Iterative typing on Scheme

Christophe, Laurent

Contents:

- Introduction to type theory
- Gradual typing
- Iterative typing

Introduction to type theory

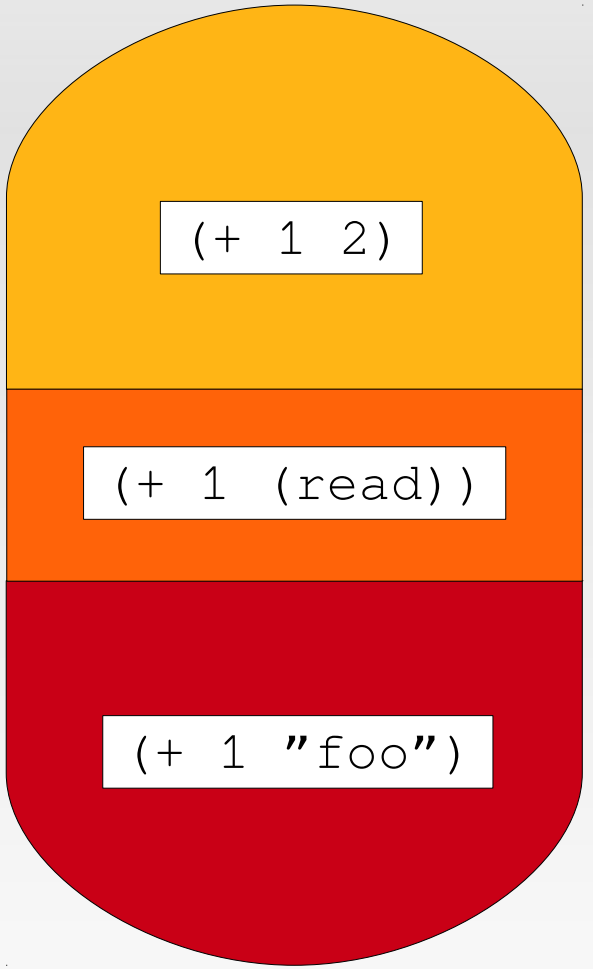
Some definitions:

- Type
- Type error
- Type system
- Type inference
- Type checking

Introduction to type theory

Some definitions:

- Type
- Type error
- Type system
- Type inference
- Type checking
- Well / Ill typed programs



Type inference & type annotation

$$\frac{\dfrac{\texttt{+::(n→n→n), x::n, 1.0::f} \vdash \texttt{+::(n→n→n), x::n, 1.0::f} \quad \texttt{n} \leq \texttt{n} \quad \texttt{f} \leq \texttt{n}}{\texttt{+::(n→n→n), x::n, 1.0::f} \vdash \texttt{(+ x 1.0)::n}}}{\texttt{+::(n→n→n), 1::n} \vdash \texttt{(lambda(x)(+ x 1.0))::n→n}}$$

Type inference & type annotation

$$\frac{\dfrac{\texttt{+::(n→n→n), x::n, 1.0::f} \vdash \texttt{+::(n→n→n), x::n, 1.0::f} \qquad \texttt{n} \leq \texttt{n} \quad \texttt{f} \leq \texttt{n}}{\texttt{+::(n→n→n), x::n, 1.0::f} \vdash \texttt{(+ x 1.0)::n}}}{\texttt{+::(n→n→n), 1::n} \vdash \texttt{(lambda(x)(+ x 1.0))::n→n}}$$

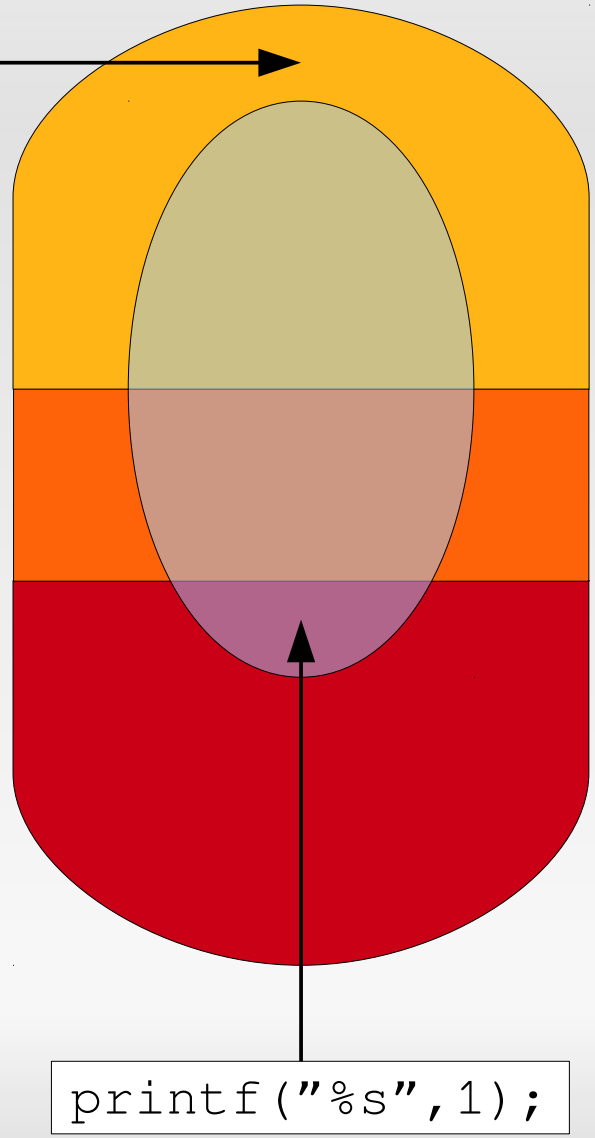
	Few annotations	A lot of annotations
Type inference	- Difficult	+ Easier
Documentation	- Poor	+ Good
Flexibility	+ Better	- Bad

Type checking: static vs dynamic

	Static checking	Dynamic checking
During	Compilation	Running
Dicriminate	Program	Execution

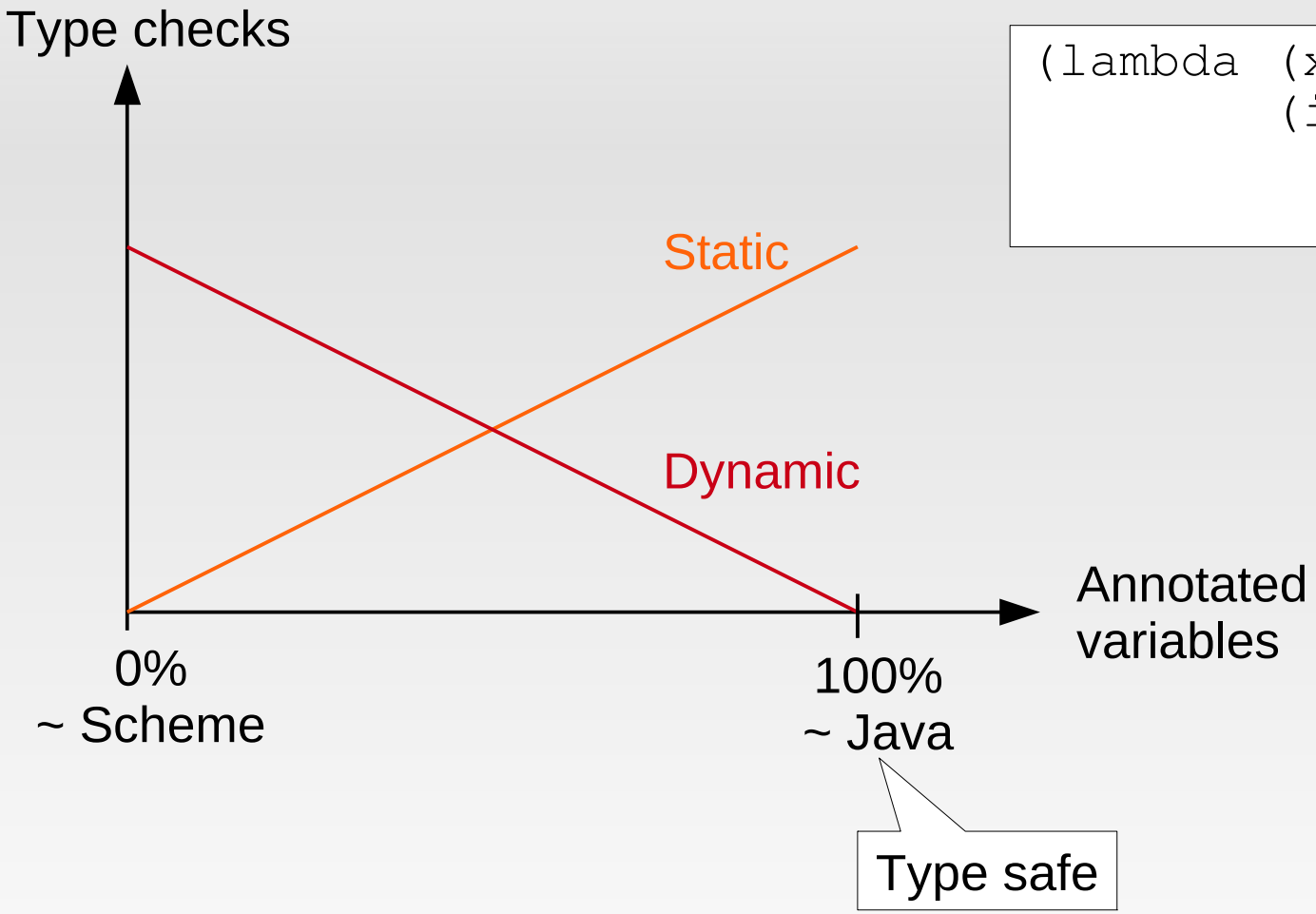
Type checking: static vs dynamic

```
if (TRUE){
    1;
} else {
    1 + "foo";
}
```



	Static checking	Dynamic checking
During	Compilation	Running
Dicriminate	Program	Execution
	- Accepted programs ≠ well typed	+ Only accept type safe executions
Detection	+ Sooner	- Later
Speed	+ Faster	- Slower

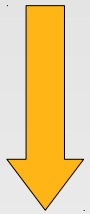
Gradual typing: use




```
(lambda (x :: number)
        (if (< x 0)
            "negative"
            "positive"))
```

Gradual typing: how it works?

```
(lambda (x)
        (if (< x 0)
            ”negative”
            ”positive”))
```

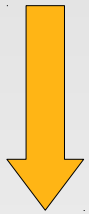


```
(lambda (x :: ?)
        (if (< x 0)
            ”negative”
            ”positive”))
```

Runtime check:
(? → number)

Gradual typing: how it works?

```
(lambda (x)
        (if (< x 0)
            ”negative”
            ”positive”))
```



```
(lambda (x :: ?)
        (if (< x 0)
            ”negative”
            ”positive”))
```

Runtime check:
(? → number)

- Consistency relation

```
((f :: a → b) (x :: c))
```

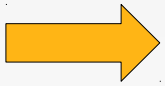
```
a ~ c
```

- Some Axioms

```
? ~ t
t ~ ?
t ~ t
```

- No transitivity!

```
t ~ ?
? ~ u
```

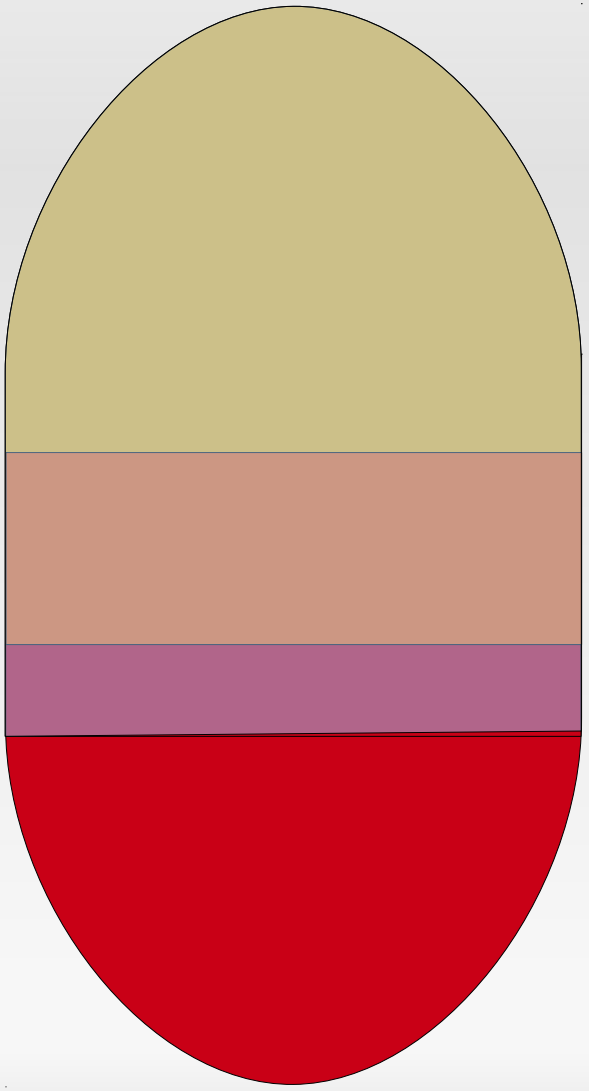


```
t ~ u
```

Iterative typing: leitmotiv

Goals

- As permisive as dynamic languages
- Detect type errors as soon as possible
- No type annotation



Some pieces of code make type inference very difficult; to resolve those uncertainties, we need runtime informations. As soon as those informations are available the inference is refined and remaining type checks performed.

Iterative typing: behaviour

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

```
(define input [(read) :: num])
.
.
.
(+ 1 input)
```

```
(+ 1 ”hell yeah”)
```

Get rejected like a boss

Iterative typing: behaviour

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

```
(define input [(read) :: num])
.
.
.
(+ 1 input)
```

```
(+ 1 "hell yeah")
```

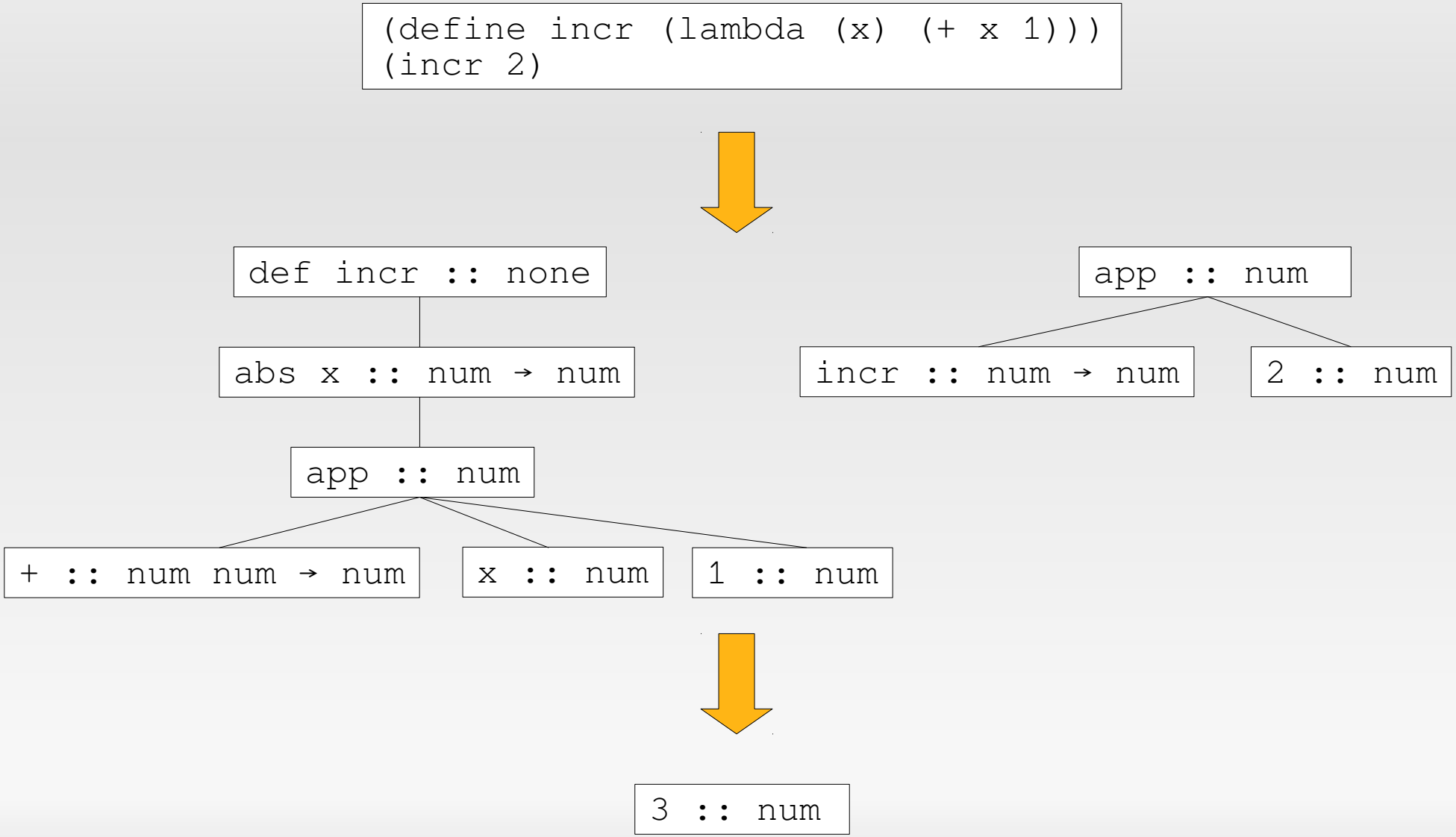
Get rejected like a boss

```
(define zeros
 (lambda (x)
  (if (equal? x 0)
      '()
      (cons 0 (zeros (- x 1))))))
.
.
.
(car (zeros 1))
(cdr (zeros 0))
```

```
(define zeros
 (lambda (x)
  (if (equal? x 0)
      '()
      (cons 0 (zeros (- x 1))))))
.
.
.
(car [(zeros 1) :: a.b])
(cdr [(zeros 0) :: a.b])
```

Always fail

Iterative typing: my work so far



Happy end

What remains?

- Introduce the dynamic type into my framework
- Place runtime checks
- Justify my algorithm

Questions?