

# CFAL = Lazy FAE

```
<CFAL> ::= <num>
        | { + <CFAL> <CFAL> }
        | { - <CFAL> <CFAL> }
        | <id>
        | { fun { <id> } <CFAL> }
        | { <CFAL> <CFAL> }
```

{ { fun { x } 0 } { + 1 { fun { y } 2 } } } ⇒ 0

{ { fun { x } x } { + 1 { fun { y } 2 } } } ⇒ *error*

# Implementing CFAL

Option #1: Run the FAE interpreter in PLAI Lazy!

```
; interp : CFAL SubCache -> CFAL-Value
(define (interp expr sc)
  ...
  [app (fun-expr arg-expr)
       (local [(define fun-val
                  (interp fun-expr sc))
                (define arg-val
                  (interp arg-expr sc))]
             (interp (closureV-body fun-val)
                     (aSub (closureV-param fun-val)
                           arg-val
                           (closureV-sc fun-val))))])])
```

`arg-val` never used  $\Rightarrow$  `interp` call never evaluated

# Implementing CFAL

Option #2: Use PLAI Advanced and explicitly delay `arg-expr` interpretation

```
; interp : CFAL SubCache -> CFAL-Value
(define (interp expr sc)
  ...
  [app (fun-expr arg-expr)
    (local [(define fun-val
              (interp fun-expr sc))
            (define arg-val
              (exprV arg-expr sc))]
      (interp (closureV-body fun-val)
              (aSub (closureV-param fun-val)
                    arg-val
                    (closureV-sc fun-val))))])])
```

where `exprV` is a new kind of `CFAL-Value`

# CFAL Values

```
(define-type CFAL-Value
  [numV (n number?)]
  [closureV (param symbol?)
             (body CFAL?)
             (sc SubCache?)]
  [exprV (expr CFAL?)
         (sc SubCache?)])
```

# Forcing Evaluation for Number Operations

```
(interp {{fun {x} {+ 1 x}} 10} (mtSub))
```

⇒ *error: expected numV, got exprV*

```
(define (num-op op op-name x y)
  (numV (op (numV-n (strict x))
            (numV-n (strict y)))))

(define (num+ x y) (num-op + '+ x y))
(define (num- x y) (num-op - '- x y))

; strict : CFAL-Value -> CFAL-Value
(define (strict v)
  (type-case CFAL-Value v
    [exprV (expr sc) (strict (interp expr sc))]
    [else v]))
```

# Forcing Evaluation for Application

```
(interp {{fun {f} {f 1}} {fun {x} {+ x 1}}}}  
  (mtSub))
```

```
; interp : CFAL SubCache -> CFAL-Value  
(define (interp expr sc)  
  ...  
  [app (fun-expr arg-expr)  
    (local [(define fun-val  
              (strict (interp fun-expr sc)))  
            (define arg-val  
              (exprV arg-expr sc))]  
      (interp (closureV-body fun-val)  
              (aSub (closureV-param fun-val)  
                    arg-val  
                    (closureV-sc fun-val))))])])
```

# Redundant Evaluation

```
{ { fun {x} {+ {+ x x} {+ x x}} }  
  {- {+ 4 5} {+ 8 9}} }
```

How many times is `{+ 8 9}` evaluated?

Since the result is always the same, we'd like to evaluate `{- {+ 4 5} {+ 8 9}}` at most once

# Caching Strict Results

```
(define-type CFAL-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body CFAL?)
            (sc SubCache?)]
  [exprV (expr CFAL?)
         (sc SubCache?)
         (value (box-of (false-or CFAL-Value?)))]])

; strict : CFAL-Value -> CFAL-Value
(define (strict v)
  (type-case CFAL-Value v
    [exprV (expr sc value-box)
     (if (false? (unbox value-box))
         (local [(define v (strict (interp expr sc)))]
             (begin
               (set-box! value-box v)
               v))
         (unbox value-box))]
    [else v]))
```

# Etc.

```
(define (false? v)
  (and (boolean? v)
        (not v)))
```

```
(define (box-of p)
  (lambda (v)
    (and (box? v)
          (p (unbox v))))))
```

```
(define (false-or p)
  (lambda (v)
    (or (false? v)
         (p v))))
```

```
(define (interp expr sc)
  ...
  [app ...
    (exprV arg-expr sc (box #f))
    ...])
```