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~~ Types

PCF

PCF

```
(quotient 10 (if0 7 2 3))
    │ if0-f
    ▼
(quotient 10 3)
    │ δ
    ▼
3
```

$$\begin{tikzcd}
\texttt{(quotient 10 (if0 7 2 0))} \arrow[d, "\text{if0-f}"] \\
\texttt{(quotient 10 0)} \arrow[d, "\delta"] \\
\texttt{(err "Divide by zero")}
\end{tikzcd}$$

PCF

```
((λ ((x : nat)) (quotient 10 x))
 (if0 7 2 3))
```

$\downarrow$ if0-f

```
((λ ((x : nat)) (quotient 10 x)) 3)
```

$\downarrow$ β

```
(quotient 10 3)
```

$\downarrow$ δ

```
3
```

PCF

```
((λ ((f : (nat -> nat))) (f 3))
 (λ ((x : nat)) (quotient 10 x)))
```

$\downarrow \beta$

```
((λ ((x : nat)) (quotient 10 x)) 3)
```

$\downarrow \beta$

```
(quotient 10 3)
```

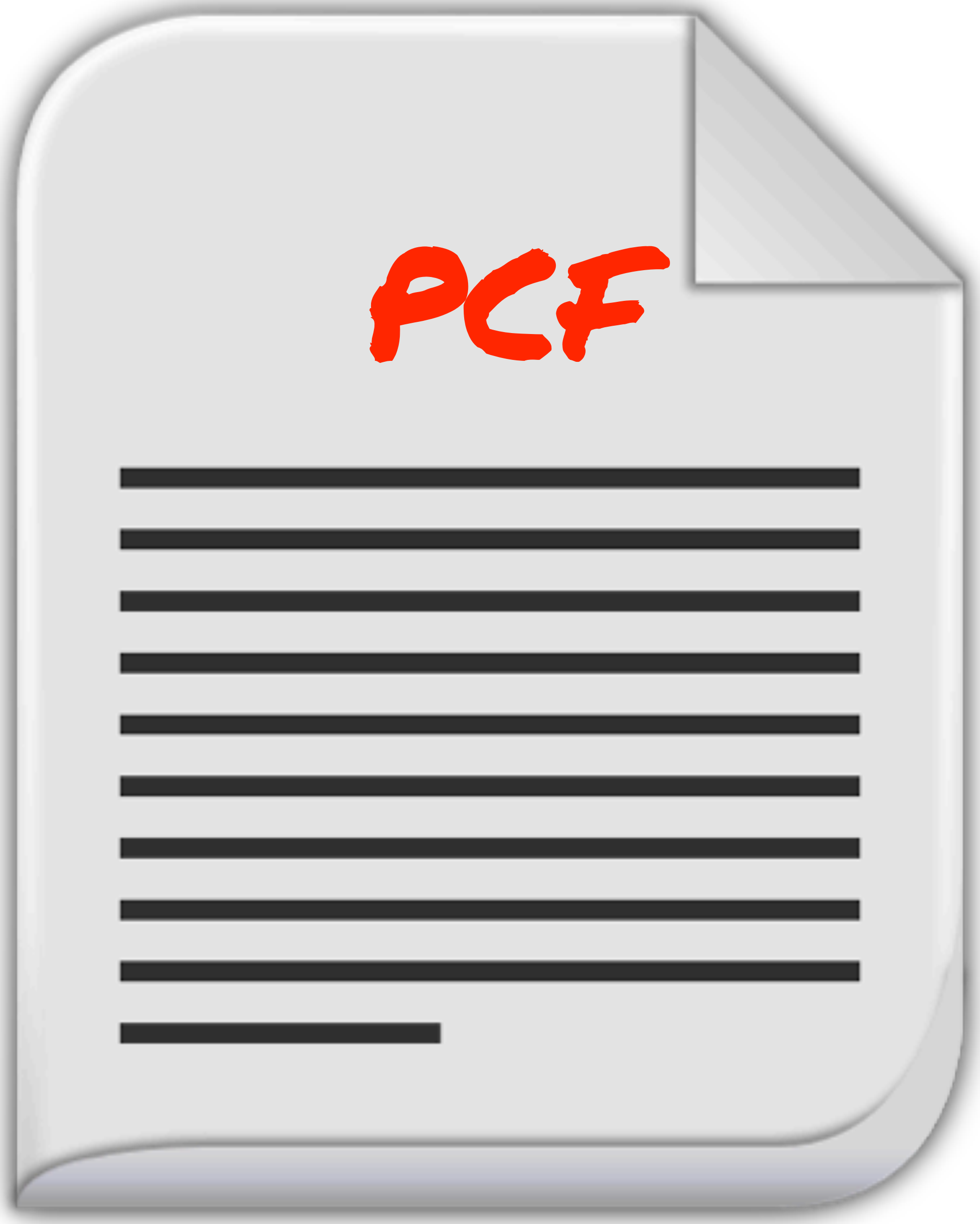
$\downarrow \delta$

```
3
```

SYMBOLIC PCF

'PCF

Abstraction



ABSTRACTION

Abstraction


```
(λ ([f : (nat -> nat)])
   (f 3))
```

Abstraction


```
(• ((nat -> nat) -> nat))
```

Abstraction

Abstraction

Concretization

Concretization

Concretization

(• ((nat -> nat) -> nat))

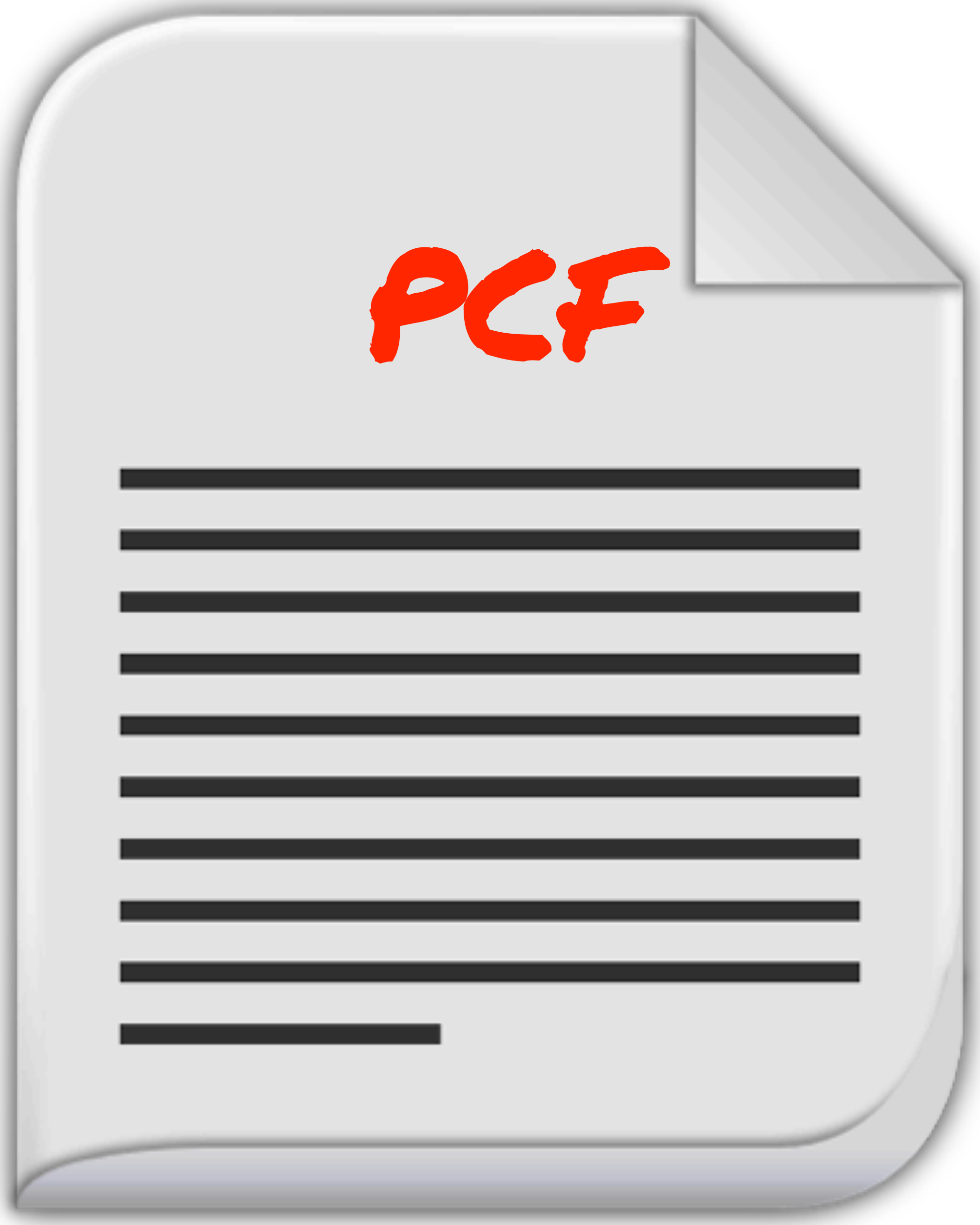
Concretization

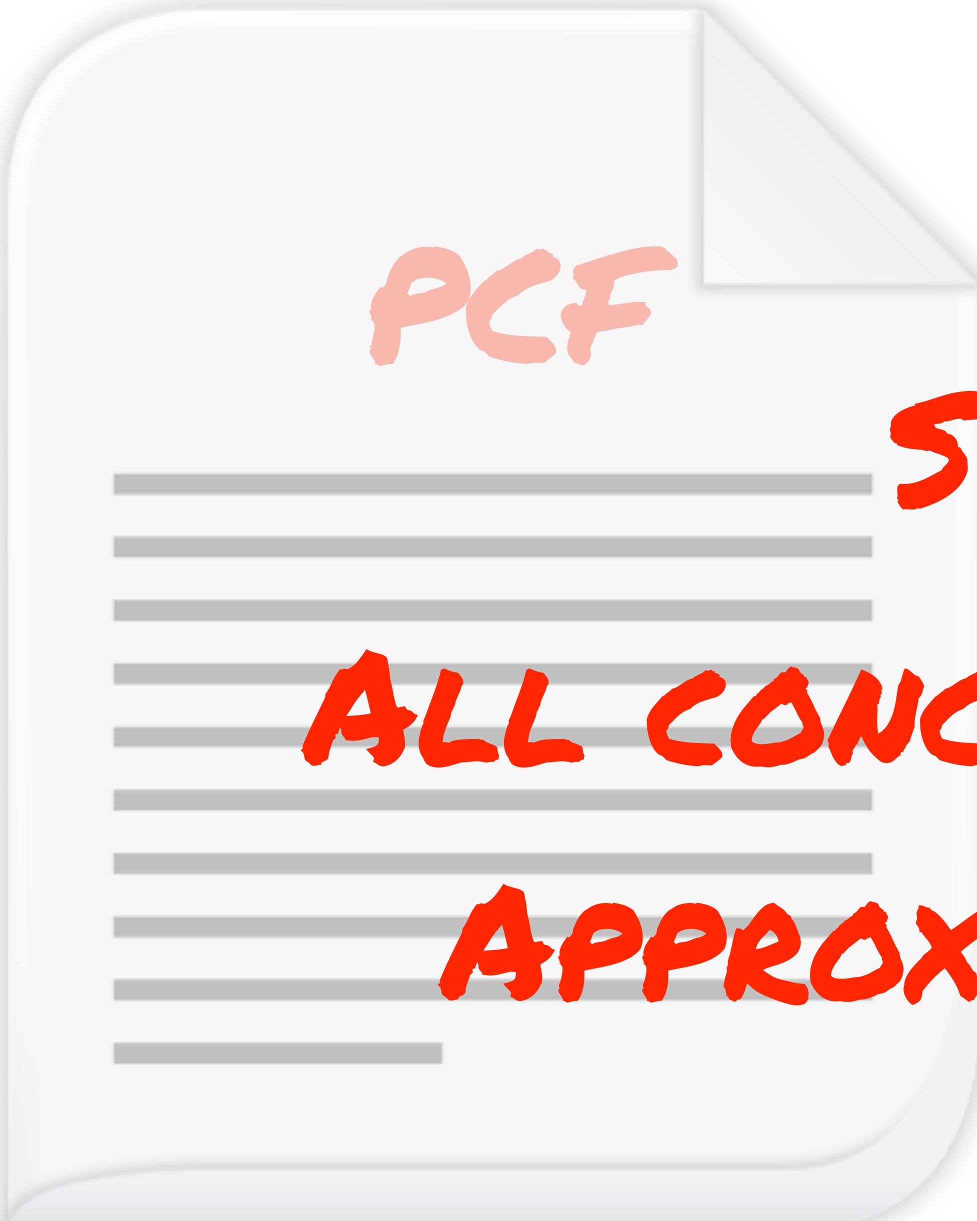

```
(λ ([g : (nat -> nat)])
   (g (g 0)))
```

Concretization

Concretization

Concretization





Soundness:

All concretizations Are Approximated by 'PCF

PCF

`(quotient 10 (if0 7 2 3))`

$\Big\downarrow$ if0-f

`(quotient 10 3)`

$\Big\downarrow$ δ

`3`

'PCF

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

N != 0

'PCF

```
(quotient (• nat) (if0 7 2 3))
        |
        | if0-f
        v
(quotient (• nat) 3)
        |
        | δ
        v
(• nat)
```

N != 0

'PCF

```
(quotient (• nat) (if0 7 2 3))
        │
        │ if0-f
        ▼
(quotient (• nat) 3)
        │
        │ δ
        ▼
(• nat)
```

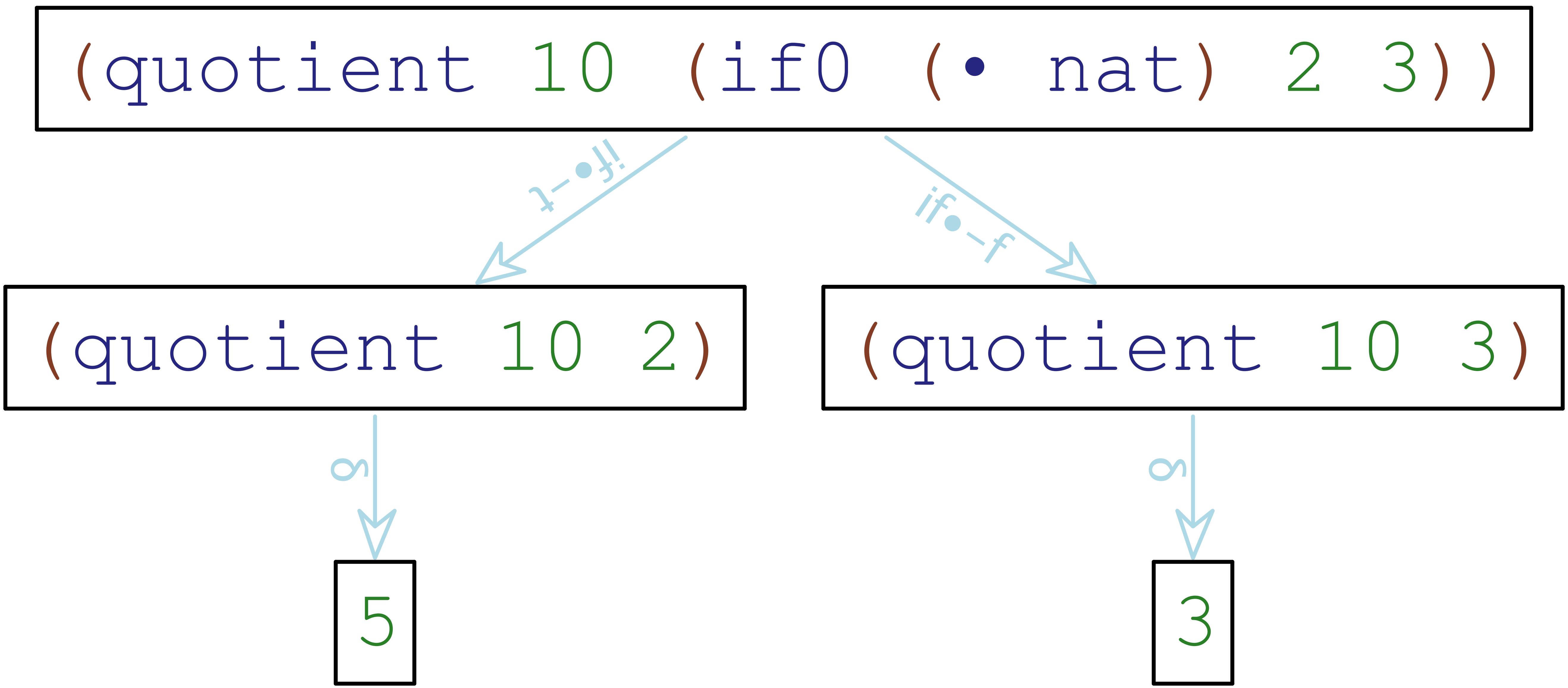
(quotient (• nat) N) δ (• nat)

N != 0

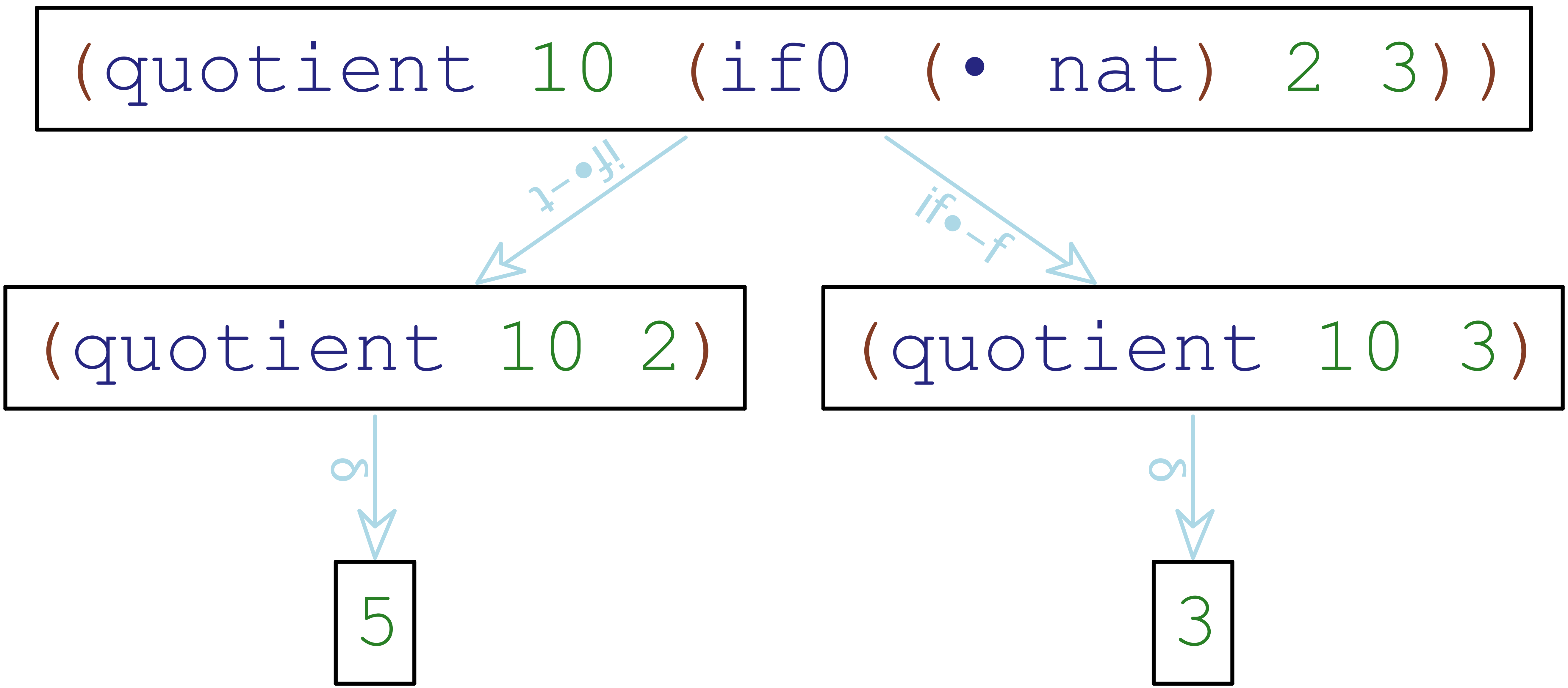
'PCF

```
(quotient 10 (if0 (• nat) 2 3))
```

'PCF


(quotient 10 (if0 (• nat) 2 3))
if•-t
if•-f
(quotient 10 2)
(quotient 10 3)
δ
δ
5
3

'PCF



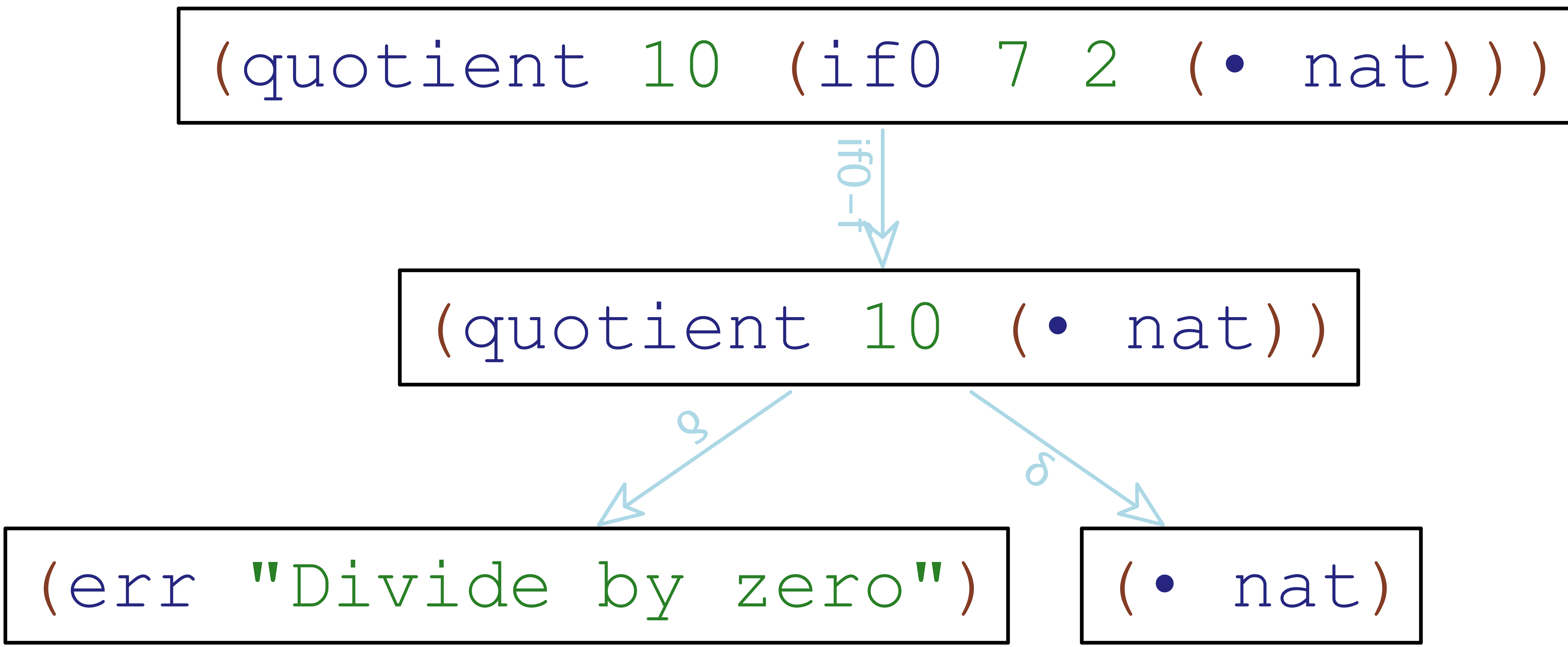
(if0 (• nat) M_0 M_1) if• M_0

(if0 (• nat) M_0 M_1) if• M_1

'PCF

```
(quotient 10 (if0 7 2 (• nat)))
```

'PCF


(quotient 10 (if0 7 2 (• nat)))
if0-f
(quotient 10 (• nat))
δ
δ
(err "Divide by zero")
(• nat)

'PCF

$$\begin{tikzcd} & \texttt{(quotient 10 (if0 7 2 (• nat)))} \arrow[d, "\text{if0-f}"'] & \\ & \texttt{(quotient 10 (• nat))} \arrow[dl, "\delta"'] \arrow[dr, "\delta"] & \\ \texttt{(err "Divide by zero")} & & \texttt{(• nat)} \end{tikzcd}$$

```
(quotient N (• nat))  δ (• nat)
(quotient N (• nat))  δ (err "Divide by 0")
```

'PCF

```
((• (nat -> nat)) 7)
```

'PCF

```
((• (nat -> nat)) 7)
```

$$\Big\downarrow{\scriptstyle \beta\bullet}$$

```
(• nat)
```

'PCF

$$\begin{tikzcd} \boxed{\texttt{((}\bullet\texttt{ (nat -> nat)) 7)}} \arrow[d, "\beta\bullet"] \\ \boxed{\texttt{(}\bullet\texttt{ nat)}} \end{tikzcd}$$

((• (T_0 ..._1 -> T)) v ..._1) β• (• T)

'PCF

```
(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 ...)))])
```

'PCF

```
(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 ...)))])
```

UNSOUND

'PCF

```
((• ((nat -> nat) -> nat))
 (λ ((x : nat)) (quotient 10 x)))
```

$\downarrow$ β•

```
(• nat)
```

```
((• ((nat -> nat) -> nat))
 (λ ((x : nat)) (quotient 10 x)))
```

β•

```
(• nat)
```

```
(• ((nat -> nat) -> nat))
```

'PCF

```
((• ((nat -> nat) -> nat))
 (λ ((x : nat)) (quotient 10 x)))
```

β•

```
(• nat)
```

```
(λ ((f : (nat -> nat))) (f 3))
```

'PCF

```
((• ((nat -> nat) -> nat))
 (λ ((x : nat)) (quotient 10 x)))
```

```
(• nat)
```

```
(λ ((g : (nat -> nat))) (g 0))
```

'PCF

```
((• ((nat -> nat) -> nat))
 (λ ((x : nat)) (quotient 10 x)))
```

$\Big\downarrow \beta$

```
(• nat)
```

```
((λ ((g : (nat -> nat))) (g 0))
 (λ ((x : nat)) (quotient 10 x)))
```

$\Big\downarrow \beta$

```
((λ ((x : nat)) (quotient 10 x)) 0)
```

$\Big\downarrow \beta$

```
(quotient 10 0)
```

$\Big\downarrow \delta$

```
(err "Divide by zero")
```

'PCF

```
(((• ((nat -> nat) -> nat))
  (λ ((x : nat)) (quotient 10 x)))
```

$\downarrow \beta\bullet$

```
(• nat)
```

'PCF

```
((• ((nat -> nat) -> nat))
 (λ ((x : nat)) (quotient 10 x)))
```

$\big\downarrow \beta\bullet$

```
(• nat)
```

```
(λ ((x : nat)) (quotient 10 x))
```

'PCF

```
((λ ((x : nat)) (quotient 10 x))
 7)
```

'PCF

```
((λ ((x : nat)) (quotient 10 x))
 7)
```

'PCF

```
((λ ((x : nat)) (quotient 10 x))
 8)
```

'PCF

```
((λ ((x : nat)) (quotient 10 x))
 625)
```

'PCF

```
((λ ((x : nat)) (quotient 10 x))
 0)
```

'PCF

```
((λ ((x : nat)) (quotient 10 x))
 (• nat))
```

'PCF

```
((λ ((x : nat)) (quotient 10 x))
 (• nat))

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

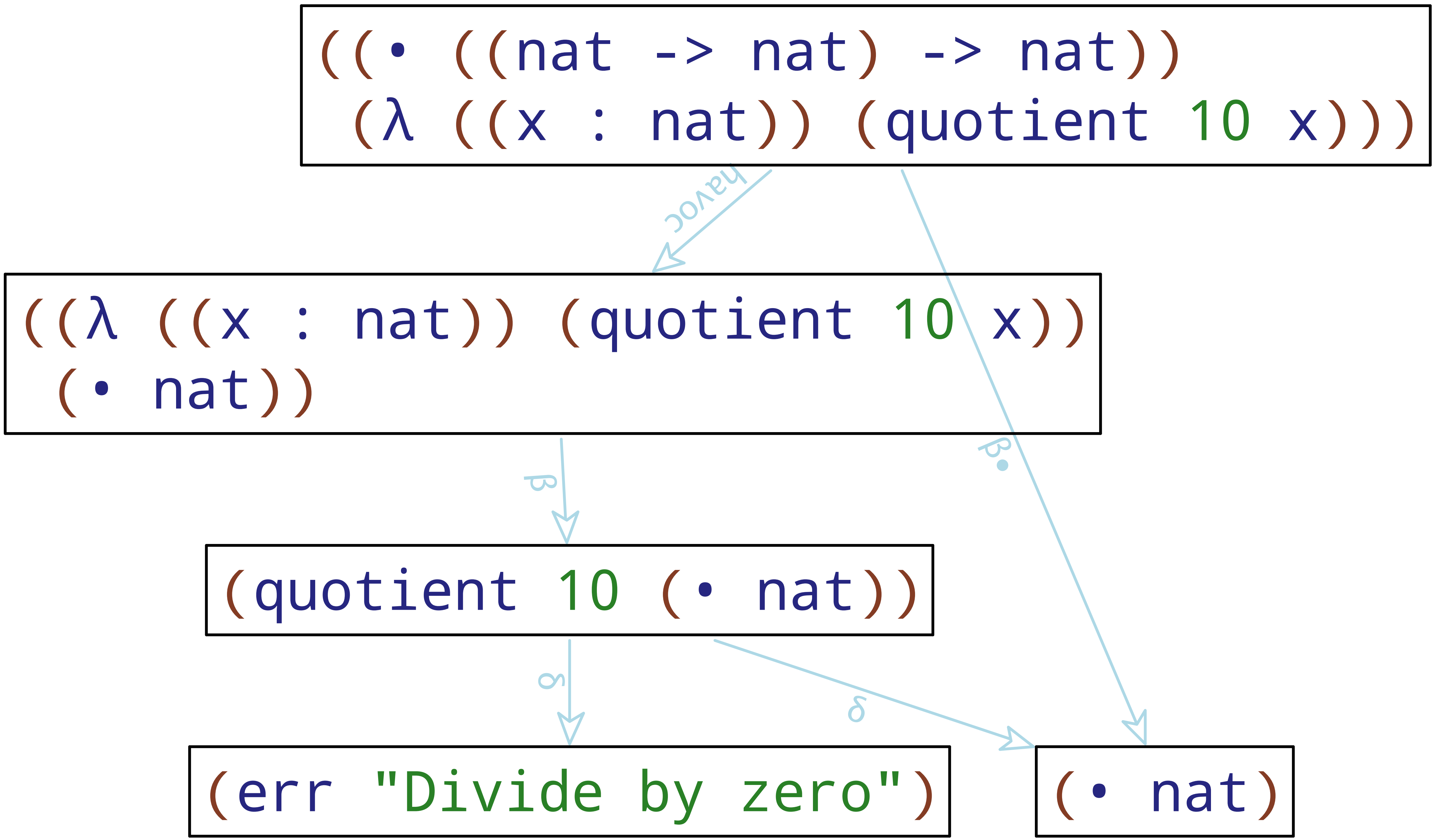
'PCF

```
((λ ((x : nat)) (quotient 10 x))
 (• nat))

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

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

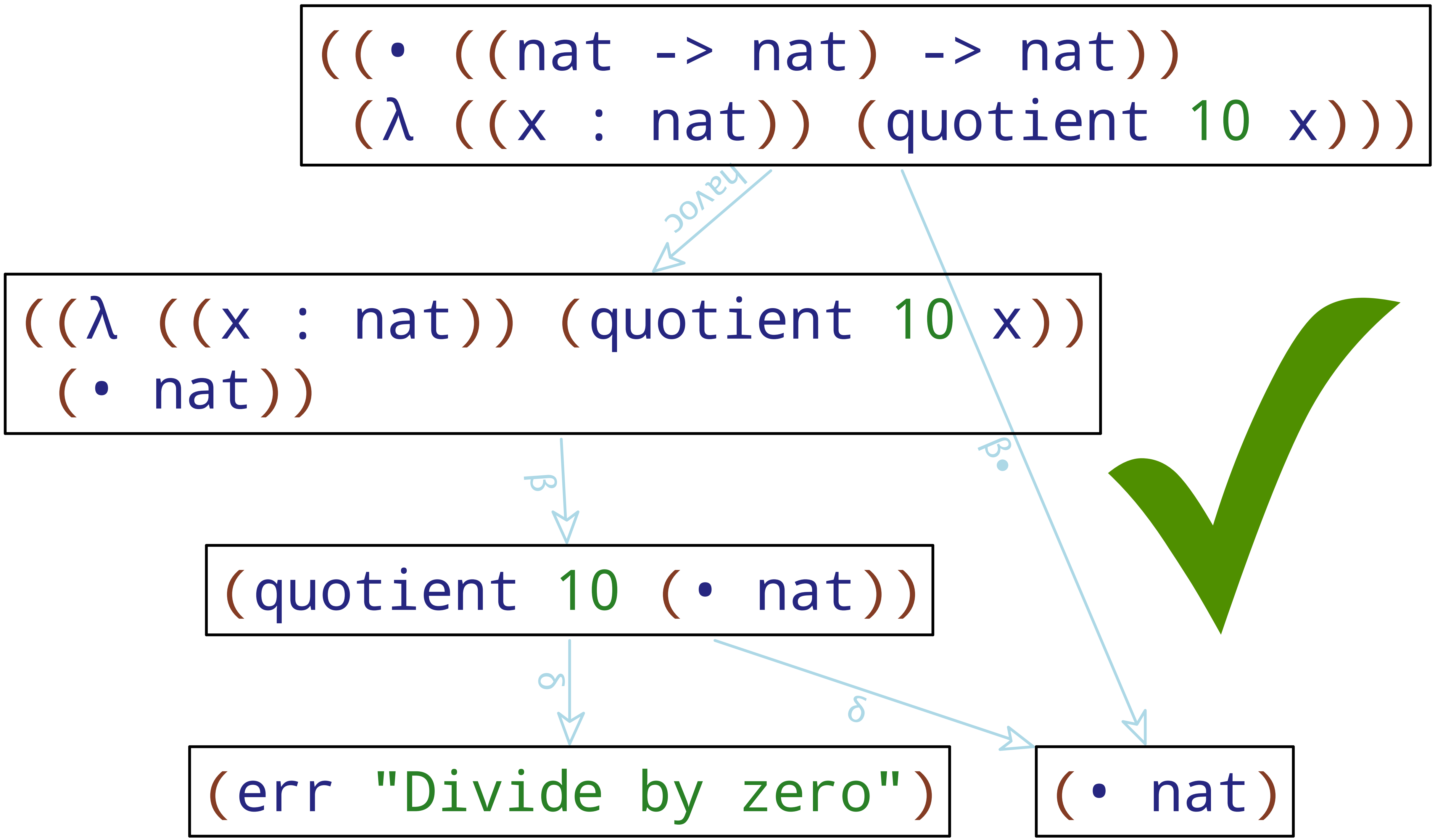
'PCF

```
((λ ((x : nat)) (quotient 10 x))
 (• nat))

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

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

'PCF


((• ((nat -> nat) -> nat))
(λ ((x : nat)) (quotient 10 x)))
havoc
((λ ((x : nat)) (quotient 10 x))
(• nat))
β
β•
(quotient 10 (• nat))
δ
δ
(err "Divide by zero")
(• nat)

'PCF



((• ((nat -> nat) -> nat))
(λ ((x : nat)) (quotient 10 x)))
havoc
β•
((λ ((x : nat)) (quotient 10 x))
(• nat))
β
(quotient 10 (• nat))
δ
δ
(err "Divide by zero")
(• nat)

'PCF

```
(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) ...))])
```

'PCF

```
(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 O)
  [(δ quotient ((• nat) (• 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) ...))])
```

SOUND

Soundness:

All concretizations Are Approximated by 'PCF

'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

Think hard about modularity

Analysis

Semantics

Analysis

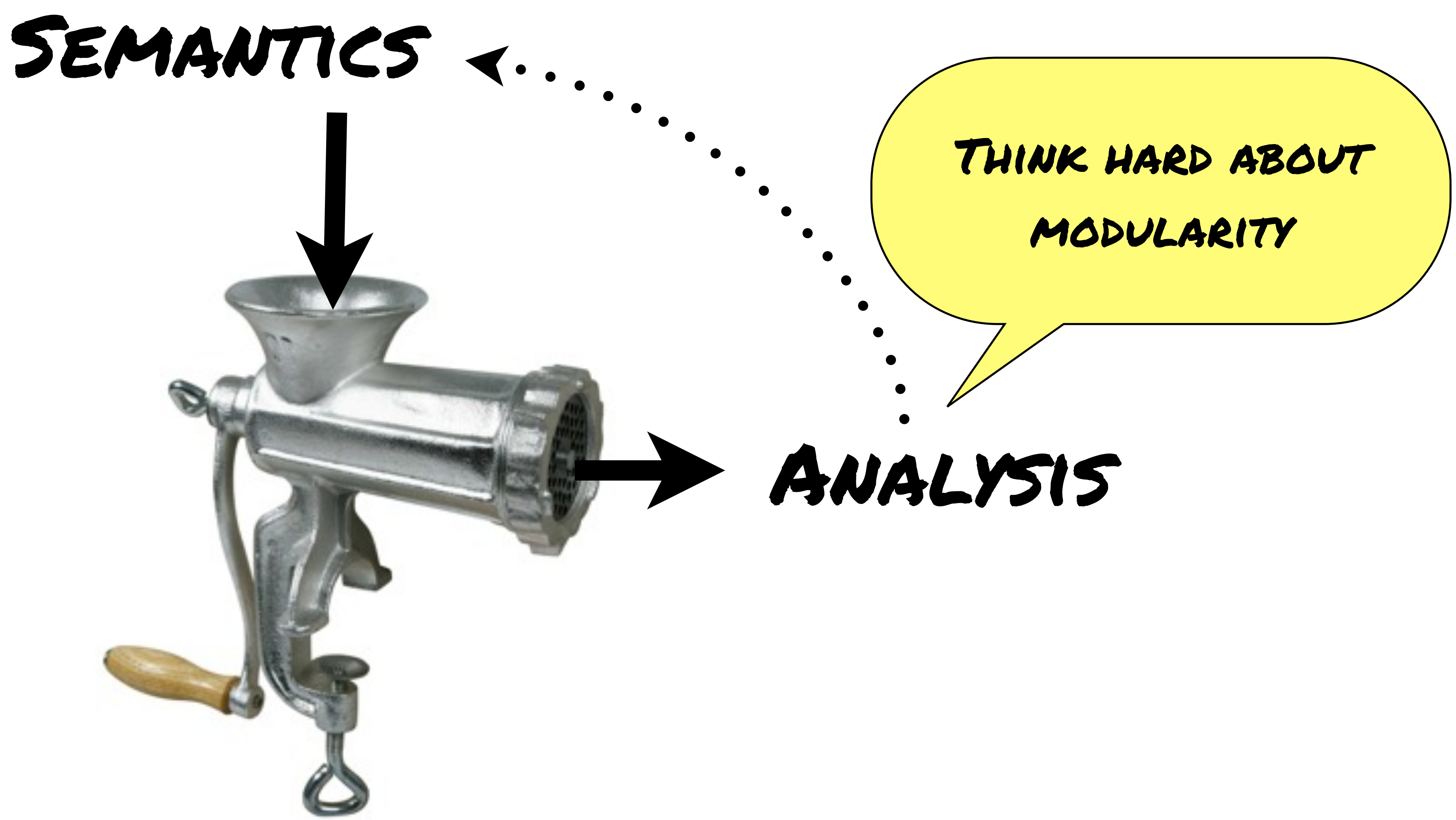
Think hard about modularity

Semantics

Think hard about modularity

Analysis

Abstracting abstract machines, CACM'11


Semantics
Think hard about modularity
Analysis

Think hard about modularity

Semantics

Analysis

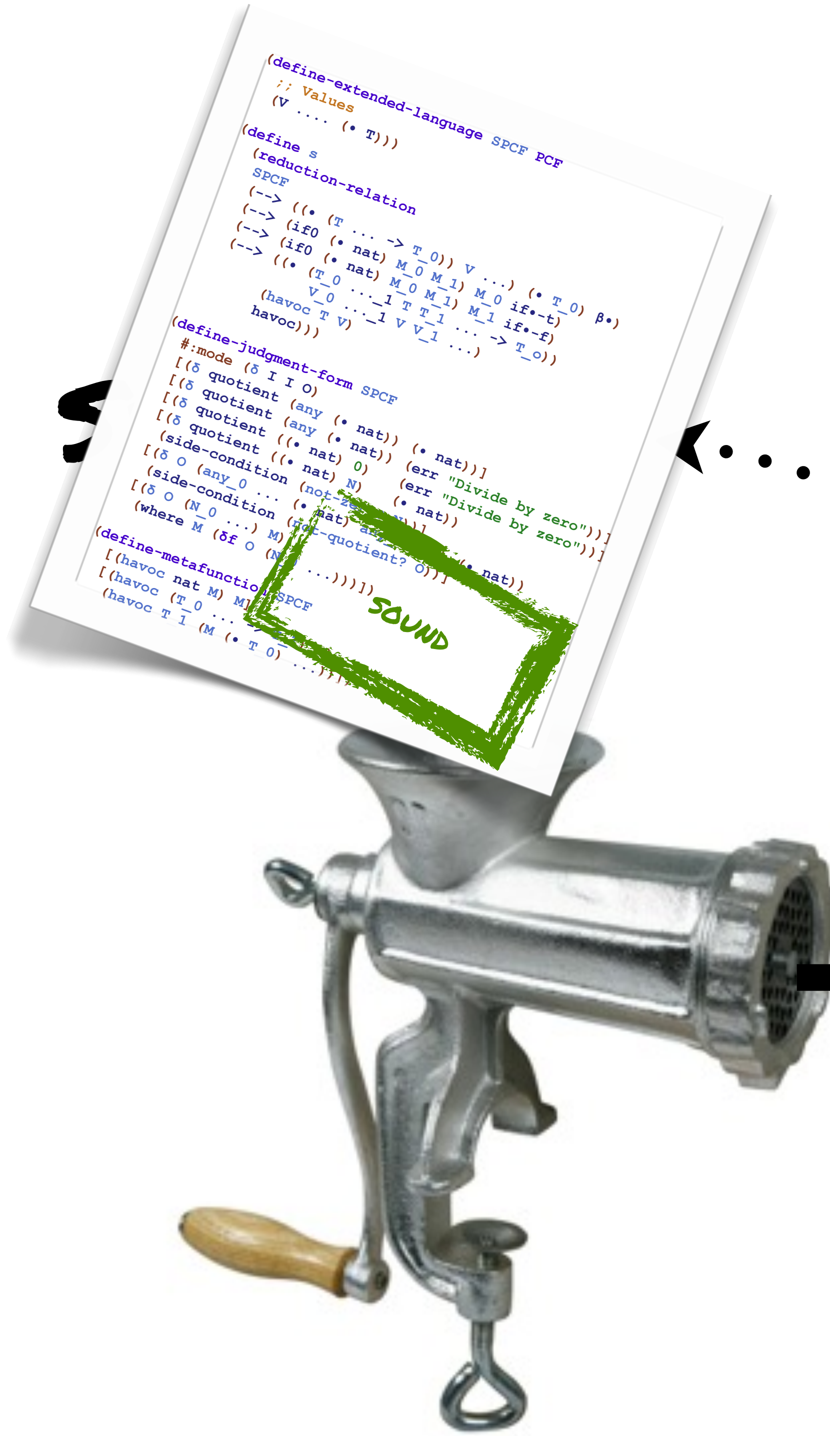
Whole-Program
Semantics

Whole-Program
Analysis

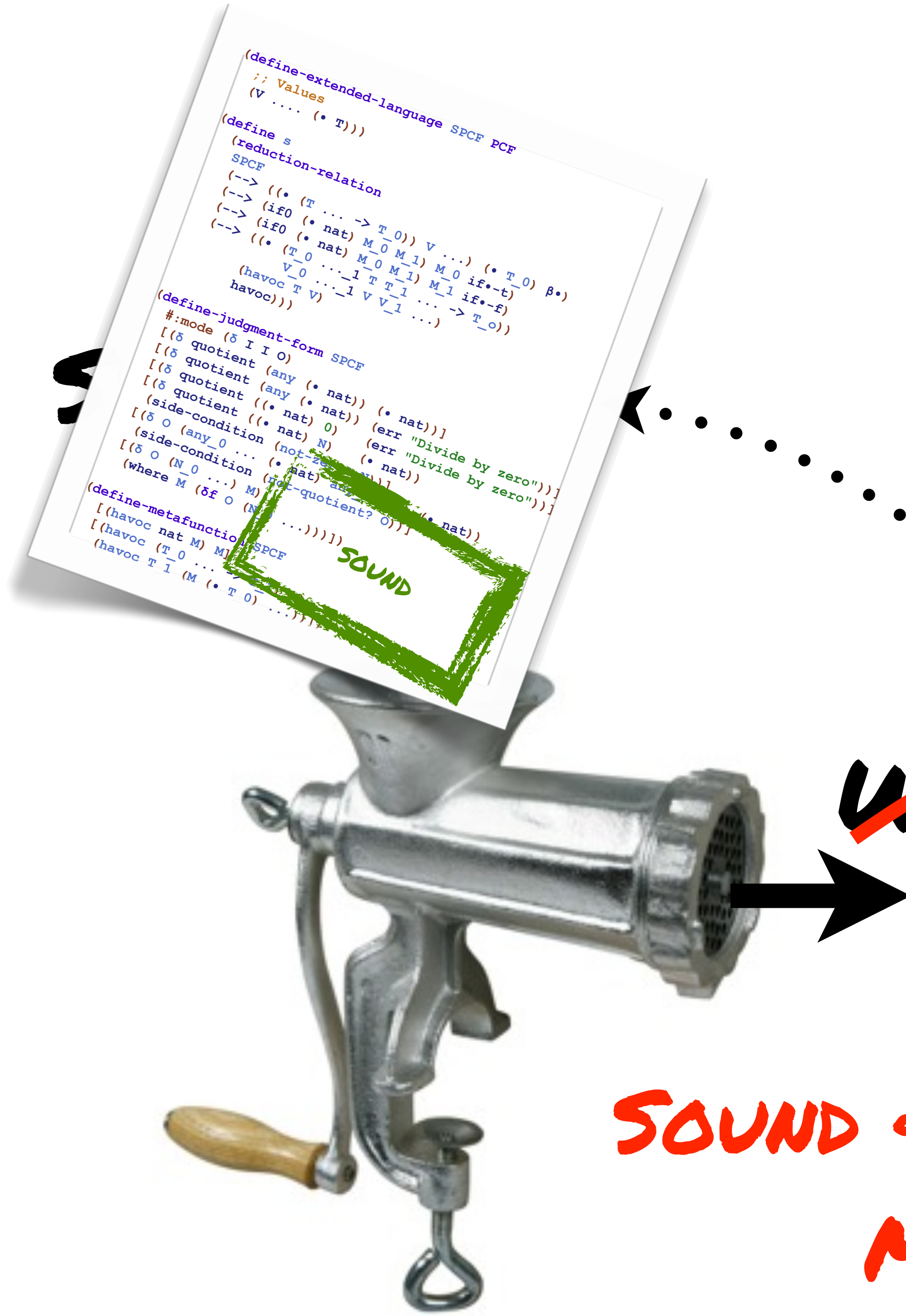
```
(define-extended-language SPCF PCF
  ;; Values
  (V .... (• T)))

(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) ...))])
```

SOUND

LE-PROGRAM
NALYSIS



Whole-Program 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

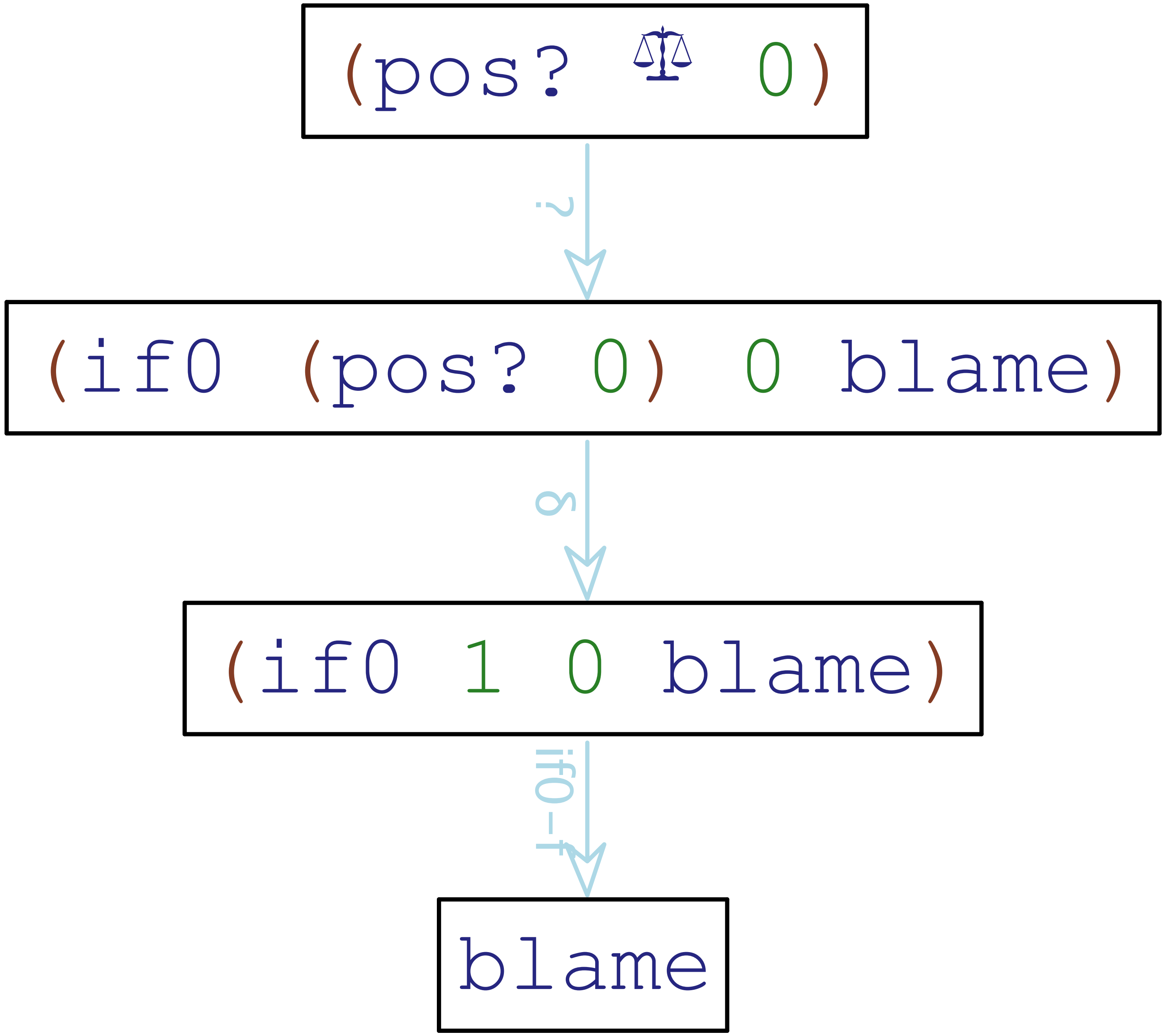
CPCF

```
;; Terms
M ::= .... (C ⚖ M) blame
;; Contracts
C :: = M (C ... -> C)
```

CPCF

```
(pos? ⚖ 7)
    ↓ ?
(if0 (pos? 7) 7 blame)
    ↓ δ
(if0 0 7 blame)
    ↓ if0-t
7
```

CPCF



CPCF

```
((prime? -> even?) ⚖ (λ (x) ...))
```

```
(λ (x) (even? ⚖ ((λ (x) ...) (prime? ⚖ x))))
```

'CPCF

Abstraction


```
(λ ([f : (nat -> nat)])
   (f 3))
```

Abstraction


```
(• ((nat -> nat) -> nat))
```

Abstraction

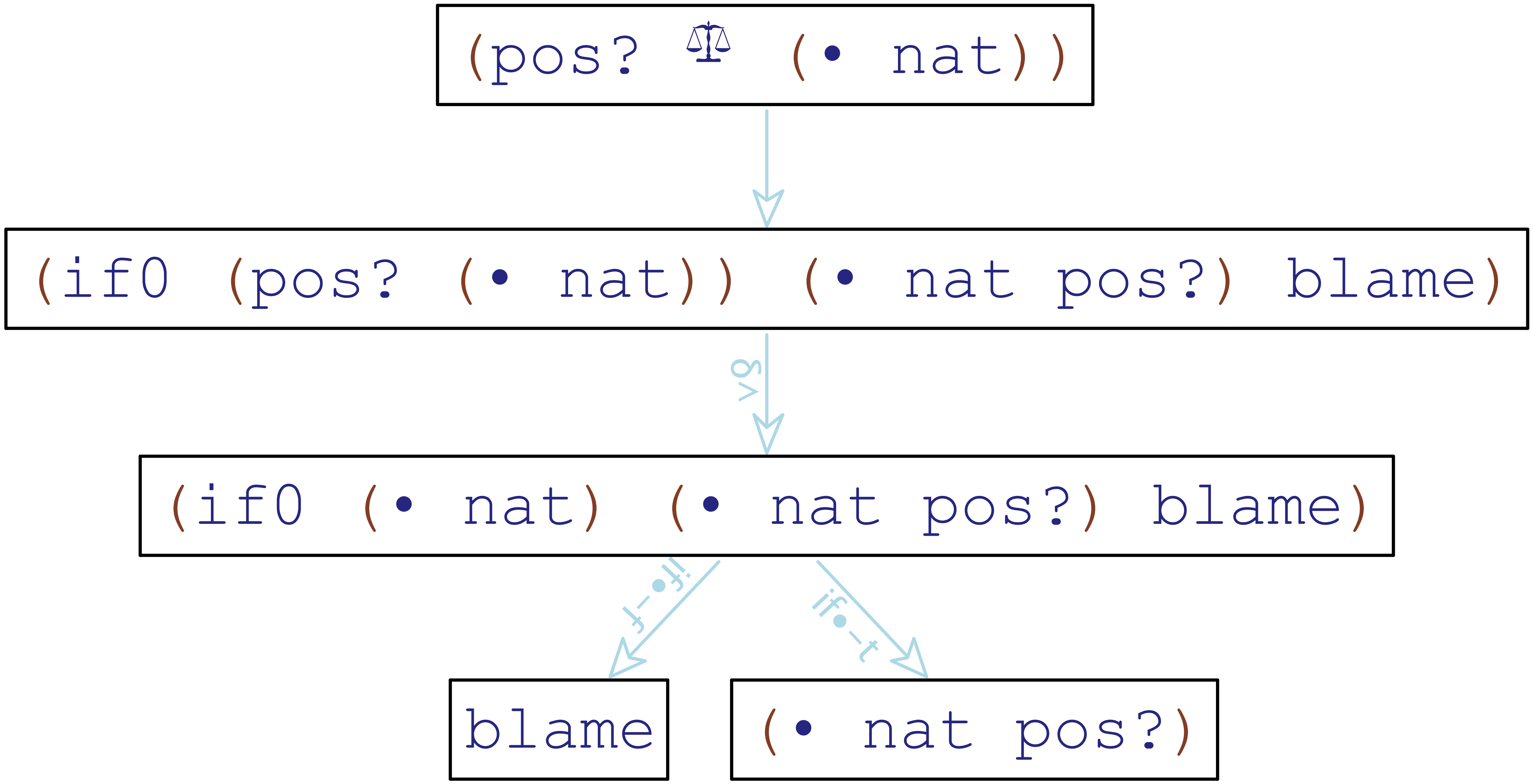

```
(• ((nat -> nat) -> nat)
   ((pos? -> prime?) -> prime?))
```

ABSTRACTION


```
(• ((nat -> nat) -> nat)
   ((pos? -> prime?) -> prime?))
```

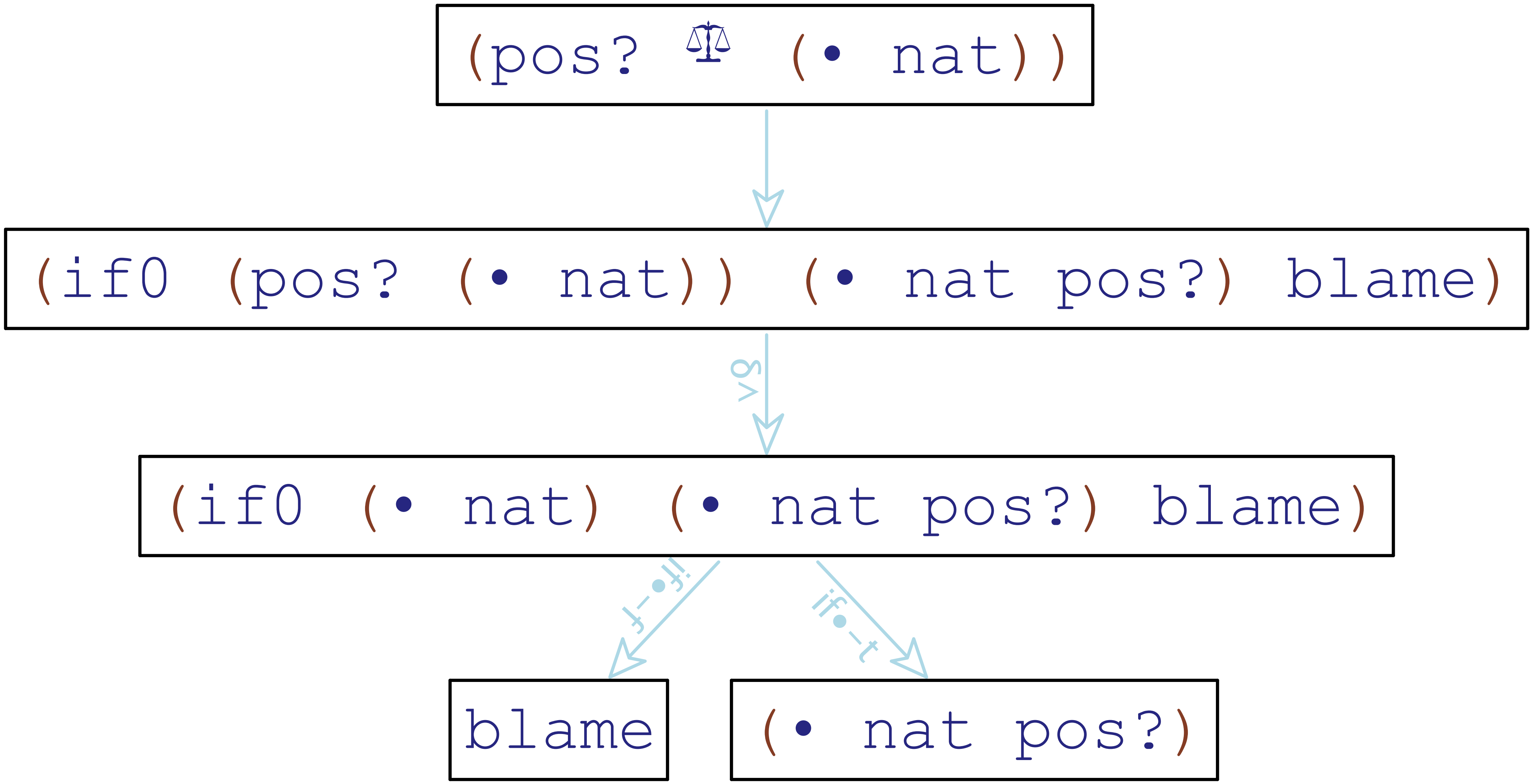
SYMBOLIC VALUES ARE SETS OF CONTRACTS

'CPCF



(pos? ⚖ (• nat))
(if0 (pos? (• nat)) (• nat pos?) blame)
βv
(if0 (• nat) (• nat pos?) blame)
if•-f
if•-t
blame
(• nat pos?)

'CPCF



Symbolic values Remember Contracts they've Satisfied

'CPCF

```
(pos? ⚖ (• nat pos?))
```

↓

```
(• nat pos?)
```

```
(pos? ⚖ (• nat pos?))
```

↓

```
(• nat pos?)
```

Contracts influence computation

'CPCF

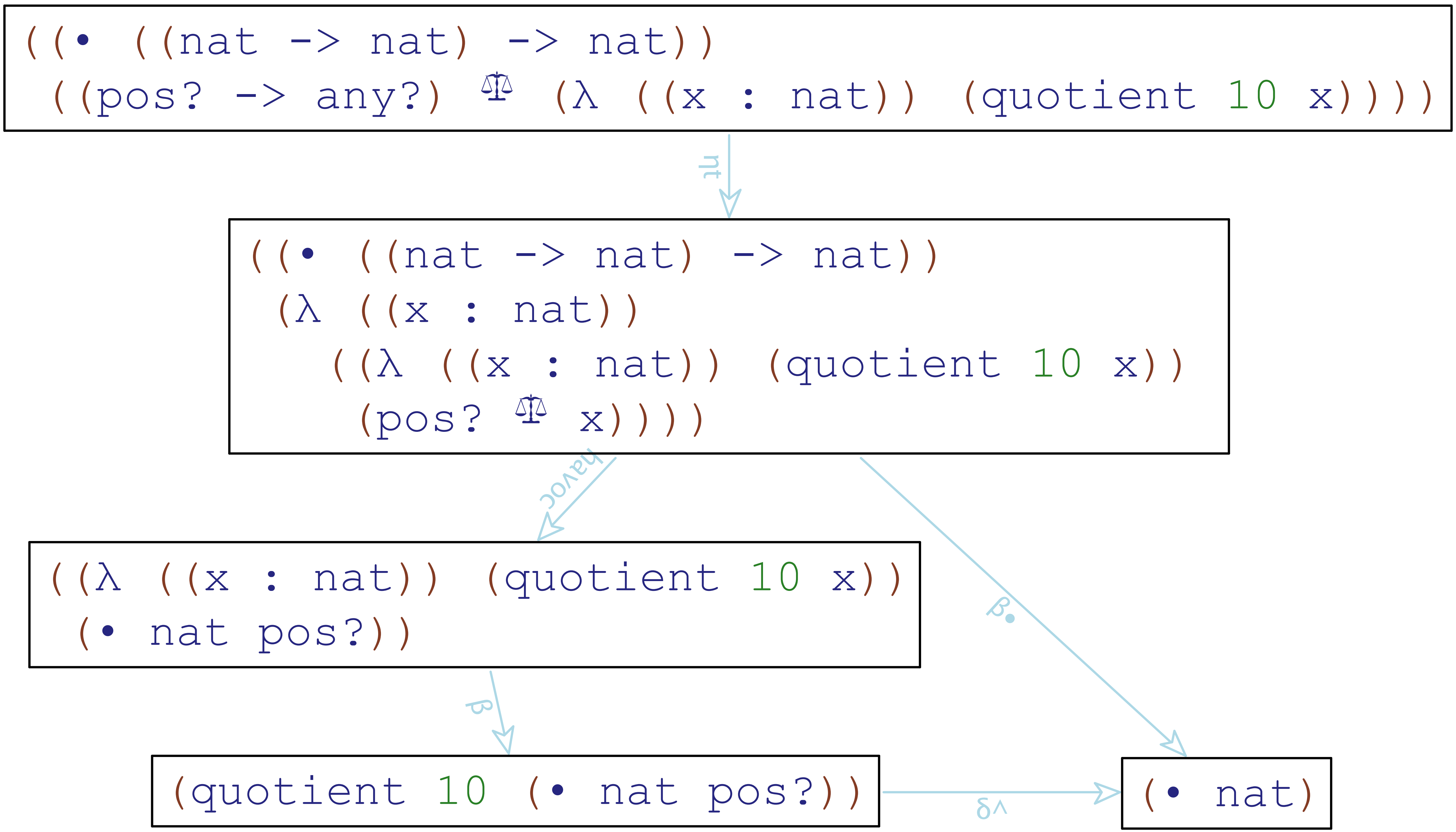
```
(quotient 10 (• nat pos?))
```

v9

```
(• nat)
```

Contracts influence computation

'CPCF


((• ((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))))
havoc
((λ ((x : nat)) (quotient 10 x))
(• nat pos?))
β•
β
(quotient 10 (• nat pos?))
(• nat)

'CPCF

```
((• ((nat -> nat) -> nat))
 ((pos? -> any?) ⚖ (λ ((x : nat)) (quotient 10 x))))
```

ηt ↓

```
((• ((nat -> nat) -> nat))
 (λ ((x : nat))
   ((λ ((x : nat)) (quotient 10 x))
    (pos? ⚖ x))))
```

havoc ↓

```
((λ ((x : nat)) (quotient 10 x))
 (• nat pos?))
```

β ↓

```
(quotient 10 (• nat pos?))
```

δ^ →

```
(• nat)
```

(β• : from the second term directly to `(• 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

'Racket

* Rich Language
* Rich Contracts
* Interactive Verification environment

Rich Language:

* Module system
* Untyped
* Data Structures
* Many Base Types
* Many Primitive Operations

Rich Contracts:

* Dependent Functions
* Data Structures
* Conjunction
* Disjunction
* Recursive Contracts

'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?))]))
```

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

'Racket

```
(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?))]))
```

'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?))]))
```

```
(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]))
```

'Racket

```
(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?))]))
```

'RACKET

```
> (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

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

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 semantics that facilitates fully automated contract verification. Our implementation is capable of verifying contracts expressed in existing programs, and of justifying contract-elimination optimizations.

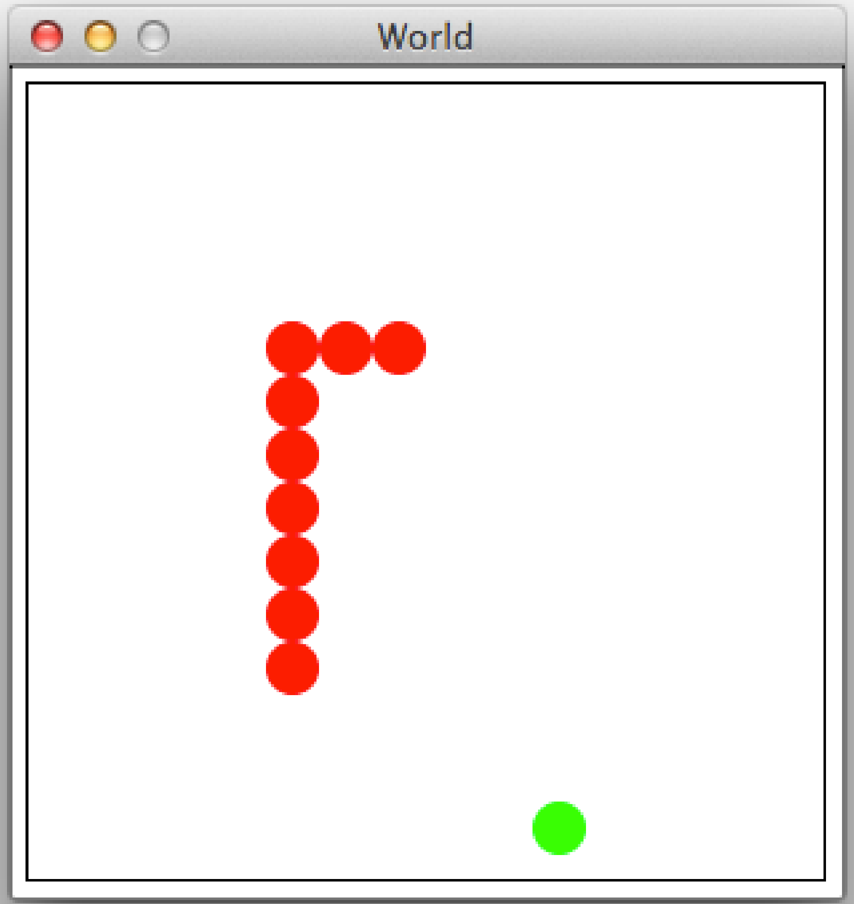
Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory, Verification

Keywords Higher-order contracts, symbolic execution, reduction semantics

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

* Semantics for 'Racket
* Soundness for 'Racket
* Analysis for 'Racket
* Interactive Verification Environment


World

snake.rktl – DrRacket

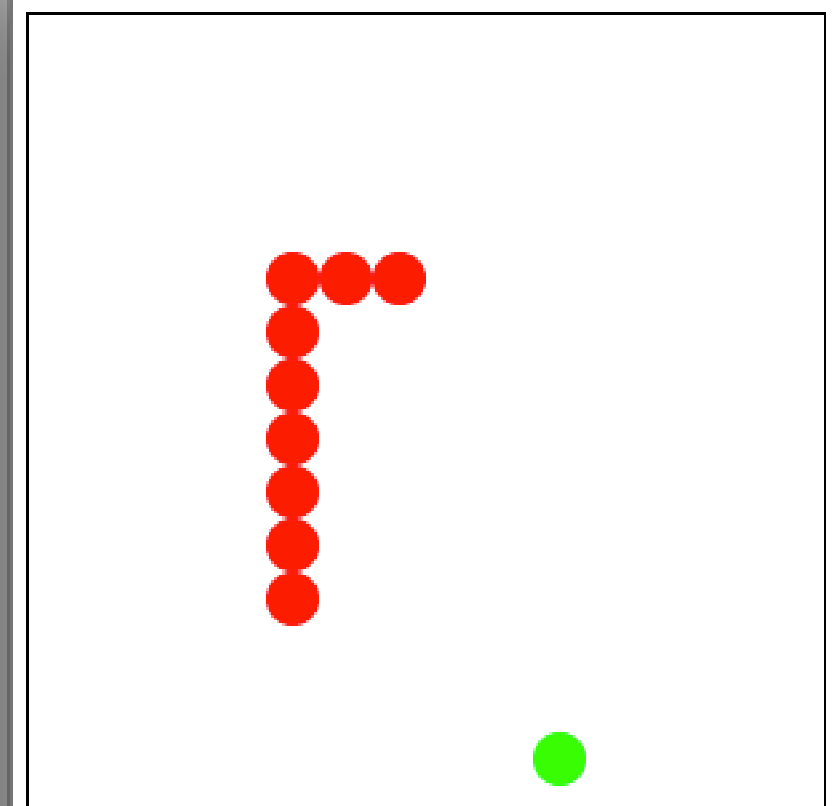
snake.rktl (define ...) Debug Check Syntax Macro Stepper Run Stop

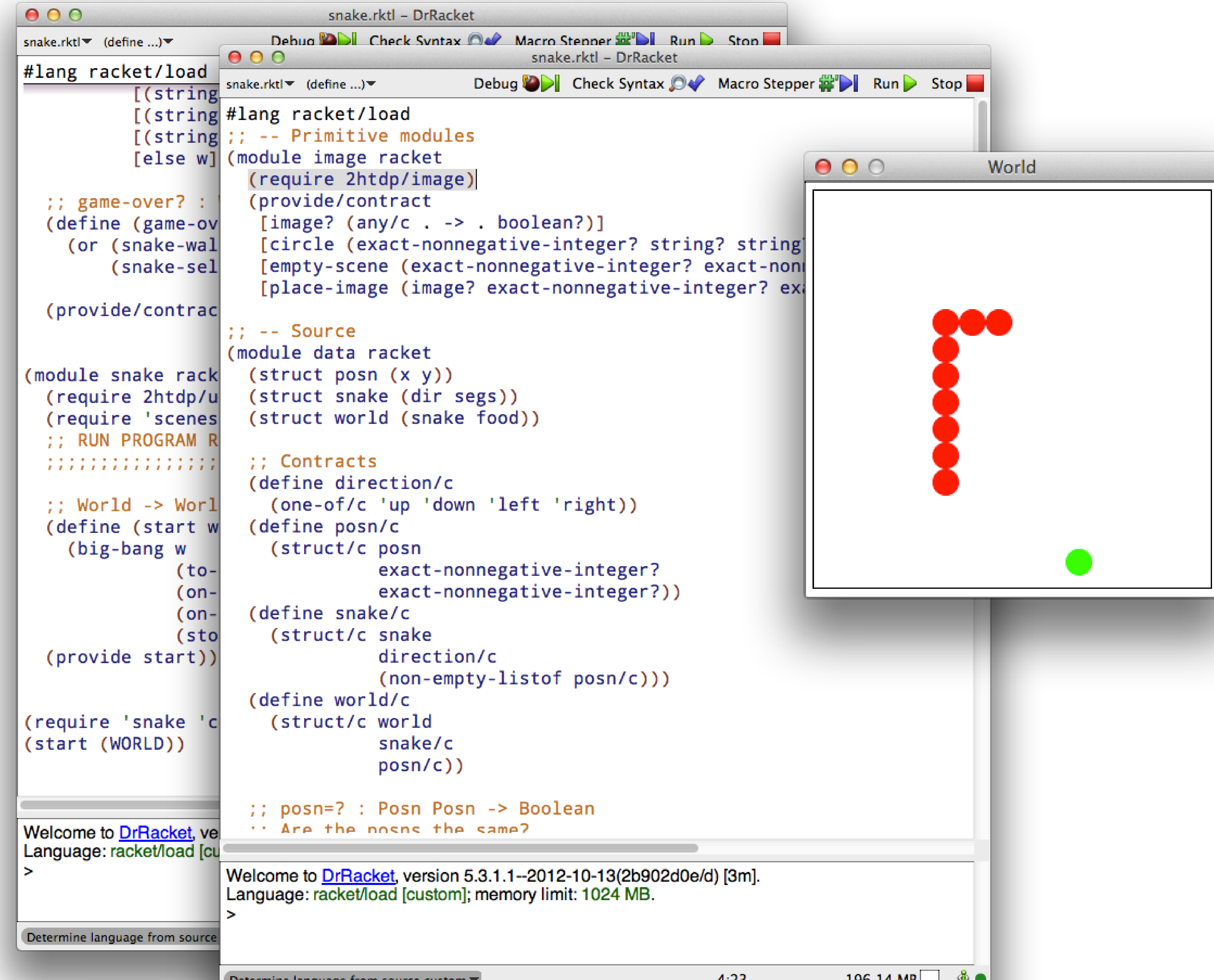
```
#lang racket/load
          [(string=? ke "s") (world-change-dir w 'down)]
          [(string=? ke "a") (world-change-dir w 'left)]
          [(string=? ke "d") (world-change-dir w 'right)]
          [else w]))

  ;; 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))
```

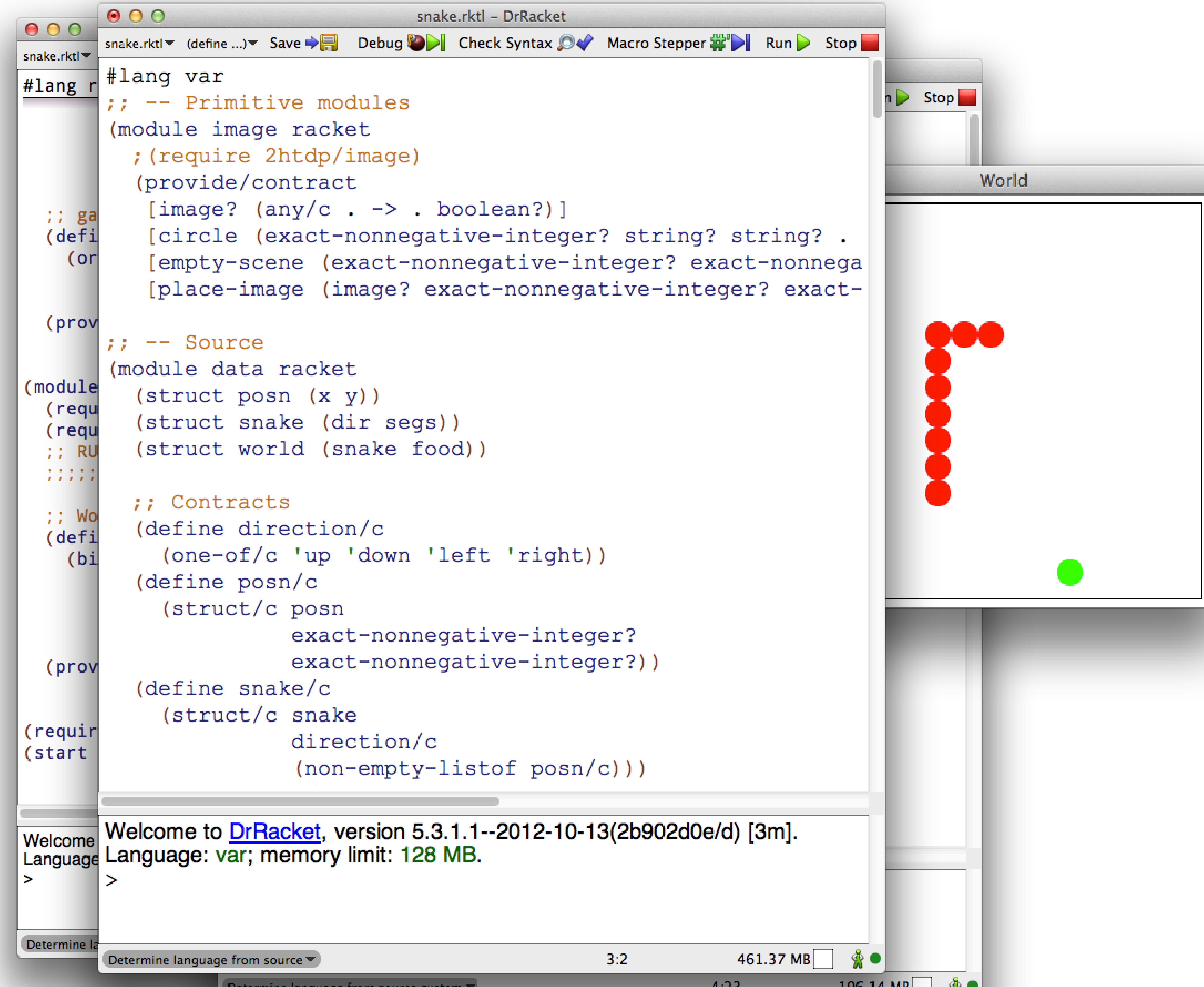
Welcome to DrRacket, version 5.3.1.1--2012-10-13(2b902d0e/d) [3m].
Language: racket/load [custom]; memory limit: 1024 MB.
>

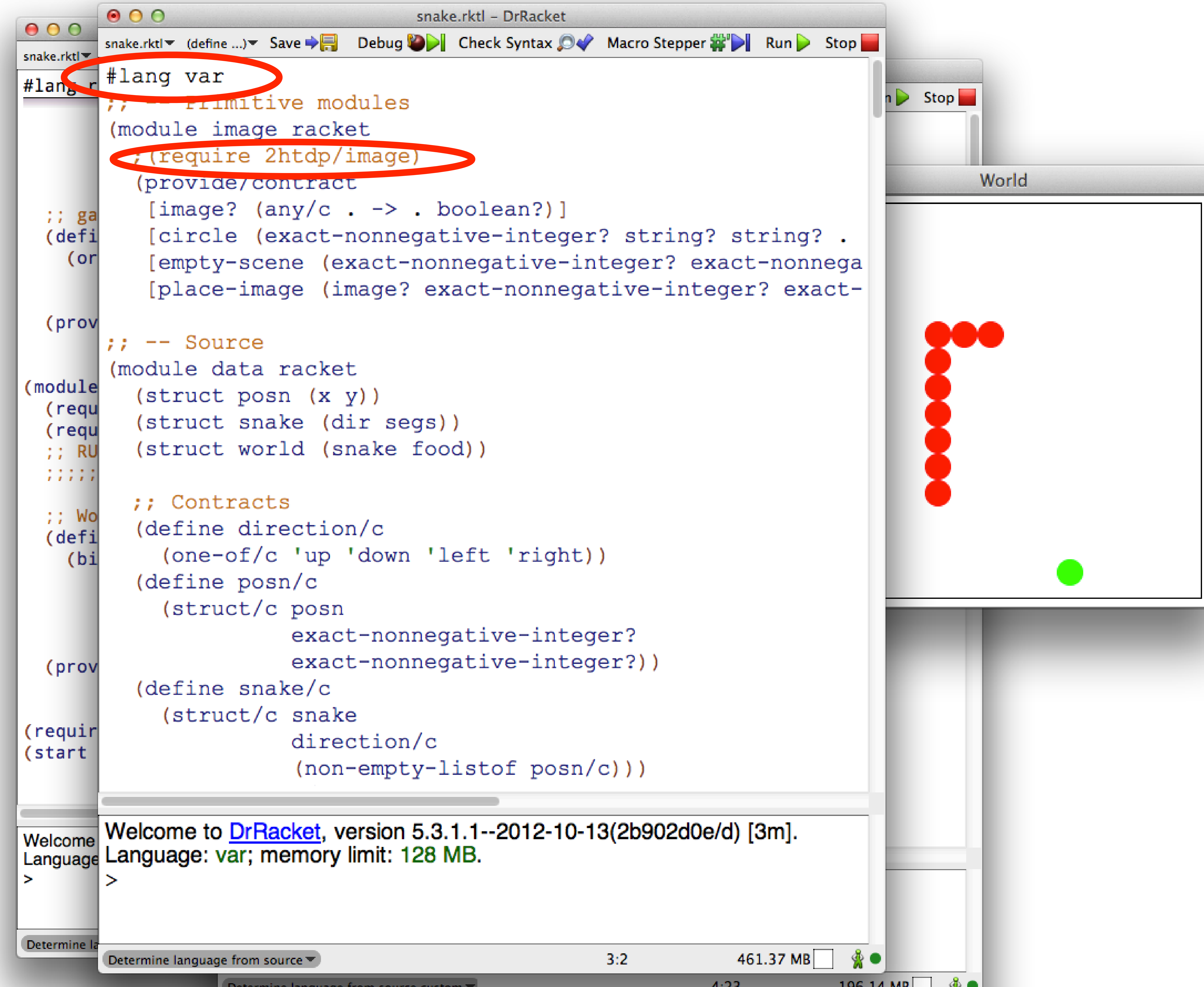
Determine language from source custom 75:22 196.14 MB

World




snake.rktl – DrRacket
snake.rktl (define ...)
Debug Check Syntax Macro Stepper Run Stop
#lang racket/load
;; -- Primitive modules
(module image racket
(require 2htdp/image)
(provide/contract
[image? (any/c . -> . boolean?)]
[circle (exact-nonnegative-integer? string? string
[empty-scene (exact-nonnegative-integer? exact-non
[place-image (image? exact-nonnegative-integer? ex
;; -- 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)))
(define world/c
(struct/c world
snake/c
posn/c))
;; posn=? : Posn Posn -> Boolean
Welcome to DrRacket, version 5.3.1.1--2012-10-13(2b902d0e/d) [3m].
Language: racket/load [custom]; memory limit: 1024 MB.
>
World


snake.rktl – DrRacket
snake.rktl (define ...) Save Debug Check Syntax Macro Stepper Run Stop
#lang var
;; -- Primitive modules
(module image racket
;(require 2htdp/image)
(provide/contract
[image? (any/c . -> . boolean?)]
[circle (exact-nonnegative-integer? string? string? .
[empty-scene (exact-nonnegative-integer? exact-nonnega
[place-image (image? 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.
>
Determine language from source
3:2
461.37 MB
World
Stop

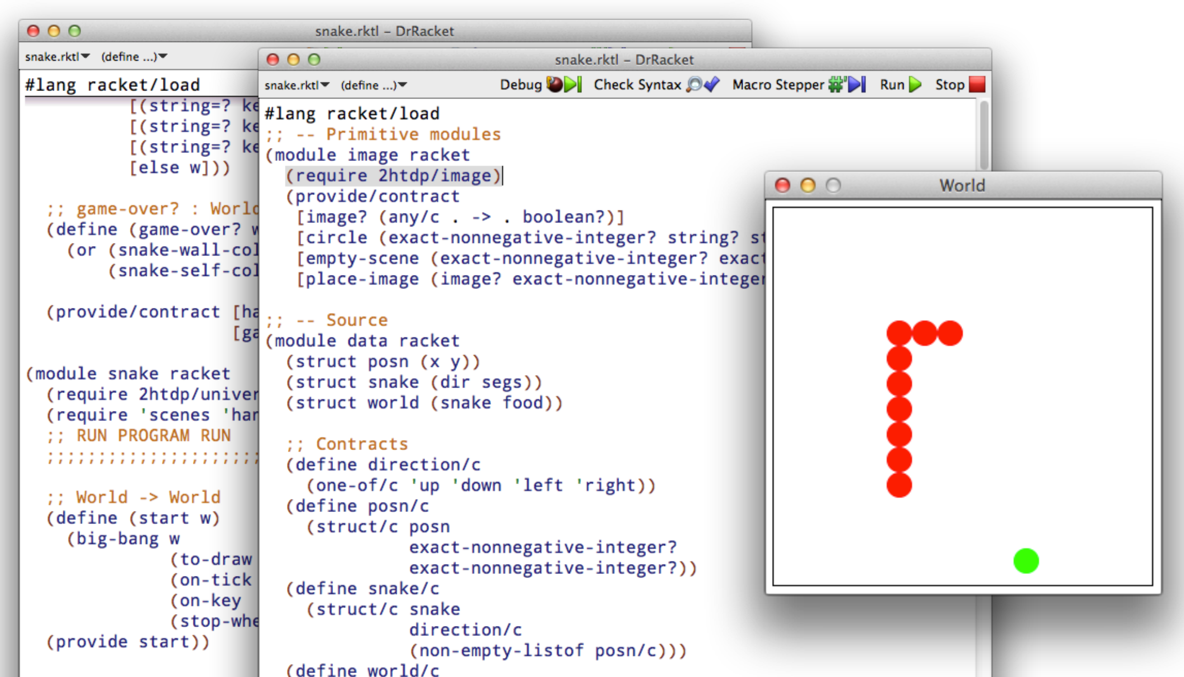

snake.rktl – DrRacket
snake.rktl (define ...) Save Debug Check Syntax Macro Stepper Run Stop
#lang var
;; -- Primitive modules
(module image racket
;(require 2htdp/image)
(provide/contract
[image? (any/c . -> . boolean?)]
[circle (exact-nonnegative-integer? string? string? .
[empty-scene (exact-nonnegative-integer? exact-nonnega
[place-image (image? 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.
>
Determine language from source
3:2
461.37 MB
World
Stop

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/