Gradual Typing with Efficient Object Casts

Extended abstract

Michael M. Vitousek [*]

Adviser: Jeremy G. Siek [†]
University of Colorado at Boulder
michael.vitousek@colorado.edu

1. Problem and motivation

Gradual type systems meld dynamic typing with optional static types, moderating between the two with statically inserted casts. Casting is the *éminence grise* of such systems — it enables swift detection of type errors in dynamic code without enforcing runtime checks throughout a program, and in combination with blame tracking it allows such errors to be traced to their origin. However, such casts accumulate as data is passed between static and dynamic portions of a program, and the naïve solution of fully applying casts whenever data flows through them — and therefore creating new, casted values — is highly inefficient in both space and time. Worse, this naïve solution makes a copy of the object, which is incompatible with the semantics of imperative object-oriented languages. This problem must be solved if gradual typing is to become practical for common dynamic languages.

2. Background and related work

Siek and Taha introduced gradual typing to functional languages [4] and extended it to object-oriented languages [5]. Another approach to mixed static/dynamic typing of imperative objects was proposed by Flanagan, Freund, and Tomb [2]. Gradual typing has been applied to the dynamic language ActionScript [3], and work is under way at the University of Colorado at Boulder to introduce it to Jython. It is also used in Typed Scheme [7].

Siek and Wadler's space-efficient "threesome" casts [6] allow multiple casts to be safely compressed into a single cast and allow a single wrapper function to handle any number of function casts, with each cast after the first costing only a constant space overhead. My work extends that result to object casts by using threesomes as first-class values.

3. Approach and uniqueness

The basic structure of objects with space-efficient casts, shown in Figure 1, is:

- a reference ℓ to a dictionary o containing the members of the object,
- a threesome $\mathcal{K}$ which tracks the casts applied to the object,
- and a delegation function d which has a label and a first-class threesome parameter and which is invoked whenever a member is accessed, taking the label of the requested member and the object's stored threesome as arguments.

The content of these elements depends on whether the object has been cast. If not, then o only contains the initial, non-wrapped members of the object; d will ignore its threesome parameter and simply return the member of o pointed to by its label parameter; and $\mathcal{K}$ will be an identity cast.

If a cast has occurred, however, then o will contain both the original members and generically wrapped

[*] Graduate student
ACM # 3541296

3120 Corona Trl, Apt 307
Boulder, CO, 80301, USA
[†] jeremy.siek@colorado.edu

objects	j	$::=$	$\langle \ell, d \rangle :: \mathcal{K}$
delegators	d	$::=$	$\lambda(label, cast).e$
heaps	σ	$::=$	$\{\overline{\ell := o}\}$
dictionaries	o	$::=$	$\{\overline{x = e}\}$
types	T	$::=$	$\ldots \mid \mathsf{dyn} \mid \{\overline{x{:}T}\}$
expressions	e	$::=$	$\ldots \mid j \mid \mathcal{K} \mid (\mathcal{K})e \mid e.x \mid$ $e.x = e$
threesomes	$\mathcal{K}$	$::=$	$T \overset{T}{\Rightarrow} T$

Figure 1. Syntax for gradual object casts

versions of them, associated with labels generated by a static hash function. When invoked, the delegator d will retrieve the wrapped version of whatever member is requested and apply its threesome parameter to the generic wrapper, resulting in the cast being efficiently applied before the wrapped value is returned. Finally, $\mathcal{K}$ will record both the original type of the object and the target type of the applied cast.

The delegate function and the stored cast become relevant when objects are used. The delegation function described above handles object accesses directly, so we now consider object casts and field updates.

The initial cast of an object installs the delegation function described above, creates generic wrappers around each member of the object and adds them to o, and stores the applied threesome. These operations clearly have a high space complexity. However, further casts only need to extend the stored threesome, which is much less expensive.

After a field update occurs, the object must still obey the constraints its previous casts impose. The stored threesome $\mathcal{K}$ records this information, so when a member is updated, a new cast is constructed from $\mathcal{K}$ by tracing the types it requires from the updated member, from the final type to the original — reversing the cast. This threesome is applied to the new field value, catching any type errors. For example,

```
$foo(obj : \{x{:}\mathsf{dyn}\})$ :
    $y{:}\mathsf{dyn}$ = "pure evil";
    $obj.x = y$

$bar(obj : \{x{:}\mathsf{int}\})$ :
    $foo(obj)$;
    $obj.x + 10$
```

This program will pass static typechecking because *obj* has been cast to dyn. However, when the update is performed at runtime,

$$obj = (\langle \ell, d \rangle :: \{x{:}\mathsf{int}\} \xRightarrow{\{x:\mathsf{int}\}} \{x{:}\mathsf{dyn}\})$$

We therefore cast y by the extracted threesome $\mathsf{dyn} \overset{\mathsf{int}}{\Rightarrow} \mathsf{int}$. This cast fails, and the dangerous update is prevented. If y were 42, the cast would succeed and the update would proceed.

This approach to object casts is related to our approach to function casts, which also involves creating a wrapper and maintaining a stored threesome which is applied at the call site. However, due to the mutability of objects, the use of delegation functions and cast inversion in field updates are required to preserve the normal semantics of objects and ensure their safety. Previous approaches to gradual typing with objects either did not handle imperative objects [5] or did not focus on space efficiency [2].

4. Results and contributions

I have presented an approach to gradually-typed object casts which supports common use cases of objects in dynamic languages. In particular, this approach

- preserves the semantics of object updates,
- requires minimal space overhead on object casts after the initial cast,
- and minimizes the overhead of accessing and updating non-casted objects.

In combination with our work on space efficient gradual function casts, and with techniques such as blame-tracking [1] and gradual type inference [3], this work increases the practicality and feasibility of integrating gradual typing into object-oriented dynamic languages such as ActionScript and Python and of designing new dynamic languages with gradual typing in mind.

References

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *POPL '11*, pages 201–214, New York, New York, USA, 2011. ACM.

[2] Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid Types, Invariants, and Refinements For Imperative Objects. In *FOOL/WOOD*, 2006.

[3] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *POPL '12*, pages 481–494. ACM, 2012.

[4] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop '06*, pages 81–92. ACM, 2006.

[5] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP '07*, pages 2–27. Springer, 2007.

[6] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *POPL '10*, pages 365–376. ACM, 2010.

[7] Sam Tobin-Hochstadt and Robert Bruce Findler. Cycles without pollution: a gradual typing poem. In *STOP '09*, pages 47–57. ACM, 2009.