Toward Efficient Sound Gradual Typing

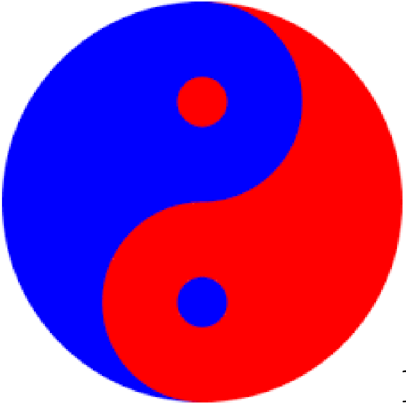
Deyaaeldeen Almahallawi

Indiana University Bloomington

April 24, 2018

Joint work with Andre Kuhlenschmidt and Jeremy Siek

Gradual Typing



Static

Dynamic

Efficient gradual typing

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

There is tension in the design space

$$\begin{tikzcd} \textit{Efficient} \arrow[rr, leftrightarrow] \arrow[dr, leftrightarrow] & & \textit{Sound} \arrow[dl, leftrightarrow] \\ & \textit{Interoperable} & \end{tikzcd}$$

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

Approach: insert casts

Compile the GTLC to STLC + casts.

A cast has the form

$$e' : T_1 \Rightarrow T_2$$

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

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

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : \star \qquad \Gamma \vdash e_2 \rightsquigarrow e'_2 : T_2}{\Gamma \vdash e_1\ e_2 \rightsquigarrow (e_1 : \star \Rightarrow \star \to \star)\ (e_2 : T_2 \Rightarrow \star) : \star}$$

Operational semantics of casts

$$
\begin{array}{llcl}
\text{Base types} & B & ::= & \texttt{Int} \mid \texttt{Bool} \\
\text{Injection types} & I & ::= & B \mid \star \to \star \\
\text{Values} & v & ::= & n \mid b \mid \lambda x{:}T.\, e \mid \\
 & & & v : I \Rightarrow \star \mid v : T \to T \Rightarrow T \to T
\end{array}
$$

$$
\begin{aligned}
(v : I \Rightarrow \star) : \star \Rightarrow I &\longrightarrow v \\
(v : I \Rightarrow \star) : \star \Rightarrow I' &\longrightarrow \texttt{blame} \quad \text{if } I \neq I' \\
v : B \Rightarrow B &\longrightarrow v \\
v : \star \Rightarrow \star &\longrightarrow v \\
(v : T_1 \to T_2 \Rightarrow T_1' \to T_2')\ v' &\longrightarrow v\ (v' : T_1' \Rightarrow T_1) : T_2 \Rightarrow T_2' \\
v : T \Rightarrow \star &\longrightarrow (v : T \Rightarrow I) : I \Rightarrow \star\ ^{\dagger} \\
v : \star \Rightarrow T &\longrightarrow (v : \star \Rightarrow I) : I \Rightarrow T\ ^{\dagger}
\end{aligned}
$$

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

Example

$$
\begin{array}{l}
(\texttt{let}\ ((f\ (\lambda(y:\texttt{Int}).\,(+1\ y)))\\
\qquad\quad (g\ (\lambda(g:\star).\,(g:\star\Rightarrow\star\to\star\ \ (\texttt{true}:\texttt{Bool}\Rightarrow\star)))))\\
\ \ (h\ (f:\texttt{Int}\to\texttt{Int}\Rightarrow\star)))\\
\longrightarrow^{*}\\
(f:\texttt{Int}\to\texttt{Int}\Rightarrow\star\to\star\ )\ \ (\texttt{true}:\texttt{Bool}\Rightarrow\star)\\
\longrightarrow\\
(f\ \ (\texttt{true}:\texttt{Bool}\Rightarrow\star\Rightarrow\texttt{Int}):\texttt{Int}\Rightarrow\star\\
\longrightarrow\\
(f\ \ \texttt{blame}\ ):\texttt{Int}\Rightarrow\star\\
\longrightarrow\\
\texttt{blame}
\end{array}
$$

Casts can consume unbounded space

$$
\begin{array}{l}
(\text{letrec } ((\mathit{even}\ \lambda(n : \texttt{Int}) : \star.\ (\text{if } (=\ n\ \texttt{0}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{\#t} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{odd}\ (-\ n\ \texttt{1})))) \\
\qquad\quad (\mathit{odd}\ \lambda(n : \texttt{Int}) : \texttt{Bool}.\ (\text{if } (=\ n\ \texttt{0}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{\#f} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{even}\ (-\ n\ \texttt{1}))))) \\
\quad (\mathit{even}\ \ldots))
\end{array}
$$

Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

Casts can consume unbounded space

$$
\begin{array}{l}
(\text{letrec}\ ((\mathit{even}\ \lambda(n:\texttt{Int}):\star.\ (\text{if}\ (=\ n\ 0)\\
\qquad\qquad\qquad\qquad\qquad\qquad (\#\text{t} : \texttt{Bool} \Rightarrow \star)\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{odd}\ (-\ n\ 1) : \texttt{Bool} \Rightarrow \star)))\\
\qquad\quad (\mathit{odd}\ \lambda(n:\texttt{Int}):\texttt{Bool}.\ (\text{if}\ (=\ n\ 0)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \#\text{f}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{even}\ (-\ n\ 1) : \star \Rightarrow \texttt{Bool}))))\\
\ (\mathit{even}\ \ldots))
\end{array}
$$

Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

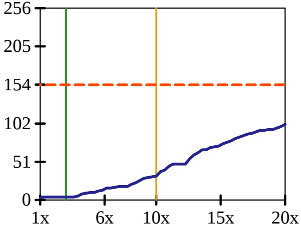
Casts can consume unbounded space

$$
\begin{aligned}
& \mathit{even}(5) \\
\longrightarrow\ & \mathit{odd}(4) : \texttt{Bool} \Rightarrow \star \\
\longrightarrow\ & \mathit{even}(3) : \star \Rightarrow \texttt{Bool} \Rightarrow \star \\
\longrightarrow\ & \mathit{odd}(2) : \texttt{Bool} \Rightarrow \star \Rightarrow \texttt{Bool} \Rightarrow \star \\
\longrightarrow\ & \mathit{even}(1) : \star \Rightarrow \texttt{Bool} \Rightarrow \star \Rightarrow \texttt{Bool} \Rightarrow \star \\
\longrightarrow\ & \mathit{odd}(0) : \texttt{Bool} \Rightarrow \star \Rightarrow \texttt{Bool} \Rightarrow \star \Rightarrow \texttt{Bool} \Rightarrow \star
\end{aligned}
$$

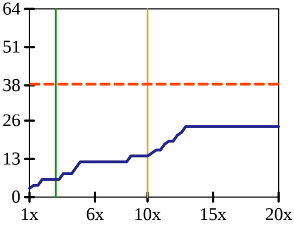
Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

Casts can cause catastrophic slowdowns

snake	(8 modules)
typed/untyped ratio	0.92x
max. overhead	121.51x
mean overhead	32.30x
3-deliverable	4 (2%)
3/10-usable	28 (11%)



suffixtree	(6 modules)
typed/untyped ratio	0.72x
max. overhead	105.27x
mean overhead	53.01x
3-deliverable	6 (9%)
3/10-usable	8 (12%)

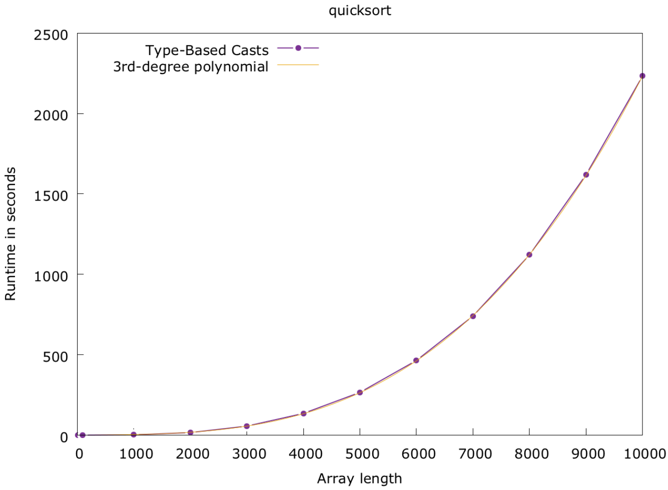


Is sound gradual typing dead? Takikawa et al. POPL 2016

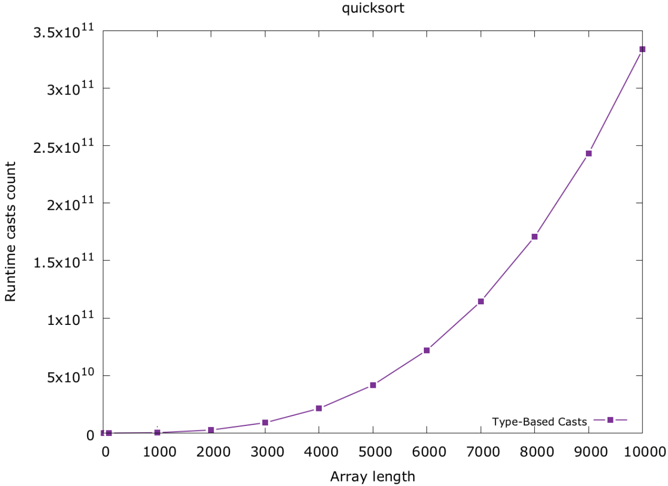
Casts can cause catastrophic slowdowns

```
(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)))))
```

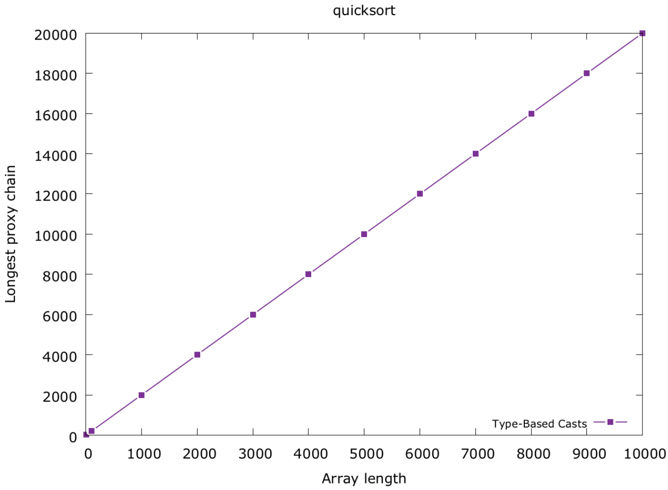
Casts can cause catastrophic slowdowns



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

$$\mathit{odd}(\mathrm{o}) : \texttt{Bool} \Rightarrow \star \stackrel{\ell}{\Rightarrow} \texttt{Bool} \Rightarrow \star \stackrel{m}{\Rightarrow} \texttt{Bool} \Rightarrow \star$$

$$\mathit{odd}(\mathrm{o})\langle\texttt{Bool!}\rangle\langle\texttt{Bool?}^{\ell}\rangle\langle\texttt{Bool!}\rangle\langle\texttt{Bool?}^{m}\rangle\langle\texttt{Bool!}\rangle$$
$$\longrightarrow$$
$$\mathit{odd}(\mathrm{o})\langle\texttt{id}_{\texttt{Bool}}\rangle\langle\texttt{Bool!}\rangle\langle\texttt{Bool?}^{m}\rangle\langle\texttt{Bool!}\rangle$$
$$\longrightarrow$$
$$\mathit{odd}(\mathrm{o})\langle\texttt{Bool!}\rangle\langle\texttt{Bool?}^{m}\rangle\langle\texttt{Bool!}\rangle$$
$$\longrightarrow$$
$$\mathit{odd}(\mathrm{o})\langle\texttt{id}_{\texttt{Bool}}\rangle\langle\texttt{Bool!}\rangle$$
$$\longrightarrow$$
$$\mathit{odd}(\mathrm{o})\langle\texttt{Bool!}\rangle$$

Coercion Calculus

Syntax

$$c, d ::= \mathtt{id}_T \mid I! \mid I?^{\ell} \mid c \to d \mid c\,;d \mid \bot^{\ell}$$

Reduction

$$
\begin{array}{rcll}
c; \mathtt{id}_T & \longrightarrow & c & \\
\mathtt{id}_T; c & \longrightarrow & c & \\
I!; I?^{\ell} & \longrightarrow & \mathtt{id}_I & \\
I!; I'?^{\ell} & \longrightarrow & \bot^{\ell} & I \neq I' \\
(c{\to}d); (c'{\to}d') & \longrightarrow & (c'; c) \to (d; d') & \\
(\mathtt{id}_T \to \mathtt{id}_{T'}) & \longrightarrow & \mathtt{id}_{T \to T'} & \\
\bot^{\ell}; c & \longrightarrow & \bot^{\ell} & \\
c; \bot^{\ell} & \longrightarrow & \bot^{\ell} & \text{if } c \neq I?^{\ell'}
\end{array}
$$

Dynamic Typing. Henglein. ESOP 1992

Normalize adjacent coercions

$$
\begin{array}{rcll}
e & ::= & \cdots \mid e\langle c\rangle & \text{Terms} \\
u & ::= & n \mid \lambda x{:}T.\, e & \text{Uncoerced Values} \\
v & ::= & u \mid u\langle c\to d\rangle \mid u\langle I!\rangle & \text{Values}
\end{array}
$$

$$
\begin{array}{rcll}
(u\langle c\to d\rangle)\ v & \longrightarrow & (u\ \ v\langle c\rangle)\langle d\rangle & \\
u\langle \mathtt{id}_T\rangle & \longrightarrow & u & \\
e\langle c\rangle\langle d\rangle & \longrightarrow & e\langle c'\rangle & \text{if } (c;d) \longrightarrow^* c' \\
u\langle \bot^\ell\rangle & \longrightarrow & \mathtt{blame}\,\ell &
\end{array}
$$

Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

Coercions in normal form & composition

$$s, t ::= \mathtt{id}_\star \mid (I?^\ell \,;\, i) \mid i$$
$$i ::= (g \,;\, I!) \mid g \mid \bot^\ell \qquad g, h ::= \mathtt{id}_{\mathtt{Int}} \mid (s \to t)$$

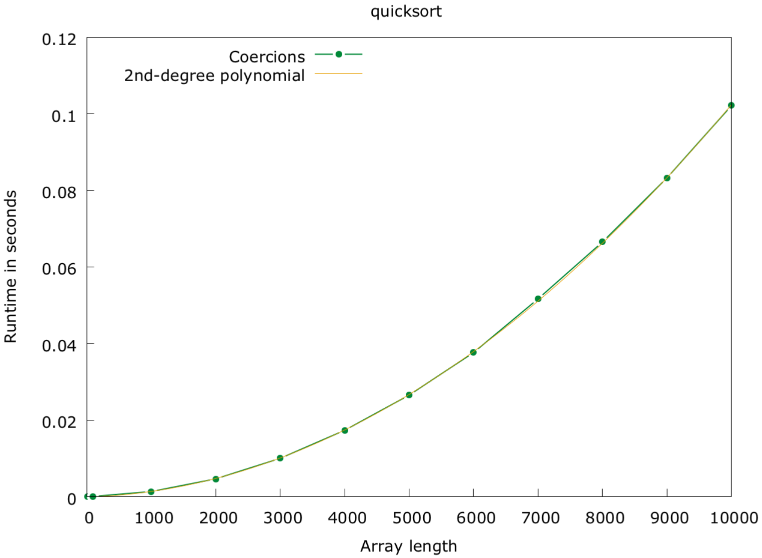
$$\boxed{s \fatsemi t = s}$$

$$\begin{aligned}
\mathtt{id}_{\mathtt{Int}} \fatsemi t = \mathtt{id}_\star \fatsemi t &= t \\
(s \to t) \fatsemi (s' \to t') &= (s' \fatsemi s) \to (t \fatsemi t') \\
(g \,;\, I!) \fatsemi \mathtt{id}_\star &= g \,;\, I! \\
(I?^\ell \,;\, i) \fatsemi t &= I?^\ell \,;\, (i \fatsemi t) \\
g \fatsemi (h \,;\, I!) &= (g \fatsemi h) \,;\, I! \\
(g \,;\, I!) \fatsemi (I?^\ell \,;\, i) &= g \fatsemi i \\
(g \,;\, I!) \fatsemi (I'?^\ell \,;\, i) &= \bot^\ell && \text{if } I \neq I' \\
\bot^\ell \fatsemi s = g \fatsemi \bot^\ell &= \bot^\ell
\end{aligned}$$

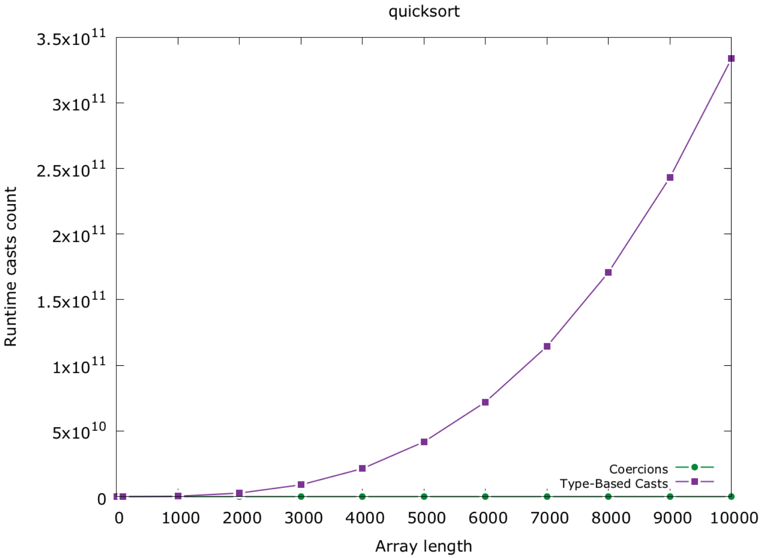
$$\mathcal{F}[e\langle s\rangle\langle t\rangle] \longrightarrow \mathcal{F}[e\langle s \fatsemi t\rangle]$$

Blame and coercion . . . Siek, Thiemann, Wadler. PLDI 2015.

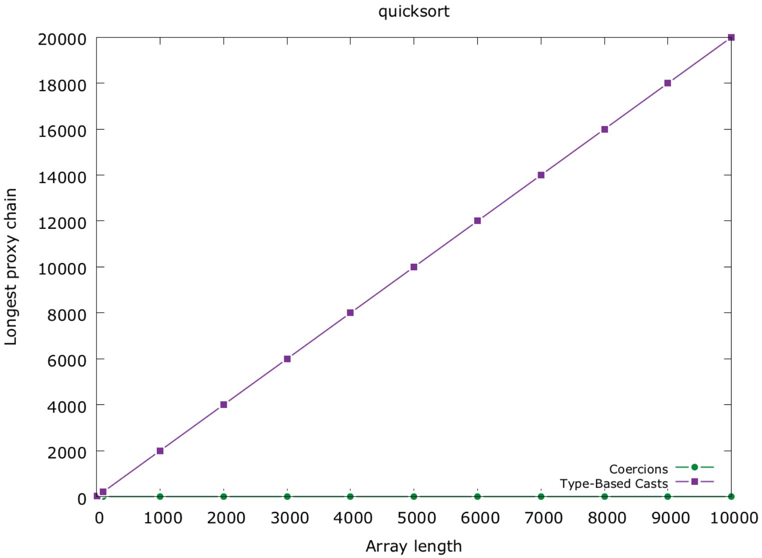
Coercions eliminate catastrophic slowdowns



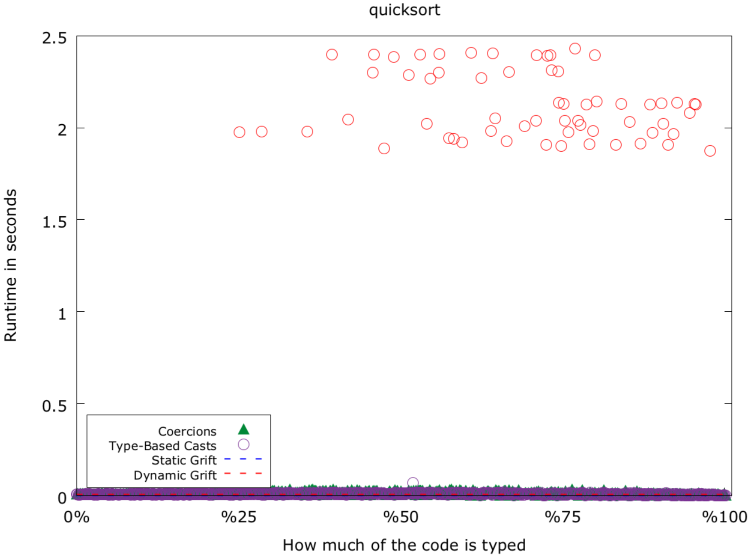
Coercions eliminate catastrophic slowdowns



Coercions eliminate catastrophic slowdowns

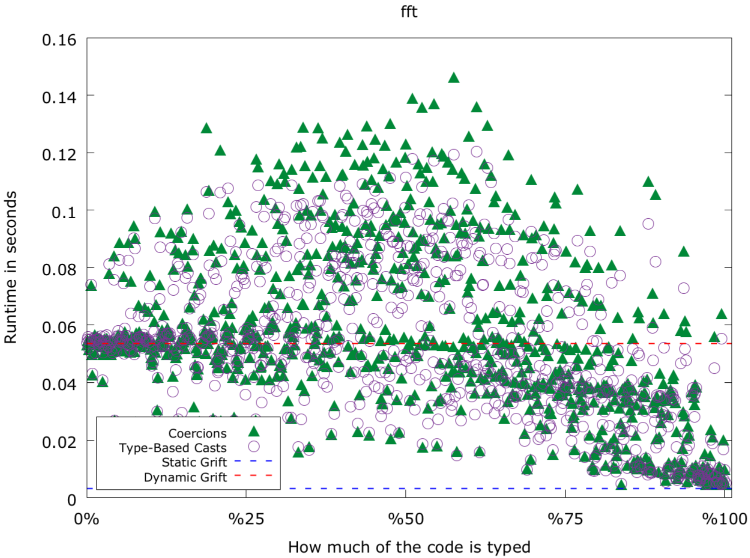


Coercions eliminate catastrophic slowdowns



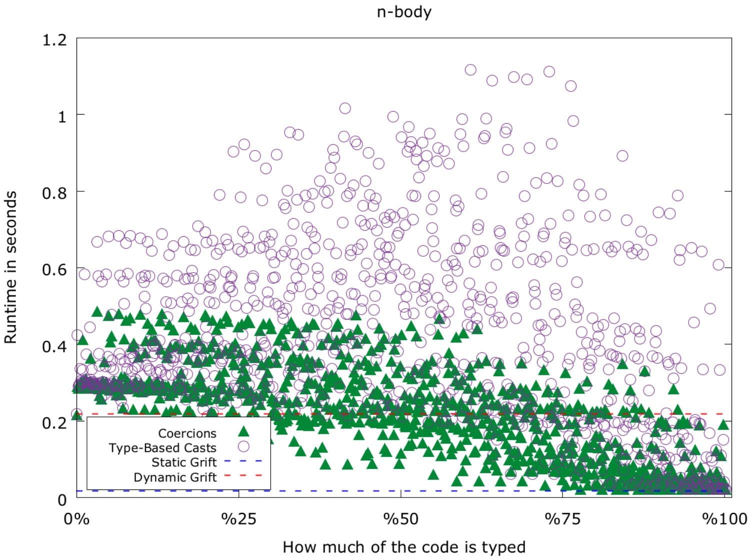
- mean speedup of 13 and max speedup of 1756

Coercions sometimes incur overhead



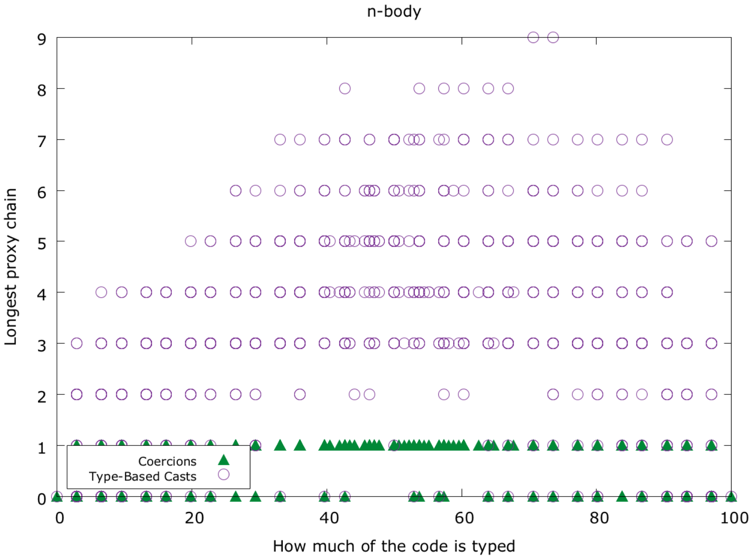
- mean slowdown of 1.1 and max slowdown of 1.6

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

$$T ::= \cdots \mid \texttt{Ref}\ T$$
$$e ::= \cdots \mid \texttt{ref}\ e \mid \texttt{!}^{\ell} e \mid e\ \texttt{:=}^{\ell}\ e$$
$$v ::= \cdots \mid a \mid a\langle \texttt{Ref}\ (c, d)\rangle$$

Consistency $\boxed{T \sim T}$

$$\ldots \qquad \frac{T_1 \sim T_2}{\texttt{Ref}\ T_1 \sim \texttt{Ref}\ T_2}$$

Coercions

$$c ::= \ldots \mid \texttt{Ref}\ (c_1, c_2)$$

Compile Casts to Coercions

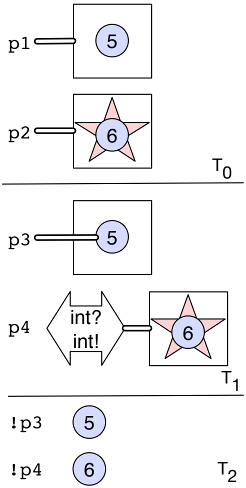
$$\langle\!\langle \texttt{Ref}\ T_1 \stackrel{\ell}{\Rightarrow} \texttt{Ref}\ T_2 \rangle\!\rangle = \texttt{Ref}\ (\langle\!\langle T_1 \stackrel{\ell}{\Rightarrow} T_2 \rangle\!\rangle, \langle\!\langle T_2 \stackrel{\ell}{\Rightarrow} T_1 \rangle\!\rangle)$$

Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

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 ⋆ ⇒ 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* $T = \texttt{Ref}\ T$, *then either*

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

Two rules for dereference

$$\texttt{!}a, \mu \longrightarrow \mu(a), \mu$$
$$\texttt{!}(a\langle \texttt{Ref}\ (c_1, c_2)\rangle), \mu \longrightarrow (\texttt{!}a)\langle c_1\rangle, \mu$$

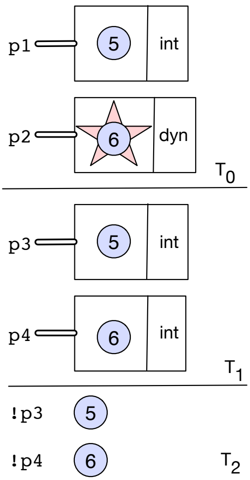
Two rules for update

$$a \texttt{ := } v, \mu \longrightarrow a, \mu(a \mapsto v)$$
$$a\langle \texttt{Ref}\ (c_1, c_2)\rangle \texttt{ := } v, \mu \longrightarrow a \texttt{ := } v\langle c_2\rangle, \mu$$

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.

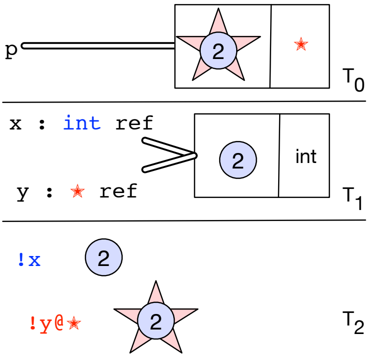
Monotonic Ref. for Efficient Gradual Typing. Siek et al. ESOP 2015

Aliasing and Static vs. Dynamic Dereference

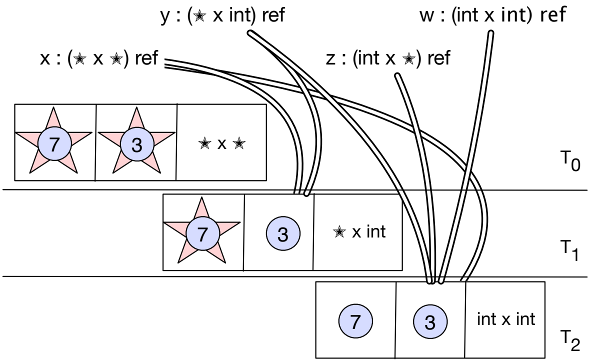
```
fun f(x:Ref Int, y:Ref ⋆)=
    !x + !y@⋆;

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

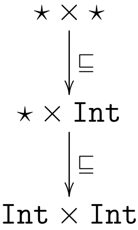
Compile-time choice:

- Fast static deref.
- Slow dynamic dereference



The Monotonic Invariant



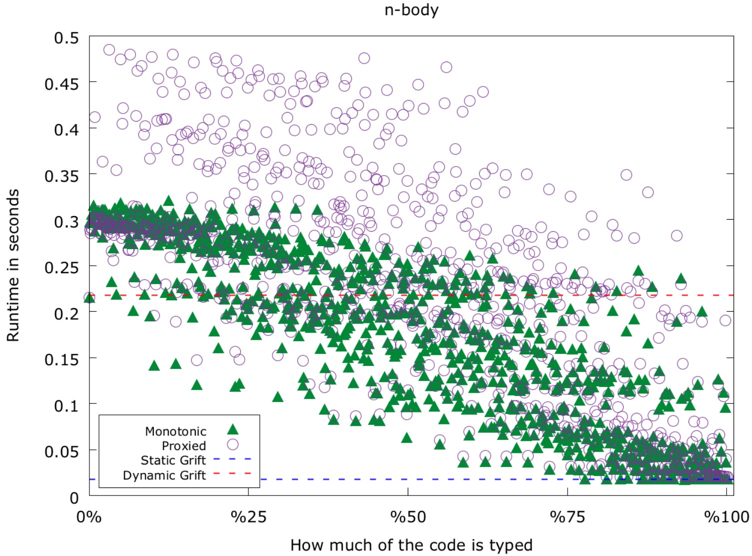


- The RTTI of a cell may become more precise.
- Every reference is less or equally precise as the RTTI.
- If a reference is fully static (e.g. w), then so is the cell.

Monotonic implementation

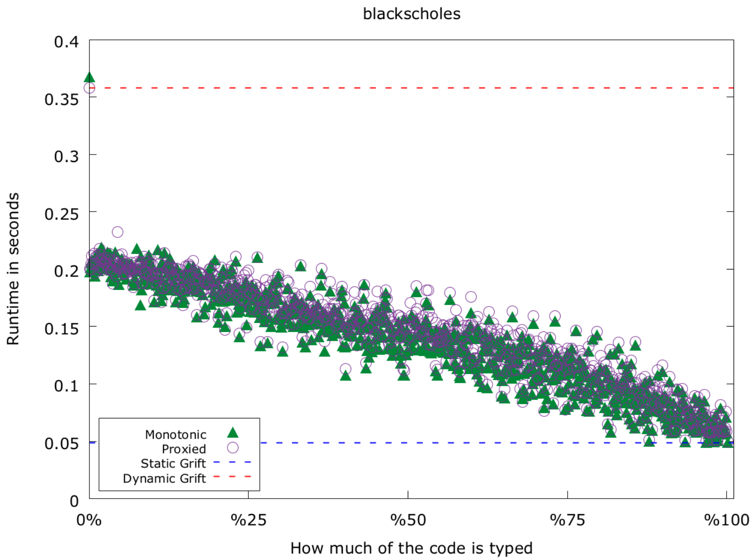
- hashconsing types at runtime to speedup the meet operation
- casted tuple values are just copies of the original tuples written to the heap and updated in place while the cast is in progress.

Monotonic eliminates overhead in static code



- mean speedup of 1.26 and max speedup of 3.24

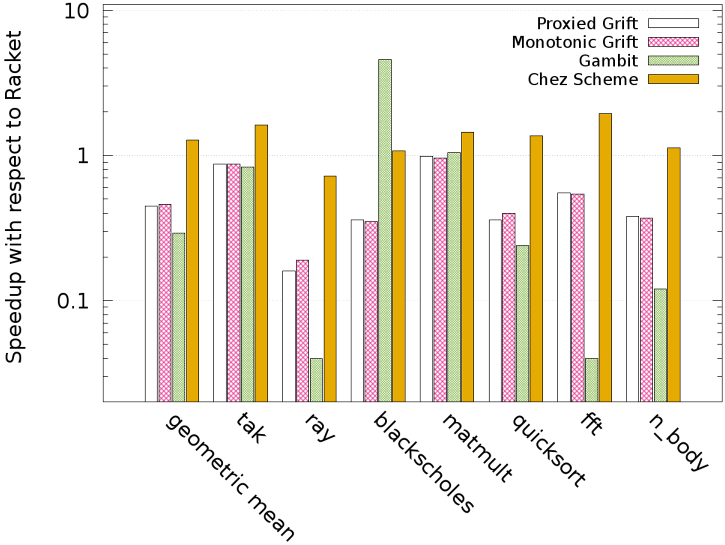
Montonic doesn’t matter for some programs



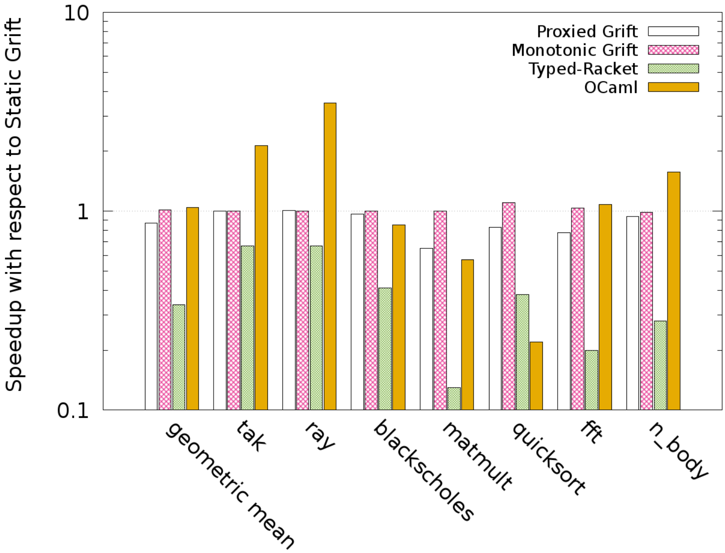
Efficient gradual typing

- 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: competative with Gambit.
- On statically typed code: competative 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!