Gradual Typing with Efficient Object Casts

Michael M. Vitousek
Department of Computer Science, University of Colorado at Boulder

Space-efficiency in gradually-typed objects

Gradual typing

Gradual type systems meld dynamic typing with optional static types, moderating between the two with **statically inserted casts**. Cast insertion 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. This problem **must be solved** if gradual typing is to become practical for common dynamic languages.

Background

Gradual typing was introduced to functional languages by Siek and Taha [2], who later extended it to objects [3]. An important step in making gradual typing practical for common languages is to support **imperative objects with update**. Eagerly applying casts to object members whenever a cast is encountered is **inefficient and incompatible with the semantics of imperative languages**. Instead we **lazily carry around casts**, and apply them only when needed. Siek and Wadler [4] developed a space-efficient design for such lazy casts on functions using *threesomes*, which collapse any number of delayed casts into two casts. However, this approach does not directly translate to gradual object casts due to the mutability of objects. This work presents a solution to **gradual casts in imperative, object-oriented languages**, in order to enable the use of gradual typing in languages like ActionScript and Python.

Syntax

objects	$j ::= \langle \ell, d \rangle :: \mathcal{K}$	gradually-castable objects
delegators	$d ::= \lambda(label, cast).e$	access delegators
heaps	$\sigma ::= \{\overline{\ell := v}\}$	mapping of addresses to values
values	$v ::= \ldots \mid \{\overline{x := \ell}\}$	values, including dictionaries
types	$T ::= \ldots \mid \textbf{dyn} \mid \{\overline{x{:}T}\}$	
expressions	$e ::= \ldots \mid j \mid \mathcal{K} \mid (\mathcal{K})e \mid e.x$	threesomes are first class
statements	$s ::= \ldots \mid e.x = e$	
threesomes	$\mathcal{K} ::= T \overset{T}{\Rightarrow} T$	

A solution: gradual object casts

Gradual object structure

The basic structure of object references with space-efficient casts is:

- an **address** ℓ pointing to a **dictionary** o in the heap,
- a **threesome** $\mathcal{K}$, which is a **first class value**,
- and a **delegation function** d, which takes a member name and a threesome as arguments.

Casts

Initial casts to previously uncasted objects cause structural changes to each part of the object:

- **generic wrappers** are installed onto each element of o,
- $\mathcal{K}$ is updated to record the cast's effect,
- and an **unwrapping and casting** function is installed in d.

These major structural changes only occur once — both the wrappers around the elements and the delegation function are **generic**, so on further casts, **only $\mathcal{K}$ needs to be updated**.

Member access and updates

Member accesses are **moderated by** d, and whenever a member is accessed, d is invoked, taking the label of the requested member and its corresponding sub-cast from $\mathcal{K}$ as arguments. If the object is uncasted, then d simply **returns the desired value**. If it has been casted, d will instead **apply its cast argument** to the value, unwrapping it and giving it the required type.

Member updates must **maintain the type** of the updated object. To ensure this, casts are applied to **values written to the object** as well as those read from it. Specifically, if a casted object has a $\mathcal{K}$ which records that the object's member x has been cast by $A \overset{B}{\Rightarrow} C$, then when a value is written to x, that value must be cast by $C \overset{B}{\Rightarrow} A$.

Semantics

Member access

$$\langle (\langle \ell, d \rangle :: \mathcal{K}).x, \sigma \rangle \longrightarrow \langle d(x,\ \mathcal{K}), \sigma \rangle$$

Member update

$$\langle (\langle \ell, d \rangle :: \mathcal{K}).x = e, \sigma \rangle \longrightarrow \langle \textbf{pass}, [\sigma(\ell)(x) := (\llbracket \mathcal{K}.x \rrbracket_\sigma)\llbracket e \rrbracket_\sigma]\sigma \rangle$$

Example and discussion

Member update

This approach protects against undetected runtime type errors, as could otherwise occur in the following example.

$$\begin{array}{l} foo(obj : \{x{:}\textbf{dyn}\}) : \\ \quad y{:}\textbf{dyn} = \text{“hello world”}; \\ \quad obj.x = y \end{array}$$

$$\begin{array}{l} bar(obj : \{x{:}\textbf{int}\}) : \\ \quad foo(obj); \\ \quad obj.x + 10 \end{array}$$

The type error in this example cannot be detected at runtime, but this approach to member updates ensures that it will be immediately **caught as a cast error** rather than a more dangerous or difficult to debug type error.

Conclusions

This approach to gradually-typed object casts

- **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 addition, we are investigating **other techniques** to achieve these results, such as monotonic objects. Either approach, in combination with techniques like **blame tracking** [4] and **gradual type inference** [1], will improve the practicality and utility of gradual typing in scripting languages like ActionScript and Python.

References

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

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

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

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

michael.vitousek@colorado.edu