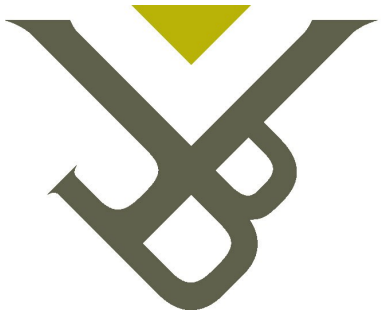
Iterative typing

Master thesis

Author: *Laurent Christophe*

Promotor: *Wolfgang De Meuter*
Advisor: *Dries Harnie*

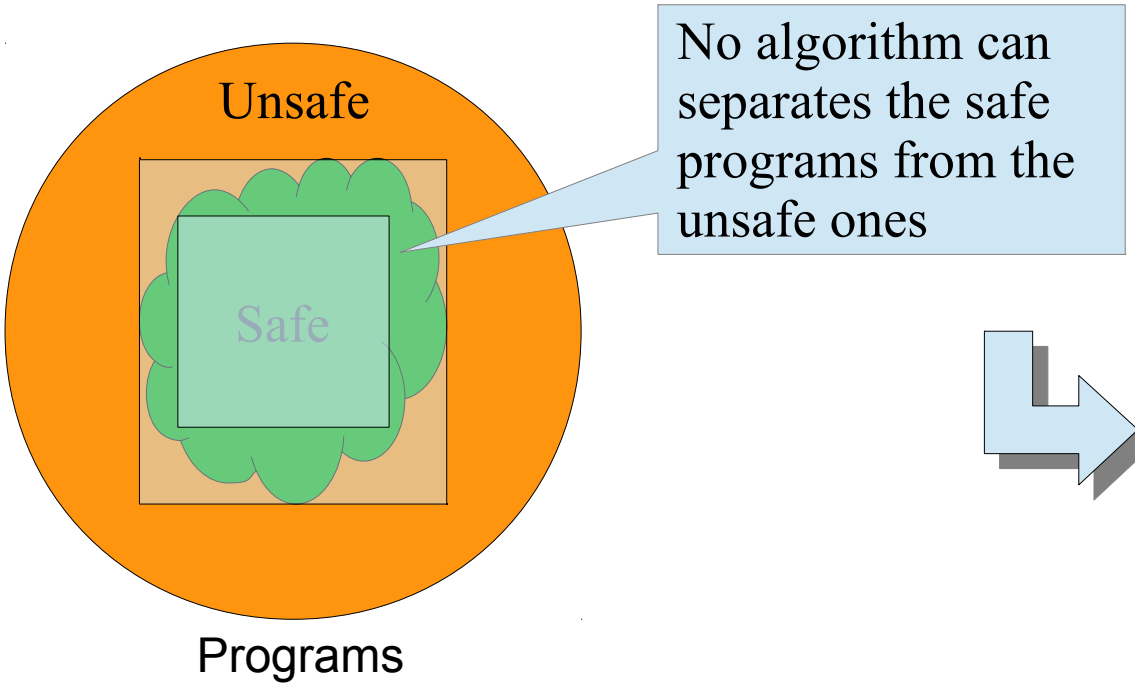


Vrije
Universiteit
Brussel

Static safety vs Expressiveness


```
square(2);
```

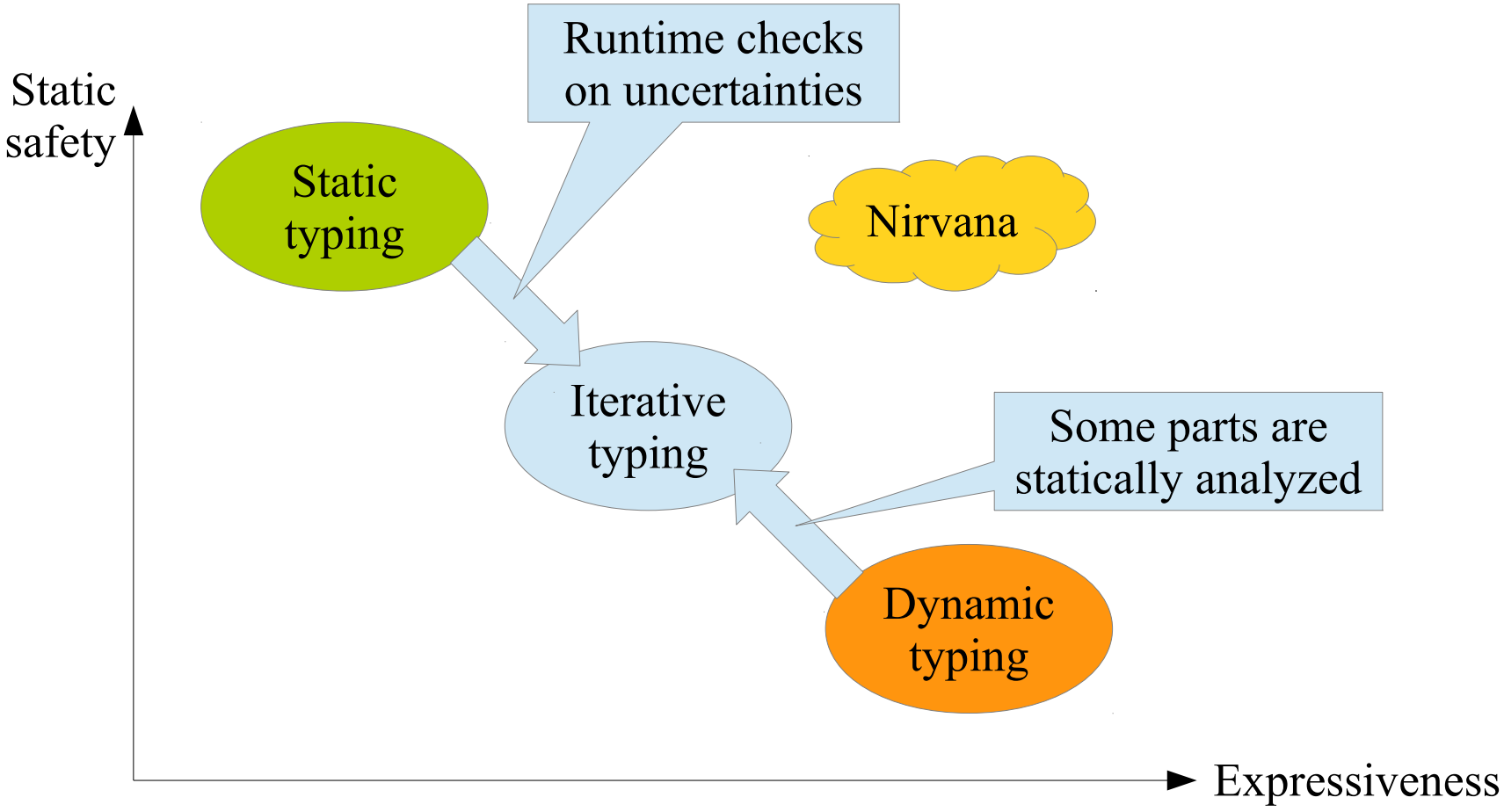
```
square("foo");
```



No algorithm can separates the safe programs from the unsafe ones

A compromise must be made between static safety and expressiveness

The iterative compromise



Iterative typing → Static typing that delays some type checks to runtime

Case study: Deserialization (1/2)

Error detected on primitive call

```
(define x (read))
(+ x 1)
```

```
>> “foo”
Error: + expects type
  <number> as 1st
  argument, given: "foo"
```

Error detected during the string interpretation

```
main = do x <- getLine
          y <- return (read x)
          return (y + 1)
```

```
>> “foo”
Prelude.read: no parse
```

Case study: Deserialization (2/2)

```
(define x (read))
...
...
(+ x 1)
```

Late error detection

Runtime type check, early error detection

Iterative compilation

```
(define x ([r => Number] read))
...
...
(+ x 1)
```

```
>> “foo”
Error on [String => Number]
```

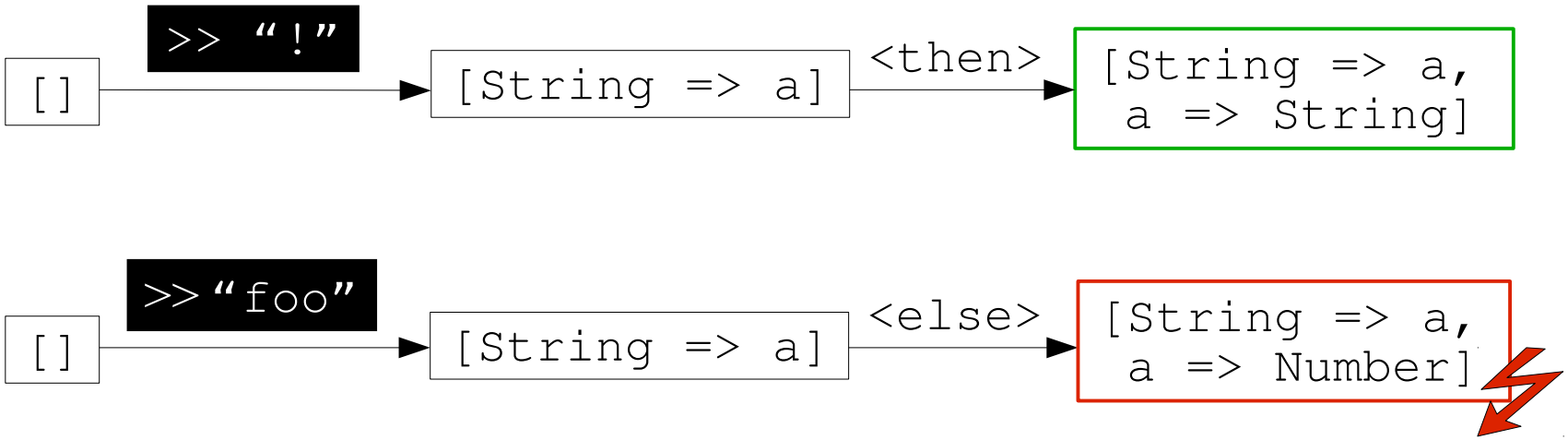
Input dispatch

```
(define input (read))
(if (equal? input “!”)
    (concat “Yo” input)
    (+ input 1))
```

```
(define input ([r => a] read))
(if (equal? input “!”)
    [a => String](concat “Yo” input)
    [a => Number](+ input 1))
```

```
>> “!”
“Yo!”
```

```
>> “foo”
Error on: [Number => String]
```



The “evil” case

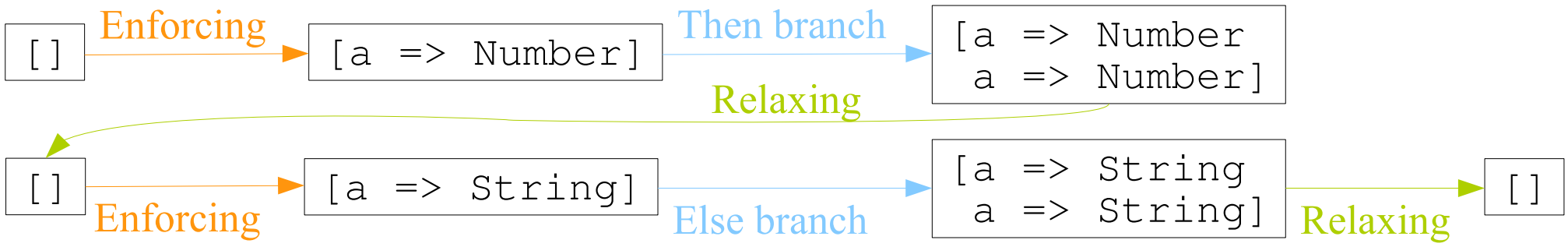
```
(define (evil x)
  (if x 1 “foo”))

(+ 1 (evil #t))
(concat “yo” (evil #f))
```

```
2
“yofoo”
```

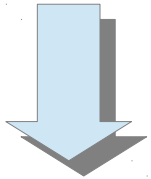
```
;; evil :: ∀ a . Bool -> a
(define (evil x)
  (if x
      [a => Number] 1
      [a => String] “foo”))

(+ 1 ([a => Number] evil #t))
(concat “yo” ([a => String] evil #f))
```

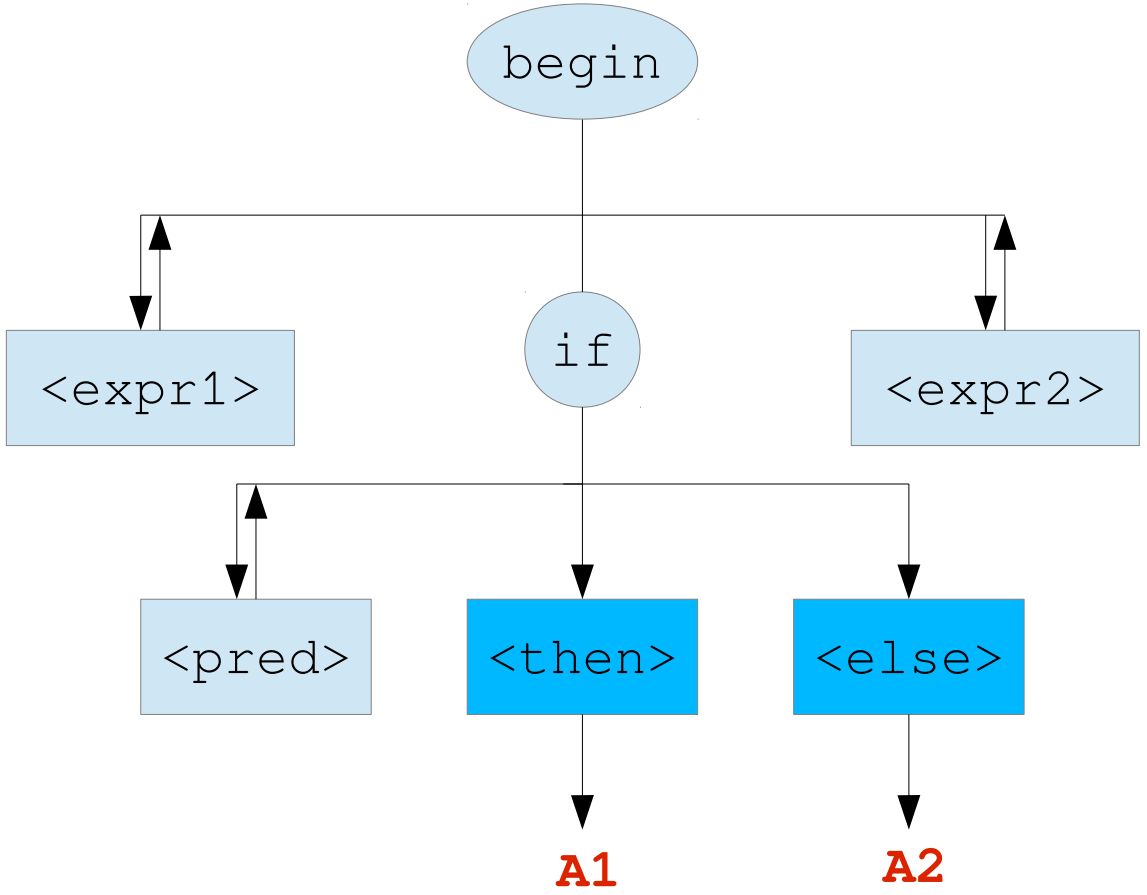


Compilation overview

```
(begin <expr1>
       (if <pred>
           <then>
           <else>)
       <expr2>)
```



```
(begin <expr1>
       (if <pred>
           A1 <then>
           A2 <else>)
       <expr2>)
```



Iterative type system

$$\text{ID}\ \frac{x : \sigma \in A}{A \vdash x : \sigma} \qquad \text{CONST}\ \frac{type(c) = \gamma}{A \vdash c : \gamma}$$

$$\text{APP}\ \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau} \qquad \text{ABS}\ \frac{A \cup \{x : \tau\} \vdash e : \tau'}{A \vdash (\lambda\ x\ e) : \tau \to \tau'}$$

$$\text{LET}\ \frac{A \cup \{x : \tau\} \vdash e : \sigma \qquad A \cup \{x : \sigma\} \vdash e' : \tau' \qquad \tau \sqsubseteq \sigma}{A \vdash (let\ x\ e\ e') : \tau'}$$

$$\text{GEN}\ \frac{A \vdash e : \forall\{\alpha_i\}.\tau \to \tau' \qquad \alpha \notin free(A)}{A \vdash e : \forall\alpha.\forall\{\alpha_i\}.\tau \to \tau'} \qquad \text{SPE}\ \frac{A \vdash e : \forall\{\alpha_i\}.\tau \qquad \tau' = [\alpha_i \mapsto \tau_i]\,\tau}{A \vdash [\alpha_i \mapsto \tau_i]\ e : \tau'}$$

Only procedures can be polymorphic

```
;; read :: ∀ a . () -> a
(+ 1 ([a => Number] read))
```

Conditional rule

Encapsulation of assumptions made inside the “then” branche

$$\text{COND}\ \frac{A \vdash e_p : \tau_p \qquad [\alpha'_i \mapsto \tau'_i]\, A \vdash e' : \tau \qquad [\alpha''_i \mapsto \tau''_i]\, A \vdash e'' : \tau}{A \vdash (if\ e_p\ [\alpha'_i \mapsto \tau'_i]\ e'\ [\alpha''_i \mapsto \tau''_i]\ e'') : \tau}$$

The assumptions made inside the “then” branch are inserted into the code

```
(if x
    1
    "foo")
```

$$\text{COND}\ \frac{\text{ID}\ \dfrac{x : B \in \{x : B\}}{\{x : B\} \vdash x : B} \qquad \text{CONST}\ \dfrac{type(1) = N}{\{x : B\} \vdash 1 : N} \qquad \text{CONST}\ \dfrac{type(\text{"}foo\text{"}) = S}{\{x : B\} \vdash \text{"}foo\text{"} : S}}{\{x : B\} \vdash (if\ x\ [\alpha \mapsto N]\ 1\ [\alpha \mapsto S]\ \text{"}foo\text{"}) : \alpha}$$

Static conditional structure

Preserving side effects

```
(unsafe <body>)
```

If a type error is predicted, the iterative part of the interpretation is shut down instead of raising an error

```
(define x (read))
(display “Log info: ”)
(display (typeOf x))
(+ 1 x)
```

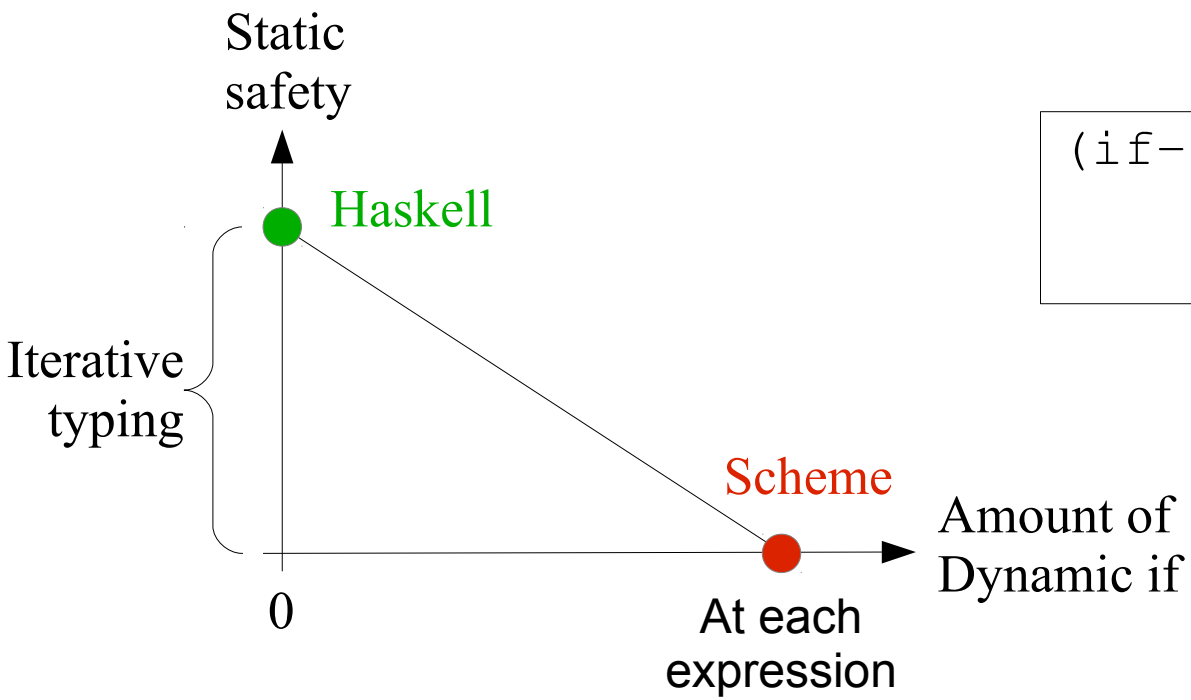
```
>> “foo”
Error on [String => Number]
```

```
(unsafe
  (define x (read))
  (print “Log info: ”)
  (print (typeOf x))
  (+ 1 x))
```

```
>> “foo”
Log info: String
Error: + expects type
  <number>  as 1st
  argument, given: "foo"
```

Conclusion

Static conditional structure




```
(if-static <pred>
           <cons>
           <alt>)
```

Both branches must be type-compatible

```
(define (map f xs)
  (if-static (null? xs)
             null
             (cons (f (car xs))
                   (map f (cdr xs)))))
```

Predicting the right path

```
(cond-viable
  <attempt1>
  ...
  <attemptN>)
```

Look for a branch compatible with the current locus

```
(define (over-arg x)
  (cond-viable
    (+ 1 x)
    (concat “yo” x)
    (or #f x)
    (error “???”)))

(over-arg 2)
(over-arg “foo”)
```

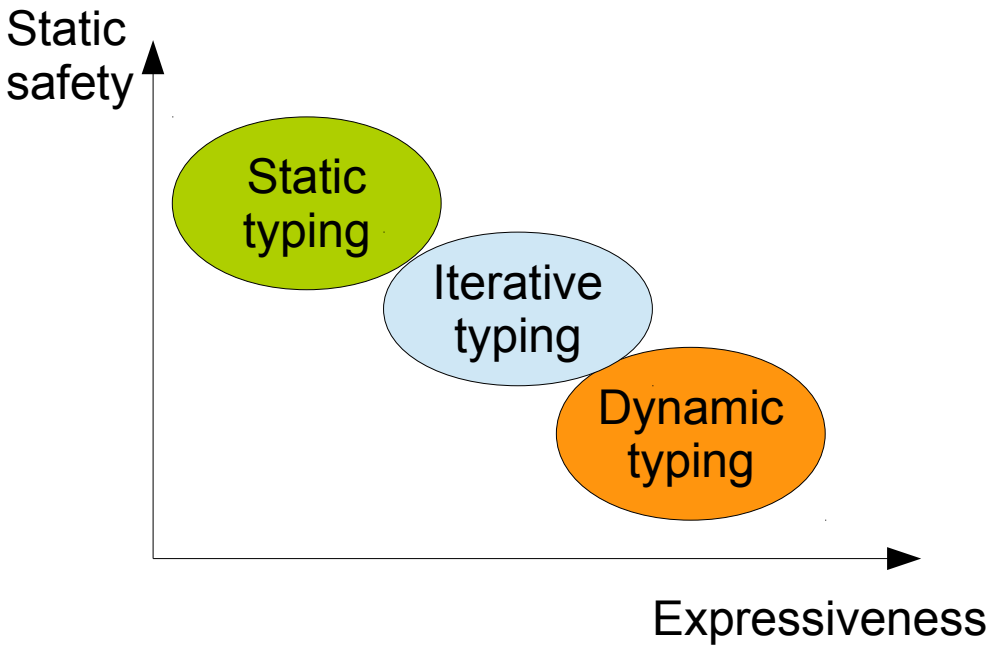
```
3
“yofoo”
```

```
(define (over-res)
  (cond-viable
    1
    “foo”
    #t
    (error “???”)))

(+ 2 (over-res))
(or #f (over-res))
```

```
3
#t
```

Conclusion




```
(if <pred>
    <A-then> <then>
    <A-else> <else>)
```

Questions?