

# Towards Gradual Typing in Jython

Michael M. Vitousek    Shashank Bharadwaj    Jeremy G. Siek

University of Colorado at Boulder

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

## 1. Introduction

The Jython implementation of Python for the JVM [1] stands to benefit greatly from the introduction of gradual typing. In particular, it may lead to improved program efficiency, static detection of type errors, and improved modularity of runtime error detection through blame tracking [2]. However, there are tensions between these goals. For example, the addition of type annotations to a program often causes the compiler to insert casts to mediate between static and dynamically typed code and these casts incur runtime overhead [3, 7].

Researchers have demonstrated space efficient blame tracking for casts involving first-class functions but the technique required a second value-form at function type, a casted function [4, 5]. The consequence for compilation is that generated code for function application must perform a dispatch and thereby incurs overhead, even for function applications residing in statically typed code. At Dagstuhl in January, Siek suggested a solution to this problem by storing the “threesome” as part of the closure representation and by moving the work of casting from the caller to the callee. In Section 2 we elaborate on this solution.

After functions, the next major challenge for efficient gradual typing in Python is how to efficiently implement casts involving objects. Objects are problematic because Python, as an imperative and dynamically typed language, allows *strong updates*, that is, changes to objects that affect their type. This problem has appeared in other guises, such as covariant arrays in Java and changes in tystate [6]. The approaches to date either guard every read access by a runtime type test or require reference counting. Both approaches impose significant runtime overhead, even for statically typed code. In Section 3 we propose an alternative in which we only guard strong updates and only allow the type of an object to change in a monotonically decreasing fashion with respect to naïve subtyping.

## 2. Function casts

We begin by considering the implementation of function casts. The naïve approach would be to have every cast on a function create a new function that “wraps” up the function, but this method is not scalable to situations where a function is repeatedly passed from static code to dynamic code and vice versa. Consider the following partially-typed Gradual Jython code, which traverses a file directory tree, applies a function *fun* to each file in the structure, and prints its output:

```
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)
```

Because a new wrapper is added around *fun* every time it passed through from one function to another, there is a  $O(n)$  space blowup, rendering this approach infeasible.

Siek and Wadler developed a partial solution to this issue by attaching a “threesome” to a casted function and by merging threesomes when a casted function is cast yet again [4]. While this eliminates the  $O(n)$  blowup of function wrapping, it also increases the overhead of function calls, because the compiled call site needs to check if the value being called is a threesome-wrapped function or a bare function. As such, the rule for compiling functions becomes

$$\begin{aligned} \llbracket e_1(e_2) \rrbracket &= \\ \text{let } f &= \llbracket e_1 \rrbracket \text{ in} \\ \text{case } f &\text{ of} \\ &| \text{Casted } f' \mathcal{K} \Rightarrow f'(\llbracket e_2 \rrbracket : \text{dom}(\mathcal{K})) : \text{cod}(\mathcal{K}) \\ &| \text{Function } f' \Rightarrow f'.fun(f'.fvs, \llbracket e_2 \rrbracket) \end{aligned}$$

We see a path to solving this problem by using a combination of the naïve wrapping approach and the threesomes approach. In this formulation, function closures contain pointers to a stored threesome, in addition to the typical function code and values for free variables. In a closure that has not been cast, the threesome is an identity cast. Applying a cast to a closure for the first time installs a generic wrapper function that performs casting on the argument, calls the original function, and casts the return value. Additional casts applied to the closure simply alter the threesome.

With this modification to function closures, call sites are returned to their simple form, with the exception that we pass the entire closure into the function instead of just its free variables:

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

The wrapper installed onto casted function bodies accesses the closure’s threesome to cast the function from its original to its final type. Uncasted functions, lacking wrappers, simply ignore the threesome. To enable this treatment of threesomes as data, we depart from previous work and make threesomes into first-class entities.

We believe this approach to function casts will suffice to eliminate overhead at the call site of uncasted functions while continuing to support blame tracking to produce useful error messages.

## 3. Object casts

Our implementation of function casts relies on the fact that Python functions are immutable — a property that does not hold for Python objects. This complicates the design of gradual object casts in Jython, requiring a different approach from that used for functions.<sup>1</sup>

### 3.1 Motivation

The complicating effects of imperative update on gradual typing are reflected in the following code. This program constructs an object

<sup>1</sup> Python does not have mutable reference cells, but a language with *refs* would have to consider similar issues.

obj including a member  $x$  with value 10 and calls `get_ref` which returns a function that is holding a reference to `obj`. The reference to `obj` relies on member  $x$  having type `int`. Upon returning from `get_ref`, the program mutates `obj.x` to be a string value and then calls `x.ref`, which tries to reference member  $x$  as an integer.

```

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

```

This program should fail because `obj.x` is updated to contain a string but has been casted to `int`. However, we would like to detect this error without incurring too much overhead on member accesses, and while still being able to blame the code ultimately responsible for violating the constraints on the types. This choice has ramifications for the internal representation of Jython objects.

### 3.2 Approaches to object casts

We review the traditional approach that relies on access checking and then present our new approach to object casts.

#### 3.2.1 Access checking

One solution to this problem is to use additional inserted casts to check that object members conform to their expected types when accessed. This approach would require a cast or runtime check to be inserted at the access `obj.x` at line 3 above. In this case, `obj.x` would be cast to `int` — and in this particular program, the cast would fail, since at the time that  $x$  is finally dereferenced (at line 6) its value is a `str`.

This approach maintains the “Pythonic” flexibility of object updates even when typed variables view objects. On the other hand, casting members at their access sites adds overhead, and free field mutation may make it difficult to use an object representation conducive to fast field access. While we believe we can use JVM techniques such as `InvokeDynamic` or `MethodHandles` to minimize this overhead, we have the additional problem that the code point blamed is the access site (line 3), not the location of the update that invalidated the access’ assumptions (line 5).

#### 3.2.2 Monotonic objects

A second method for performing function casts involves permanently restricting the types of object members when casts are applied. This approach, which we call “monotonic objects”, requires that the object itself record the most specific type (the *meet* with respect to naïve subtyping [4]) that its members have been viewed through. Successful casts update the *meet* as needed. This system detects the error in the above example at the update — when `obj` is passed to `get_ref`, the  $x$  field of `obj` is forever after restricted to containing ints, and so the update at line 5 fails and is blamed.

This approach enables a representation for objects that enable fast field lookup from static code. Objects consist of a dictionary, required for dynamic accesses, and an array of values and their *meet* types, as shown in Figure 1. When another, differently-typed reference is made to an object, its *meet* types mutate.

Wolff et al. [6] offer an alternative approach in which strong updates are allowed so long as they respect the types of all current references to the object. Their approach requires a form of reference counting that induces significant runtime overhead and it makes the semantics of the program depend on the speed of the garbage collector. Our monotonic object approach provides a more efficient, but in some ways less flexible, alternative.

We plan to further investigate monotonic objects and access checking. The monotonic approach promises more efficient field

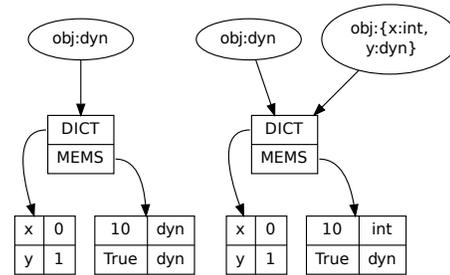


Figure 1. Representations of `obj` at line 1 and after line 4.

accesses, but the flow-sensitive restrictions it places on object values may be problematic in practice, and alternatively we may be able to reduce the runtime overhead of access checking.

## 4. Conclusions

Handling casts on nontrivial program data is a critical challenge for implementing gradual typing in a language like Jython. We have identified several of the problems that need to be solved to make gradual typing feasible in Jython — specifically, correct and efficient function and object casts — and have laid out our current strategies for confronting these challenges. More work is required to determine the best approaches, but our work thus far seems promising and we are confident that these challenges can be solved.

## References

- [1] The Jython Project. URL <http://jython.org>.
- [2] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming*, October 2002.
- [3] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.
- [4] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *POPL '10*, pages 365–376. ACM, 2010.
- [5] J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *ESOP '09*, pages 17–31. Springer, 2009.
- [6] R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual tpestate. In *Proceedings of the 25th European conference on Object-oriented programming, ECOOP'11*, pages 459–483, Berlin, Heidelberg, 2011. Springer-Verlag.
- [7] T. Wrigstad, F. Z. Nardelli, S. Lebesne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2010. ACM.