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

\[ \lambda \quad GTLC \quad \lambda \rightarrow \]

\[ e \downarrow r \quad e \downarrow? r \quad e \downarrow\rightarrow r \]

\[ \Gamma \vdash? e : T \quad \Gamma \vdash\rightarrow e : T \]

\[ T ::= ? \mid B \mid T \rightarrow T \quad B \mid T \rightarrow T \]
Gradual typing includes dynamic typing

An untyped program:

```plaintext
let
  \( f = \lambda y. 1 + y \)
  \( h = \lambda g. g \ 3 \)
in
  \( h f \)
```

? 4
Gradual typing includes dynamic typing

A buggy untyped program:

\[
\begin{align*}
\text{let} & \quad f = \lambda y. 1 + y \\
& \quad h = \lambda g. g \text{ true} \\
\text{in} & \quad hf \\
\downarrow & \quad \text{blame } \ell_2
\end{align*}
\]

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

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

Theorem (Embedding of $\lambda$-calculus)

Suppose that $e$ is a term of the $\lambda$-calculus.

$\emptyset \vdash ? [e] : ?$

$e \Downarrow r \iff [e] \Downarrow ? [r]$
Gradual typing includes static typing

A typed program:

\[
\text{let } f = \lambda y:\text{int}. 1 + y \\
\quad h = \lambda g:\text{int} \rightarrow \text{int}. g 3 \\
\text{in } h f \\
\quad \rightarrow 4
\]
Gradual typing includes static typing

An ill-typed program:

```latex
let
f = \lambda y: \text{int}. 1 + y
h = \lambda g: \text{int} \rightarrow \text{int}. g \text{ true}

in
hf
```

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.

- $\emptyset \vdash e : T \iff \emptyset \vdash ? e : T$
- $e \ll e \ll r \iff e \ll ? r$
Gradual typing enables migration

\[ \begin{align*}
P(T_1, T_2) & \equiv \\
& \text{let } f = \lambda y : T_1. 1 + y \\
& \quad \text{let } h = \lambda g : T_2. g 3 \\
& \quad \text{in } h f
\end{align*} \]
The Precision Relation

Precision on Types

\[ T \sqsubseteq T \]

\[ ? \sqsubseteq T \quad \text{int} \sqsubseteq \text{int} \]

\[ \frac{T_1 \sqsubseteq T'_1 \quad T_2 \sqsubseteq T'_2}{T_1 \rightarrow T_2 \sqsubseteq T'_1 \rightarrow T'_2} \]

Precision on Terms

\[ T \sqsubseteq T' \quad e_1 \sqsubseteq e_2 \]

\[ \lambda x : T . e_1 \sqsubseteq \lambda x : T' . e_2 \]

\[ e_1 \sqsubseteq e_2 \quad e'_1 \sqsubseteq e'_2 \]

\[ (e_1 \; e'_1) \sqsubseteq (e_2 \; e'_2) \]

\[ \ldots \]

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)

\( \text{If } e' \sqsubseteq e \text{ and } \emptyset \vdash e : T, \text{ then } \emptyset \vdash e' : T' \text{ and } T' \sqsubseteq T. \)
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 $e \Downarrow ? \text{blame } \ell$. 
Gradual typing protects type invariants

A buggy, partially typed program:

\[
\text{let } f = \lambda y: \text{int}.1 + y \\
\quad h = \lambda g. g \text{ true} \\
\text{in} \\
\quad h f \\
\rightarrow \\
\quad \text{blame } \ell_3
\]

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 $\rho \vdash e \Downarrow \forall v$, then $\Gamma \vdash \forall v : T$. 
Toward Efficient Gradual Typing

- Criteria for Gradually Typed Languages
- Efficiency Problems, Solutions in Theory
- Implementations & the Grift Compiler
- Performance Evaluation
let rec even(n:int) : ? =
    if n = 0 then true
    else odd(n - 1)

let rec odd(n:int) : bool =
    if n = 0 then false
    else even(n - 1)
let rec even(n:int) : ? =
    if n = 0 then (true : bool ⇒ ?)
    else (odd(n - 1) : bool ⇒ ?)

let rec odd(n:int) : bool =
    if n = 0 then false
    else (even(n - 1) : ? ⇒ bool)
Space & Time Overhead of Higher-Order Casts

\[ \text{even}(5) \]
\[ \rightarrow \text{odd}(4) : \text{bool} \Rightarrow ? \]
\[ \rightarrow \text{even}(3) : ? \Rightarrow \text{bool} \Rightarrow ? \]
\[ \rightarrow \text{odd}(2) : \text{bool} \Rightarrow ? \Rightarrow \text{bool} \Rightarrow ? \]
\[ \rightarrow \text{even}(1) : ? \Rightarrow \text{bool} \Rightarrow ? \Rightarrow \text{bool} \Rightarrow ? \]
\[ \rightarrow \text{odd}(0) : \text{bool} \Rightarrow ? \Rightarrow \text{bool} \Rightarrow ? \Rightarrow \text{bool} \Rightarrow ? \]
A Solution in Theory: Coercion Calculus

ground types  \[ G, H ::= \text{int} | \text{bool} | ? \rightarrow ? \]
coercions  \[ c, d ::= \text{id} | G! | G?^\ell | c \rightarrow d | c ; d | \bot^\ell \]

\[ c ; \text{id} \rightarrow c \]
\[ \text{id} ; c \rightarrow c \]
\[ G! ; G?^\ell \rightarrow \text{id} \]
\[ G! ; H?^\ell \rightarrow \bot^\ell \quad G \neq H \]
\[ (c \rightarrow d) ; (c' \rightarrow d') \rightarrow (c' ; c) \rightarrow (d ; d') \]
\[ \text{id} \rightarrow \text{id} \rightarrow \text{id} \]
\[ \bot^\ell ; c \rightarrow \bot^\ell \]
\[ c ; \bot^\ell \rightarrow \bot^\ell \quad \text{if } c \neq G?^{\ell'} \]

Dynamic Typing. Henglein. ESOP 1992
Closer to practice: the compose algorithm

\[ s, t ::= \text{id} \mid (G?^{\ell} ; i) \mid i \]
\[ i ::= (g ; G!) \mid g \mid \bot^{\ell} \]
\[ g, h ::= \text{id} \mid (s \rightarrow t) \]

\[ \text{id} \cdot \text{id} = \text{id} \]
\[ (s \rightarrow t) \cdot (s' \rightarrow t') = (s' \cdot s) \rightarrow (t \cdot t') \]
\[ \text{id} \cdot t = t \]
\[ (g ; G!) \cdot \text{id} = g ; G! \]
\[ (G?^{\ell} ; i) \cdot t = G?^{\ell} ; (i \cdot t) \]
\[ g \cdot (h ; G!) = (g \cdot h) ; G! \]
\[ (g ; G!) \cdot (G?^{\ell} ; i) = g \cdot i \]
\[ (g ; G!) \cdot (H?^{\ell} ; i) = \bot^{\ell} \quad \text{if } G \neq G' \]
\[ \bot^{\ell} \cdot s = \bot^{\ell} \]
\[ g \cdot \bot^{\ell} = \bot^{\ell} \]

Blame and coercion ... Siek, Thiemann, Wadler. PLDI 2015.
Compose Adjacent Coercions

\[ e ::= \cdots | e\langle c \rangle \]

**Terms**

\[ u ::= n | \lambda x : T . e \]

**Uncoerced Values**

\[ v ::= u | u\langle c \to d \rangle | u\langle g ; G! \rangle \]

**Values**

\[
\begin{align*}
(u\langle c \to d \rangle) v & \rightarrow (u \circ v\langle c \rangle)\langle d \rangle \\
u\langle id \rangle & \rightarrow u \\
u\langle \bot^\ell \rangle & \rightarrow \text{blame } \ell \\
e\langle c \rangle\langle d \rangle & \rightarrow e\langle c \# d \rangle
\end{align*}
\]
Quicksort with and without coercions

![Graph showing runtime and longest proxy chain vs array length]
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

Efficient ← Gradual Guarantee → Sound

<table>
<thead>
<tr>
<th>Approach</th>
<th>Sound</th>
<th>Efficient</th>
<th>Gradual Guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erase types</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Insert casts</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Limit interop.</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
</tbody>
</table>
## Implementation Landscape

<table>
<thead>
<tr>
<th>System</th>
<th>Sound</th>
<th>Gradual Guarantee</th>
<th>$O(1)$</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradualtalk</td>
<td>●</td>
<td>●</td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>Reticulated (G)</td>
<td>●</td>
<td>●</td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>Nom</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Grift</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>TypeScript</td>
<td>○</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Reticulated (T)</td>
<td>○</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Safe TypeScript</td>
<td>●</td>
<td>○</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Typed Racket</td>
<td>●</td>
<td>○</td>
<td></td>
<td>○</td>
</tr>
</tbody>
</table>
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

int  61-bit integer stored in 64 bits

float double precision floating pointer number

bool 0 or 1 stored in 64 bits

$T_1 \rightarrow T_2$ 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.

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.

? 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

\[ T^p \] 2 × 64 bits for pointer to type \( T \) and blame label.

\[ T! \] 64 bits for pointer to type.

\[ c_1 \ldots c_n \rightarrow c_r \] \((n + 2) \times 64\) bits for secondary tag (with arity), \( n \) parameter coercions, and return coercion.

\[ \text{ref } c d \] 3 × 64 bits for tag and 2 coercions.

\[ c ; d \] 2 × 64 bits 2 coercions.

\[ \bot^p \] 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

![Graph showing speedup with respect to Racket for various benchmarks: Grift, Gambit, Chez Scheme.]
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