

Toward Efficient Sound Gradual Typing

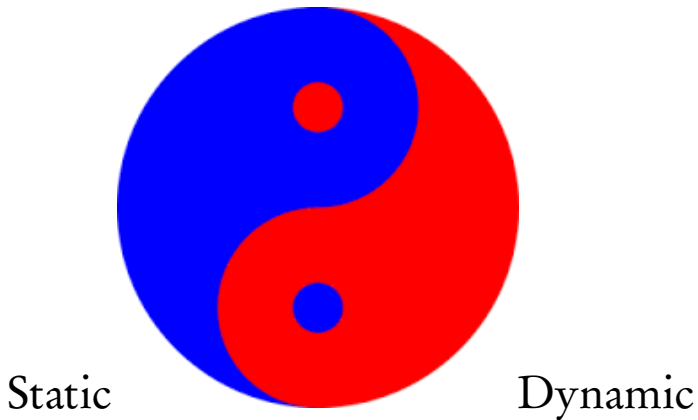
Deyaaeldeen Almahallawi

Indiana University Bloomington

April 23, 2018

Joint work with Andre Kuhlenschmidt and Jeremy Siek

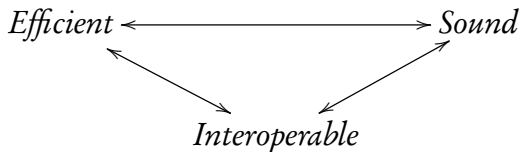
Gradual Typing



Efficient gradual typing

- ▶ Challenges to Efficiency
- ▶ Challenges to Evaluation
- ▶ Space-efficient Coercions
- ▶ Monotonic Coercions
- ▶ Performance Comparison

There is tension in the design space



Approach	Sound	Efficient	Interoperable
erase types	◐	◐	●
insert casts	●	◐	●
limit interop.	●	●	◐

Approach: insert casts

Compile the GTLC to STLC + casts.

A cast has the form

$$e' : T_1 \Rightarrow T_2$$

$$\boxed{\Gamma \vdash e \rightsquigarrow e' : T}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : T \rightarrow T' \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : T_2 \quad T_2 \sim T}{\Gamma \vdash e_1 \ e_2 \rightsquigarrow e'_1 \ (e'_2 : T_2 \Rightarrow T) : T'}$$
$$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : \star \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : T_2}{\Gamma \vdash e_1 \ e_2 \rightsquigarrow (e'_1 : \star \Rightarrow \star \rightarrow \star) \ (e'_2 : T_2 \Rightarrow \star) : \star}$$

Operational semantics of casts

Base types	$B ::= \text{Int} \mid \text{Bool}$
Injection types	$I ::= B \mid \star \rightarrow \star$
Values	$v ::= n \mid b \mid \lambda x:T. e \mid$ $v : I \Rightarrow \star \mid v : T \rightarrow T \Rightarrow T \rightarrow T$

$$(v : I \Rightarrow \star) : \star \Rightarrow I \longrightarrow v$$

$$(v : I \Rightarrow \star) : \star \Rightarrow I' \longrightarrow \text{blame} \quad \text{if } I \neq I'$$

$$v : B \Rightarrow B \longrightarrow v$$

$$v : \star \Rightarrow \star \longrightarrow v$$

$$(v : T_1 \rightarrow T_2 \Rightarrow T'_1 \rightarrow T'_2) \ v' \longrightarrow v \ (v' : T'_1 \Rightarrow T_1) : T_2 \Rightarrow T'_2$$

$$v : T \Rightarrow \star \longrightarrow (v : T \Rightarrow I) : I \Rightarrow \star^\dagger$$

$$v : \star \Rightarrow T \longrightarrow (v : \star \Rightarrow I) : I \Rightarrow T^\dagger$$

† if $T \sim I$, $T \neq I$, and $T \neq \star$

Example

```
(let ((f (λ(y : Int). (+1 y)))  
      (g (λ(g : ★). (g : ★ ⇒ ★ → ★ (true : Bool ⇒ ★))))  
      (h (f : Int → Int ⇒ ★)))  
→*  
(f : Int → Int ⇒ ★ → ★) (true : Bool ⇒ ★)  
→  
(f (true : Bool ⇒ ★ ⇒ Int) : Int ⇒ ★  
→  
(f blame) : Int ⇒ ★  
→  
blame
```

Casts can consume unbounded space

```
(letrec ((even λ(n : Int) : ★. (if (= n 0)
                                     #t
                                     (odd (- n 1))))
         (odd λ(n : Int) : Bool. (if (= n 0)
                                     #f
                                     (even (- n 1)))))
  (even ...))
```


Casts can consume unbounded space

```
(letrec ((even  $\lambda(n : \text{Int}) : \star$ . (if (= n 0)
                                     (#t : Bool  $\Rightarrow \star$ )
                                     (odd (- n 1) : Bool  $\Rightarrow \star$ ))))
  (odd  $\lambda(n : \text{Int}) : \text{Bool}$ . (if (= n 0)
                                   #f
                                   (even (- n 1) :  $\star \Rightarrow \text{Bool}$ ))))
(even ...))
```

Casts can consume unbounded space

$even(5)$

$\longrightarrow odd(4) : Bool \Rightarrow \star$

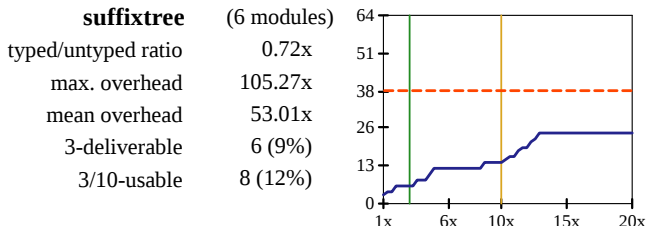
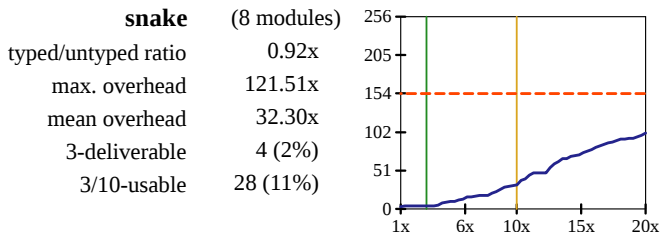
$\longrightarrow even(3) : \star \Rightarrow Bool \Rightarrow \star$

$\longrightarrow odd(2) : Bool \Rightarrow \star \Rightarrow Bool \Rightarrow \star$

$\longrightarrow even(1) : \star \Rightarrow Bool \Rightarrow \star \Rightarrow Bool \Rightarrow \star$

$\longrightarrow odd(0) : Bool \Rightarrow \star \Rightarrow Bool \Rightarrow \star \Rightarrow Bool \Rightarrow \star$

Casts can cause catastrophic slowdowns

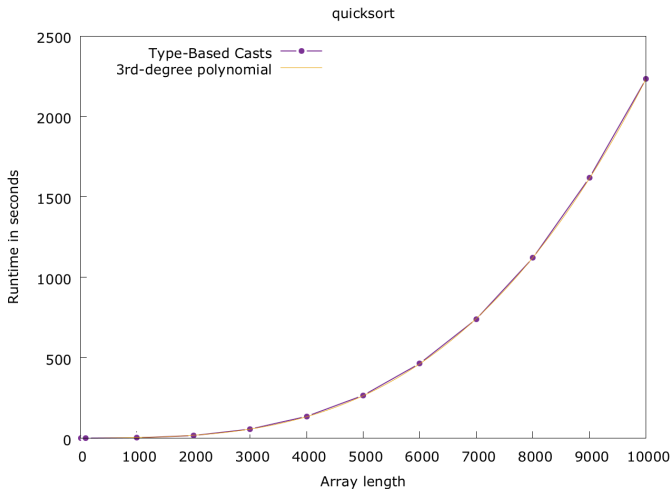


Is sound gradual typing dead? Takikawa et al. POPL 2016

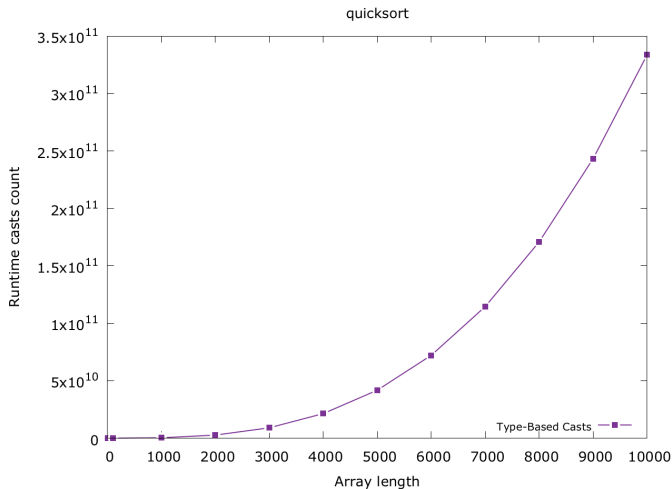
Casts can cause catastrophic slowdowns

```
(define sort!  
  : ((Vectorof Dyn) Int Int -> ())  
  (lambda ([v : (Vectorof Int)]  
    [lo : Int][hi : Int])  
    (when (< lo hi)  
      (let ([pivot : Int (partition! v lo hi)])  
        (sort! v lo (- pivot 1))  
        (sort! v (+ pivot 1) hi))))))
```

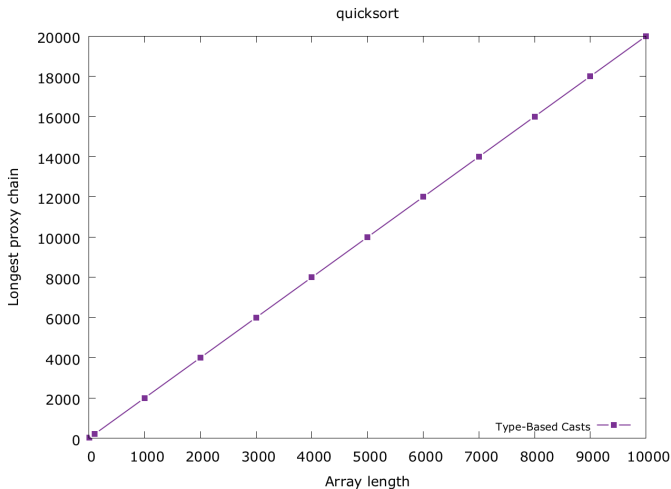
Casts can cause catastrophic slowdowns



Casts can cause catastrophic slowdowns



Casts can cause long proxy chains



Efficient gradual typing

- ▶ Challenges to Efficiency
- ▶ **Challenges to Evaluation**
- ▶ Space-efficient Coercions
- ▶ Monotonic Coercions
- ▶ Performance Comparison

Challenges to Performance Evaluation

- ▶ Takikawa et al. POPL 2016 proposes to run all the combinations of making each module typed or untyped. There are 2^n configurations for a program that consists of n modules.
- ▶ This should work for most benchmarks where n is relatively small.

Challenges to Performance Evaluation

- ▶ Takikawa et al. POPL 2016 proposes to run all the combinations of making each module typed or untyped. There are 2^n configurations for a program that consists of n modules.
- ▶ This should work for most benchmarks where n is relatively small.
- ▶ But what if the language supports fine-grained gradual typing, where the programmer may opt to not write some type annotations or put Dyn inside some?

Challenges to Performance Evaluation

- ▶ Takikawa et al. POPL 2016 proposes to run all the combinations of making each module typed or untyped. There are 2^n configurations for a program that consists of n modules.
- ▶ This should work for most benchmarks where n is relatively small.
- ▶ But what if the language supports fine-grained gradual typing, where the programmer may opt to not write some type annotations or put Dyn inside some?
- ▶ The size of the configuration space for a textbook quicksort is 248832000000. For n-body, it is 6914086267191872901144038355222134784.

Grift: an experimental compiler

- ▶ An ahead-of-time optimizing compiler for gradual typing.
- ▶ The source is a functional language with tuples and mutable vectors and references, and the target is C.
- ▶ The runtime system implements **coercions** and **monotonic references**.
- ▶ The compiler specializes casts when their source and/or target type is known at compile time.
- ▶ The compiler defers coercion creation until it is actually needed.

Grift Performance Evaluation

- ▶ a number of configurations are sampled from across the spectrum of type precision.
- ▶ Grift is compared on partially typed code to a variant of Grift where it is statically typed and does not support gradual typing (Static Grift) and to Grift on fully untyped code (Dynamic Grift)
- ▶ Grift is compared on fully typed benchmarks to fully typed languages.
- ▶ Grift is compared on fully untyped benchmarks to dynamically typed languages.

Efficient gradual typing

- ▶ Challenges to Efficiency
- ▶ Challenges to Evaluation
- ▶ **Space-efficient Coercions**
- ▶ Monotonic Coercions
- ▶ Performance Comparison

Compress casts via coercion reduction

$$\text{odd}(\circ) : \text{Bool} \Rightarrow \star \xRightarrow{\ell} \text{Bool} \Rightarrow \star \xRightarrow{m} \text{Bool} \Rightarrow \star$$

$$\text{odd}(\circ) \langle \text{Bool}! \rangle \langle \text{Bool}?\ell \rangle \langle \text{Bool}! \rangle \langle \text{Bool}?^m \rangle \langle \text{Bool}! \rangle$$

\longrightarrow

$$\text{odd}(\circ) \langle \text{id}_{\text{Bool}} \rangle \langle \text{Bool}! \rangle \langle \text{Bool}?^m \rangle \langle \text{Bool}! \rangle$$

\longrightarrow

$$\text{odd}(\circ) \langle \text{Bool}! \rangle \langle \text{Bool}?^m \rangle \langle \text{Bool}! \rangle$$

\longrightarrow

$$\text{odd}(\circ) \langle \text{id}_{\text{Bool}} \rangle \langle \text{Bool}! \rangle$$

\longrightarrow

$$\text{odd}(\circ) \langle \text{Bool}! \rangle$$

Coercion Calculus

Syntax

$$c, d ::= \text{id}_T \mid I! \mid I?^\ell \mid c \rightarrow d \mid c ; d \mid \perp^\ell$$

Reduction

$$c ; \text{id}_T \longrightarrow c$$

$$\text{id}_T ; c \longrightarrow c$$

$$I! ; I?^\ell \longrightarrow \text{id}_I$$

$$I! ; I'?^\ell \longrightarrow \perp^\ell \qquad I \neq I'$$

$$(c \rightarrow d) ; (c' \rightarrow d') \longrightarrow (c' ; c) \rightarrow (d ; d')$$

$$(\text{id}_T \rightarrow \text{id}_{T'}) \longrightarrow \text{id}_{T \rightarrow T'}$$

$$\perp^\ell ; c \longrightarrow \perp^\ell$$

$$c ; \perp^\ell \longrightarrow \perp^\ell \qquad \text{if } c \neq I?^{\ell'}$$

Normalize adjacent coercions

$e ::= \dots \mid e\langle c \rangle$	Terms
$u ::= n \mid \lambda x:T. e$	Uncoerced Values
$v ::= u \mid u\langle c \rightarrow d \rangle \mid u\langle I! \rangle$	Values

$$(u\langle c \rightarrow d \rangle) \ v \longrightarrow (u \ v\langle c \rangle)\langle d \rangle$$

$$u\langle \text{id}_T \rangle \longrightarrow u$$

$$e\langle c \rangle\langle d \rangle \longrightarrow e\langle c' \rangle \quad \text{if } (c; d) \longrightarrow^* c'$$

$$u\langle \perp^\ell \rangle \longrightarrow \text{blame } \ell$$

Coercions in normal form & composition

$$s, t ::= \text{id}_\star \mid (I^{?\ell} ; i) \mid i$$

$$i ::= (g ; I!) \mid g \mid \perp^\ell$$

$$g, h ::= \text{id}_{\text{Int}} \mid (s \rightarrow t)$$

$$\boxed{s \circ t = s}$$

$$\text{id}_{\text{Int}} \circ t = \text{id}_\star \circ t = t$$

$$(s \rightarrow t) \circ (s' \rightarrow t') = (s' \circ s) \rightarrow (t \circ t')$$

$$(g ; I!) \circ \text{id}_\star = g ; I!$$

$$(I^{?\ell} ; i) \circ t = I^{?\ell} ; (i \circ t)$$

$$g \circ (h ; I!) = (g \circ h) ; I!$$

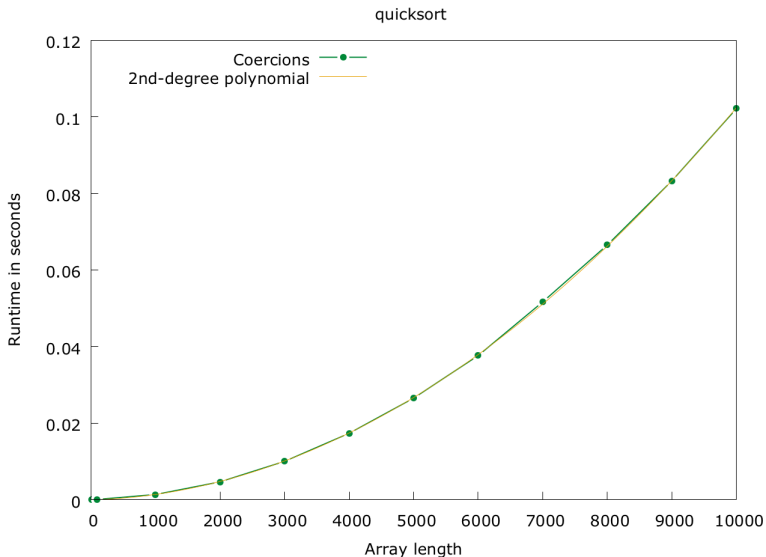
$$(g ; I!) \circ (I^{?\ell} ; i) = g \circ i$$

$$(g ; I!) \circ (I'^{?\ell} ; i) = \perp^\ell \quad \text{if } I \neq I'$$

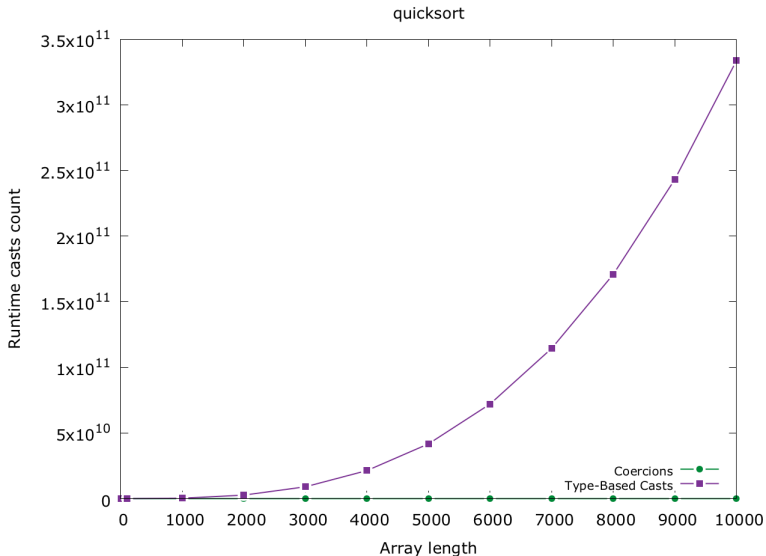
$$\perp^\ell \circ s = g \circ \perp^\ell = \perp^\ell$$

$$\mathcal{F}[e\langle s \rangle \langle t \rangle] \longrightarrow \mathcal{F}[e\langle s \circ t \rangle]$$

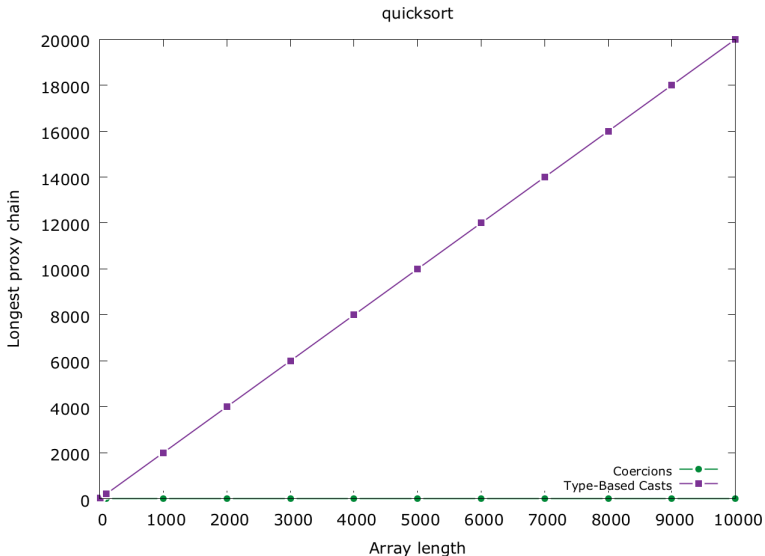
Coercions eliminate catastrophic slowdowns



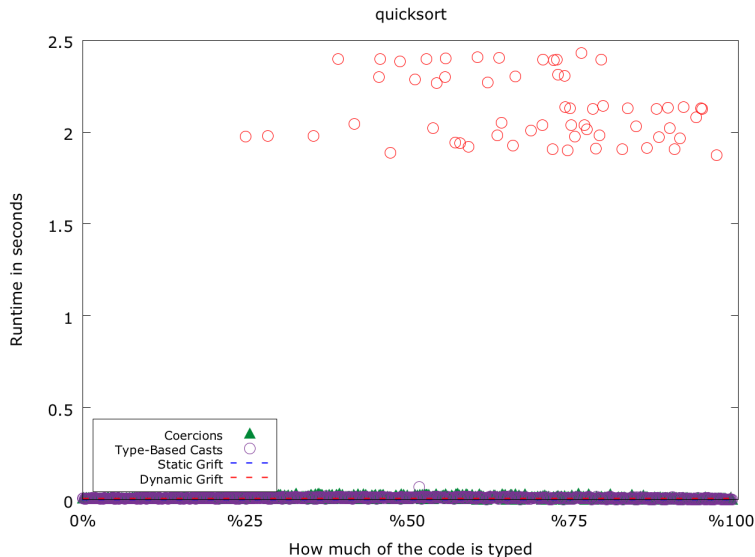
Coercions eliminate catastrophic slowdowns



Coercions eliminate catastrophic slowdowns

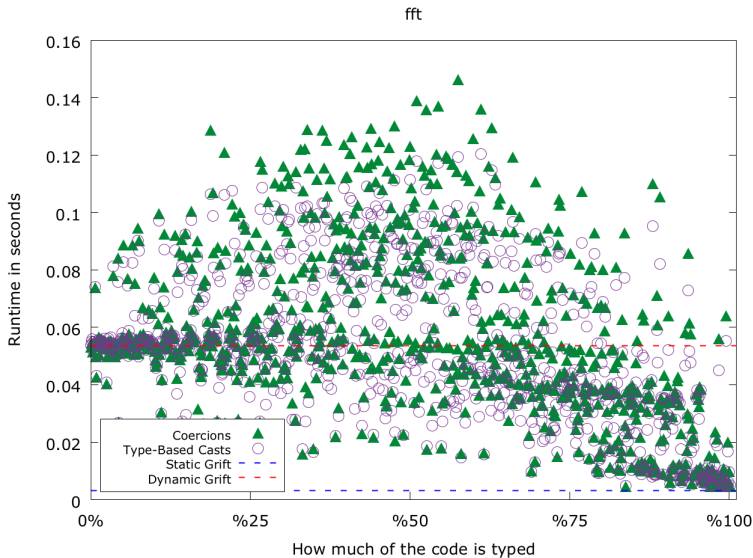


Coercions eliminate catastrophic slowdowns



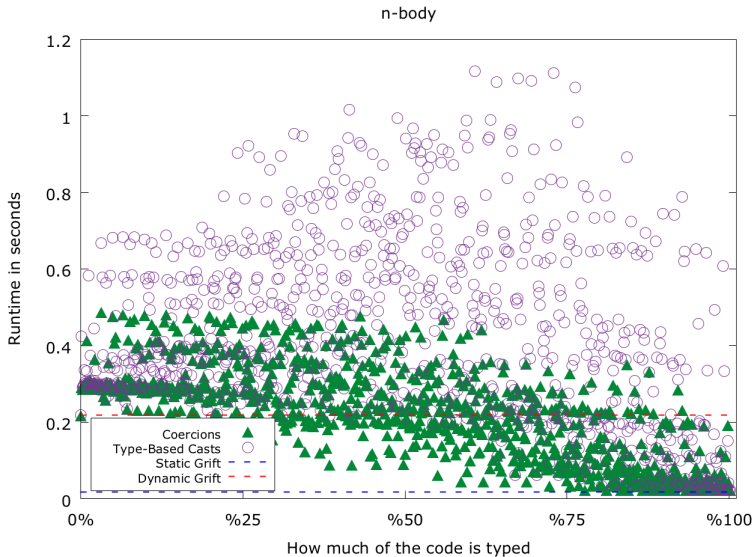
- mean speedup of 13 and max speedup of 1756

Coercions sometimes incur overhead



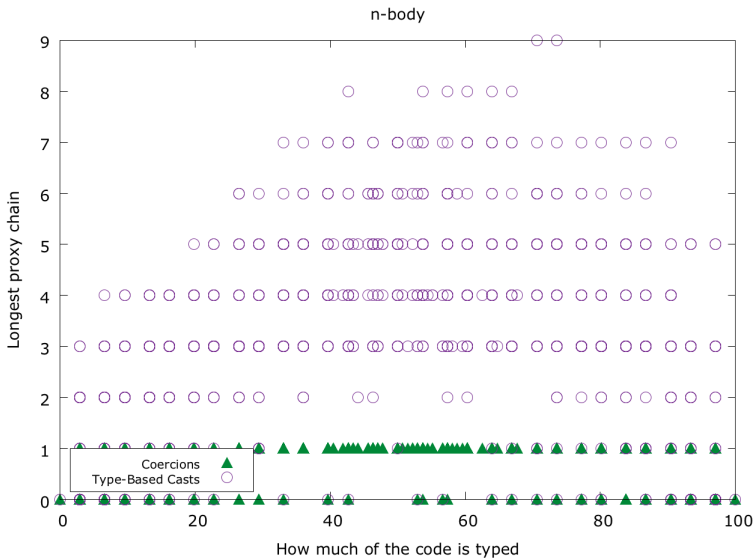
- mean slowdown of 1.1 and max slowdown of 1.6

Coercions sometimes pay for themselves



- mean speedup of 1.9 and max speedup of 28

Coercions sometimes pay for themselves



Efficient gradual typing

- ▶ Challenges to Efficiency
- ▶ Challenges to Evaluation
- ▶ Space-efficient Coercions
- ▶ **Monotonic Coercions**
- ▶ Performance Comparison

Gradual typing with mutable references

$$T ::= \dots \mid \text{Ref } T$$

$$e ::= \dots \mid \text{ref } e \mid !^\ell e \mid e :=^\ell e$$

$$v ::= \dots \mid a \mid a \langle \text{Ref } (c, d) \rangle$$

Consistency

$$\boxed{T \sim T}$$

$$\dots \quad \frac{T_1 \sim T_2}{\text{Ref } T_1 \sim \text{Ref } T_2}$$

Coercions

$$c ::= \dots \mid \text{Ref } (c_1, c_2)$$

Compile Casts to Coercions

$$\langle\langle \text{Ref } T_1 \Rightarrow^\ell \text{Ref } T_2 \rangle\rangle = \text{Ref } (\langle\langle T_1 \Rightarrow^\ell T_2 \rangle\rangle, \langle\langle T_2 \Rightarrow^\ell T_1 \rangle\rangle)$$

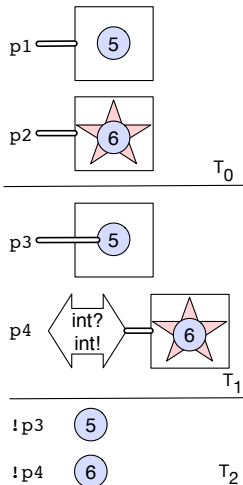
Example of overhead in reference access

```
fun f(p3:Ref Int, p4:Ref Int)=  
    !p3 + !p4;
```

```
val p1 = ref 5;  
val p2 = ref (6<Int!>);
```

```
f(p1, p2<Ref(Int?,Int!)>);
```

```
ref(Int?,Int!)  
  : Ref *  $\Rightarrow$  Ref Int
```



Problem: generated code for `!p3` and `!p4` must branch at run-time for the two kinds of references.

Root of the problem

Theorem (Canonical Forms)

Suppose $\emptyset \vdash v : T$. If $T = \text{Ref } T$, then either

- ▶ *$v = a$ for some address a , or*
- ▶ *$v = a \langle \text{Ref } (c_1, c_2) \rangle$.*

Two rules for dereference

$$\begin{aligned} !a, \mu &\longrightarrow \mu(a), \mu \\ !(a \langle \text{Ref } (c_1, c_2) \rangle), \mu &\longrightarrow (!a) \langle c_1 \rangle, \mu \end{aligned}$$

Two rules for update

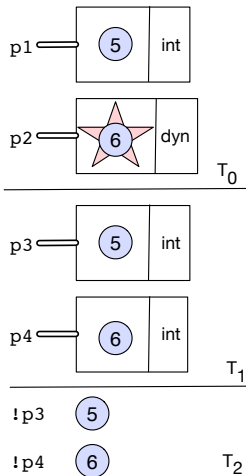
$$\begin{aligned} a := v, \mu &\longrightarrow a, \mu(a \mapsto v) \\ a \langle \text{Ref } (c_1, c_2) \rangle := v, \mu &\longrightarrow a := v \langle c_2 \rangle, \mu \end{aligned}$$

Monotonic References

```
fun f(p3:Ref Int, p4:Ref Int)=  
    !p3 + !p4;
```

```
val p1 = ref 5;  
val p2 = ref (6<Int!>);
```

```
f(p1, p2<Ref(Int)>);
```



Update the reference cell to the meet of the current RTTI and the target of the cast.

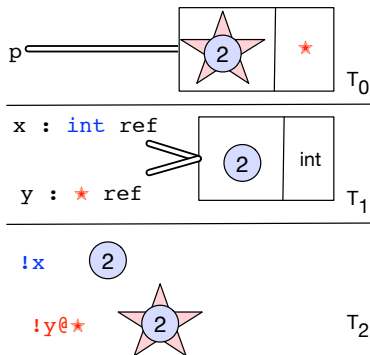
Aliasing and Static vs. Dynamic Dereference

```
fun f(x:Ref Int, y:Ref ★)=  
    !x + !y@★;
```

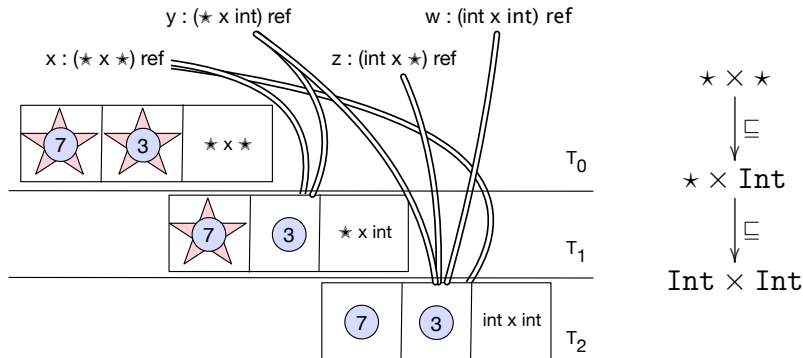
```
p = ref (2<Int!>);  
f(p, p);
```

Compile-time choice:

- Fast static deref.
- Slow dynamic dereference



The Monotonic Invariant

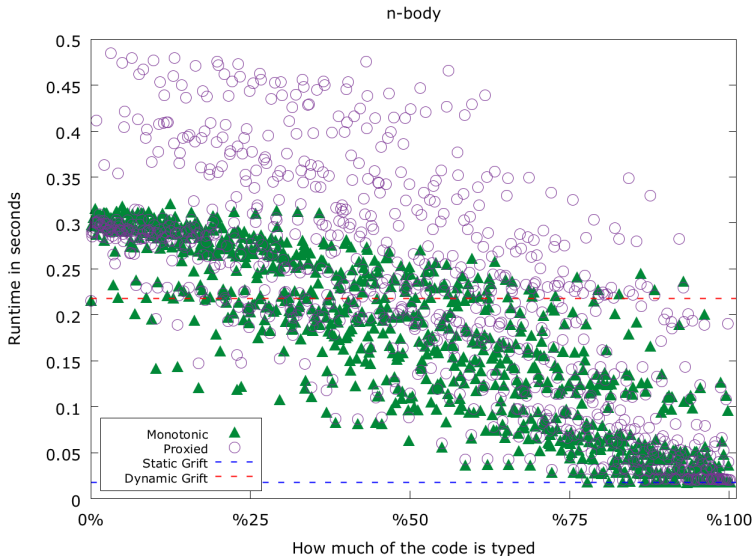


- ▶ The RTTI of a cell may become more precise.
- ▶ Every reference is less or equally precise as the RTTI.
- ▶ If a reference is fully static (e.g. w), then so is the cell.

Monotonic implementation

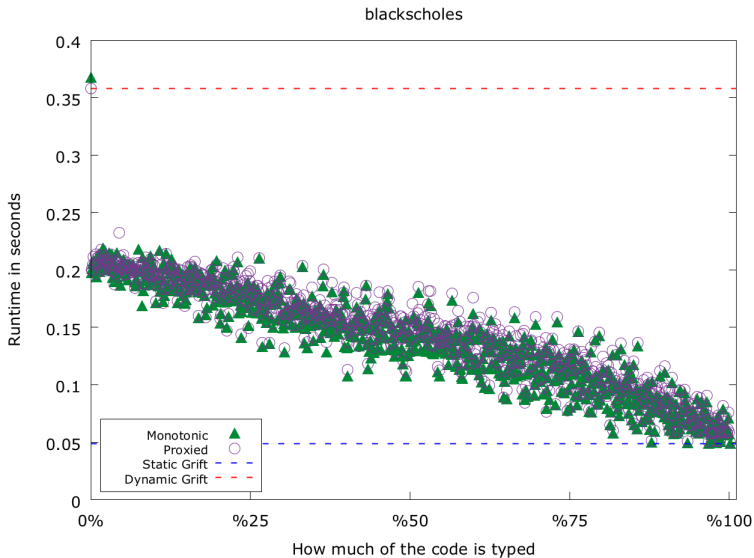
- ▶ hashconsing types at runtime to speedup the meet operation
- ▶ casted tuple values are just copies of the original tuples written to the heap and updated in place while the cast is in progress.

Monotonic eliminates overhead in static code



- mean speedup of 1.26 and max speedup of 3.24

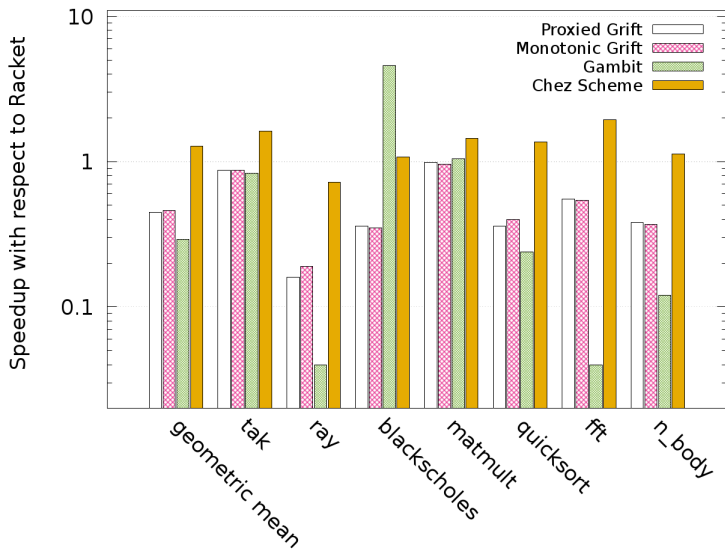
Monotonic doesn't matter for some programs



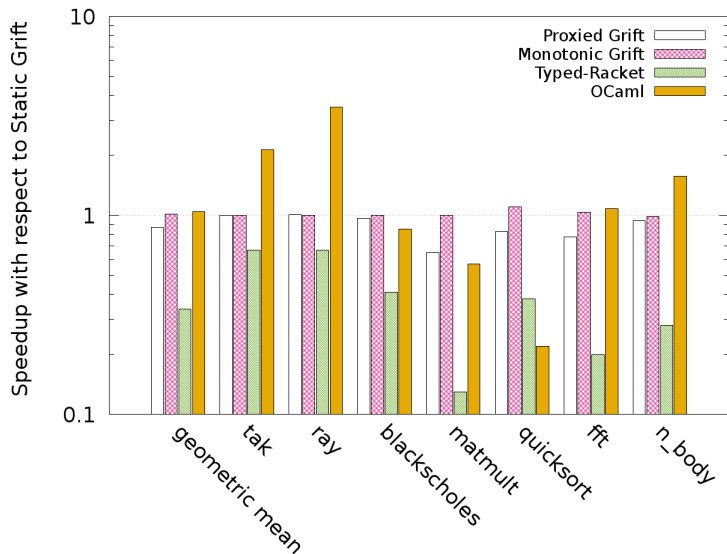
Efficient gradual typing

- ▶ Challenges to Efficiency
- ▶ Challenges to Evaluation
- ▶ Space-efficient Coercions
- ▶ Monotonic Coercions
- ▶ **Performance Comparison**

Comparison on dynamically typed programs



Comparison on statically typed programs



Conclusion

- ▶ Gradual typing can be efficient!
- ▶ On dynamically typed code: competitive with Gambit.
- ▶ On statically typed code: competitive with OCaml.
- ▶ On partially typed code, performance varies roughly between %50 of untyped impl. to %100 of typed impl.
- ▶ Use **coercions** to eliminate catastrophic slowdowns in partially typed code.
- ▶ Use **monotonic references** to reduce overhead in statically typed code.

Grift is open source: github.com/Gradual-Typing/Grift

Thank you!