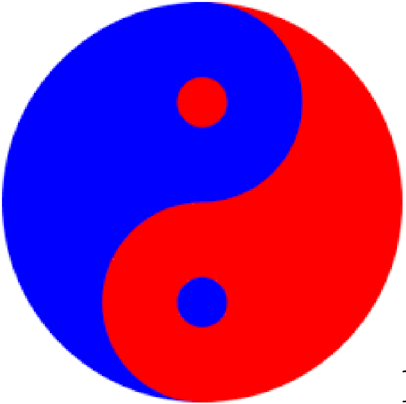
The State of the Art in Gradual Typing

Jeremy G. Siek
Indiana University, Bloomington

SICSA Summer School
on Practical Types
University of St. Andrews
August 2015

Integrating Static and Dynamic Typing



Static

Dynamic

State of the Art in Gradual Typing

Outline:

- Functions
  - Type System
  - Operational Semantics
  - Gradual Type Safety
  - Space and Time Efficiency
- Mutable References
- Objects
- Parametric Polymorphism

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:

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

$$\longrightarrow$$

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

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

Gradual typing includes static typing

A typed program:

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

$$\longrightarrow$$

$$\texttt{4}$$

Gradual typing includes static typing

An ill-typed program:

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

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

Gradual typing provides fine-grained mixing

A partially typed program:

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

$$\longrightarrow$$

$$4$$

Gradual typing protects type invariants

A buggy, partially typed program:

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

$\longrightarrow$

$\texttt{blame}\ \ell_3$

Gradual typing enables migration

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

$$\begin{tikzcd}
 & P(\star, \star) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
P(\star, \texttt{int}{\to}\texttt{int}) \arrow[d, dash] & P(\texttt{int}, \star) \arrow[dl, dash] & P(\texttt{bool}, \star) & P(\star, \texttt{int}{\to}\texttt{bool}) \\
P(\texttt{int}, \texttt{int}{\to}\texttt{int}) & & &
\end{tikzcd}$$

Why support static typing?

- **Communication**
  Machine-checked documentation of module interfaces.
- **Reliability**
  - Early error detection.
  - Protects abstractions and establishes invariants.
- **Productivity**
  Aids auto-completion and guides refactoring.
- **Efficiency**

Why support dynamic typing?

- ~~Don't have to write type annotations.~~
- **Expressiveness**
  Sometimes the most elegant and reusable expression of a software component won't type check.
- **Cognitive load**
  Sometimes thinking about the type system distracts from the programmer's current task.
- **Learning curve**
  For the beginner programmer, learning a static type system adds a significant hurdle.

Alternatives to Gradual Typing

- Add a **dynamic** type and **typecase** to a typed language.
  - CPL (1960's)
    "There is also a type **general** which designates an item whose type is not fixed and may, therefore, vary at run time." — D. W. Barron et al.
  - CLU (1970's)
  - Amber (1980's)
  - Modula-3 (1990's)
- Add an **object** type and subtyping (implicit upcast) to a typed language.

Alternatives to Gradual Typing, cont'd

- Type annotations trusted by an optimizing compiler.
  - Common LISP (1990)
  - Dylan (1996)
- Infer types (statically) from unannotated programs.
  - Hindley-Milner (1970's)
  - Soft Typing (1990's)
- Design a static type system for a dynamic language.
  - LISP (1970's)
  - Smalltalk (1980's and 1990's)
  - Erlang (1990's)
  - Scheme, Python, Ruby (2000's)

Integrating static & dynamic typing

Approach	Static	Dynamic	Migration
dynamic type & typecase	●	○	○
subtyping & downcast	●	○	○
type hints	○	●	○
soft typing	●	●	○
types for dyn. lang.	●	○	○
gradual typing	●	●	●

State of the Art in Gradual Typing

Outline:

- Functions
  - **Type System**
  - Operational Semantics
  - Gradual Type Safety
  - Space and Time Efficiency
- Mutable References
- Objects
- Parametric Polymorphism

A False Start

Notation: I write the **dynamic** type as $\star$.
Augment subtyping to allow implicit down-casts

$$\frac{}{T <: \star} \qquad \frac{}{\star <: T} \qquad \cdots$$

- Quasi-static Typing. Satish Thatte. POPL 1990.
- Sage and Hybrid Typing. Gronski, Knowles, Tomb, Freund, and Flanagan. SFP 2006.

But subtyping is transitive, so `int` $<:$ `string`!

- Thatte adds a "plausibility checking" post-processor.
- Gronski et al. specify a subtyping algorithm that differs from their declarative subtype relation.

Implementations, but no theory

Reflective method calls for receivers of type $\star$.

$$
\begin{array}{l}
\texttt{method}\ m(x : \star)\{ \\
\quad x.move(5, 3) \\
\}
\end{array}
$$

- Cecil. Chambers et al. Technical Report 2004.
- Visual Basic.NET. Meijer and Drayton. OOPSLA Workshop 2004.
- ProfessorJ. Gray, Findler, Flatt. OOPSLA 2005.

Gradual Type Systems

New "consistency" relation governs implicit casts involving $\star$.

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

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

- For structural type systems
  Gradually Typed Lambda Calculus (GTLC).
  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.

Replace Equality with Consitency

Rule for application in STLC:

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \; e_2 : T'}$$

Rules for application in the GTLC:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T_2 \\ T_2 \sim T\end{array}}{\Gamma \vdash e_1 \; e_2 : T'} \qquad \frac{\Gamma \vdash e_1 : \star \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \; e_2 : \star}$$

Exercise

Easier: What are the gradually typed versions of the typing rules for pairs?

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \\ \Gamma \vdash e_2 : T_2\end{array}}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash \mathsf{fst}\, e : T_1} \qquad \frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash \mathsf{snd}\, e : T_2}$$

Harder: What is the gradually typed version of the typing rule for disjoint sum elimination?

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 + T_2 \\ \Gamma, x : T_1 \vdash e_2 : T \qquad \Gamma, x : T_2 \vdash e_3 : T\end{array}}{\Gamma \vdash (\texttt{case}\, e_1\, \texttt{of}\, \texttt{inl}\, x \Rightarrow\ e_2 \mid \texttt{inr}\, x \Rightarrow\ e_3) : T}$$

Solution

Pairs:

$$\frac{\Gamma \vdash e : T \qquad T \rhd T_1 \times T_2}{\Gamma \vdash \mathsf{fst}\, e : T_1} \quad \frac{\Gamma \vdash e : T \qquad T \rhd T_1 \times T_2}{\Gamma \vdash \mathsf{snd}\, e : T_2}$$

where

$$\frac{}{(T_1 \times T_2) \rhd (T_1 \times T_2)} \qquad \frac{}{\star \rhd (\star \times \star)}$$

Sums:

$$\frac{\begin{array}{ccc} \Gamma \vdash e_1 : T_4 & & T_4 \rhd T_1 + T_2 \\ \Gamma, x : T_1 \vdash e_2 : T' & \Gamma, x : T_2 \vdash e_3 : T'' & T = T' \sqcap T'' \end{array}}{\Gamma \vdash (\texttt{case}\, e_1\, \texttt{of}\, \texttt{inl}\, x \Rightarrow e_2 \mid \texttt{inr}\, x \Rightarrow e_3) : T}$$

Greatest lower bound with respect to the less dynamic (imprecision) relation (e.g., $T \sqsubseteq \star$).

State of the Art in Gradual Typing

Outline:

- Functions
  - Type System
  - **Operational Semantics**
  - Gradual Type Safety
  - Space and Time Efficiency
- Mutable References
- Objects
- Parametric Polymorphism

Protecting the Static from the Dynamic

Recall the following buggy, partially typed program:

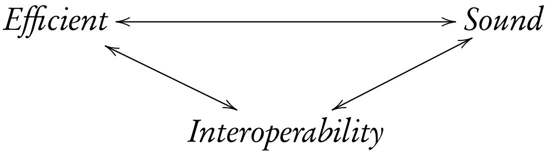
$$
\begin{array}{l}
\texttt{let} \\
\quad f = \lambda y : \texttt{int}.\, 1 + y \\
\quad h = \lambda g.\, g\ \texttt{true} \\
\texttt{in} \\
\quad h\ f
\end{array}
$$

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

Alternative ways to deal with this:

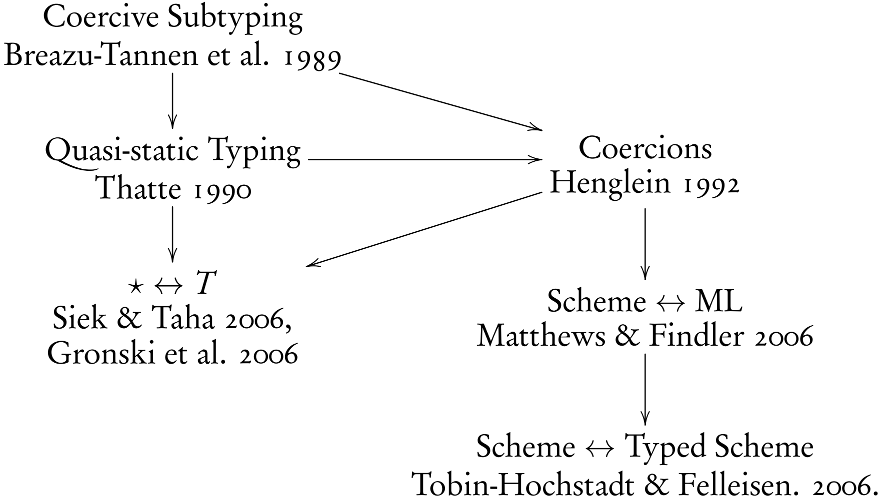
- Erase types.
- Insert casts.
- Limit interoperability.

Tensions in the Design Space



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

Approach: Insert Casts



Approach: Insert Casts

Compile the GTLC to STLC + casts (CC for Cast Calculus).

A cast has the form

$$e' : T_1 \Rightarrow T_2$$

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

$$\frac{\Gamma \vdash e_1 \leadsto e_1' : T \to T' \qquad \Gamma \vdash e_2 \leadsto e_2' : T_2 \qquad T_2 \sim T}{\Gamma \vdash e_1\ e_2 \leadsto e_1'\ \langle\!\langle e_2' : T_2 \Rightarrow T \rangle\!\rangle : T'}$$

$$\frac{\Gamma \vdash e_1 \leadsto e_1' : \star \qquad \Gamma \vdash e_2 \leadsto e_2' : T_2}{\Gamma \vdash \langle\!\langle e_1 : \star \Rightarrow \star \to \star \rangle\!\rangle\ \langle\!\langle e_2 : T_2 \Rightarrow \star \rangle\!\rangle : \star}$$

where

$$\langle\!\langle e' : T_1 \Rightarrow T_2 \rangle\!\rangle = \begin{cases} e' & \text{if } T_1 = T_2 \\ e' : T_1 \Rightarrow T_2 & \text{otherwise} \end{cases}$$

Operational Semantics of Casts

Ground types

$$G ::= \texttt{int} \mid \star{\to}\star$$

Values

$$v ::= n \mid \lambda x{:}T.f \mid v : G \Rightarrow \star$$

Reduction rules

$$\begin{aligned}
v : G \Rightarrow \star \Rightarrow G &\longrightarrow v \\
v : G \Rightarrow \star \Rightarrow G' &\longrightarrow \texttt{blame} \quad \text{if } G \neq G' \\
v : \texttt{int} \Rightarrow \texttt{int} &\longrightarrow v \\
v : \star \Rightarrow \star &\longrightarrow v \\
v : T_1{\to}T_2 \Rightarrow T_1'{\to}T_2' &\longrightarrow \lambda x{:}T_1'.\,(v\ \ (x : T_1' \Rightarrow T')) : T_2 \Rightarrow T_2' \\
v : T \Rightarrow \star &\longrightarrow v : T \Rightarrow G \Rightarrow \star\ ^{\dagger} \\
v : \star \Rightarrow T &\longrightarrow v : \star \Rightarrow G \Rightarrow T\ ^{\dagger}
\end{aligned}$$

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

The Buggy Example Revisited

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

Gradual Typing Protects Static Types

Every expression in a gradually typed program evaluates to a value whose type is equal to the static type of the expression.

Let $\rho \vdash e \Downarrow v$ be the environment-passing big-step semantics of CC. 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* $\emptyset \vdash v : T$.

Theorem (Canonical Forms)

Suppose $\emptyset \vdash v : T$.

- *If* $T = \mathtt{int}$, *then* $v = n$ *for some integer* n.
- *If* $T = T_1 \to T_2$, *then* $v = \langle \lambda x{:}T_1 .\, e, \rho \rangle$ *for some* x, e', *and* ρ.
- *If* $T = \star$, *then* $v = (v' : G \Rightarrow \star)$ *for some* v' *and* G.

Alternative: 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*.

Reminder: gradual typing enables migration

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

$$P(\star, \star)$$

$$P(\star, \texttt{int}{\rightarrow}\texttt{int}) \qquad P(\texttt{int}, \star) \qquad P(\texttt{bool}, \star) \qquad P(\star, \texttt{int}{\rightarrow}\texttt{bool})$$

$$P(\texttt{int}, \texttt{int}{\rightarrow}\texttt{int})$$

The Less Dynamic (Imprecision) Relation

Less Dynamic $\boxed{T \sqsubseteq T}$

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

Less Dynamic on Term $\boxed{e \sqsubseteq e}$

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

The Gradual Guarantee

Semantics of GTLC:

$$e \Downarrow v \equiv \exists e', T.\ \emptyset \vdash e \leadsto e' : T \text{ and } e' \longrightarrow^* v$$

Theorem (Gradual Guarantee)

Suppose $e \sqsubseteq e'$ and $\emptyset \vdash e : T$.

- $\emptyset \vdash e' : T'$ *and* $T \sqsubseteq T'$.
- *If* $e \Downarrow v$, *then* $e' \Downarrow v'$ *and* $v \sqsubseteq v'$.
  If e diverges then so does e'.
- *If* $e' \Downarrow v'$, *then either* $e \Downarrow v$ *and* $v \sqsubseteq v'$ *or* $e \Downarrow \texttt{blame}\,\ell$.
  If e' diverges, then either e diverges or $d \Downarrow \texttt{blame}\,\ell$.

Open problem: characterize when adding types is OK.

Exercise

What should the operational semantics for pairs look like?

- What should the ground types be?
- What should the values be?
- What are the reduction rules?

To get started, of course we need:

$$\mathsf{fst}\,(v_1, v_2) \longrightarrow v_1$$
$$\mathsf{snd}\,(v_1, v_2) \longrightarrow v_2$$

Solutions

Solution 1:

$$G ::= \cdots \mid \star \times \star$$
$$v ::= \cdots \mid (v, v)$$

$$v : T_1 \times T_2 \Rightarrow T_1' \times T_2' \longrightarrow ((\mathsf{fst}\, v) : T_1 \Rightarrow T_1', (\mathsf{snd}\, v) : T_2 \Rightarrow T_2')$$

Solution 2:

$$G ::= \cdots \mid \star \times \star$$
$$v ::= \cdots \mid (v, v) \mid v : T \times T \Rightarrow T \times T$$

$$\mathsf{fst}\,(v : T_1 \times T_2 \Rightarrow T_1' \times T_2') \longrightarrow (\mathsf{fst}\, v) : T_1 \Rightarrow T_1'$$
$$\mathsf{snd}\,(v : T_1 \times T_2 \Rightarrow T_1' \times T_2') \longrightarrow (\mathsf{snd}\, v) : T_2 \Rightarrow T_2'$$

State of the Art in Gradual Typing

Outline:

- Functions
  - Type System
  - Operational Semantics
  - **Gradual Type Safety**
  - Space and Time Efficiency
- Mutable References
- Objects
- Parametric Polymorphism

Type Safety for Gradual Typing

Is the following theorem precise enough?

Theorem (Type Safety)

If $\emptyset \vdash e : T$, *then either*

- $e \longrightarrow^* v$ *and* $\emptyset \vdash v : T$ *for some* v, *or*
- $e \longrightarrow^*$ `blame`, *or*
- e *diverges.*

No! This theorem is no stronger than a type safety theorem for a dynamically typed language. We want to know that

> "code in statically typed regions can't go wrong"
> — Tobin-Hochstadt and Fellseisen. DLS 2006.

Blame Tracking

Attach a *blame label* to each cast

$$e : T_1 \stackrel{\ell}{\Rightarrow} T_2$$

that represents source position information, for example

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

Contracts for Higher-Order Functions. Findler & Felleisen. ICFP 2002.

Blame Tracking

When a cast fails, include the label in the error report:

$$v : G \stackrel{\ell}{\Rightarrow} \star \stackrel{\ell'}{\Rightarrow} G' \longrightarrow \texttt{blame}\,\ell' \quad \text{if } G \neq G'$$

Propagate labels when reducing higher-order casts:

$$\begin{array}{l} v : T_1 \to T_2 \stackrel{\ell}{\Rightarrow} T_1' \to T_2' \\ \longrightarrow \\ \lambda x{:}T_1'.\,(v\ \ (x : T_1' \stackrel{\ell}{\Rightarrow} T')) : T_2 \stackrel{\ell}{\Rightarrow} T_2' \end{array}$$

Gradual Type Safety

Definition (Static Region)

An expression e' is a *statically typed region* of program e, written $static(e', e)$, if e' is a subexpression of e and e' is typable in the STLC.

$$static(e', e) \equiv \exists C.\ e = C[e'] \text{ and } \Gamma \vdash_{STLC} e' : T'$$

$\boxed{labels(e)}$

$$\begin{aligned}
labels(x) &= \emptyset \\
labels(n) &= \emptyset \\
labels((e_1\ \ e_2)^{\ell}) &= \{\ell\} \cup labels(e_1) \cup labels(e_2) \\
labels(\lambda x{:}T.\, e) &= labels(e)
\end{aligned}$$

Theorem (Gradual Type Safety)

If $\emptyset \vdash e \leadsto e' : T$*, then either*

- $e' \longrightarrow^* v$ *and* $\emptyset \vdash_{CC} v : T$ *for some* v*, or*
- $e' \longrightarrow^* \texttt{blame}\, \ell$ *and* $\forall e''$*,* $\mathit{static}(e'', e)$ *implies* $\ell \notin \mathit{labels}(e'')$*, or*
- e' *diverges.*

Proof Sketch for Gradual Type Safety

Lemma (Static Regions Produce no Labels)
If $\Gamma \vdash_{STLC} e : T$ *and* $\Gamma \vdash e \leadsto e' : T$, *then* $\mathit{labels}(e') = \emptyset$.

Lemma (Monotonicity of Labels)
If $e'_1 \longrightarrow e'_2$, *then* $\mathit{labels}(e'_2) \subseteq \mathit{labels}(e'_1)$.

Adapted from Siek, Garcia, and Taha. ESOP 2009.

Blame-Subtyping Theorem

But even some partially-typed regions are safe: regions that only involve implicit up-casts.

$$\boxed{T <: T}$$

$$\texttt{int} <: \texttt{int} \qquad \star <: \star \qquad \frac{T <: G}{T <: \star} \qquad \frac{S_1 <: T_1 \qquad T_2 <: S_2}{T_1 \to T_2 <: S_1 \to S_2}$$

$$\boxed{\Gamma \vdash e : T \; \mathit{safe} \; \ell}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T \to T' \; \mathit{safe} \; \ell \qquad \Gamma \vdash e_2 : T_2 \; \mathit{safe} \; \ell \\ (\ell \neq \ell' \text{ and } T_2 \sim T) \text{ or } (\ell = \ell' \text{ and } T_2 <: T)\end{array}}{\Gamma \vdash (e_1 \;\; e_2)^{\ell'} : T' \; \mathit{safe} \; \ell}$$

$$\vdots$$

Well-typed programs can't be blamed. Wadler & Findler. ESOP 2009

Theorem (Blame-Subtyping Theorem)

If

- $\emptyset \vdash e : T$ *safe* ℓ,
- $\emptyset \vdash e \leadsto e' : T$, *and*
- $e' \longrightarrow^* \texttt{blame}\, \ell'$,

then $\ell \neq \ell'$.

State of the Art in Gradual Typing

Outline:

- Functions
  - Type System
  - Operational Semantics
  - Gradual Type Safety
  - **Space and Time Efficiency**
- Mutable References
- Objects
- Parametric Polymorphism

Space Consumption of Casts

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

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

Space Consumption of Casts

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

Space Consumption of Casts

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

Coercion Calculus

Coercions

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

Terms

$$e ::= \cdots \mid e\langle c \rangle$$

Reduction

$$\begin{aligned}
v\langle \mathtt{id}_T \rangle &\longrightarrow v \\
(v\langle c \to d \rangle)\;\; W &\longrightarrow (v\;\; W\langle c \rangle)\langle d \rangle \\
v\langle G! \rangle\langle G?^\ell \rangle &\longrightarrow v \\
v\langle G! \rangle\langle G'?^\ell \rangle &\longrightarrow \mathtt{blame}\,\ell && \text{if } G \neq G' \\
v\langle c \,;\, d \rangle &\longrightarrow v\langle c \rangle\langle d \rangle \\
v\langle \bot^\ell \rangle &\longrightarrow \mathtt{blame}\,\ell
\end{aligned}$$

Dynamic Typing. Henglein. ESOP 1992
Blame and coercion . . . Siek, Thiemann, Wadler. PLDI 2015.

Compile Casts to Coercions

$$\boxed{\langle\!\langle T \overset{\ell}{\Rightarrow} T \rangle\!\rangle = c}$$

$$\begin{aligned}
\langle\!\langle \mathtt{int} \overset{\ell}{\Rightarrow} \mathtt{int} \rangle\!\rangle &= \mathtt{id}_{\mathtt{int}} \\
\langle\!\langle T_1 {\to} T_2 \overset{\ell}{\Rightarrow} T_1' {\to} T_2' \rangle\!\rangle &= \langle\!\langle T_1' \overset{\ell}{\Rightarrow} T_1 \rangle\!\rangle {\to} \langle\!\langle T_2 \overset{\ell}{\Rightarrow} T_2' \rangle\!\rangle \\
\langle\!\langle \star \overset{\ell}{\Rightarrow} \star \rangle\!\rangle &= \mathtt{id}_{\star} \\
\langle\!\langle G \overset{\ell}{\Rightarrow} \star \rangle\!\rangle &= G! \\
\langle\!\langle T \overset{\ell}{\Rightarrow} \star \rangle\!\rangle &= \langle\!\langle T \overset{\ell}{\Rightarrow} G \rangle\!\rangle \,;\, G! \qquad \dagger \\
\langle\!\langle \star \overset{\ell}{\Rightarrow} G \rangle\!\rangle &= G?^{\ell} \\
\langle\!\langle \star \overset{\ell}{\Rightarrow} T \rangle\!\rangle &= G?^{\ell} \,;\, \langle\!\langle G \overset{\ell}{\Rightarrow} T \rangle\!\rangle \qquad \dagger
\end{aligned}$$

$\dagger$ if $T \neq \star, T \neq G, T \sim G$

Normalized Coercions

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

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

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

Normalize Adjacent Coercions

$$\begin{array}{rcll}
u & ::= & n \mid \lambda x{:}T.\,e & \text{Uncoerced Values} \\
v & ::= & u \mid u\langle s \to t\rangle \mid u\langle g\,;\,G!\rangle & \text{Values} \\
\mathcal{E} & ::= & \mathcal{F} \mid \mathcal{F}[\square\langle c\rangle] & \text{Evaluation contexts} \\
\mathcal{F} & ::= & \square \mid \mathcal{E}[\square\ e] \mid \mathcal{E}[v\ \square] & \text{Cast-free contexts}
\end{array}$$

$$\begin{array}{rcll}
\mathcal{E}[(u\langle s \to t\rangle)\ v] & \longrightarrow & \mathcal{E}[(u\ v\langle s\rangle)\langle t\rangle] & \\
\mathcal{F}[u\langle \mathtt{id}\rangle] & \longrightarrow & \mathcal{F}[u] & \\
\mathcal{F}[e\langle s\rangle\langle t\rangle] & \longrightarrow & \mathcal{F}[e\langle s \mathbin{⨾} t\rangle] & \\
\mathcal{F}[u\langle \bot^{\ell}\rangle] & \longrightarrow & \mathtt{blame}\,\ell & \\
\mathcal{E}[\mathtt{blame}\,\ell] & \longrightarrow & \mathtt{blame}\,\ell & \text{if } \mathcal{E} \neq \square
\end{array}$$

Time Overhead in Function Application

Theorem (Canonical Forms)

Suppose $\emptyset \vdash v : T$.

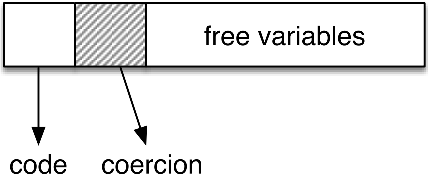
- *If* $T = T_1 \rightarrow T_2$, *then* $v = \lambda x{:}T_1.\, e$ *for some* x *and* e'. *or* $v = u\langle s \rightarrow t\rangle$.

Compiler has to insert a branch to decide which of the following two reduction rules to apply.

$$\mathcal{E}[(\lambda x{:}T.\, e)\ \ v] \longrightarrow \mathcal{E}[[x \mapsto v]e]$$

$$\mathcal{E}[(u\langle s \rightarrow t\rangle)\ \ v] \longrightarrow \mathcal{E}[(u\ \ v\langle s\rangle)\langle t\rangle]$$

Hybrid Closure Representation



Interpretations of the GTLC. Siek & Garcia. SFP 2012.

CEK Machine for the STLC

$$e ::= x \mid \lambda x{:}T.\, e \mid e\ e$$
$$v ::= n \mid \langle \lambda x{:}T.\, e, \rho \rangle$$

$$\begin{aligned}
\langle x, \rho, \mathcal{E} \rangle &\longmapsto \langle \rho(x), \rho, \mathcal{E} \rangle \\
\langle \lambda x{:}T.\, e, \rho, \mathcal{E} \rangle &\longmapsto \langle \langle \lambda x{:}T.\, e, \rho \rangle, \rho, \mathcal{E} \rangle \\
\langle (e_1\ e_2), \rho, \mathcal{E} \rangle &\longmapsto \langle e_1, \rho, \mathcal{E}[\Box\ \langle e_2, \rho \rangle] \rangle \\
\langle v, \rho, \mathcal{E}[\Box\ \langle e, \rho' \rangle] \rangle &\longmapsto \langle e, \rho', \mathcal{E}[v\ \Box] \rangle \\
\langle v, \rho, \mathcal{E}[\langle \lambda x{:}T.\, e, \rho' \rangle\ \Box] \rangle &\longmapsto \langle e, \rho'[x \mapsto v], \mathcal{E} \rangle
\end{aligned}$$

CEK Machine for the CC

$$
\begin{aligned}
e &::= x \mid \lambda x{:}T.\, e \mid e\ e \mid e : T \overset{\ell}{\Rightarrow} T \\
u &::= n \mid \langle \lambda x{:}T.\, e, \rho, [c] \rangle \mid \\
v &::= u \mid u\langle G! \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \lambda x{:}T.\, e, \rho, \mathcal{E} \rangle &\longmapsto \langle \langle \lambda x{:}T.\, e, \rho, () \rangle, \rho, \mathcal{E} \rangle \\
\langle v, \rho, \mathcal{E}[\langle \lambda x{:}T.\, e, \rho', c \rangle\ \square] \rangle &\longmapsto \langle e, \rho'[x \mapsto v, \mathsf{c} \mapsto c], \mathcal{E} \rangle \\
\langle e : T_1 \overset{\ell}{\Rightarrow} T_2, \rho, \mathcal{E} \rangle &\longmapsto \langle e, \rho, \mathcal{E}[\square \langle\!\langle T_1 \overset{\ell}{\Rightarrow} T_2 \rangle\!\rangle] \rangle \\
\langle e, \rho, \mathcal{F}[\square\langle c_1 \rangle][\square\langle c_2 \rangle] \rangle &\longmapsto \langle e, \rho, \mathcal{F}[\square\langle c_1 \fatsemi c_2 \rangle] \rangle \\
\langle v, \rho, \mathcal{F}[\square\langle c \rangle] \rangle &\longmapsto \langle v', \rho, \mathcal{F} \rangle \quad \text{if } \mathit{cast}(v, c) = v' \\
\langle v, \rho, \mathcal{F}[\square\langle c \rangle] \rangle &\longmapsto \mathtt{blame}\ \ell \quad \text{if } \mathit{cast}(v, c) = \mathtt{blame}\ \ell
\end{aligned}
$$

Apply Cast to Value

$\boxed{cast(v, c) = r}$

$$
\begin{aligned}
cast(u, G!) &= u\langle G!\rangle \\
cast(u\langle G!\rangle, G'?^{\ell}) &= \begin{cases} u & \text{if } G = G' \\ \mathtt{blame}\,\ell & \text{otherwise} \end{cases} \\
cast(\langle \lambda x.\, e, \rho, ()\rangle, c_2) &= \langle \lambda y.\, e', \rho, c_2\rangle \\
\text{where } e' &\equiv \mathtt{let}\ x = y\langle dom(\rho(\mathtt{c}))\rangle \mathtt{in}\ e\langle rng(\rho(\mathtt{c}))\rangle \\
cast(\langle \lambda x.\, e, \rho, c_1\rangle, c_2) &= \langle \lambda x.\, e, \rho, c_1 \mathbin{;} c_2\rangle \\
cast(v, \mathtt{id}) &= v
\end{aligned}
$$

State of the Art in Gradual Typing

Outline:

- Functions
  - Type System
  - Operational Semantics
  - Gradual Type Safety
  - Space and Time Efficiency
- **Mutable References**
- Objects
- Parametric Polymorphism

Mutable References

GTLC + 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

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

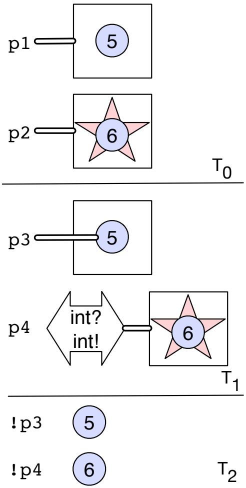
Space-Efficient Gradual Typing. Herman, Tomb, Flanagan. TFP 2006.

Example of overhead in reference access

```
fun f(p3:int ref, p4:int ref) =
    !p3 + !p4;
```

```
val p1 = ref 5;
val p2 = ref (6<int!>);

f(p1, p2<ref(int?,int!)>);
```

```
ref(int?,int!)
  : dyn ref ⇒ int ref
```



Problem: generated code for `!p3` and `!p4` must branch at run-time for the two kinds of references.

The Root of the Problem

Theorem (Canonical Forms)

Suppose $\emptyset \vdash v : T$.

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

Two rules for dereference

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

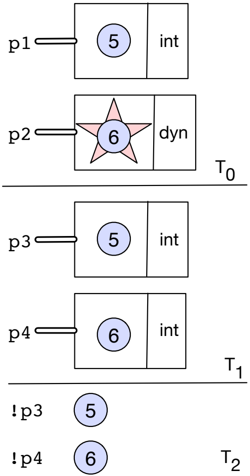
Two rules for update

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

Monotonic References

```
fun f(p3:int ref, p4:int ref)=
    !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.

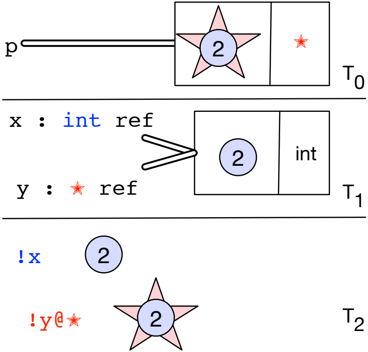
Monotonic Ref. for Efficient Gradual Typing. Siek et al. ESOP 2015

Aliasing and Static vs. Dynamic Dereference

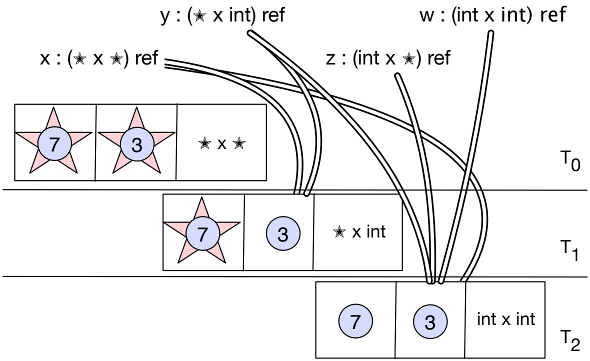
```
fun f(x:int ref, y:⋆ ref) =
    !x + !y@⋆;

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.

Reduction Rules for Casting References

Casting References $\mu(a) = cv : T_1$

$$\frac{T_3 = T_1 \sqcap T_2 \qquad T_3 \neq T_1}{a\langle \mathtt{ref}(T_2)\rangle, \mu \longrightarrow a, \mu(a \mapsto (cv\langle [\![T_1 \Rightarrow T_3]\!]\rangle) : T_3)}$$

$$\frac{T_3 = T_1 \sqcap T_2 \qquad T_3 = T_1}{a\langle \mathtt{ref}(T_2)\rangle, \mu \longrightarrow a, \mu}$$

$$\frac{T_1 \sqcap T_2 = \bot}{a\langle \mathtt{ref}(T_2)\rangle, \mu \longrightarrow \mathtt{error}, \mu}$$

Reduction Rules for Accessing References

Deference $\qquad\qquad\qquad\qquad \mu(a) = v : T$

$$!a, \mu \longrightarrow v, \mu$$

$$!a@T', \mu \longrightarrow v\langle\!\langle [\![T \Rightarrow T']\!] \rangle\!\rangle, \mu$$

Update

$$a := v', \mu \longrightarrow a, \mu(a \mapsto v' : T)$$

$$a := v'@T', \mu \longrightarrow a, \mu(a \mapsto (v'\langle [\![T' \Rightarrow T]\!] \rangle) : T)$$

Reduction Rules for Heap Quiescence

$$
\begin{array}{llcl}
\text{Casted Values} & cv & ::= & v \mid cv\langle c\rangle \\
\text{Heap} & \mu & ::= & \emptyset \mid \mu(a \mapsto v : T) \\
\text{Evolving Heap} & \nu & ::= & \emptyset \mid \nu(a \mapsto cv : T)
\end{array}
$$

$$
\frac{\nu(a) = cv : T \qquad cv, \nu \longrightarrow cv', \nu' \qquad \nu'(a)_{\mathsf{rtti}} = T}{e, \nu \longrightarrow e, \nu'(a \mapsto cv' : T)}
$$

$$
\frac{\nu(a) = cv : T \qquad cv, \nu \longrightarrow cv', \nu' \qquad \nu'(a)_{\mathsf{rtti}} \neq T}{e, \nu \longrightarrow e, \nu'}
$$

(omitted error handling rules)

Stay tuned...

- ... for performance evaluations.
- We are developing a compiler for the GTLC in which to empirically test these solutions.
- The PLT folks are evaluating and improving the efficiency of contracts.

Towards absolutely efficient gradually typed languages. Kuhlenschmidt et al. STOP 2015
Towards Practical Gradual Typing. Takikawa et al. ECOOP 2015

State of the Art in Gradual Typing

Outline:

- Functions
  - Type System
  - Operational Semantics
  - Gradual Type Safety
  - Space and Time Efficiency
- Mutable References
- **Objects**
- Parametric Polymorphism

Gradual Typing and Objects

$$e ::= \cdots \mid [m_i{:}T_i = \varsigma(x_i)e_i \;^{i\in 1..n}] \mid e.m(e) \mid e.m_T := \varsigma(x)e$$

- Recall that we use *consistency* for implicit casts to and from $\star$, not *subtyping*.
- But what if we want subtyping for other reasons?
- How can consistency and subtyping co-exist?

Answer: treat $\star$ like a basic type (e.g. `int`), not as the "top" type. Add subtyping and subsumption to your gradually typed language to make it object oriented.

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

Challenge: Algorithm Type Checking

- The subsumption rule is not syntax directed.
- So one has to remove it and use subtyping in place of type equality.

Example: STLC with subtyping:

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'}$$

becomes

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 <: T}{\Gamma \vdash e_1\ e_2 : T'}$$

Types and Programming Languages. Pierce 2002

Algorithm Type Checking: First Attempt

For the GTLC:

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 \sim T}{\Gamma \vdash e_1 \; e_2 : T'}$$

becomes

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 \sim T_2' \qquad T_2' <: T}{\Gamma \vdash e_1 \; e_2 : T'}$$

- But this rule is still not syntax directed!
- T_2' comes out of nowhere.

The Consistent-Subtyping Relation

Can we create a decision procedure, $T_1 \lesssim T_3$, for

$$\exists T_2.\, T_1 \sim T_2 \text{ and } T_2 <: T_3$$

Yes, take a syntax-directed definition of a subtype relation and add these two axioms for $\star$:

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

Example of a Consistent-Subtyping Relation

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

$$\frac{T'_1 \lesssim T_1 \qquad T_2 \lesssim T'_2}{T_1 \to T_2 \lesssim T'_1 \to T'_2} \qquad \frac{T_1 \lesssim T'_1 \qquad T_2 \lesssim T'_2}{T_1 \times T_2 \lesssim T'_1 \times T'_2}$$

Abadi-Cardelli object types:

$$\frac{T_i \sim T'_i \quad \forall i \in 1..n}{[m_i : T_i \ ^{i \in 1..n+m}] \lesssim [m_i : T'_i \ ^{i \in 1..n}]}$$

(No depth subtyping, Abadi-Cardelli objects can be updated.)

Gradual Type for Objects. Siek and Taha. ECOOP 2007

Wrappers and Object Identity

$$u ::= \cdots \mid [m_i{:}T_i = \varsigma(x_i)e_i \;^{i \in 1..n}] \qquad \text{Uncoerced Values}$$
$$v ::= u \mid \cdots \mid u\langle [m_i : s_i, t_i \;^{i \in 1..n}] \rangle \qquad \text{Values}$$

Naively, wrapped object has different identity (address) than the underlying object.

- Change *identity* to make the wrappers transparent. (Handling foreign functions is hard, Python $\leftrightarrow$ C.)
- Change the semantics to avoid wrappers:
  - Monotonic Casts
  - Transient Casts

Transparent Object Proxies in JS. Keil and Thiemann. ECOOP 2015
Design and Eval. of Grad. Typing for Python. Vitousek et al. DLS 2014

State of the Art in Gradual Typing

Outline:

- Functions
  - Type System
  - Operational Semantics
  - Gradual Type Safety
  - Space and Time Efficiency
- Mutable References
- Objects
- **Parametric Polymorphism**

Gradual Typing and Parametric Polymorphism

Extend the Cast Calculus with type abstraction and application:

$$e ::= \cdots \mid \Lambda X.\, e \mid e \;\; T$$

Allow casts between $\star$ and $\forall X.\, T$:

$$v : T_1 \overset{\ell}{\Rightarrow} (\forall X.\, T_2) \longrightarrow \Lambda X.\, (v : T_1 \overset{\ell}{\Rightarrow} T_2) \qquad \text{(GENERALIZE)}$$
$$\text{if } X \notin \mathrm{ftv}(T_1)$$

$$v : (\forall X.\, T_1) \overset{\ell}{\Rightarrow} T_2 \longrightarrow (v \;\; \star) : T_1[X \mapsto \star] \overset{\ell}{\Rightarrow} T_2$$
$$\text{(INSTANTIATE)}$$
$$\text{if } T_2 \neq \star \text{ and } T_2 \neq \forall X'.\, T_2' \text{ for any } X', T_2'$$

Blame for All. Ahmed et al. POPL 2011

The Problem with Type Substition

Recall the traditional reduction rule:

$$(\Lambda X.\, e)\;\; T \longrightarrow [X \mapsto T]e$$

Consider casting the constant function

$$K^{\star} = \lambda x{:}\, \star\, .\, \lambda y{:}\, \star\, .\, x$$

to the following polymorphic types.

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

The first cast should succeed. The second should fail because of parametricity.

The Problem with Type Substition

$$
\begin{aligned}
&(K^\star : \star \overset{\ell}{\Rightarrow} \forall X.\, \forall Y.\, X \to Y \to X)\ \texttt{int}\ \texttt{int}\ 2\ 3 \\
\longrightarrow^*\ &(K^\star : \star \overset{\ell}{\Rightarrow} \texttt{int} \to \texttt{int} \to \texttt{int})\ 2\ 3 \\
\longrightarrow^*\ &2
\end{aligned}
$$

$$
\begin{aligned}
&(K^\star : \star \overset{\ell}{\Rightarrow} \forall X.\, \forall Y.\, X \to Y \to Y)\ \texttt{int}\ \texttt{int}\ 2\ 3 \\
\longrightarrow^*\ &(K^\star : \star \overset{\ell}{\Rightarrow} \texttt{int} \to \texttt{int} \to \texttt{int})\ 2\ 3 \\
\longrightarrow^*\ &2
\end{aligned}
$$

Explicit Binding

$$(\Lambda X.\, v)\;\; T \longrightarrow \nu X \mapsto T.\, v \qquad \text{(TyBeta)}$$

Values pass through the ν binder:

$$\nu X \mapsto T.\, (n) \longrightarrow n \qquad \text{(NuInt)}$$

$$\nu X \mapsto T_1.\, (\lambda y{:}T_2.\, e) \longrightarrow \lambda y{:}[X \mapsto T_1]T_2.\, (\nu X \mapsto T_1.\, e) \qquad \text{(NuAbs)}$$

$$\nu X \mapsto T.\, (\Lambda Y.\, v) \longrightarrow \Lambda Y.\, (\nu X \mapsto T.\, v) \qquad \text{(NuTyAbs)}$$

$$\text{if } Y \neq X \text{ and } Y \notin \mathrm{ftv}(T)$$

$$\nu X \mapsto A.\, (v : G \Rightarrow \star) \longrightarrow (\nu X \mapsto A.\, v) : G \Rightarrow \star \qquad \text{(NuDyn)}$$

$$\text{if } G \neq X$$

$$\nu X \mapsto A.\, (v : X \Rightarrow \star) \longrightarrow \texttt{blame}\, p_\nu \qquad \text{(NuErr)}$$

Properties of the Polymorphic Blame Calculus

- ✓ Type Safety
- ✓ Blame Theorem (weak subtyping)

$$\frac{[X \mapsto \star]T_1 <: T_2}{(\forall X.\, T_1) <: T_2}$$

- □ Blame Theorem (strong subtyping)

$$\frac{[X \mapsto T]T_1 <: T_2}{(\forall X.\, T_1) <: T_2}$$

  (Incorrect proof in POPL 2011.)

- □ Parametricity

Conclusion

- We have just scratched the surface of the recent work. See Sam Tobin-Hochstadt's online bibliography.
- The "typing" part of gradual typing is relatively easy.
- The runtime behavior has been much more challenging.
- Is it possible for a gradually typed language to be efficient?
- Have we got the blame tracking right?
- How does gradual typing interact with other features such as:
  - recursive types
  - type operators
  - dependent types (some partial answers here)