Where are you going with those types?

Vincent St-Amour, Sam Tobin-Hochstadt,
Matthew Flatt, Matthias Felleisen

PLT / Northeastern University
Boston, MA, USA

PLT / University of Utah
Salt Lake City, UT, USA

IFL 2010 - September 3rd, 2010

Generating fast code in the presence of ad-hoc polymorphism is hard.

Case study: generic arithmetic

```
(+ 2      2)
```

Case study: generic arithmetic

```
(+ 2     2)

(+ 2.3 2.4)
```

Case study: generic arithmetic

```
(+ 2     2)

(+ 2.3 2.4)

(+ 2.3 2)
```

Case study: generic arithmetic

```
(+ 2   2)

(+ 2.3 2.4)

(+ 2.3 2)

(+ 2+3i 2+4i)
```

Case study: generic arithmetic

```
(+ 2   2)

(+ 2.3 2.4)

(+ 2.3 2)

(+ 2+3i 2+4i)
```

Types of arguments are not known statically.

Case study: generic arithmetic

```
#lang racket
(+ 2.3 2.4)
```

Case study: generic arithmetic

```
#lang racket
(+ 2.3 2.4)
```

```
(define (add x y)
  (cond ((and (float? x) (float? y))
         (let* ([val-x  (strip-type-tag x)]
                [val-y  (strip-type-tag y)]
                [result (add-floats val-x val-y)])
           (tag-as-float result)))

        ((and (integer? x) (integer? y))
         ...)

        ((and (complex? x) (complex? y))
         ...)

        (else (error))))
```

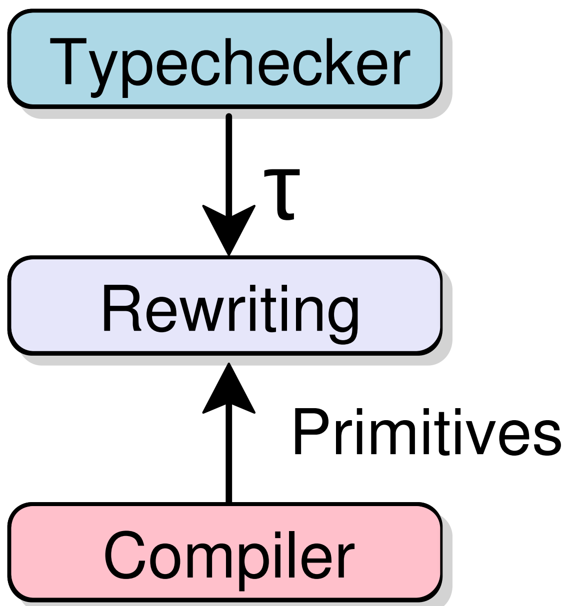
Case study: generic arithmetic

```
#lang racket
(+ 2.3 2.4)
```

```
(define (add x y)
  (cond ((and (float? x) (float? y))
         (let* ([val-x  (strip-type-tag x)]
                [val-y  (strip-type-tag y)]
                [result (add-floats val-x val-y)])
           (tag-as-float result)))

        ((and (integer? x) (integer? y))
         ...)

        ((and (complex? x) (complex? y))
         ...)

        (else (error))))
```

Our solution

- Type-specialized primitives
- Composition of:
  - Type-driven rewriting
  - Primitives drive optimization



Implementation

- Typed Racket
- Higher-order functional language
- Generic arithmetic (and complexes)

Implementation

- Typed Racket
- Higher-order functional language
- Generic arithmetic (and complexes)

Applicable to other languages

Type-specialized primitives

```
#lang racket
(fl+ x    y)
```

```
#lang racket
(fl+ 2.3 2.4)
```

```
#lang racket
(fl+ 2.3 2.4)
```

```
4.7
```

```
#lang racket
(fl+ 2   2)
```

```
#lang racket
(fl+ 2   2)
```

segmentation fault

```
#lang typed/racket
(let: ([x : Float 2.3]
       [y : Float 2.4])
  (fl+ x y))
```

```
#lang typed/racket
(let: ([x : Float 2.3]
       [y : Float 2.4])
  (fl+ x y))
```

```racket
#lang typed/racket
(let: ([x : Float 2.3]
       [y : Float 2.4])
  (fl+ x y))
```

4.7

```
#lang typed/racket
(let: ([x : Integer 2]
       [y : Integer 2])
  (fl+ x y))
```

```
#lang typed/racket
(let: ([x : Integer 2]
       [y : Integer 2])
  (fl+ x y))
```

```
#lang typed/racket
(let: ([x : Integer 2]
       [y : Integer 2])
  (fl+ x y))
```

```
Type Checker: No function domains matched in
function application:
Domains: Float Float
Arguments: Integer Integer
 in: (fl+ x y)
```

Type-driven rewriting

```
#lang typed/racket
(let: ([x : Float 2.3]
       [y : Float 2.4])
  (+   x y))
```

```
#lang typed/racket
(let: ([x : Float 2.3]
       [y : Float 2.4])
  (+  x y))
```

```
#lang typed/racket
(let: ([x : Float 2.3]
       [y : Float 2.4])
  (fl+ x y))
```

```
#lang typed/racket
(let: ([x : Float 2.3]
       [y : Float 2.4])
  (fl+ x y))
```

```
4.7
```

Primitives drive optimization

```
#lang typed/racket
(* (+ x y)
   (+ z w))
```

```
#lang typed/racket
(* (+ x y)
   (+ z w))
```

➡

```
load $x $r1
load $y $r2
...
fadd $r1 $r2 $r3
...
sto  $r3 $tmp1
load $z $r4
load $w $r5
...
fadd $r4 $r5 $r6
...
sto  $r6 $tmp2
load $tmp1 $r7
load $tmp2 $r8
...
fmul $r7 $r8 $r9
...
sto  $r9 $tmp3
```

```
#lang typed/racket
(* (+ x y)
   (+ z w))
```

➡

```
load $x $r1
load $y $r2
...
fadd $r1 $r2 $r3
...
sto  $r3 $tmp1
load $z $r4
load $w $r5
...
fadd $r4 $r5 $r6
...
sto  $r6 $tmp2
load $tmp1 $r7
load $tmp2 $r8
...
fmul $r3 $r6 $r9
...
sto  $r9 $tmp3
```

```
#lang typed/racket
(* (+ x y)
   (+ z w))
```

```
#lang typed/racket
(fl* (fl+ x y)
     (fl+ z w))
```

```
load $x $r1
load $y $r2
...
fadd $r1 $r2 $r3
...
sto  $r3 $tmp1
load $z $r4
load $w $r5
...
fadd $r4 $r5 $r6
...
sto  $r6 $tmp2
load $tmp1 $r7
load $tmp2 $r8
...
fmul $r3 $r6 $r9
...
sto  $r9 $tmp3
```

```
#lang typed/racket
(* (+ x y)
   (+ z w))
```

```
#lang typed/racket
(fl* (fl+ x y)
     (fl+ z w))
```

```
load $x $r1
load $y $r2
...
fadd $r1 $r2 $r3
...
sto  $r3 $tmp1
load $z $r4
load $w $r5
...
fadd $r4 $r5 $r6
...
sto  $r6 $tmp2
load $tmp1 $r7
load $tmp2 $r8
...
fmul $r3 $r6 $r9
...
sto  $r9 $tmp3
```

```
#lang typed/racket
(let ([a (+ x y)])
  (* a (- z a)))
```

```
#lang typed/racket
(let ([a (+ x y)])
  (* a (- z a)))
```

```
load $x $r1
load $y $r2
...
fadd $r1 $r2 $r3
...
sto  $r3 $a
load $z $r4
load $a $r5
...
fsub $r4 $r5 $r6
...
sto  $r6 $tmp1
load $a    $r7
load $tmp1 $r8
...
fmul $r7 $r8 $r9
...
sto  $r9 $tmp2
```

```
#lang typed/racket
(let ([a (+ x y)])
  (* a (- z a)))
```

```
load $x $r1
load $y $r2
...
fadd $r1 $r2 $r3
...
sto  $r3 $a
load $z $r4
load $a $r5
...
fsub $r4 $r3 $r6
...
sto  $r6 $tmp1
load $a    $r7
load $tmp1 $r8
...
fmul $r3 $r6 $r9
...
sto  $r9 $tmp2
```

```
#lang typed/racket
(let ([a (+ x y)])
  (* a (- z a)))
```

```
#lang typed/racket
(let ([a (fl+ x y)])
  (fl* a (fl- z a)))
```

```
load $x $r1
load $y $r2
...
fadd $r1 $r2 $r3
...
sto  $r3 $a
load $z $r4
load $a $r5
...
fsub $r4 $r3 $r6
...
sto  $r6 $tmp1
load $a    $r7
load $tmp1 $r8
...
fmul $r3 $r6 $r9
...
sto  $r9 $tmp2
```

```
#lang typed/racket
(let loop ([acc 0.0])
  (if (> acc x)
      acc
      (loop (+ y acc))))
```

```racket
#lang typed/racket
(let loop ([acc 0.0])
  (if (> acc x)
      acc
      (loop (+ y acc))))
```

```
    mov  0.0 $r1
    ...
    sto   $r1 $acc


loop:
    load  $acc $r2
    load  $x   $r3
    ...
    flcmp $r2 $r3
    jgt   end
    load  $y   $r4
    load  $acc $r5
    ...
    fadd $r4 $r5 $r6
    ...
    sto   $r6 $acc
    jmp   loop
end:
```

```
#lang typed/racket
(let loop ([acc 0.0])
  (if (> acc x)
      acc
      (loop (+ y acc))))
```

```
    mov   0.0 $r1
    ...
    sto   $r1 $acc


loop:
    load  $acc $r2
    load  $x   $r3
    ...
    flcmp $r2 $r3
    jgt   end
    load  $y   $r4
    load  $acc $r5
    ...
    fadd $r4 $r5 $r6
    ...
    sto   $r6 $acc
    jmp   loop
end:
```

```
#lang typed/racket
(let loop ([acc 0.0])
  (if (> acc x)
      acc
      (loop (+ y acc))))
```

```
#lang typed/racket
(let loop ([acc 0.0])
  (if (fl> acc x)
      acc
      (loop (fl+ y acc))))
```

```
    mov  0.0 $r1
    ...
    sto   $r1 $acc

loop:
    load  $acc $r2
    load  $x   $r3
    ...
    flcmp $r1 $r3
    jgt   end
    load  $y   $r4
    load  $acc $r5
    ...
    fadd $r4 $r1 $r6
    ...
    sto   $r6 $acc
    jmp   loop
end:
    sto $r1 $acc
```

```
#lang typed/racket
(let loop ([acc 0.0])
  (if (> acc x)
      acc
      (loop (+ y acc))))
```

```
#lang typed/racket
(let loop ([acc 0.0])
  (if (fl> acc x)
      acc
      (loop (fl+ y acc))))
```

```
    mov   0.0 $r1
    ...
    sto   $r1 $acc


loop:
    load  $acc $r2
    load  $x   $r3
    ...
    flcmp $r1 $r3
    jgt   end
    load  $y   $r4
    load  $acc $r5
    ...
    fadd $r4 $r1 $r6
    ...
    sto   $r6 $acc
    jmp   loop
end:
    sto $r1 $acc
```

```
#lang typed/racket
(let loop ([acc 0.0])
  (if (> acc x)
      acc
      (loop (+ y acc))))
```

```
#lang typed/racket
(let loop ([acc 0.0])
  (if (fl> acc x)
      acc
      (loop (fl+ y acc))))
```

```
    mov   0.0 $r1
    ...
    sto   $r1 $acc
    load  $x $r3
    load  $y $r4
loop:
    load  $acc $r2
    load  $x   $r3
    ...
    flcmp $r1 $r3
    jgt   end
    load  $y   $r4
    load  $acc $r5
    ...
    fadd $r4 $r1 $r6
    ...
    sto   $r6 $acc
    jmp   loop
end:
    sto $r1 $acc
```

Results

		Speedup
benchmarks	pseudoknot	**2.50**
	mandelbrot	**15.72**
	nbody	**2.16**
	takl	**1.24**
	FFT	**15.91**
data structures	banker's queue	**1.60**
	leftist heap	**1.34**
industrial application	FFT	**2.78**

Bigger is better

Average of 5 runs on x86

In-depth look: Industrial FFT

```racket
#lang racket
(let* ([x       2.3+2.4i]
       [y       2.5+2.6i]
       [z       2.7+2.8i])
  (- (+ x y) z))
```

```racket
#lang racket
(let* ([x      2.3+2.4i]
       [y      2.5+2.6i]
       [z      2.7+2.8i]
       [x-real (real-part x)]
       [x-imag (imag-part x)]
       [y-real (real-part y)]
       [y-imag (imag-part y)]
       [z-real (real-part z)]
       [z-imag (imag-part z)])
  (make-rectangular
   (-   (+   x-real y-real) z-real)
   (-   (+   x-imag y-imag) z-imag)))
```

```racket
#lang racket
(let* ([x      2.3+2.4i]
       [y      2.5+2.6i]
       [z      2.7+2.8i]
       [x-real (real-part x)]
       [x-imag (imag-part x)]
       [y-real (real-part y)]
       [y-imag (imag-part y)]
       [z-real (real-part z)]
       [z-imag (imag-part z)])
  (make-rectangular
   (fl- (fl+ x-real y-real) z-real)
   (fl- (fl+ x-imag y-imag) z-imag)))
```

```racket
#lang racket
(let* ([x      2.3+2.4i]
       [y      2.5+2.6i]
       [z      2.7+2.8i]
       [x-real (real-part x)]
       [x-imag (imag-part x)]
       [y-real (real-part y)]
       [y-imag (imag-part y)]
       [z-real (real-part z)]
       [z-imag (imag-part z)])
  (make-rectangular
   (fl- (fl+ x-real y-real) z-real)
   (fl- (fl+ x-imag y-imag) z-imag)))
```

Significant manual labor

```racket
#lang racket
(let* ([x      2.3+2.4i]
       [y      2.5+2.6i]
       [z      2.7+2.8i]
       [x-real (real-part x)]
       [x-imag (imag-part x)]
       [y-real (real-part y)]
       [y-imag (imag-part y)]
       [z-real (real-part z)]
       [z-imag (imag-part z)])
  (make-rectangular
   (fl- (fl+ x-real y-real) z-real)
   (fl- (fl+ x-imag y-imag) z-imag)))
```

Significant manual labor

Error prone

```racket
#lang typed/racket
(let* ([x 2.3+2.4i]
       [y 2.5+2.6i]
       [z 2.7+2.8i])
  (- (+ x y) z))
```

```
#lang typed/racket
(let* ([x 2.3+2.4i]
       [y 2.5+2.6i]
       [z 2.7+2.8i])
  (- (+ x y) z))
```

Unboxed intermediate results

```
#lang typed/racket
(let* ([x 2.3+2.4i]
       [y 2.5+2.6i]
       [z 2.7+2.8i])
  (- (+ x y) z))
```

Unboxed intermediate results

Unboxed let bindings

```
#lang typed/racket
(let* ([x 2.3+2.4i]
       [y 2.5+2.6i]
       [z 2.7+2.8i])
  (- (+ x y) z))
```

Unboxed intermediate results

Unboxed let bindings

Unboxed loop variables

```
#lang typed/racket
(let* ([x 2.3+2.4i]
       [y 2.5+2.6i]
       [z 2.7+2.8i])
  (- (+ x y) z))
```

Unboxed intermediate results

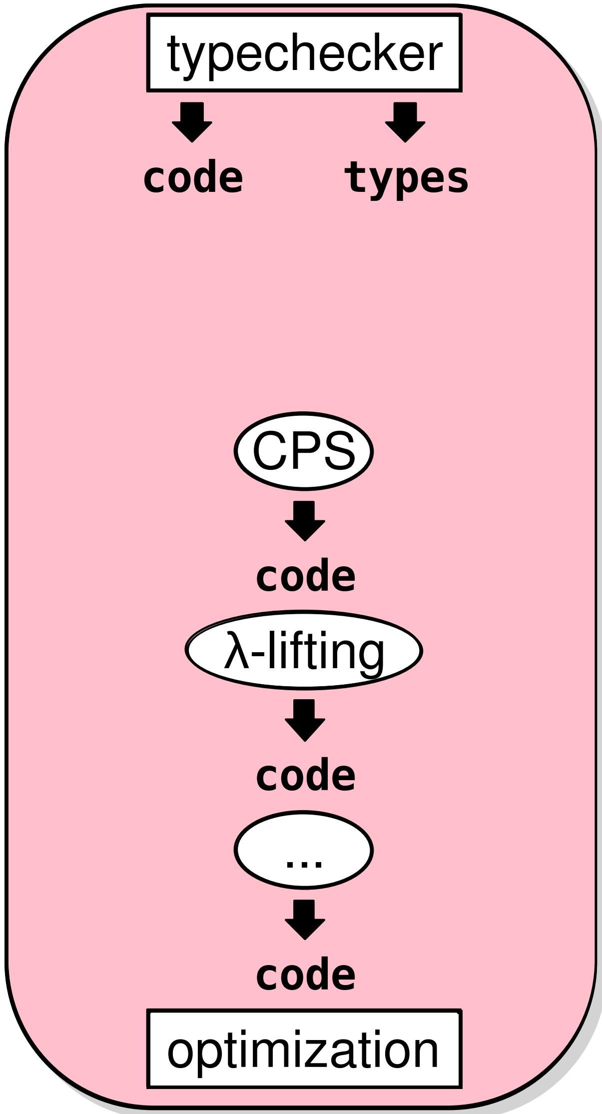
Unboxed let bindings

Unboxed loop variables

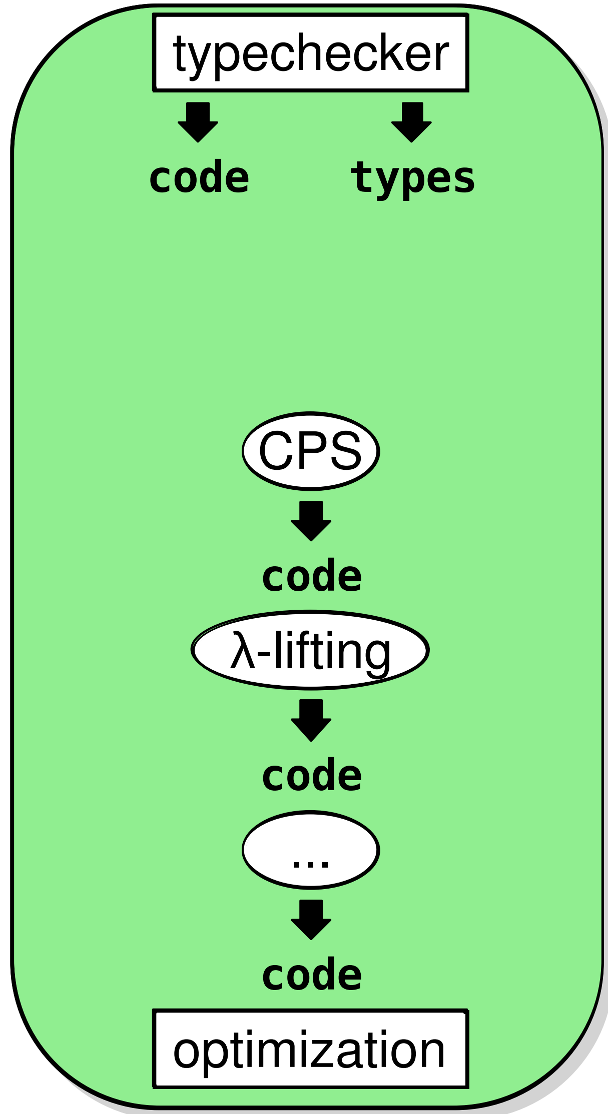
Faster than hand-optimized code

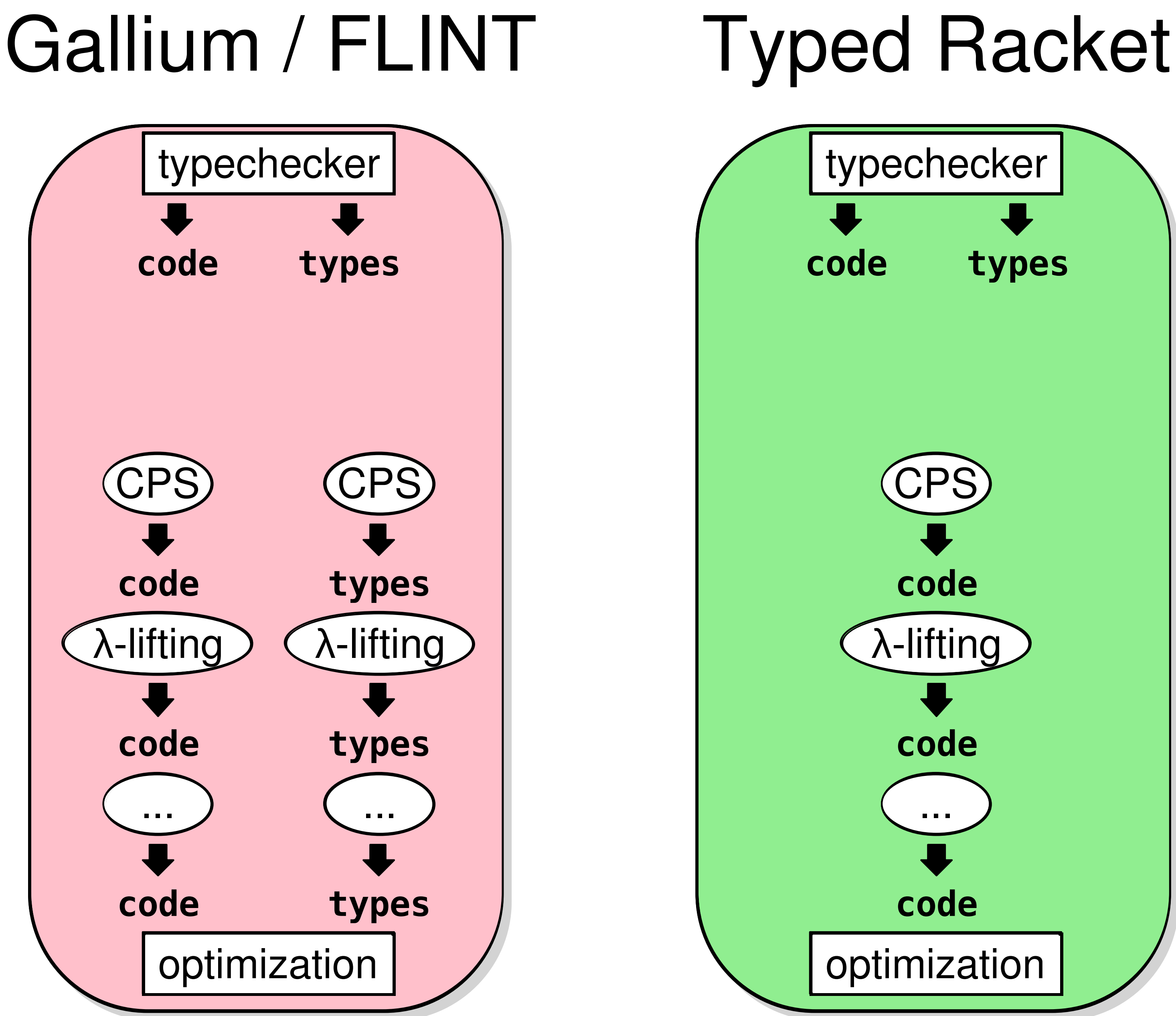
Related Work

Gallium / FLINT

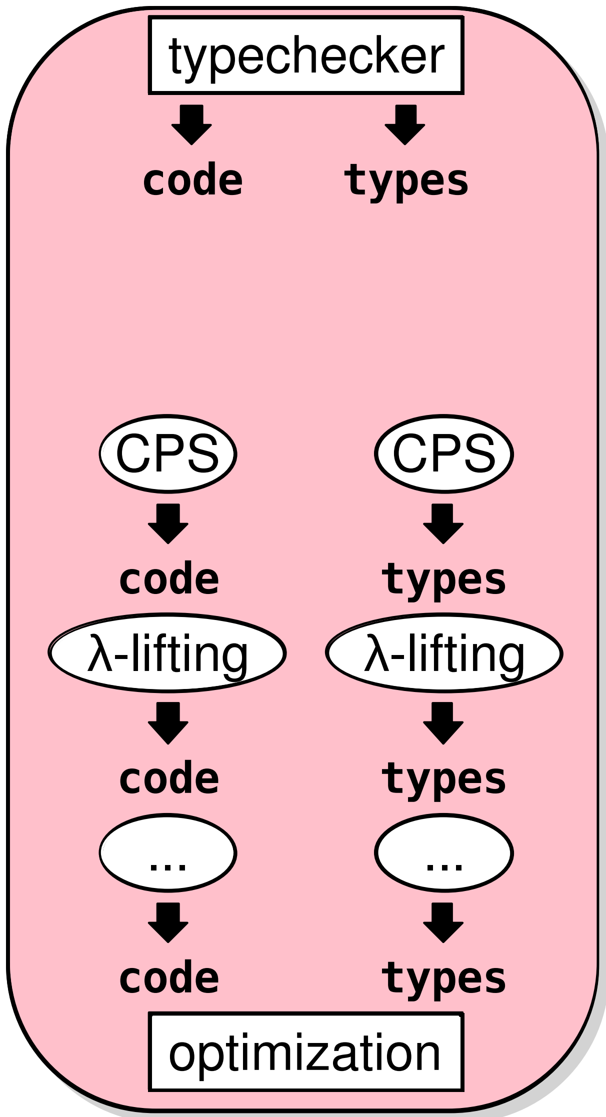


Typed Racket

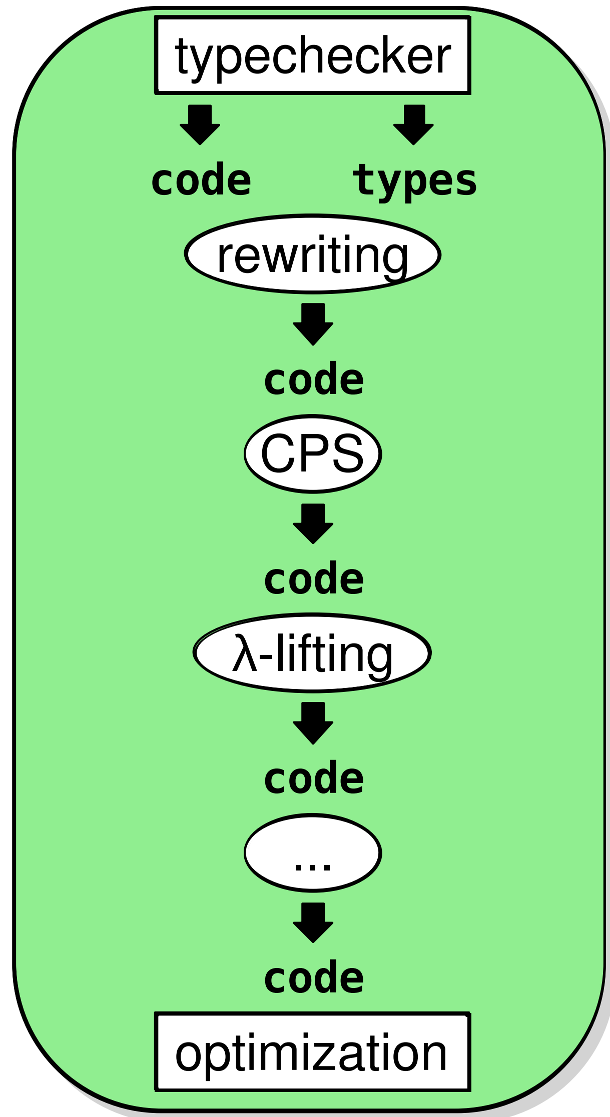



Gallium / FLINT
typechecker
code
types
CPS
CPS
code
types
λ-lifting
λ-lifting
code
types
...
...
code
types
optimization
Typed Racket
typechecker
code
types
CPS
code
λ-lifting
code
...
code
optimization

Gallium / FLINT



Typed Racket



OCaml	Typed Racket
`2   +  2`	`(+ 2   2)`
`2.3 +. 2.4`	`(+ 2.3 2.4)`

OCaml	Typed Racket
`2   +  2`	`(+ 2   2)`
`2.3 +. 2.4`	`(+ 2.3 2.4)`

OCaml limits specialization to integer vs float.

Common LISP

```lisp
(defun
  f (x)
  (declare (type function x)
           (optimize (speed 3)
                     (safety 0)))
  (funcall x 3))

(defun
  g (fun)
  (declare (type function fun))
  (funcall fun "a"))

(g #'f)
```

Typed Racket

```racket
#lang typed/racket

(define:
  (f (x : (Integer -> Integer)))
     : Integer


  (x 3))

(define:
  (g (fun : (String -> String)))
     : String
  (fun "a"))

(g f)
```

Common LISP

```
(defun
  f (x)
  (declare (type function x)
           (optimize (speed 3)
                     (safety 0)))
  (funcall x 3))

(defun
  g (fun)
  (declare (type function fun))
  (funcall fun "a"))

(g #'f)
```

Typed Racket

```
#lang typed/racket

(define:
  (f (x : (Integer -> Integer)))
     : Integer


  (x 3))

(define:
  (g (fun : (String -> String)))
     : String
  (fun "a"))

(g f)
```

Common LISP

```
(defun
  f (x)
  (declare (type function x)
           (optimize (speed 3)
                     (safety 0)))
  (funcall x 3))

(defun
  g (fun)
  (declare (type function fun))
  (funcall fun "a"))

(g #'f)
```

```
Unhandled memory fault at #x610000.
```

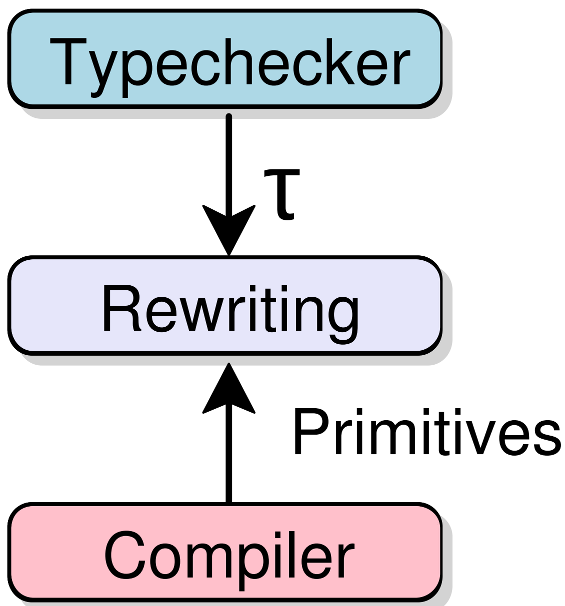
Typed Racket

```
#lang typed/racket

(define:
  (f (x : (Integer -> Integer)))
     : Integer


  (x 3))

(define:
  (g (fun : (String -> String)))
     : String
  (fun "a"))

(g f)
```

```
Type Checker:
Expected (String -> String),
but got ((Integer -> Integer)
          -> Integer)
in: f
```

Our solution

- Type-specialized primitives
- Composition of:
  - Type-driven rewriting
  - Primitives drive optimization



racket-lang.org