Toward Efficient Gradual Typing

Jeremy G. Siek,
Andre Kuhlenschmidt,
Deyaaeldeen Almahallawi

Indiana University, Bloomington
Visiting Université Paris Diderot, IRIF

INRIA Gallium
21 February 2019

Toward Efficient Gradual Typing

- **Criteria for Gradually Typed Languages**
- Efficiency Problems, Solutions in Theory
- Implementations & the Grift Compiler
- Performance Evaluation

Three Languages

untyped		gradual		static
untyped	$\longleftrightarrow$	gradual	$\longleftrightarrow$	static
λ		$GTLC$		$\lambda^{\rightarrow}$
$e \Downarrow r$		$e \Downarrow_{?} r$		$e \Downarrow_{\rightarrow} r$
		$\Gamma \vdash_{?} e : T$		$\Gamma \vdash_{\rightarrow} e : T$
$T ::=$		$? \mid B \mid T \rightarrow T$		$B \mid T \rightarrow T$

Gradual typing includes dynamic typing

An untyped program:

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

Gradual typing includes dynamic typing

A buggy untyped program:

$$\begin{array}{l}
\texttt{let} \\
\quad f = \lambda y.\, \mathtt{1} + y \\
\quad h = \lambda g.\, g\ \boxed{\texttt{true}} \\
\texttt{in} \\
\quad h\, f \\
\Downarrow_{?} \\
\texttt{blame}\ \ell_2
\end{array}$$

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

Gradual typing includes dynamic typing

Let $\lfloor \cdot \rfloor$ be an embedding of the λ-calculus into the GTLC that casts every value to the unknown type.

Theorem (Embedding of λ-calculus)

Suppose that e is a term of the λ-calculus.

- $\emptyset \vdash_? \lfloor e \rfloor : ?$
- $e \Downarrow r \iff \lfloor e \rfloor \Downarrow_? \lfloor r \rfloor$

Gradual typing includes static typing

A typed program:

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

$$\longrightarrow$$

$$4$$

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 includes static typing

Definition (Static)

A type is *static* if it does not contain $?$.
A term is *static* if its type annotations do not contain $?$.

Theorem (Equivalence to $\lambda^{\rightarrow}$ on static terms)

Suppose e *is a static term and* T *is a static type.*

- $\emptyset \vdash_{\rightarrow} e : T \iff \emptyset \vdash_{?} e : T$
- $e \Downarrow_{\rightarrow} r \iff e \Downarrow_{?} r$

Gradual typing enables migration

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

$$
\begin{tikzcd}
P(\texttt{int}, \texttt{int}{\rightarrow}\texttt{int}) \arrow[d, dash] \arrow[dr, dash] & & & \\
P(?, \texttt{int}{\rightarrow}\texttt{int}) & P(\texttt{int}, ?) & P(\texttt{bool}, ?) & P(?, \texttt{int}{\rightarrow}\texttt{bool}) \\
& P(?, ?) \arrow[ul, dash] \arrow[u, dash] \arrow[ur, dash] \arrow[urr, dash] & &
\end{tikzcd}
$$

The Precision Relation

Precision on Types $\boxed{T \sqsubseteq T}$

$$? \sqsubseteq T \qquad \texttt{int} \sqsubseteq \texttt{int} \qquad \frac{T_1 \sqsubseteq T_1' \qquad T_2 \sqsubseteq T_2'}{T_1 \to T_2 \sqsubseteq T_1' \to T_2'}$$

Precision on Terms $\boxed{e \sqsubseteq e}$

$$\frac{T \sqsubseteq T' \qquad e_1 \sqsubseteq e_2}{\lambda x{:}T.\, e_1 \sqsubseteq \lambda x{:}T'.\, e_2} \qquad \frac{e_1 \sqsubseteq e_2 \qquad e_1' \sqsubseteq e_2'}{(e_1\ e_1') \sqsubseteq (e_2\ e_2')} \qquad \cdots$$

AKA naive subtyping, less-informative, and materialization.
Some authors put ? on top instead of bottom.

Gradual Guarantee, Part 1

Decreasing precision preserves type checking.

Theorem (Static Gradual Guarantee)
If $e' \sqsubseteq e$ *and* $\emptyset \vdash_{?} e : T$*, then* $\emptyset \vdash_{?} e' : T'$ *and* $T' \sqsubseteq T$.

Gradual Guarantee, Part 2

Decreasing precision preserves program behavior.

Increasing precision either preserves behavior or causes a runtime type error.

Theorem (Dynamic Gradual Guarantee)

Suppose $e' \sqsubseteq e$ *and* $\emptyset \vdash_? e : T$.

- *If* $e \Downarrow_? v$, *then* $e' \Downarrow_? v'$ *and* $v' \sqsubseteq v$.
- *If* $e' \Downarrow_? v'$, *then either* $e \Downarrow_? v$ *and* $v' \sqsubseteq v$ *or* $e \Downarrow_? \texttt{blame}\ \ell$.

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.

Soundness: gradual typing protects types

The result of an expression agrees with its type.

Let $\Gamma \vdash \rho$ be well-typed environments.

Theorem (Type Soundness)

If $\Gamma \vdash_? e : T$, $\Gamma \vdash \rho$, *and* $\rho \vdash e \Downarrow_? v$, *then* $\Gamma \vdash_? v : T$.

Toward Efficient Gradual Typing

- Criteria for Gradually Typed Languages
- **Efficiency Problems, Solutions in Theory**
- Implementations & the Grift Compiler
- Performance Evaluation

Space & Time Overhead of Higher-Order Casts

$$
\begin{array}{l}
\texttt{let rec}\ \mathit{even}(n{:}\texttt{int}) : {\color{red}?} = \\
\quad \texttt{if}\ n = 0\ \texttt{then true} \\
\quad \texttt{else}\ \mathit{odd}(n - 1)
\end{array}
$$

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

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

Space & Time Overhead of Higher-Order Casts

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

Space & Time Overhead of Higher-Order Casts

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

A Solution in Theory: Coercion Calculus

$$
\begin{array}{lrcl}
\text{ground types} & G, H & ::= & \mathtt{int} \mid \mathtt{bool} \mid ? \to ? \\
\text{coercions} & c, d & ::= & \mathtt{id} \mid G! \mid G?^{\ell} \mid c \to d \mid c\,;d \mid \perp^{\ell}
\end{array}
$$

$$
\begin{array}{rcll}
c; \mathtt{id} & \longrightarrow & c & \\
\mathtt{id}; c & \longrightarrow & c & \\
G!; G?^{\ell} & \longrightarrow & \mathtt{id} & \\
G!; H?^{\ell} & \longrightarrow & \perp^{\ell} & G \neq H \\
(c{\to}d); (c'{\to}d') & \longrightarrow & (c'; c) \to (d; d') & \\
\mathtt{id} \to \mathtt{id} & \longrightarrow & \mathtt{id} & \\
\perp^{\ell}; c & \longrightarrow & \perp^{\ell} & \\
c; \perp^{\ell} & \longrightarrow & \perp^{\ell} & \text{if } c \neq G?^{\ell'}
\end{array}
$$

Dynamic Typing. Henglein. ESOP 1992
Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

Closer to practice: the compose algorithm

$$
\begin{aligned}
s, t &::= \mathtt{id} \mid (G?^{\ell} \,;\, i) \mid i \\
i &::= (g \,;\, G!) \mid g \mid \perp^{\ell} \\
g, h &::= \mathtt{id} \mid (s \to t)
\end{aligned}
$$

$$\boxed{s \mathbin{⨟} t = s}$$

$$
\begin{aligned}
\mathtt{id} \mathbin{⨟} \mathtt{id} &= \mathtt{id} \\
(s \to t) \mathbin{⨟} (s' \to t') &= (s' \mathbin{⨟} s) \to (t \mathbin{⨟} t') \\
\mathtt{id} \mathbin{⨟} t &= t \\
(g \,;\, G!) \mathbin{⨟} \mathtt{id} &= g \,;\, G! \\
(G?^{\ell} \,;\, i) \mathbin{⨟} t &= G?^{\ell} \,;\, (i \mathbin{⨟} t) \\
g \mathbin{⨟} (h \,;\, G!) &= (g \mathbin{⨟} h) \,;\, G! \\
(g \,;\, G!) \mathbin{⨟} (G?^{\ell} \,;\, i) &= g \mathbin{⨟} i \\
(g \,;\, G!) \mathbin{⨟} (H?^{\ell} \,;\, i) &= \perp^{\ell} && \text{if } G \neq G' \\
\perp^{\ell} \mathbin{⨟} s &= \perp^{\ell} \\
g \mathbin{⨟} \perp^{\ell} &= \perp^{\ell}
\end{aligned}
$$

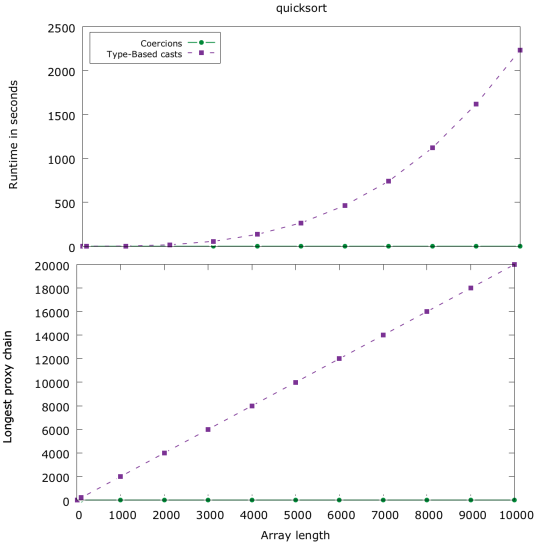
Blame and coercion … Siek, Thiemann, Wadler. PLDI 2015.

Compose Adjacent Coercions

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

$$
\begin{aligned}
(u\langle c{\to}d\rangle)\ v &\longrightarrow (u\ v\langle c\rangle)\langle d\rangle \\
u\langle \mathtt{id}\rangle &\longrightarrow u \\
u\langle \perp^{\ell}\rangle &\longrightarrow \mathtt{blame}\,\ell \\
{\color{red} e\langle c\rangle\langle d\rangle} &{\color{red}\longrightarrow e\langle c \mathbin{;} d\rangle}
\end{aligned}
$$

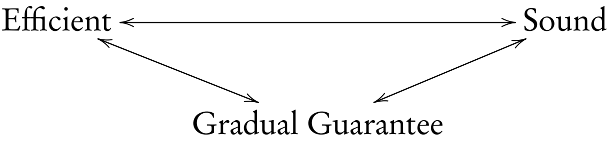
Quicksort with and without coercions



Toward Efficient Gradual Typing

- Criteria for Gradually Typed Languages
- Efficiency Problems, Solutions in Theory
- **Implementations & the Grift Compiler**
- Performance Evaluation

Tensions in the Design Space



Approach	Sound	Efficient	Gradual Guarantee
Erase types	◒	◒	●
Insert casts	●	◒	●
Limit interop.	●	●	◒

Implementation Landscape

System	Sound	Gradual Guarantee	$O(1)$ Overhead
Gradualtalk	●	●	○
Reticulated (G)	●	●	○
Nom	●	●	●
Grift	●	●	●
TypeScript	○	●	●
Reticulated (T)	◐	●	●
Safe TypeScript	●	○	●
Typed Racket	●	◐	○

Research Questions

- What is the speed of coercions wrt. regular casts?
- What is the overhead for gradual typing on:
  (1) statically typed code,
  (2) dynamically typed code, and
  (3) partially typed code?

The theory says $O(1)$, but what is the constant factor?

The Grift Compiler

- An ahead-of-time compiler. 23k LOC written in Racket.
- The source language GTLC+ includes first-class functions, mutable arrays, recursive types, tuples, integers, and floats.
- Compiles the GTLC+ to C.
- Implements coercions and compose (a C function).
- Values are 64 bits. Values of type ? are tagged.
- Specialize casts if neither source nor target is ?.
- Some optimization of function closures (e.g. direct calls).
- No global optimizations, no type inference or specialization.
- Boehm garbage collector.

Value Representation

$\boxed{\texttt{int}}$ 61-bit integer stored in 64 bits

$\boxed{\texttt{float}}$ double precision floating pointer number

$\boxed{\texttt{bool}}$ 0 or 1 stored in 64 bits

$\boxed{T_1 \to T_2}$ A 64-bit pointer to either
(1) a flat closure (function pointer and free variables), or
(2) a proxy closure, which contains three pointers to:
wrapper code, flat closure, and a coercion.

$\boxed{\texttt{ref}\ T}$ A 64-bit pointer (with 1-bit tag) to either
(1) the data,
(2) a proxy, with pointers to the data and a coercion.

$\boxed{?}$ A 64-bit value with 3-bits for a type tag.
Payload is stored in-line for types that can fit.
For others, payload is a pointer to a pair with the full type and a pointer to the value.

Coercion Representation

$\boxed{T?^p}$ 2×64 bits for pointer to type T and blame label.

$\boxed{T!}$ 64 bits for pointer to type.

$\boxed{c_1 \ldots c_n \to c_r}$ $(n+2) \times 64$ bits for secondary tag (with arity), n parameter coercions, and return coercion.

$\boxed{\texttt{ref}\; c\, d}$ 3×64 bits for tag and 2 coercions.

$\boxed{c \,;\, d}$ 2×64 bits 2 coercions.

$\boxed{\perp^p}$ 64 bits for blame label.

- Coercions are heap allocated objects, some during initialization and some at runtime.
- Types are heap allocated during program initialization) and we apply hash consing.

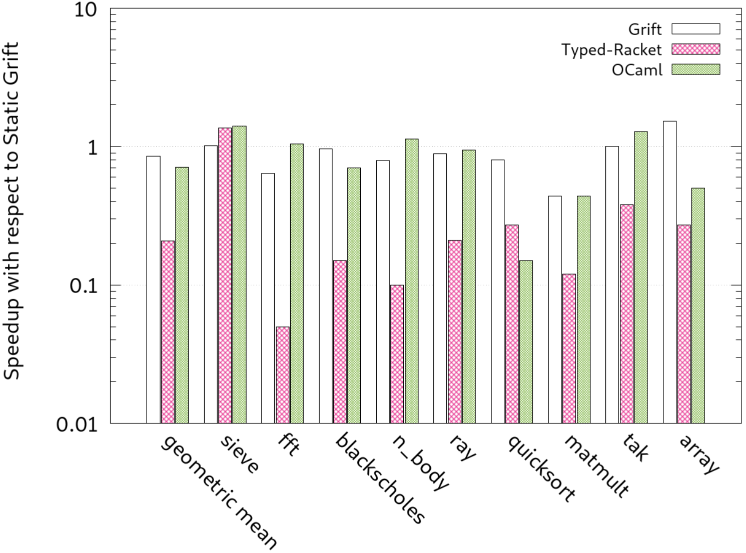
The Compose Procedure

```
crcn compose(crcn fst, crcn snd) {
  if (fst == ID) { return snd; }
  else if (snd == ID) { return fst; }
  else if (TAG(fst) == SEQUENCE_TAG) {
    sequence s1 = UNTAG_SEQ(fst);
    if (TAG(s1->fst) == PROJECT_TAG) {
      return MK_SEQ(s1->fst, compose(s1->snd, snd)); }
    else if (TAG(snd) == FAIL_TAG) { return snd; }
    else { sequence s2 = UNTAG_SEQ(snd);
      type src = UNTAG_INJ(s1->snd)->type;
      type tgt = UNTAG_PRJ(s2->fst)->type;
      blame lbl = UNTAG_PRJ(s2->fst)->lbl;
      crcn c = mk_crcn(src, tgt, lbl);
      return compose(compose(seq->fst, c), s2->snd); }
  } else if (TAG(snd) == SEQUENCE_TAG) {
    if (TAG(fst) == FAIL) { return fst; }
    else { crcn c = compose(fst, s2->fst);
      return MK_SEQ(c, UNTAG_SEQ(seq)->snd); }
  } else if (TAG(snd) == FAIL) {
      return TAG(fst) == FAIL ? fst : snd; }
  } else if (TAG(fst) == HAS_2ND_TAG) {
    snd_tag tag = UNTAG_2ND(fst)->second_tag;
    if (tag == FUN_COERCION_TAG) {
      return compose_fun(fst, snd);
    } else if (tag == REF_COERCION_TAG) {
      ref_crcn r1 = UNTAG_REF(fst);
      ref_crcn r2 = UNTAG_REF(snd);
      if (read == ID && write == ID) return ID;
      else { crcn c1 = compose(r1->read, r2->read);
        crcn c2 = compose(r2->write, r1->write);
        return MK_REF_COERCION(c1, c2); } }
  } else { raise_blame(UNTAG_FAIL(fst)->lbl); }
}
```

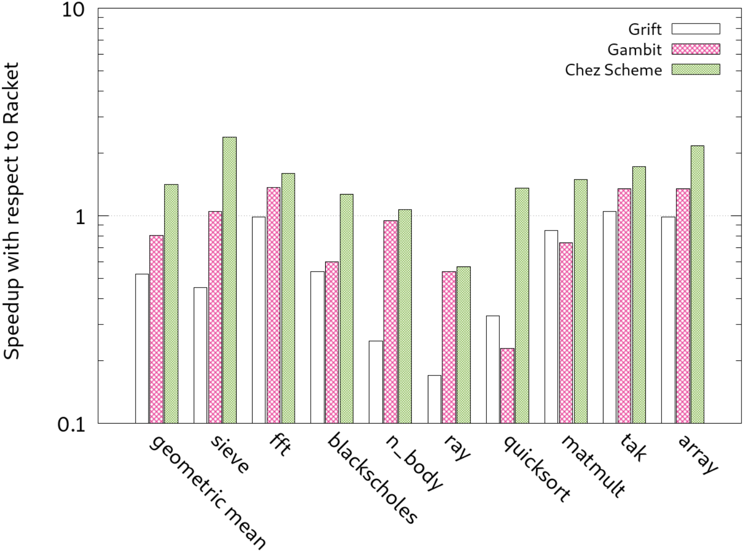
Toward Efficient Gradual Typing

- Criteria for Gradually Typed Languages
- Efficiency Problems, Solutions in Theory
- Implementations & the Grift Compiler
- **Performance Evaluation**

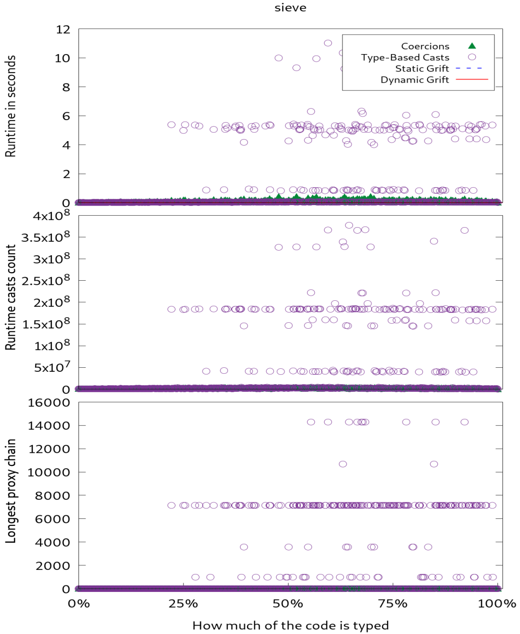
Situating Grift among Typed Languages



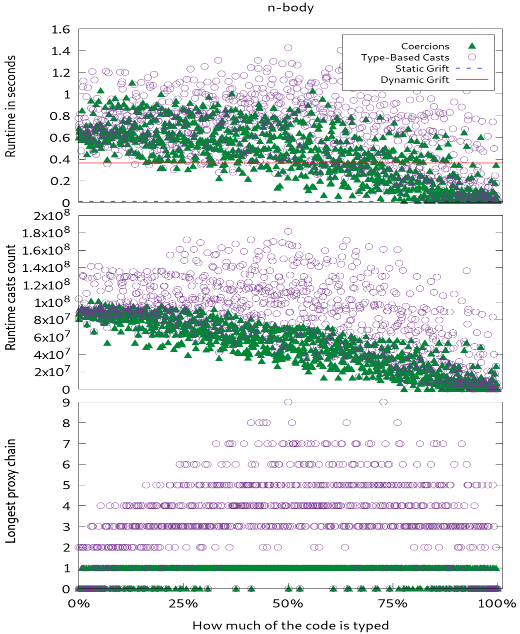
Situating Grift among Untyped Languages



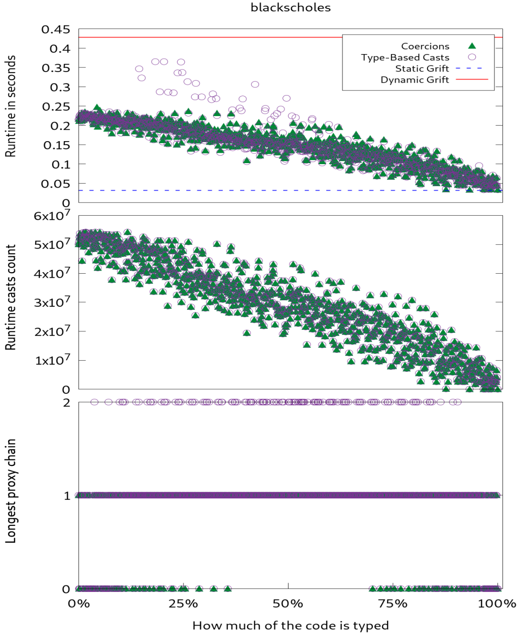
Partially-typed Sieve w/ & w/o coercions



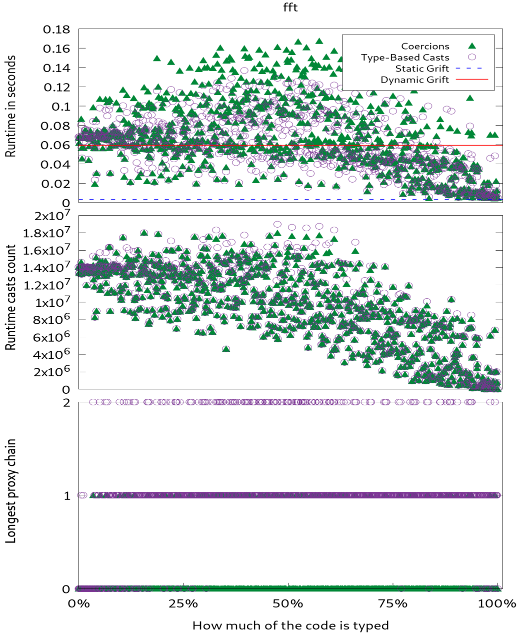
Partially-typed N-Body w/ & w/o coercions



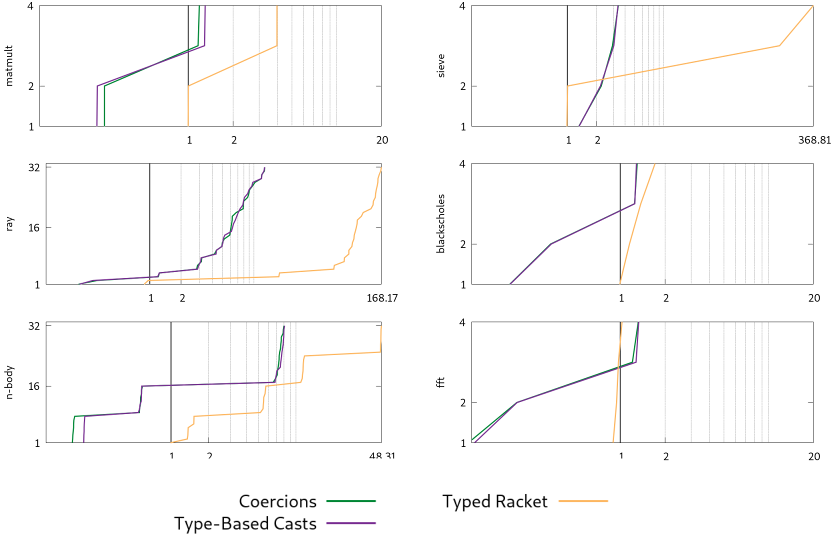
Partially-typed Blackscholes w/&w/o coercions



Partially-typed FFT w/ & w/o coercions



Comparison to Typed Racket



X-axis: slowdown wrt. Racket, Y-axis: number of configurations

Conclusion

- What is the speed with coercions wrt. regular casts?
  Much better on programs with proxy-chains.
  Similar on programs without proxy-chains.
- What is the overhead for Grift on:
  (1) statically typed code: up to 20% (matmult)
  (2) dynamically typed code: up to $5\times$ (ray), often $< 2\times$
  (3) partially typed code: up to $20\times$ (ray), often $< 2\times$
- Next steps:
  – Improve representation of coercions.
  – Reduce overhead in static code via monotonic pointers.
  – Optimizations such as type inference and inlining.

Draft of our PLDI 2019 paper:
`https://www.dropbox.com/s/eors60h9t15uv1h/grift-submission-nov-2019.pdf?dl=1`