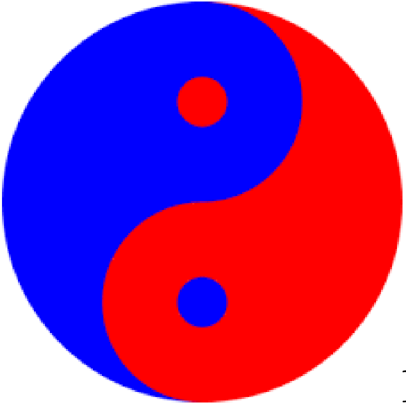
Challenges and Progress Toward Efficient Gradual Typing

Jeremy G. Siek
Indiana University, Bloomington

Andre Kuhlenschmidt
Deyaaeldeen Almahallawi

Dynamic Languages Symposium
October 2017

Gradual Typing



Static Dynamic

Efficient gradual typing

- Background on Gradual Typing
- Challenges to Efficiency
- Space-efficient Coercions
- Monotonic Coercions
- Performance Comparison

Gradual typing includes dynamic typing

An untyped program:

$$
\begin{array}{l}
\texttt{let} \\
\quad f = \lambda y.\, \texttt{1} + y \\
\quad h = \lambda g.\, g\;\; 3 \\
\texttt{in} \\
\quad h\; f
\end{array}
$$

$$\longrightarrow$$

$$\texttt{4}$$

Gradual typing includes dynamic typing

A buggy untyped program:

$$\begin{array}{l} \texttt{let} \\ \quad f = \lambda y.\, 1 + y \\ \quad h = \lambda g.\, g\ \boxed{\texttt{true}} \\ \texttt{in} \\ \quad h\ f \end{array}$$

$$\longrightarrow$$

$$\texttt{blame}\ \ell_2$$

Just like dynamic typing, the error is caught at run time.

Gradual typing includes static typing

A typed program:

$$
\begin{array}{l}
\texttt{let} \\
\quad f = \lambda y{:}\texttt{Int}.\, 1 + y \\
\quad h = \lambda g{:}\texttt{Int}{\to}\texttt{Int}.\, g\ \texttt{3} \\
\texttt{in} \\
\quad h\ f
\end{array}
$$

$$\longrightarrow$$

$$\texttt{4}$$

Gradual typing includes static typing

An ill-typed program:

$$
\begin{array}{l}
\texttt{let} \\
\quad f = \lambda y{:}\texttt{Int}.\, 1 + y \\
\quad h = \lambda g{:}\texttt{Int}{\to}\texttt{Int}.\, g\ \boxed{\texttt{true}} \\
\texttt{in} \\
\quad h\ f
\end{array}
$$

Just like static typing, the error is caught at compile time.

Gradual typing provides fine-grained mixing

A partially typed program:

$$
\begin{array}{l}
\texttt{let} \\
\quad f = \lambda y{:}\texttt{Int}.\, 1 + y \\
\quad h = \lambda g.\, g\ \ 3 \\
\texttt{in} \\
\quad h\ f \\
\longrightarrow \\
4
\end{array}
$$

Gradual typing protects type invariants

A buggy, partially typed program:

$$
\begin{array}{l}
\texttt{let} \\
\quad f = \lambda y{:}\,\texttt{Int}\,.\,1 + y \\
\quad h = \lambda g.\,g\ \texttt{true} \\
\texttt{in} \\
\quad h\ f \\
\longrightarrow \\
\texttt{blame}\ \ell_3
\end{array}
$$

The error is caught at runtime when the value is cast to an inconsistent type.

Gradual typing enables migration

$$
P(T_1, T_2) \equiv \begin{array}{l}
\texttt{let} \\
\quad f = \lambda y{:}T_1.\, \texttt{1} + y \\
\quad h = \lambda g{:}T_2.\, g\ \texttt{3} \\
\texttt{in} \\
\quad h\ f
\end{array}
$$

$$
\begin{tikzcd}[column sep=small]
 & & {\color{red}P(\star, \star)} \arrow[dll, no head] \arrow[d, no head] \arrow[dr, no head] \arrow[drr, no head] & & \\
P(\star, \texttt{Int}{\to}\texttt{Int}) \arrow[d, no head] & & P(\texttt{Int}, \star) \arrow[dll, no head] & P(\texttt{Bool}, \star) & P(\star, \texttt{Int}{\to}\texttt{Bool}) \\
{\color{blue}P(\texttt{Int}, \texttt{Int}{\to}\texttt{Int})} & & & &
\end{tikzcd}
$$

"Configuration Lattice"

Gradual type systems

The **consistency** relation governs implicit casts involving $\star$.

- For nominal type systems
  Anderson and Drossopoulou, WOOD 2003.

$$T_1 \sim T_2 \text{ iff } T_1 = T_2 \text{ or } T_1 = \star \text{ or } T_2 = \star$$

- For structural type systems
  Siek and Taha, SFP 2006.

$$\frac{}{T \sim \star} \qquad \frac{}{\star \sim T} \qquad \texttt{Int} \sim \texttt{Int}$$

$$\frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$$

Consistency is symmetric but not transitive.

1. Replace equality with consitency
2. Add rules for the unknown type $\star$

Typing rules for functions:

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash (\lambda x{:}T.e) : T \rightarrow T'} \qquad \frac{\Gamma \vdash e_1 : T{\rightarrow}T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'}$$

Gradual typing rules for functions:

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash (\lambda x{:}T.e) : T \rightarrow T'}$$

$$\frac{\Gamma \vdash e_1 : T{\rightarrow}T' \qquad \Gamma \vdash e_2 : T_2 \qquad \boxed{T \sim T_2}}{\Gamma \vdash e_1\ e_2 : T'} \qquad \frac{\Gamma \vdash e_1 : \boxed{\star} \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : \boxed{\star}}$$

Protecting the static from the dynamic

Recall the following buggy, partially typed program:

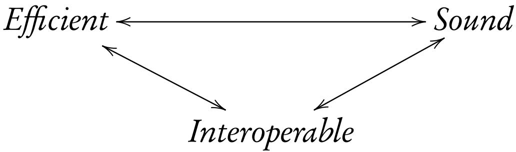
$$
\begin{array}{l}
\texttt{let} \\
\quad f = \lambda y : \texttt{Int}.\, 1 + y \\
\quad h = \lambda g.\, g\ \boxed{\texttt{true}} \\
\texttt{in} \\
\quad h\ f
\end{array}
$$

The untyped code tries to pass the Boolean `true` to parameter y of type `Int`.

Alternative ways to deal with this:

- erase types,
- insert casts, or
- limit interoperability.

There is tension in the design space



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

Approach: limit interoperability

A number of proposed designs place restrictions on passing values between static and dynamic regions.

- Siek and Taha. SFP 2006. (wrt. mutable references)
- Wrigstad et al. POPL 2010.
- Allende et al. OOPSLA 2014.
- Swamy et al. POPL 2014.

It's debatable whether these designs support gradual typing.

In particular, they do not satisfy the *gradual guarantee*. (*Refined Criteria for Gradual Typing.* Siek et al. SNAPL 2015.)

Approach: insert casts

$$
\begin{tikzcd}
\begin{array}{c}\text{Coercive Subtyping}\\ \text{Breazu-Tannen et al. 1989}\end{array} \arrow[d] \arrow[dr] & \\
\begin{array}{c}\text{Quasi-static Typing}\\ \text{Thatte 1990}\end{array} \arrow[r] \arrow[d] & \begin{array}{c}\text{Coercions}\\ \text{Henglein 1992}\end{array} \arrow[dl] \arrow[d] \\
\begin{array}{c}\star \leftrightarrow T\\ \text{Siek \& Taha 2006,}\\ \text{Gronski et al. 2006}\end{array} & \begin{array}{c}\text{Scheme} \leftrightarrow \text{ML}\\ \text{Matthews \& Findler 2006}\end{array} \arrow[d] \\
 & \begin{array}{c}\text{Scheme} \leftrightarrow \text{Typed Scheme}\\ \text{Tobin-Hochstadt \& Felleisen. 2006.}\end{array}
\end{tikzcd}
$$

Approach: insert casts

Compile the GTLC to STLC + casts.

A cast has the form

$$e' : T_1 \Rightarrow T_2$$

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

$$\frac{\Gamma \vdash e_1 \leadsto e_1' : T \to T' \qquad \Gamma \vdash e_2 \leadsto e_2' : T_2 \qquad T_2 \sim T}{\Gamma \vdash e_1\ e_2 \leadsto e_1'\ (e_2' : T_2 \Rightarrow T) : T'}$$

$$\frac{\Gamma \vdash e_1 \leadsto e_1' : \star \qquad \Gamma \vdash e_2 \leadsto e_2' : T_2}{\Gamma \vdash e_1\ e_2 \leadsto (e_1 : \star \Rightarrow \star \to \star)\ (e_2 : T_2 \Rightarrow \star) : \star}$$

Operational semantics of casts

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

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

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

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

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

$$(v : T_1{\to}T_2 \Rightarrow T'_1{\to}T'_2)\ v' \longrightarrow v\ (v' : T'_1 \Rightarrow T_1) : T_2 \Rightarrow T'_2$$

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

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

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

The Buggy Example Revisited

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

Efficient gradual typing

- Background on Gradual Typing
- **Challenges to Efficiency**
- Space-efficient Coercions
- Monotonic Coercions
- Performance Comparison

Casts can consume unbounded space

$$
\begin{array}{l}
\texttt{let}\,\texttt{rec}\ \mathit{even}(n{:}\texttt{Int}) : \star = \\
\quad \texttt{if}\ n = 0\ \texttt{then}\ \texttt{true} \\
\quad \texttt{else}\ \mathit{odd}(n - 1) \\
\\
\texttt{let}\,\texttt{rec}\ \mathit{odd}(n{:}\texttt{Int}) : \texttt{Bool} = \\
\quad \texttt{if}\ n = 0\ \texttt{then}\ \texttt{false} \\
\quad \texttt{else}\ \mathit{even}(n - 1)
\end{array}
$$

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

Casts can consume unbounded space

$$
\begin{array}{l}
\texttt{let rec}\ \mathit{even}(n{:}\texttt{Int}) : \star = \\
\quad \texttt{if}\ n = 0\ \texttt{then}\ (\texttt{true} : \texttt{Bool} \Rightarrow \star) \\
\quad \texttt{else}\ (\mathit{odd}(n - 1) : \texttt{Bool} \Rightarrow \star)
\end{array}
$$

$$
\begin{array}{l}
\texttt{let rec}\ \mathit{odd}(n{:}\texttt{Int}) : \texttt{Bool} = \\
\quad \texttt{if}\ n = 0\ \texttt{then false} \\
\quad \texttt{else}\ (\mathit{even}(n - 1) : \star \Rightarrow \texttt{Bool})
\end{array}
$$

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

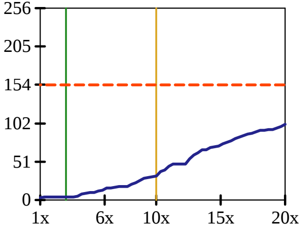
Casts can consume unbounded space

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

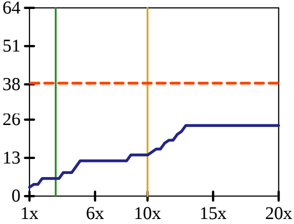
Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

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

Grift: an experimental compiler

- An ahead-of-time compiler for gradual typing.
- The input is a minimal functional language, output is C (e.g. no continuations).
- Benchmarking to-date is limited due to amount of language support.
- The compiler specializes casts when their source and/or target type is known at compile time.
- The runtime system implements **coercions** and **monotonic references**.

Efficient gradual typing

- Background on Gradual Typing
- Challenges to Efficiency
- **Space-efficient Coercions**
- Monotonic Coercions
- Performance Comparison

Compress casts via coercion reduction

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

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

Coercion Calculus

Syntax

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

Reduction

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

Dynamic Typing. Henglein. ESOP 1992

Normalize adjacent coercions

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

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

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

Coercions in normal form & composition

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

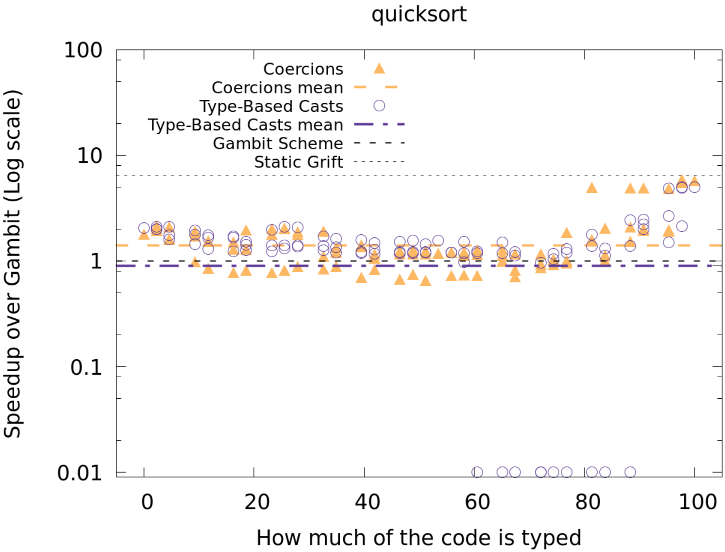
$$\boxed{s \fatsemi t = s}$$

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

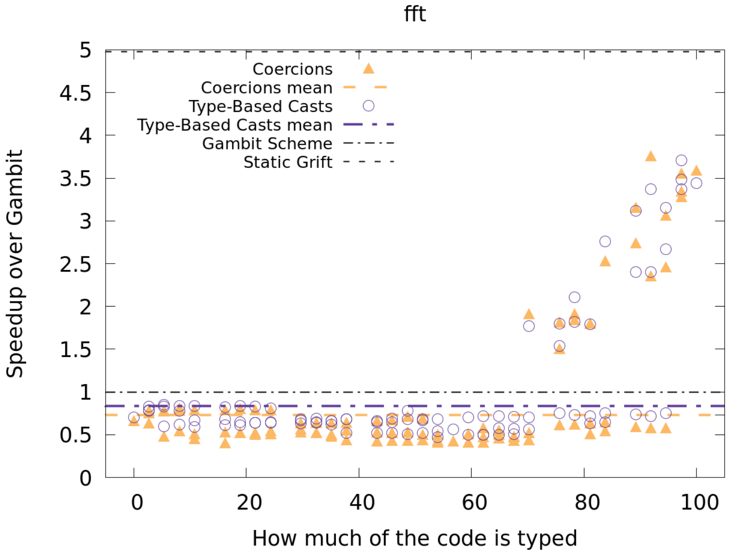
$$\mathcal{F}[e\langle s\rangle\langle t\rangle] \longrightarrow \mathcal{F}[e\langle s \fatsemi t\rangle]$$

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

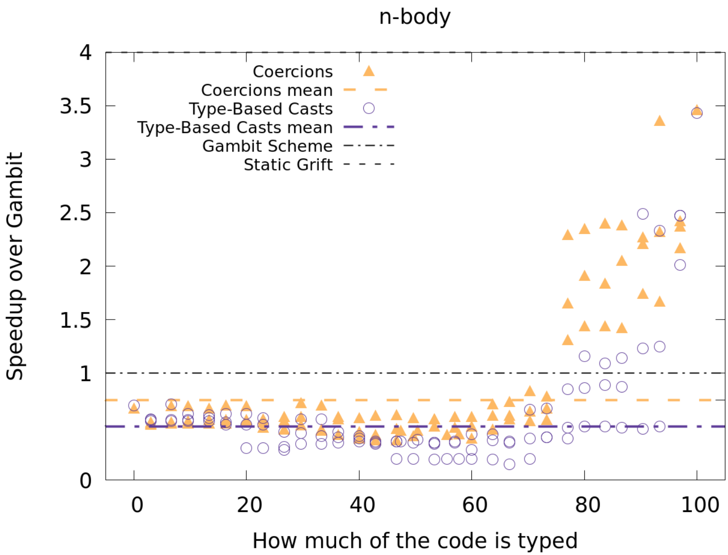
Coercions eliminate catastrophic slowdowns



Coercions sometimes incur overhead



Coercions sometimes pay for themselves



Efficient gradual typing

- Background on Gradual Typing
- Challenges to Efficiency
- Space-efficient Coercions
- **Monotonic Coercions**
- Performance Comparison

Gradual typing with mutable references

$$T ::= \cdots \mid \texttt{Ref}\ T$$
$$e ::= \cdots \mid \texttt{ref}\, e \mid \texttt{!}^{\ell} e \mid e \texttt{ :=}^{\ell}\ e$$

Consistency $\boxed{T \sim T}$

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

Coercions

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

Compile Casts to Coercions

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

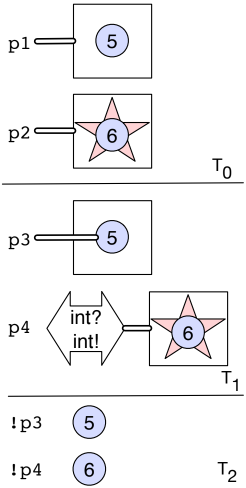
Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

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

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

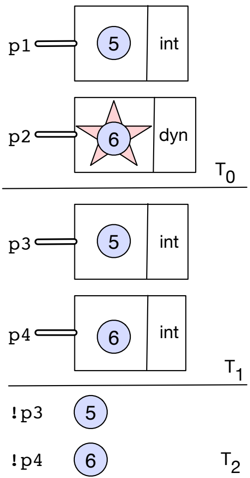
Two rules for update

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

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.

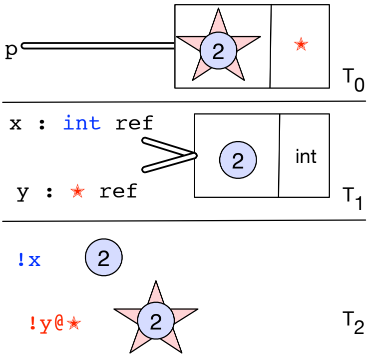
Monotonic Ref. for Efficient Gradual Typing. Siek et al. ESOP 2015

Aliasing and Static vs. Dynamic Dereference

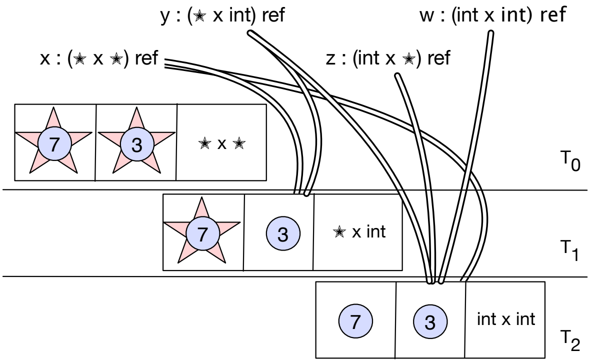
```
fun f(x:Ref Int, y:Ref ⋆)=
    !x + !y@⋆;

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

Compile-time choice:

- Fast static deref.
- Slow dynamic dereference



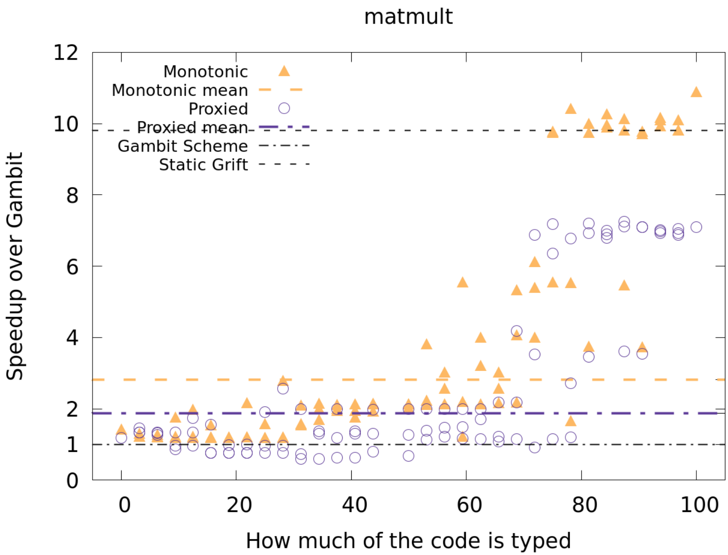
The Monotonic Invariant



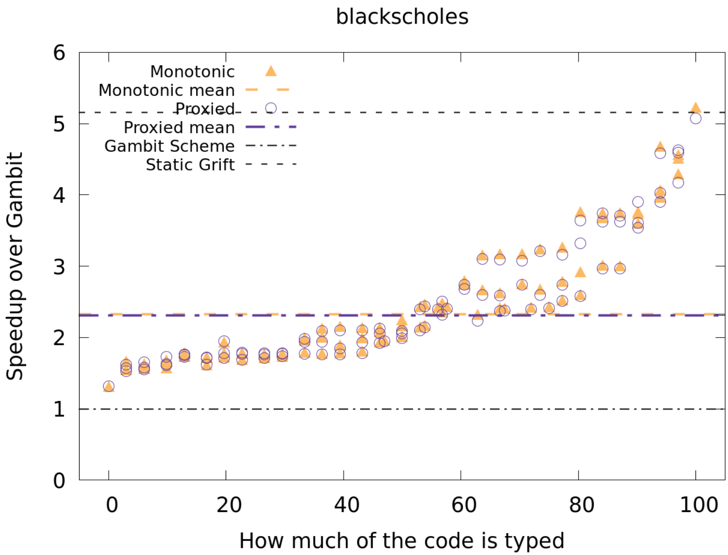
$$\star \times \star \;\sqsubseteq\; \star \times \mathtt{Int} \;\sqsubseteq\; \mathtt{Int} \times \mathtt{Int}$$

- 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 eliminates overhead in static code



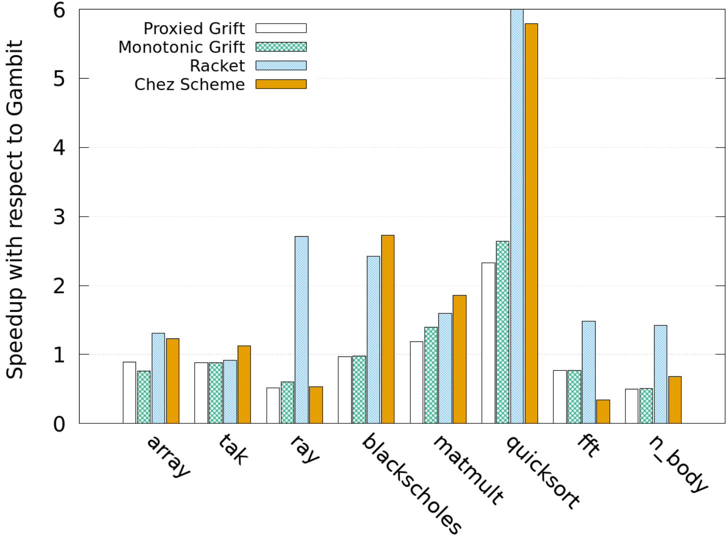
Montonic doesn't matter for some programs



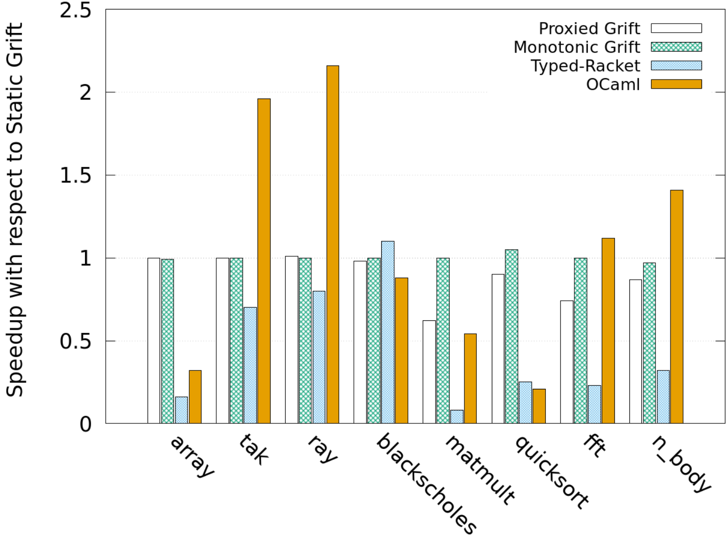
Efficient gradual typing

- Background on Gradual Typing
- Challenges to Efficiency
- 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.

Caveat: on the 8 benchmarks we have working so far.