Gradual Typing Roundup

6 Gradually-Typed Programs

Jeremy G. Siek

University of Colorado at Boulder

Dagstuhl, January 2012

Gradual Typing Basics[1]

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

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

[1] Gradual Typing for Functional Languages, Siek and Taha, 2006.

Translation to the Blame Calculus

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

def distance(x: ⋆, y: ⋆)→ ⋆:
  return abs((x : ⋆ ⇒ Int) - (y : ⋆ ⇒ Int)) : Int ⇒ ⋆
```

Gradual Typing Goes Higher Order[2]

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

def fun(y):
  if y > 0:
    return y ** 3 - y - 1
  else:
    return "yikes"

deriv(0.01, fun, 3.0)
deriv(0.01, fun, -3.0)
```

[2] Contracts for higher-order functions, Findler and Felleisen, 2002

Gradual Typing Isn't Space Efficient[3]

```
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)
```

[3]Space-Efficient Gradual Typing. Herman, et al., 2006

Gradual Typing Becomes Space Efficient[4,5]

$$
\begin{aligned}
& v : A \stackrel{P}{\Longrightarrow} B \stackrel{Q}{\Longrightarrow} C \\
\longrightarrow\ & v : A \stackrel{P \circ Q}{\Longrightarrow} C
\end{aligned}
$$

$$
\begin{aligned}
& (v_1 : A_1 \to A_2 \stackrel{P \to Q}{\Longrightarrow} B_1 \to B_2)\ v_2 \\
\longrightarrow\ & v_1\,(v_2 : B_1 \stackrel{P}{\Longrightarrow} A_1) : A_2 \stackrel{Q}{\Longrightarrow} B_2
\end{aligned}
$$

[4] Space-Efficient Gradual Typing. Herman, et al. 2006

[5] Threesomes, with and without blame. Siek, Wadler 2010.

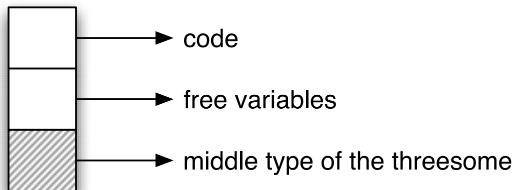
But There is Still Overhead[6]

$$v ::= \ldots \mid \lambda x{:}A.\, e \mid v : A_1 \to A_2 \stackrel{P \to Q}{\Longrightarrow} B_1 \to B_2$$

```
def inc(x: Int)→Int:
  return x + 1

def dec(x):
  return x - 1

def apply(f: Int→Int, y: Int)→Int:
  return f(y)

apply(inc, 41)
apply(dec, 43)
```

[6] Integrating typed and untyped code. Wrigstad et al. 2010

Solution Idea



But What About Arrays?[7]

```
def f(v: Array(Int))→Int:
  return v[0]

x = 3
y = "hi"
a = [x,y]
f(a)
```

[7] Integrating typed and untyped code. Wrigstad et al. 2010

A New Weapon on the JVM: invokedynamic

- A new bytecode for method invocation
- Provides inline caching for method dispatch
- Programmer controls:
  - method choice
  - cache invalidation via switchpoints
- JIT's specialize the caller, reducing overhead to zero!
- Not just for methods, see Charles Nutter's work on JRuby, such as caching global variables.
- Idea: use invokedynamic for statically typed array accesses, with unboxed access as the default version and boxed access as the fallback.

Strong Updates

```
def f(o: { m: Int→Int })→ Int:
  g(o)
  return o.m(42)

def g(o):
  del o.m
```

Type Safety for Gradual Typing

Theorem (The Blame Theorem[8])

If $A <: B$ *and* $\ell \notin \mathit{labels}(C) \cup \mathit{labels}(e)$*, then* $C[e : A \Rightarrow^{\ell} B] \not\longmapsto^{*} \mathsf{blame}\,\ell$*.*

We say expression e is *safe* if every implicit cast in e respects subtyping.

Corollary (Shallow Blame Safety)

If e *is safe and* $\mathit{labels}(e) \cap \mathit{labels}(C) = \emptyset$*, then* $C[e] \not\longmapsto \mathsf{blame}\,\ell$ *where* $\ell \in \mathit{labels}(e)$*.*

There are several choices of blame tracking strategy and corresponding definitions of subtyping[9].

[8] Well-typed programs can't be blamed. Wadler, Findler, 2009.
[9] Exploring the Design Space of Casts. Siek, Garcia, Taha, 2009.

Deep Blame Safety and Effects

Effects (capabilities) are sets of blame labels

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 : S \rightarrow T, \varphi_1 & \Gamma \vdash e_2 : S', \varphi_2 \\ S \sim S' & S' \not<: S\end{array}}{\Gamma \vdash (e_1 e_2)^{\ell} : T, \varphi_1 \cup \varphi_2 \cup \{\ell\}}$$

Conjecture (Deep Blame Safety)

If $\vdash e : S, \varphi$*, then either*

1. $E[e] \longmapsto^* E[\mathrm{blame}^{\ell}]$ *and* $\ell \in \varphi$*, or*
2. $E[e] \longmapsto^* E[v]$*, or*
3. $E[e] \Uparrow$.

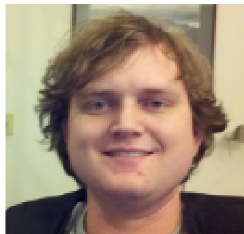
Reflection as a Capability

- As in eval, stack inspection, attributes by string, strong updates, etc.
- Reflection normally comes with a heavy cost, whether or not a program uses it.
- For example, in the Jython implementation of Python, the support for stack introspection dominates the run-time of some microbenchmarks.
- We've added a "frameless" annotation for functions in Jython to turn off support for stack inspection. We get a $3\times$ performance improvement on Fibonnaci.
- We'd like to generalize to other forms of reflection.

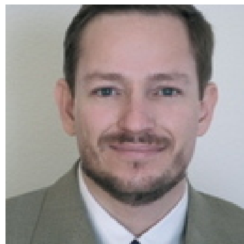
Feature Interaction, Gradual Typing and X

- Subtyping (structural and nominal)
  - Gradual Typing for Objects, ECOOP 2007, Siek, Taha.
  - BabyJ, WOOD 2003, Anderson and Drossopoulou.
  - Integrating typed and untyped code. POPL 2010. Wrigstad et al.
- Parametric Polymorphism:
  - Blame for All, POPL 2011, Ahmed et al.
  - Gradual Typing for Generics, OOPSLA 2011, Ina and Igarashi.
- Refinement types, dependent types
  - Sage, Scheme 2006, Gronski et al.
  - Contracts made manifest, POPL 2010, Greenberg et al.
  - Correct blame for contracts, POPL 2011, Dimoulas et al
- Typestate (ECOOP 2011, Roger et al.)
- Information flow (STOP 2011, Disney and Flanagan)
- Ownership types (IWACO 2011, Sergey and Clark)

The Gradual Python Team


Michael Vitousek

Shashank Bharadwaj


Jim Baker