ 11)))))))))
```

```
(define example3 (parse '{let {{add1 {lambda add1 {x} {+ 1 x}}}}
                           {let {{add2 {lambda add2 {x} {+ 2 x}}}}
                             {let {{apply {lambda applyf {f} {lambda applyg {g} {f g}}}}}
                               {let {{another_add1 {apply add1}}}
                                 {let {{another_add2 {apply add2}}}
                                   {let {{two {another_add1 1}}}
                                     {let {{three {another_add1 two}}}
                                       {another_add2 2}}}}}}}}}))

; reject becuase {another_add2 2} does not satisfy precondition of add2
(verify-runtime example3 contract3)
```

Thanks!