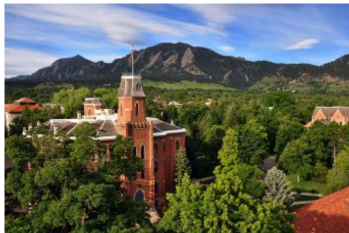
The gradual typing approach to mixing static and dynamic typing

Jeremy G. Siek

University of Colorado $\Longrightarrow$ Indiana University

TFP 2013 at Provo, Utah, May 2013



The Goals of Gradual Typing

- Enjoy the benefits of static & dynamic typing in different parts of the same program.
- Provide seamless interoperability between the static & dynamic parts.

Static Typing	Dynamic Typing
Reliability & Efficiency	Productivity

Overview

Gradual Typing:

- Basics
- History
- Functions
- Objects
- Generics
- Mutable State
- The Future

How can Static and Dynamic Coexist?

```
def abs(n: int) int:
  return -n if n<0 else n

def dist(x, y):
  return abs(x - y)
```

Gradual Typing for Functional Languages, Siek and Taha, SFP 2006.

How can Static and Dynamic Coexist?

```
def abs(n: int) int:
  return -n if n<0 else n

def dist(x :★, y :★) ★:
  return abs(x - y)
```

$$\star \sim \mathtt{int}$$

Consistency:

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

$$\frac{}{\mathtt{int} \sim \mathtt{int}} \qquad \frac{}{\mathtt{str} \sim \mathtt{str}}$$

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

Type rule for application:

$$\frac{\Gamma \vdash e_1 : T_1 \to T_3 \quad \Gamma \vdash e_2 : T_2 \qquad T_1 \sim T_2}{\Gamma \vdash e_1\, e_2 : T_3}$$

Gradual Typing for Functional Languages, Siek and Taha, SFP 2006.

Properly Catch Static Errors

```
def abs(n: int) int:
  return -n if n<0 else n

x : str = input_string()
...
abs(x)
```

$\texttt{str} \not\sim \texttt{int}$

Consistency:

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

$$\frac{}{\texttt{int} \sim \texttt{int}} \qquad \frac{}{\texttt{str} \sim \texttt{str}}$$

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

Compiler Performs Cast Insertion

```
def abs(n: int) int:
  return -n if n<0 else n

def dist(x :⋆, y :⋆) ⋆:
  return abs(x - y :⋆ ⇒ int) :int ⇒ ⋆

dist(7 :int ⇒ ⋆, 3 :int ⇒ ⋆)
```

The dynamic semantics specifies the behavior of casts, e.g.,

$$(7 : \mathtt{int} \Rightarrow \star) : \star \Rightarrow \mathtt{int} \longrightarrow 7$$
$$(7 : \mathtt{int} \Rightarrow \star) : \star \Rightarrow \mathtt{str} \longrightarrow \mathtt{error}$$

Overview

Gradual Typing:

- Basics
- **History**
- Functions
- Objects
- Generics
- Mutable State
- The Future

The Prehistory of Gradual Typing

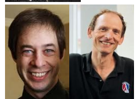
Lisp, early 1980's types as optimization hints

Abadi et al., 1991 defined a "dynamic" type with explicit injection and projection terms.

Cartwright & Fagan, 1991 soft typing: static analysis of dynamic programs

Henglein, 1992 expressed "dynamic" casts using an algebra of coercions.

Thatte, 1994 introduced implicit casts based on subtyping, which didn't quite work.



Findler & Felleisen, 2002 designed contracts for higher-order functions.

History of Gradual Typing (Abbreviated)

*Siek & Taha, 2006 implicit casts, consistency
Herman et al., 2006 space-efficient casts
*Siek & Taha, 2007 gradual typing & objects
Adobe, 2006 ActionScript becomes gradual
Sam TH & Felleisen, 2008 Typed Scheme
Wadler & Findler, 2009 the Blame Theorem
*Garcia et. al, 2009 space-efficient blame
Larry Wall, 2009 Perl 6 becomes gradual
Bierman et al, 2010 C# becomes gradual
*Ahmed, et al., 2011 gradual typing & generics
Hejlsberg, 2012 Microsoft releases TypeScript
*Siek et al., 2012 gradual typing & mutable state

Overview

Gradual Typing:

- Basics
- History
- **Functions**
- Objects
- Generics
- Mutable State
- The Future

Higher-Order Functions are Hard

```
def deriv(d: float, f: float→float) float:
  return lambda(x:float): (f(x+d)-f(x-d)) / (2.0*d)
```

```
1 def g(y):
2   if y > 0:
3     return y**3 - y - 1
4   else:
5     return "yikes"
6
7 deriv(0.01, g)(3.0)
8 deriv(0.01, g)(-3.0)
```

Higher-Order Functions and Blame

```
def deriv(d: float, f: float→float) float:
  return lambda(x:float): (f(x+d)-f(x-d)) / (2.0*d)
```

```
1 def g(y):
2   if y > 0:
3     return y**3 - y - 1
4   else:
5     return "yikes"
6
7 deriv(0.01, g)(3.0)
8 deriv(0.01, g)(-3.0)
```

Casting a function creates a "wrapper":

$$g : \star \to \star \Rightarrow^{8} \texttt{float} \to \texttt{float}$$

$$\longrightarrow$$

$$\lambda p : \texttt{float}.\ g\,(p : \texttt{float} \Rightarrow^{8} \star) : \star \Rightarrow^{8} \texttt{float}$$

$$\text{"yikes"} : \texttt{str} \Rightarrow \star \Rightarrow^{8} \texttt{float} \longrightarrow \textbf{blame}\ 8$$

Is Gradual Typing Unsound?

"adding type annotations at random places is unsound"
— Matthias Felleisen, PLT Mailing List, June 10, 2008.

Is Gradual Typing Unsound?

> *"adding type annotations at random places is unsound"*
> *— Matthias Felleisen, PLT Mailing List, June 10, 2008.*

Too weak:
If $\vdash e : T$, then either e diverges, $e \longrightarrow^* v$ and $\vdash v : T$, or
$\underline{e \longrightarrow^* \textbf{error}}$.

Is Gradual Typing Unsound?

"adding type annotations at random places is unsound"
— Matthias Felleisen, PLT Mailing List, June 10, 2008.

Too weak:
If $\vdash e : T$, then either e diverges, $e \longrightarrow^* v$ and $\vdash v : T$, or $\underline{e \longrightarrow^* \textbf{error}}$.

Too strong:
If $\vdash e : T$, then either e diverges or $e \longrightarrow^* v$ and $\vdash v : T$.

Is Gradual Typing Unsound?

> *"adding type annotations at random places is unsound"*
> *— Matthias Felleisen, PLT Mailing List, June 10, 2008.*

Too weak:
If $\vdash e : T$, then either e diverges, $e \longrightarrow^* v$ and $\vdash v : T$, or $\underline{e \longrightarrow^* \textbf{error}}$.

Too strong:
If $\vdash e : T$, then either e diverges or $e \longrightarrow^* v$ and $\vdash v : T$.

Just right:
If $\vdash e : T$, then either e diverges, $e \longrightarrow^* v$ and $\vdash v : T$, or $\underline{e \longrightarrow^* \textbf{blame}\,\ell}$ where $e = C[e' : T_1 \Rightarrow^\ell T_2]$ and $T_1 \not<: T_2$.

Well-typed Program's Can't be Blamed, Wadler & Findler, ESOP 2009
Exploring the Design Space of H.O. Casts, Siek et al., ESOP 2009

Blame and Subtyping

Wadler & Finder, 2009:

$$\frac{}{\texttt{int} <: \texttt{int}} \qquad \frac{}{\star <: \star} \qquad \frac{}{\texttt{int} <: \star}$$

$$\frac{T <: \star}{\star \to T <: \star} \qquad \frac{T_3 <: T_1 \qquad T_2 <: T_4}{T_1 \to T_2 <: T_3 \to T_4}$$

Siek, Garcia, & Taha, 2009:

$$\frac{}{\texttt{int} <: \texttt{int}} \qquad \frac{}{T <: \star}$$

$$\frac{T_3 <: T_1 \qquad T_2 <: T_4}{T_1 \to T_2 <: T_3 \to T_4}$$

But Wrappers are Not Space Efficient

```
def even(n: int, k: ⋆ → Bool) Bool:
   if n == 0:
     return k(True)
   else:
     return odd(n - 1, k)

def odd(n: int, k: Bool→Bool) Bool:
   if n == 0:
     return k(False)
   else:
     return even(n - 1, k)
```

Space-Efficient Gradual Typing. Herman, et al., TFP 2006

Toward Efficient Casts: Reified Wrappers

Regular wrappers:

$$v ::= \ldots \mid \lambda x : T.\, e$$

$$v : T_1 \to T_2 \Rightarrow T_3 \to T_4 \longrightarrow \lambda x : T_3.\, (v\,(x : T_3 \Rightarrow T_1)) : T_2 \Rightarrow T_4$$

$$\Downarrow$$

Reified wrappers:

$$v ::= \ldots \mid \lambda x : T.\, e \mid \color{red}{v : T_1 \to T_2 \Rightarrow T_3 \to T_4}$$

$$(v_1 : T_1 \to T_2 \Rightarrow T_3 \to T_4)\ v_2 \longrightarrow (v_1\ (v_2 : T_3 \Rightarrow T_1)) : T_2 \Rightarrow T_4$$

Compressing a Sequence of Casts

$$
\begin{array}{llllll}
v : & & \star & \to & \star & \\
& \Rightarrow & \star & \to \texttt{str} \to & \star & \\
& \Rightarrow & \texttt{int} \to & \star & \to & \star \\
& \Rightarrow & \star & \to & \star & \to \texttt{int}
\end{array}
\qquad
\begin{array}{ll}
v : & \star \to \star \\
\Rightarrow & \texttt{int} \to \texttt{str} \to \texttt{int} \\
\Rightarrow & \star \to \star \to \texttt{int}
\end{array}
$$

Define an information ordering:

$$
\frac{}{\star \sqsubseteq T} \qquad \frac{}{\texttt{int} \sqsubseteq \texttt{int}} \qquad \frac{}{\texttt{str} \sqsubseteq \texttt{str}} \qquad \frac{T_1 \sqsubseteq T_3 \qquad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4}
$$

Take the least upper bound to obtain a "triple":

$$
\begin{array}{c}
e : T_1 \Rightarrow T_2 \Rightarrow \cdots \Rightarrow T_{n-1} \Rightarrow T_n \\
e : T_1 \Rightarrow \sqcup\{T_2, \ldots, T_{n-1}\} \Rightarrow T_n
\end{array}
$$

Threesomes, with and without blame. Siek & Wadler, POPL 2010.

Space Efficiency

Notation: $|e|$ erases the casts from e.

Theorem (Space Efficiency)
For any program e there is a constant factor c such that if $e \longmapsto^* e'$, then $size(e') \leq c \cdot size(|e'|)$.

Overview

Gradual Typing:

- Basics
- History
- Functions
- **Objects**
- Generics
- Mutable State
- The Future

Gradual Typing and Subtyping

At the heart of most OO languages is a subsumption rule:

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

Thatte's early attempt at gradual typing didn't use consistency but instead put the dynamic type at the top and bottom of the subtyping relation.

$$T <: \star \qquad \star <: T$$

The Problem with Subtyping

Subtyping is transitive, so we have:

$$\frac{\texttt{str} <: \star \qquad \star <: \texttt{int}}{\texttt{str} <: \texttt{int}}$$

In general, for *any types* T_1 and T_2 we have $T_1 <: T_2$.

So the type checker accepts all programs!
(Even ones that get stuck.)

Consistency and Subtyping are Orthogonal

Let subtyping deal with object types:

$$[l_i : T_i \ ^{i \in 1..n+m}] <: [l_i : T_i \ ^{i \in 1..n}] \qquad \star <: \star$$

Let consistency deal with the dynamic type:

$$T \sim \star \qquad \star \sim T$$

Include the subsumption rule

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

and use consistency instead of equality:

$$\frac{\Gamma \vdash e_1 : T_1 \to T_3 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 \sim T_2}{\Gamma \vdash e_1\, e_2 : T_3}$$

Gradual Typing for Objects, Siek and Taha, ECOOP 2007

An Algorithmic Type System

The usual trick is to remove the subsumption rule and use subtyping in place of equality.

$$\frac{\Gamma \vdash e_1 : T_1 \to T_3 \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 <: T_1}{\Gamma \vdash e_1\, e_2 : T_3}$$

but for gradual typing, this would look like

$$\frac{\Gamma \vdash e_1 : T_1 \to T_3 \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 <: T_1' \qquad T_1' \sim T_1}{\Gamma \vdash e_1\, e_2 : T_3}$$

which is not syntax directed. We need a relation that composes the two:

$$\frac{\Gamma \vdash e_1 : T_1 \to T_3 \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 \lesssim T_1}{\Gamma \vdash e_1\, e_2 : T_3}$$

Consistent-Subtyping

$$\frac{}{T \lesssim \star} \qquad \frac{}{\star \lesssim T}$$

$$\frac{}{\texttt{int} \lesssim \texttt{int}} \qquad \frac{}{\texttt{str} \lesssim \texttt{str}}$$

$$\frac{T_3 \lesssim T_1 \qquad T_2 \lesssim T_4}{T_1 \to T_2 \lesssim T_3 \to T_4} \qquad \frac{T_i \lesssim T_i' \qquad \forall i \in 1..n}{[l_i : T_i \ ^{i \in 1..n+m}] \lesssim [l_i : T_i' \ ^{i \in 1..n}]}$$

(This is a more direct definition than the one I gave in *Gradual Typing for Objects*.)

Overview

Gradual Typing:

- Basics
- History
- Functions
- Objects
- **Generics**
- Mutable State
- The Future

Gradual Typing & Polymorphism

Review of System F:

$$\begin{array}{rcl} T & ::= & \ldots \mid \forall X.\, T \\ e & ::= & \ldots \mid \Lambda X.\, e \mid e[T] \end{array}$$

$$(\Lambda X.\, e)[T] \longrightarrow e[X{:=}T]$$

$$\frac{\Gamma, X \vdash e : T}{\Gamma \vdash \Lambda X.\, e : \forall X.\, T} \qquad \frac{\Gamma \vdash e : \forall X.\, T_1}{\Gamma \vdash e[T_2] : T_1[X{:=}T_2]}$$

Parametric Polymorphism (Generics)

Ahmed, Findler, and Wadler proposed a design at STOP 2009.

Their goals:

- Seamless interoperability.

$$v : (\forall X.\, S) \Rightarrow T \longrightarrow v[\star] : S[X{:=}\star] \Rightarrow T$$
$$v : S \Rightarrow (\forall X.\, T) \longrightarrow \Lambda X.\, (v : S \Rightarrow T)$$

- Retain relational parametricity (i.e., theorems for free).
- Provide a natural subtyping relation and blame theorem.

I helped refine the design for the POPL 2011 paper.

Challenges to Parametricity

$$K^\star = (\lambda x{:}\star.\lambda y{:}\star.x) : \star\to\star\to\star \Rightarrow \star$$

Consider two casts:

$$K^\star : \star \Rightarrow^m \forall X.\forall Y.\, X \to Y \to X$$
$$K^\star : \star \Rightarrow^\ell \forall X.\forall Y.\, X \to Y \to \color{red}{Y}$$

The second cast should lead to a cast failure.
But a naive semantics lets it go through.

$$(K^\star : \star \Rightarrow^\ell \forall X.\forall Y.\, X \to Y \to Y)[\texttt{int}][\texttt{int}]\,2\,3$$
$$\longrightarrow^* (K^\star : \star \Rightarrow^\ell \texttt{int} \to \texttt{int} \to \texttt{int})\,2\,3$$
$$\longrightarrow^* 2 : \texttt{int} \Rightarrow \star \Rightarrow^\ell \texttt{int}$$
$$\longrightarrow 2$$

Enforcement of Parametricity

$$
\begin{aligned}
& (K^\star : \star \Rightarrow^\ell \forall X. \forall Y. X \to Y \to X)[\mathtt{int}][\mathtt{int}]\, \mathtt{2}\, \mathtt{3} \\
\longrightarrow^* \; & (\nu X{:=}\mathtt{int}. \nu Y{:=}\mathtt{int}.\, K^\star : \star \Rightarrow^\ell X \to Y \to X)\, \mathtt{2}\, \mathtt{3} \\
\longrightarrow^* \; & (\nu X{:=}\mathtt{int}. \nu Y{:=}\mathtt{int}.\, \mathtt{2} : X \Rightarrow \star \Rightarrow^\ell X) \\
\longrightarrow \; & \mathtt{2}
\end{aligned}
$$

$$
\begin{aligned}
& (K^\star : \star \Rightarrow^\ell \forall X. \forall Y. X \to Y \to Y)[\mathtt{int}][\mathtt{int}]\, \mathtt{2}\, \mathtt{3} \\
\longrightarrow^* \; & (\nu X{:=}\mathtt{int}. \nu Y{:=}\mathtt{int}.\, K^\star : \star \Rightarrow^\ell X \to Y \to Y)\, \mathtt{2}\, \mathtt{3} \\
\longrightarrow^* \; & (\nu X{:=}\mathtt{int}. \nu Y{:=}\mathtt{int}.\, \mathtt{2} : X \Rightarrow \star \Rightarrow^\ell Y) \\
\longrightarrow \; & \mathbf{blame}\, \ell
\end{aligned}
$$

This mechanism “should” work, but the parametricity theorem is an open problem.

Overview

Gradual Typing:

- Basics
- History
- Functions
- Objects
- Generics
- **Mutable State**
- The Future

Gradual Typing & Mutable State

Consider ML-style references

$$\begin{array}{rcl} T & ::= & \ldots \mid \texttt{ref}\ T \\ e & ::= & \ldots \mid \texttt{ref}\ e \mid e := e \mid !e \end{array}$$

with a permissive rule for consistency of reference types:

$$\frac{T_1 \sim T_2}{\texttt{ref}\ T_1 \sim \texttt{ref}\ T_2}$$

The Standard Semantics Incurs Overhead

The Herman TFP 2006 semantics induces overhead, even in statically-typed regions of code.

$$\begin{array}{lcl} a & \in & \mathbb{N} \\ v & ::= & \ldots \mid a \mid v : \texttt{ref}\ T_1 \Rightarrow \texttt{ref}\ T_2 \end{array}$$

$$\texttt{ref}\ v \mid \mu \longmapsto a \mid \mu(a := v) \quad \text{if } a \notin \text{dom}(\mu)$$

$$!v \mid \mu \longmapsto \begin{cases} \mu(a) \mid \mu & \text{if } v = a \\ (!v') : T_1 \Rightarrow T_2 \mid \mu & \text{if } v = v' : \texttt{ref}\ T_1 \Rightarrow \texttt{ref}\ T_2 \end{cases}$$

$$v_1 := v_2 \mid \mu \longmapsto \begin{cases} v_2 \mid \mu(a := v_2) & \text{if } v_1 = a \\ v_1' := (v_2 : T_2 \Rightarrow T_1) \mid \mu & \text{if } v_1 = v_1' : \texttt{ref}\ T_1 \Rightarrow \texttt{ref}\ T_2 \end{cases}$$

Monotonic References

$$e ::= \ldots \mid \texttt{ref}\, e \mid e := e \mid !e \mid e := e@T \mid !e@T$$
$$v ::= \ldots \mid a$$

```
let r1 = ref (42 : int ⇒ ⋆) in
let r2 = r1 : ref ⋆ ⇒ ref int in
  (!r1@⋆, !r2)
```

$\texttt{ref}\,\star$ → $(42 : \texttt{int} \Rightarrow \star, \star)$

$\texttt{ref}\,\star \Longrightarrow \texttt{ref int}$ → $(42, \texttt{int})$

Standard vs. Monotonic

```
1 let r1 = ref (1 : int ⇒ ⋆) in
2 let r2 = r1 : ref ⋆ ⇒ ref int in
3 let r3 = r1 : ref ⋆ ⇒ ref bool in
4 let x = !r2 in
5 r3 := true;
6 let y = !r3 in
7   (x,y)
```

$\longmapsto^*$ $(1, \texttt{true})$ (standard)

$\longmapsto^*$ $\texttt{blame}$ 3 (monotonic)

$\boxed{e \mid \mu \longrightarrow e' \mid \Delta}$

Monotonic References

$$\texttt{ref}_T\, v \mid \mu \longrightarrow a \mid a := (v, T) \qquad \text{if } a \notin \mathrm{dom}(\mu)$$

$$!a \mid \mu \longrightarrow \mu(a)_1 \mid \epsilon$$

$$a := v \mid \mu \longrightarrow a \mid a := (v, \mu(a)_2)$$

$$a : \texttt{ref}\ T_1 \Rightarrow \texttt{ref}\ T_2 \mid \mu \longrightarrow \texttt{error} \mid \epsilon \qquad \text{if } T_2 \not\sim \mu(a)_2$$

$$a : \texttt{ref}\ T_1 \Rightarrow \texttt{ref}\ T_2 \mid \mu \longrightarrow a \mid \epsilon \qquad \text{if } T_2 \sqsubseteq \mu(a)_2$$

$$a : \texttt{ref}\ T_1 \Rightarrow \texttt{ref}\ T_2 \mid \mu \longrightarrow a \mid a := (e, \mu(a)_2 \sqcup T_2)$$

$$\text{if } T_2 \not\sqsubseteq \mu(a)_2,\ e = \mu(a)_1 : \mu(a)_2 \Rightarrow \mu(a)_2 \sqcup T_2$$

$$!a@T \mid \mu \longrightarrow (\mu(a)_1 : \mu(a)_2 \Rightarrow T) \mid \epsilon$$

$$a := v@T \mid \mu \longrightarrow a \mid a := (v : T \Rightarrow \mu(a)_2, \mu(a)_2)$$

$\boxed{e \mid \mu \longmapsto e' \mid \mu'}$

$$\frac{e \mid \mu \longrightarrow e' \mid \Delta}{e \mid \mu \longmapsto e' \mid \Delta(\mu)} \qquad \frac{\mu(a){=}(e_1, T) \quad e_1 \mid \mu \longrightarrow e_1' \mid \Delta}{e \mid \mu \longmapsto e \mid \Delta(\mu(a := (e_1', T)))}$$

Overview

Gradual Typing:

- Basics
- History
- Functions
- Objects
- Generics
- Mutable State
- **The Future**

The Future

- Gradually-typed Python (Michael Vitousek)
- Monotonic references with blame (some ideas, not easy)
- Monotonic objects (draft)
- "Putting it all together", e.g. can we maintain space efficiency with polymorphic blame? (no idea)
- Parametricity for the Polymorphic Blame Calculus (Amal Ahmed is part way there)
- Compiling and optimizing gradually-typed programs (e.g. Rastogi, Chaudhuri, and Hosmer, POPL 2012)

Questions?