Towards Gradual Typing in Python

Michael M. Vitousek    Shashank Bharadwaj    Jeremy G. Siek

{michael.vitousek, shashank.bharadwaj, jeremy.siek}@colorado.edu

University of Colorado at Boulder
Boulder, Colorado, USA

June 11, 2012
Introduction

- Gradual typing in Python
  - Compile-time error detection
  - Blame tracking
Introduction

- Gradual typing in Python
  - Compile-time error detection
  - Blame tracking
- ... and in Jython
  - Bytecode type specialization
Introduction

- Gradual typing in Python
  - Compile-time error detection
  - Blame tracking
- ... and in Jython
  - Bytecode type specialization
- Challenges
Introduction

- Gradual typing in Python
  - Compile-time error detection
  - Blame tracking
- ... and in Jython
  - Bytecode type specialization
- Challenges
  - Making statically typed code run fast
Introduction

- Gradual typing in Python
  - Compile-time error detection
  - Blame tracking
- ... and in Jython
  - Bytecode type specialization
- Challenges
  - Making statically typed code run fast
  - Prevent dynamic code from infecting static code
Introduction

■ Gradual typing in Python
  ■ Compile-time error detection
  ■ Blame tracking
■ . . . and in Jython
  ■ Bytecode type specialization
■ Challenges
  ■ Making statically typed code run fast
  ■ Prevent dynamic code from infecting static code
  ■ Minimizing overhead of going from static to dynamic and vice versa
Outline

1 Introduction

2 Function casts
   - Motivation
   - Our approach

3 Object casts
   - Motivation
   - Monotonic objects
   - Implications

4 Status and conclusions
   - Status of Gradual Jython
   - Conclusions
Function casts: motivation and example

1:  def explore_files(files, fun):
2:    for file in files:
3:      if file.is_directory():
4:        explore_dir(file, fun)
5:      else:  print  fun(file)
6:  def explore_dir(dir:file, fun:file → str) → unit:
7:    explore_files(file.members(), fun)
Function casts: motivation and example

1: `def` `explore_files(files, fun):`
2: `for` `file` `in` `files:`
3: `if` `file.is_directory()`:  
5: `else:` `print` `fun(file)`
6: `def` `explore_dir(dir:file, fun:file → str) → unit:`
7: `explore_files(file.members(): list ⇒ ?, fun: file → str ⇒ ?)`

- Standard gradual typing approach: inserted casts  
  moderate between static and dynamic code
  - Simple for basic types (`int, float`)  
  - Harder for functions
Function casts induce overhead

- Previous approaches:

  - Casts create new wrapper functions around casted functions, or casts attach to functions and are used at call sites.
  - Coercion calculus, threesomes.
  - Both approaches have problems.

  - Installing wrappers at every cast site is space-inefficient.
  - Attached casts result in complex output from compiler.

  - We would expect to generate code like:

    ```
    J1(e2)K = let f = J1K in f.(f.FVs, J2K)
    ```

  - But instead we have to generate:

    ```
    J1(e2)K = let f = J1K in case f of |
              Casted f'K ⇒ f'(J2K:dom(K)) |
              Function f' ⇒ f'.fun(f'.FVs, J2K)
    ```
Function casts induce overhead

- Previous approaches:
  - Casts create new wrapper functions around casted functions,
Function casts induce overhead

- Previous approaches:
  - Casts create new wrapper functions around casted functions,
  - or casts attach to functions and are used at call sites
Function casts induce overhead

### Previous approaches:
- Casts create new wrapper functions around casted functions,
- or casts attach to functions and are used at call sites
  - Coercion calculus, threesomes
Function casts induce overhead

- Previous approaches:
  - Casts create new wrapper functions around casted functions,
  - or casts attach to functions and are used at call sites
    - Coercion calculus, threesomes
- Both approaches have problems
Function casts induce overhead

- Previous approaches:
  - Casts create new wrapper functions around casted functions,
  - or casts attach to functions and are used at call sites
    - Coercion calculus, threesomes
- Both approaches have problems
  - Installing wrappers at every cast site is space-inefficient
Function casts induce overhead

- Previous approaches:
  - Casts create new wrapper functions around casted functions,
  - or casts attach to functions and are used at call sites
    - Coercion calculus, threesomes
- Both approaches have problems
  - Installing wrappers at every cast site is space-inefficient
  - Attached casts result in complex output from compiler
Function casts induce overhead

- Previous approaches:
  - Casts create new wrapper functions around casted functions,
  - or casts attach to functions and are used at call sites
    - Coercion calculus, threesomes

- Both approaches have problems
  - Installing wrappers at every cast site is space-inefficient
  - Attached casts result in complex output from compiler
    - We would expect to generate code like:
      \[
      \text{let } f = \text{let } e_1 \text{ in } f \text{.fun}(f \text{.FVs}, e_2) \]
      \[\text{[e_1(e_2)] = let } f = \text{[e_1]} \text{ in f.fun}(f \text{.FVs, [e_2]})\]
Function casts induce overhead

- Previous approaches:
  - Casts create new wrapper functions around casted functions,
  - or casts attach to functions and are used at call sites
    - Coercion calculus, threesomes

- Both approaches have problems
  - Installing wrappers at every cast site is space-inefficient
  - Attached casts result in complex output from compiler
    - We would expect to generate code like:

\[
[e_1(e_2)] = \text{let } f = [e_1] \text{ in } f\text{.fun}(f\text{.FVs}, [e_2])
\]

- but instead we have to generate:

\[
[e_1(e_2)] = \\
\text{let } f = [e_1] \text{ in } \\
\text{case } f \text{ of } \\
| \text{Casted } f' \mathcal{K} \Rightarrow f'([e_2] : \text{dom}(\mathcal{K})) : \text{cod}(\mathcal{K}) \\
| \text{Function } f' \Rightarrow f'\text{.fun}(f'\text{.FVs}, [e_2])
\]
Our approach

- Function closures always contain a pointer to a first-class threesome
Our approach

- Function closures always contain a pointer to a first-class threesome
  - Null if the function is not casted
Our approach

- Function closures always contain a pointer to a first-class threesome
  - Null if the function is not casted

\[ v ::= \ldots \mid \langle \text{fun} = \lambda(x \ c).e, \text{FVs} = \rho, \text{cast} = T_1 \xrightarrow{T_2} T_3 \rangle \]
Our approach

- Function closures always contain a pointer to a first-class threesome
  - Null if the function is not casted

\[
v ::= \ldots | \langle \text{fun} = \lambda(x \ c).e, \text{FVs} = \rho, \text{cast} = T_1 \xrightarrow{T_2} T_3 \rangle
\]

- At function call sites, generated code is simple

\[
[e_1(e_2)] = \text{let } f = [e_1] \text{ in } f.\text{fun}([e_2], f)
\]
Our approach

- Function closures always contain a pointer to a first-class threesome
  - Null if the function is not casted
    \[ v ::= \ldots \mid \langle \text{fun} = \lambda(x)\,c).e, \text{FVs} = \rho, \text{cast} = T_1 \xrightarrow{T_2} T_3 \rangle \]

- At function call sites, generated code is simple
  \[ \llbracket e_1(e_2) \rrbracket = \text{let } f = \llbracket e_1 \rrbracket \text{ in } f.\text{fun}(\llbracket e_2 \rrbracket, f) \]

- Pass in entire closure instead of just the FVs
Our approach

- Function closures always contain a pointer to a first-class threesome
  - Null if the function is not casted
  
  \[ v ::= \ldots | \langle \text{fun} = \lambda(x \ c).e, \text{FVs} = \rho, \text{cast} = T_1 \xrightarrow{T_2} T_3 \rangle \]

- At function call sites, generated code is simple

\[
\llbracket e_1(e_2)\rrbracket = \text{let } f = \llbracket e_1 \rrbracket \text{ in } f.\text{fun}(\llbracket e_2 \rrbracket, f)
\]

- Pass in entire closure instead of just the FVs
- Uncasted functions simply extract the FVs from the closure, and proceed normally — very little overhead
Initial casts on bare functions install a *generic* wrapper around code
Our approach

- Initial casts on bare functions install a *generic* wrapper around code
  - Wrapper is parametrized over the cast to apply
Our approach

- Initial casts on bare functions install a *generic* wrapper around code
  - Wrapper is parametrized over the cast to apply

\[
f : T_1 \xrightarrow{T_2} T_3 \rightarrow \langle \text{fun} = \lambda(x \in c). (f(x: \text{dom}(c.\text{cast}))) : \text{cod}(c.\text{cast}), \text{FVs} = \rho, \text{cast} = T_1 \xrightarrow{T_2} T_3 \rangle
\]
Initial casts on bare functions install a *generic* wrapper around code

- Wrapper is parametrized over the cast to apply

\[
  f : T_1 \overset{T_2}{\rightarrow} T_3 \rightarrow \langle \text{fun} = \lambda(x\ c).(f(x:\text{dom}(c\.\text{cast}))):\text{cod}(c\.\text{cast}), \\
  \text{FVs} = \rho, \text{cast} = T_1 \overset{T_2}{\rightarrow} T_3 \rangle
\]

- Additional casts only update the threesome
Our approach

- Initial casts on bare functions install a generic wrapper around code
  - Wrapper is parametrized over the cast to apply

\[
f : T_1 \xrightarrow{T_2} T_3 \rightarrow \langle \text{fun} = \lambda(x \cdot c). (f(x: \text{dom}(c.\text{cast}))) : \text{cod}(c.\text{cast}), \text{FVs} = \rho, \text{cast} = T_1 \xrightarrow{T_2} T_3 \rangle
\]

- Additional casts only update the threesome

- At call site, wrapper around casted functions will extract the closure’s threesome and apply it
Outline

1. Introduction

2. Function casts
   - Motivation
   - Our approach

3. Object casts
   - Motivation
   - Monotonic objects
   - Implications

4. Status and conclusions
   - Status of Gradual Jython
   - Conclusions
Casts create invalid assumptions

1. `obj: dyn = \{x = 10, y = True\}`  # Object initialization
2. `def get_ref(obj: {x: int, y: dyn}) \to (unit \to int):`
3. `return \lambda: unit. obj.x`  # Capture typed reference
4. `x_ref: (unit \to int) = get_ref(obj)`
5. `obj.x = "Hello!"`
6. `print (x_ref() + 10)`
Casts create invalid assumptions

1: \texttt{obj:dyn = \{x = 10, y = True\}} \# Object initialization
2: \texttt{def get\_ref(obj:\{x:int, y:dyn\}) \rightarrow (unit \rightarrow int):}
3: \hspace{1em} \texttt{return \lambda:unit. obj.x} \# Capture typed reference
4: \texttt{x\_ref:(unit \rightarrow int) = get\_ref(obj)}
5: \texttt{obj.x = “Hello!”}
6: \texttt{print (x\_ref()+10)}

We want to detect the type error,
Casts create invalid assumptions

1: \textit{obj:} \texttt{dyn = \{x = 10, y = True\}} \#Object initialization
2: \textbf{def} \texttt{get\_ref(obj:{x:int, y:dyn}) \rightarrow (unit \rightarrow int):}
3: \textbf{return} \lambda_: \texttt{unit. obj.x} \#Capture typed reference
4: \texttt{x\_ref:(unit \rightarrow int) = get\_ref(obj)}
5: \texttt{obj.x = “Hello!”}
6: \textbf{print} (x\_ref() + 10)

We want to detect the type error, to allow for efficient member accesses,
Casts create invalid assumptions

1: \texttt{obj:dy}n = \{x = 10, y = \text{True}\} \ #Object initialization
2: \texttt{def get\_ref(obj:{x:int, y:dy}) \rightarrow (unit \rightarrow int):}
3: \texttt{\quad return \lambda :unit. \ obj.x} \ #Capture typed reference
4: \texttt{x\_ref:(unit \rightarrow int) = get\_ref(obj)}
5: \texttt{obj.x = “Hello!”}
6: \texttt{print (x\_ref()+10)}

We want to detect the type error, to allow for efficient member accesses, and to have the ability to blame the responsible site in code.
Object casts

- Need a solution to object casting that supports these objectives
Object casts

- Need a solution to object casting that supports these objectives
- Straightforward approaches are slow and incompatible with the semantics of imperative languages
Object casts

- Need a solution to object casting that supports these objectives
- Straightforward approaches are slow and incompatible with the semantics of imperative languages
- Existence of strong updates prevents the approach used in function casts from extending to objects
Object casts

- Need a solution to object casting that supports these objectives
- Straightforward approaches are slow and incompatible with the semantics of imperative languages
- Existence of strong updates prevents the approach used in function casts from extending to objects
- Same principles apply for mutable reference cells (but Python doesn’t have them)
An approach: monotonic objects

- Strong updates to objects resolve to trapped errors if they invalidate any view of the object
An approach: monotonic objects

- Strong updates to objects resolve to trapped errors if they invalidate any view of the object
- Monotonic objects
An approach: monotonic objects

- Strong updates to objects resolve to trapped errors if they invalidate any view of the object
- Monotonic objects
  - Objects internally maintain the *meet* of the types that have been statically specified for each member
An approach: monotonic objects

- Strong updates to objects resolve to trapped errors if they invalidate any view of the object
- Monotonic objects
  - Objects internally maintain the *meet* of the types that have been statically specified for each member
  - When an object is cast,
    - the stored meet of each member is updated (if necessary) to reflect the new type,
An approach: monotonic objects

- Strong updates to objects resolve to trapped errors if they invalidate any view of the object.
- Monotonic objects
  - Objects internally maintain the *meet* of the types that have been statically specified for each member.
  - When an object is cast,
    - the stored meet of each member is updated (if necessary) to reflect the new type,
    - and the value of each member is cast to the new meet type, or left alone if the meet has not changed.
An approach: monotonic objects

- Strong updates to objects resolve to trapped errors if they invalidate any view of the object

- Monotonic objects
  - Objects internally maintain the *meet* of the types that have been statically specified for each member
  - When an object is cast,
    - the stored meet of each member is updated (if necessary) to reflect the new type,
    - and the value of each member is cast to the new meet type, or left alone if the meet has not changed.
    - If there is no such meet type, a cast error occurs.
  - When a field update occurs, the new value is cast to the object’s meet type for that member.
    - If this cast fails, we have a trapped error.
Casts mutate object structure

1: \( \text{obj:} \text{dyn} = \{ x = 10, y = \text{True} \} \) \#Object initialization
2: \textbf{def get\_ref}(\text{obj:} \{ x: \text{int}, y: \text{dyn} \}) \rightarrow (\text{unit} \rightarrow \text{int}): \\
3: \textbf{return} \ \lambda : \text{unit}. \ \text{obj.x} \ \#\text{Capture typed reference} \\
4: \ x\_\text{ref}.(\text{unit} \rightarrow \text{int}) = \text{get\_ref}(\text{obj}) \\
5: \ \text{obj.x} = \text{“Hello!”} \\
6: \ \textbf{print} \ (x\_\text{ref}() + 10) \\

\textit{obj} initially has dynamically-typed members
Casts mutate object structure

1: \( \textit{obj}:\text{dyn} = \{x = 10, y = \text{True}\} \) \#Object initialization
2: \textbf{def} \textit{get\_ref}(\textit{obj}:{\{x:int, y:dyn}\}) \rightarrow (\text{unit} \rightarrow \text{int}): \\
3: \textbf{return} \ \lambda:\text{unit}. \textit{obj}.x \ \#Capture typed reference
4: \textit{x\_ref}: (\text{unit} \rightarrow \text{int}) = \textit{get\_ref}(\textit{obj})
5: \textit{obj}.x = “Hello!”
6: \textbf{print} \ (\textit{x\_ref}() + 10)

After it passes through a cast, its types are updated to their meets
Casts mutate object structure

1: \texttt{obj: dyn = \{x = 10, y = True\} \# Object initialization}
2: \texttt{def get\_ref(obj:\{x:int, y: dyn\}) \rightarrow (unit \rightarrow \text{int}):
3: \quad \texttt{return } \lambda\_::\text{unit. obj.x} \quad \# \text{Capture typed reference}
4: \texttt{x\_ref:(unit \rightarrow \text{int}) = get\_ref(obj)}
5: \texttt{obj.x = “Hello!”}
6: \texttt{print (x\_ref() + 10)}

\text{str} \sqcap \text{int} = \bot \quad \text{Attempted update to } x \text{ fails, blames update code}
Static reads are fast

1: obj:dyn = \{x = 10, y = True\}  # Object initialization
2: def get_ref(obj:{x:int, y:dyn}) → (unit → int):
3:     return \lambda:unit. obj.x  # Capture typed reference
4: x_ref:(unit → int) = get_ref(obj)
5: print (x_ref() + 10)

Reads of statically typed properties can directly access the object’s member values, bypassing the dictionary, using permutation vectors:

\[ obj \rightarrow \text{mems}[obj.\text{perm}(0)] \]
Implications

- Fully static references to objects allow direct access to fields
Implications

- Fully static references to objects allow direct access to fields
  - dynamically-typed references may need to be boxed
Implications

- Fully static references to objects allow direct access to fields
  - dynamically-typed references may need to be boxed
- Member updates need casts, but accesses are fast
Implications

- Fully static references to objects allow direct access to fields
  - dynamically-typed references may need to be boxed
- Member updates need casts, but accesses are fast
- Flow-sensitive
Implications

- Fully static references to objects allow direct access to fields
  - dynamically-typed references may need to be boxed
- Member updates need casts, but accesses are fast
- Flow-sensitive
- Restrictive
Implications

- Fully static references to objects allow direct access to fields
  - dynamically-typed references may need to be boxed
- Member updates need casts, but accesses are fast
- Flow-sensitive
- Restrictive
  - But avoids reference counting or dependence on GC
Implications

- Fully static references to objects allow direct access to fields
  - dynamically-typed references may need to be boxed
- Member updates need casts, but accesses are fast
- Flow-sensitive
- Restrictive
  - But avoids reference counting or dependence on GC
- Alternative: check member types at access sites
Implications

- Fully static references to objects allow direct access to fields
  - dynamically-typed references may need to be boxed
- Member updates need casts, but accesses are fast
- Flow-sensitive
- Restrictive
  - But avoids reference counting or dependence on GC
- Alternative: check member types at access sites
  - Probably greater overhead, but maybe can be optimized
Outline

1. Introduction
2. Function casts
   - Motivation
   - Our approach
3. Object casts
   - Motivation
   - Monotonic objects
   - Implications
4. Status and conclusions
   - Status of Gradual Jython
   - Conclusions
Gradual Jython

- Gradual Jython is a WIP
Gradual Jython

- Gradual Jython is a WIP
  - Static typechecking
Gradual Jython

- Gradual Jython is a WIP
  - Static typechecking
  - Type specialization for primitive types
Gradual Jython

- Gradual Jython is a WIP
  - Static typechecking
  - Type specialization for primitive types
  - Shashank: Optimized function casts using MethodHandles
Gradual Jython

- Gradual Jython is a WIP
  - Static typechecking
  - Type specialization for primitive types
  - Shashank: Optimized function casts using MethodHandles
- To be integrated (as an option) into an upcoming version of Jython
Gradual Jython

- Gradual Jython is a WIP
  - Static typechecking
  - Type specialization for primitive types
  - Shashank: Optimized function casts using MethodHandles
- To be integrated (as an option) into an upcoming version of Jython
- Some interest in releasing the static typechecker as a standalone app
Gradual Jython

- Gradual Jython is a WIP
  - Static typechecking
  - Type specialization for primitive types
  - Shashank: Optimized function casts using MethodHandles

- To be integrated (as an option) into an upcoming version of Jython

- Some interest in releasing the static typechecker as a standalone app

- Additional work on Gradual Jython done by
  - Jim Baker (Canonical)
  - Chris Poulton (University of Colorado at Boulder)
Conclusions

- Statically typed code should be as fast as possible
Conclusions

- Statically typed code should be as fast as possible
- Casts from dynamic to static will happen a lot, so we need to make them work well:
Conclusions

- Statically typed code should be as fast as possible
- Casts from dynamic to static will happen a lot, so we need to make them work well:
  - Minimize overhead of casts
Conclusions

- Statically typed code should be as fast as possible
- Casts from dynamic to static will happen a lot, so we need to make them work well:
  - Minimize overhead of casts
  - Minimize overhead of using casted values (function calls, member access)
Conclusions

- Statically typed code should be as fast as possible
- Casts from dynamic to static will happen a lot, so we need to make them work well:
  - Minimize overhead of casts
  - Minimize overhead of using casted values (function calls, member access)
  - Provide useful information when things go wrong
Conclusions

- Statically typed code should be as fast as possible
- Casts from dynamic to static will happen a lot, so we need to make them work well:
  - Minimize overhead of casts
  - Minimize overhead of using casted values (function calls, member access)
  - Provide useful information when things go wrong
- Gradual function casts and monotonic objects help us achieve these goals
Conclusions

- Statically typed code should be as fast as possible
- Casts from dynamic to static will happen a lot, so we need to make them work well:
  - Minimize overhead of casts
  - Minimize overhead of using casted values (function calls, member access)
  - Provide useful information when things go wrong
- Gradual function casts and monotonic objects help us achieve these goals
- May be other worthwhile approaches, especially to object casts
Conclusions

- Statically typed code should be as fast as possible
- Casts from dynamic to static will happen a lot, so we need to make them work well:
  - Minimize overhead of casts
  - Minimize overhead of using casted values (function calls, member access)
  - Provide useful information when things go wrong
- Gradual function casts and monotonic objects help us achieve these goals
- May be other worthwhile approaches, especially to object casts
- Figuring out these issues is critical to adding robust gradual typing to Python — and we’re well on our way!