Gradual Typing with Efficient Object Casts

Michael M. Vitousek
Adviser: Jeremy G. Siek *
University of Colorado Boulder
michael.vitousek@colorado.edu

1. Problem and motivation

Static typing and dynamic typing, the two primary typing disciplines that appear in the design of modern programming languages, have different advantages and disadvantages, and are best suited to different tasks. Static typing, in which the data type of every variable or identifier is known during compilation, lends itself to early error-detection and optimized compilation strategies. Dynamic typing, on the other hand, lacks the requirement that types be known "ahead of time," and supports rapid prototyping and program evolution. While these two approaches to typing would therefore appear to suit different classes of programs, in reality it is not uncommon for the ideal type system of a project to change over time. Initially, when project structure and requirements are in flux, a dynamic type system may be more appealing, while later on, the assurances of static typing are more beneficial. However, this switch is not possible in most current languages, which are either entirely static or entirely dynamically-typed. Instead, to achieve this goal, the entire program would have to be ported to a different language, which is usually infeasible.

Gradual typing confronts this problem by allowing static and dynamic typing to co-exist within the same program. Gradual type systems meld dynamic typing with optional static types, moderating between the two with statically inserted casts. These casts ensure that at runtime, values coming from dynamically-typed code correspond to their expected types when they flow into statically-typed code. Casting is therefore 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, the semantics of casts on object values presents a challenge: the naïve approach of applying casts to the object members whenever an object flows through a cast — therefore ensuring that the object's members are of the correct type only at the cast site — is insufficient to support the behavior of mutable objects, thanks to the possibility of there being multiple references to the same object, but with different types. The next obvious approach, of simply installing a wrapper around an object to check that its types are always correct, is highly inefficient in both space and time, because such proxies accumulate as values flow back-and-forth between static and dynamic portions of a program. In order to support a practical gradual typing system in real dynamic languages like JavaScript or Python, this problem must be addressed.

1.1 Examples

As an example to both demonstrate the use of gradual typing, and to motivate the issues that my approach confronts, consider the following example in a hypothetical gradually-typed Python variant:

```
1  class Person(object):
2      def __init__(self, id, name):
3          self.id = id
4          self.name = name
5
6  def insert_to_db(p : {id:int, name:str}) → void:
7      ... p.name ...
8
9  person1 = Person(1, Jane Doe')
10 insert_to_db(person1)
```

In this program, the function `insert_to_db` expects its argument to be an object with an `int`-typed field, `id`, and a `str`-typed field, `name`. This type restriction may be useful to ensure that, for example, this program

* jeremy.siek@colorado.edu

obeys a contract imposed by an external database that all records of persons have this type structure. However, these statically-annotated types do not appear elsewhere in the program. Therefore, at the call to insert_to_db, the object person1 needs to be cast from the "dynamic type," $\star$, to the static type {id:int, name: str}.

```
10 insert_to_db(person1: ⋆ ⇒ {id:int, name:str})
```

The cast, written as $\star \Rightarrow${id:int, name:str}, has been inserted into the code at compile time. This cast would succeed in this case, but it is not guaranteed to; if an object that lacked a name field, or one where the id field was inhabited by a string, was passed to insert_to_db, the cast would be expected to produce a runtime exception.

```
11 person2 = Person('ABC123', 'John Doe')
12 insert_to_db(person2: ⋆ ⇒ {id:int, name:str})
       # Runtime cast error
```

The mutability of objects in languages like Python complicates this picture. Suppose we create a second reference to the object pointed to by person1, but this second reference will have a static type. The original reference, having a dynamic type, may be used to modify the type of the field values of the underlying object in memory, thereby violating the type constraints that the second reference expects:

```
13 typed_person : {id:int, name:str} =
       person1: ⋆ ⇒ {id:int, name:str}
14 person1.name = 42
15 insert_to_db(typed_person) #!!
```

The cast of person1 is fine, because at the cast site the contents of the object correspond to the expected type of the cast. However, several lines later, the type of typed_person.name has been essentially changed from under it, but typed_person has no way of knowing that this is so. If insert_to_db's read of typed_person.name simply returned the value in that field, the result would be a int value with a str type — in other words, an undetected type error. Therefore, it is insufficient for an object cast to simply check once whether the members of an object correspond to its expected type — it also needs to ensure that the members remain at that type, or at least are of that type when used. Doing so, and doing so in a way that is space-efficient, is the main goal of this work.

$$T \sim T \quad \star \sim T \quad T \sim \star$$
$$\frac{T_1 \sim T_1' \qquad T_2 \sim T_2'}{T_1 \to T_2 \sim T_1' \to T_2'}$$

Figure 1. Type consistency for functional languages

2. Background and related work

Early work on the mixing of static and dynamic typing in a single language includes Satish Thatte's Quasi-static Typing [9]. From there, Siek and Taha [6] introduced the modern notion of gradual typing, based on statically-inserted casts between static and dynamic code, to functional languages in 2006. In this system, the ability for values to go back and forth between static and dynamic portions of a program is governed by the consistency relation, shown in 1. During compilation, a program is accepted by the typechecker when the actual type of an expression is *consistent* with the expected type, rather than when it is *equal* to it, as is typical in functional type systems. However, when two types are consistent but not equal at compile time, a cast is inserted from the actual type to the expected one, to ensure that at runtime, the actual value is of the correct type — or to report that the typing constraints have been violated and safely halt the program's execution.

Siek and Taha later extended this work to object-oriented languages [7], but this work only described a system for pure, immutable objects, and did not support imperative-style field mutation.

Ina and Igarashi[4] developed an approach towards implementing gradual typing in nominally-typed object oriented languages like Java, including support for generics. However, their work was not largely concerned with time- or space-efficiency, and their approach is based in a nominal type system, where the type of an object is wholly specified by what class it belongs to. Object types in languages like JavaScript and Python, however, are better described by their structure — what interfaces or methods they support.

Siek and Wadler's space-efficient "threesome" casts [8] 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. The casts considered by Siek and Wadler only supported

functions; my work aims to extend this result to include objects as well.

Gradual typing has been used in several important languages. It has been applied to the dynamic language ActionScript [5], and work is under way at the University of Colorado Boulder to introduce it to Python. It is also used in Typed Scheme [10] and in C♯[2].

3. Approach and uniqueness

This section will first broadly describe through examples the approach this work takes to gradual typing with objects. It will then formalize this approach using a minimal programming language calculus.

3.1 General approach

Let us return to the examples from Section 1.1. As we saw, it is insufficient to treat casts on objects as simple one-time checks, because there can be more than one reference to the same object, the references can have different types, and one reference may make a change to the object that is incompatible with the type of another reference. This kind of mutation may then result in an undetected type error.

3.1.1 Accounting for mutation

To overcome this obstacle, we require that a cast applied to an object must return a proxy to that object. This proxy serves as an interface between the underlying object values in memory, and the reference to the object, which may have a static type imposed on it.

```
13 typed_person : {id:int, name:str} =
       person1: ⋆ ⇒ {id:int, name:str}
```

In this scheme, when we cast person1 from ⋆ to {id:int, name:str}, the result is a proxy, containing the additional type information created by the cast, to the object referenced by person1. This proxy then becomes the value of typed_person.

```
14 person1.name = 42
15 insert_to_db(typed_person)
```

When the underlying object is mutated by person1, a cast error is not yet triggered. This is acceptable, however, because when typed_person *is* used to access the name field of the object, it will cast that value from ⋆ to str — a cast that, in this case, will fail, because name actually contains an int. It is therefore possible for an underlying object to violate the constraints placed on it by its typed proxies, but such violations can never be witnessed, because the type of the object is checked by the proxy whenever it is actually used.

3.1.2 Space efficiency

While the use of proxies solves the problem of mutation from multiple references, proxies themselves introduce a new challenge.

```
16 def even_id(pid:int, person:⋆):
17     if pid == 0:
18         return person.name
19     else: return odd_id(pid  1,
           person:⋆ ⇒{id:int, name:str})
20 def odd_id(pid:int, person:{id:int, name:str}):
21     if pid == 0:
22         return None
23     else: return even_id(pid  1,
           person:{id:int, name:str}⇒ ⋆)
24 odd_id(person1.id, person1)
```

This pair of mutually recursive functions returns None if the pid argument is an odd number and the person parameter's name if pid is even. However, the type signatures of these functions differ, and therefore the person parameter may need to be cast back and forth from its static type in odd_id to and from its dynamic type in even_id. Now consider what has to happen at line 18 when this recursion terminates (assuming that the initial value of pid was even) — since the person object has been through numerous casts during the execution of the program, the value stored in name must traverse the proxies that have been wrapped around it:

$$\text{'Jane Doe'} : \text{str} \Rightarrow \star \Rightarrow \text{str} \Rightarrow \ldots \Rightarrow \star$$

Storing these chains of casts is extremely inefficient in terms of space usage (as well as inefficient in terms of time when the access site is executed).

To solve this, we use the insight of Siek and Wadler [8] that any series of n casts may be collapsed into two casts: a cast from the original type of the value to the *most specific* type at which it has been viewed, and then a cast from that type to the target type of chain of casts[1]

$$\text{'Jane Doe'} : \text{str} \Rightarrow \text{str} \Rightarrow \star$$

In keeping with Siek and Wadler, we refer to this pair of casts as a *threesome*, and write it as such:

[1] In this case, because the most specific type is str, one of these casts is an identity cast, which may be discarded.

numbers	n	$\in$	$\mathbb{Z}$
labels	ℓ		
variables	x		
expressions	e	$::=$	$x \mid n \mid \{\overline{\ell : T = e}\} \mid e.\ell \mid$
			$e.\ell = e \mid \mathsf{let}\ x : T = e\ \mathsf{in}\ e \mid$
			$e : T \Rightarrow T$
types	T	$::=$	$\star \mid \mathsf{int} \mid \{\overline{\ell : T}\}$

Figure 2. Syntax for gradual object calculus

heap addresses	a	$\in$	$\mathbb{N}$
uncasted values	ν	$::=$	$n \mid a$
values	v	$::=$	$\nu \mid \nu : T \overset{T}{\Rightarrow} \star \mid$
			$a : \{\overline{\ell : T}\} \overset{\{\overline{\ell:T}\}}{\Longrightarrow} \{\overline{\ell : T}\}$
heaps	μ	$::=$	$[\overline{a \mapsto \{\overline{\ell = v}\}}]$

Figure 3. Values for gradual object calculus

$$\mathsf{str} \overset{\mathsf{str}}{\Rightarrow} \star$$

By combining a proxy-based approach to object casts with threesome-style cast normalization, we are able to describe a system for efficient and correct gradually-typed objects.

3.2 Semantics of gradually typed objects

Figure 2 shows the syntax for a small calculus with gradually typed objects. This minimal programming language consists of numbers, objects, variables, variable-binding expressions, and expressions for reading from or writing to objects. It also includes casts; technically, casts would not appear in the concrete syntax of gradually-typed programs but would instead be inserted automatically. We assume that the cast-insertion stage has already occurred. Over-lines are used to indicate a n-length series of the syntactic structure underneath.

Figure 3 shows the values of the gradual object calculus, as well as the structure of the heap. Objects do not appear as values — instead, they are stored on the heap, and their addresses are treated as values. Also treated as values are casted values, of which there are two kinds: any value (including numbers) that has been cast to $\star$,[2] and objects which have been casted from one object type to another (for example, from $\{\mathsf{id}{:}\star,\ \mathsf{name}{:}\ \mathsf{str}\}$ to $\{\mathsf{id}{:}\mathsf{int},\ \mathsf{name}{:}\star\}$). These casted values, when the original value is a heap address, are the proxies described previously.

Finally, Figure 4 shows the reduction rules for the gradual object calculus. The rules shown in the figure each define one possible step in the program's reduction. The rule form $e_1 \mid \mu_1 \longmapsto e_2 \mid \mu_2$ means that an expression of the form e_1, paired with the heap state μ_1, may be reduced in a single step to an expression e_2 and a new, possibly updated heap μ_2, if all side conditions are met. Evaluation terminates when a value is reached, or when there is no rule that can be chosen (in which case the evaluation is "stuck"). All sub-expressions are fully evaluated first, with the exception of the "body" in let-bindings.

When "bare," uncasted objects are read from or written to, the reduction is straightforward — the read value is extracted from the heap, or it is replaced by the new value respectively. It becomes somewhat more complex when the object has been casted (i.e., is a proxy). When a field is read from a casted object, initially the field name is looked up in the heap, just as if the object were not casted. The resulting value is then itself casted by the relevant components of the object's overall cast, resulting in a value that respects the *current* context of the object (or a cast error). The process is reversed when an object is written to: the assigned value is first cast by the *reverse* of the relevant components of the object's cast, and then added to the heap. This ensures that no field write can invalidate the *source* type of any other cast on the same underlying object.

Threesome casts are heavily used in this design, and if these casts could build up on each other in an unbounded way, this would be a serious concern for space-efficiency. However, threesomes do compose with each other, as shown in the reduction rules for casts. The meet $\sqcap$ operator in the rule for composition of casts takes two types and produces a type that is more specific — that is, has less use of $\star$ — than either of them, but which is compatible with each. It can also return $\bot$ in the situation where there is no such type — such as in a cast from int to $\{\mathsf{x}{:}\mathsf{int}\}$. When $\bot$ occurs in the middle position of a threesome, a type error is detected.

[2] Casting to $\star$ can be viewed as wrapping or boxing values.

$$\{\overline{\ell : T = v}\} \mid \mu \longmapsto a \mid \mu[a \mapsto \{\overline{\ell \mapsto v}\}]$$
where a fresh

$$\mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mu \longmapsto e[x/v] \mid \mu$$

$$a.\ell \mid \mu \longmapsto \mu(a)(\ell) \mid \mu$$

$$(a : T_1 \overset{T_2}{\Rightarrow} \star).\ell \mid \mu \longmapsto \mu(a)(\ell) : T_{1\ell} \overset{T_{2\ell}}{\Longrightarrow} \star \mid \mu$$
where $T_1 = \{\ldots, \ell : T_{1\ell}, \ldots\}$
and $T_2 = \{\ldots, \ell : T_{2\ell}, \ldots\}$

$$(a : T_1 \overset{T_2}{\Rightarrow} T_3).\ell \mid \mu \longmapsto \mu(a)(\ell) : T_{1\ell} \overset{T_{2\ell}}{\Longrightarrow} T_{3\ell} \mid \mu$$
where $T_1 = \{\ldots, \ell : T_{1\ell}, \ldots\}$
and $T_2 = \{\ldots, \ell : T_{2\ell}, \ldots\}$
and $T_3 = \{\ldots, \ell : T_{3\ell}, \ldots\}$

$$a.\ell = v \mid \mu \longmapsto a \mid \mu[a(\ell) \mapsto v]$$

$$(a : T_1 \overset{T_2}{\Rightarrow} \star).\ell = v \mid \mu \longmapsto a.\ell = (v : \star \overset{T_{2\ell}}{\Longrightarrow} T_{1\ell}) \mid \mu$$
where $T_1 = \{\ldots, \ell : T_{1\ell}, \ldots\}$
and $T_2 = \{\ldots, \ell : T_{2\ell}, \ldots\}$

$$(a : T_1 \overset{T_2}{\Rightarrow} T_3).\ell = v \mid \mu \longmapsto a.\ell = (v : T_{3\ell} \overset{T_{2\ell}}{\Longrightarrow} T_{1\ell}) \mid \mu$$
where $T_1 = \{\ldots, \ell : T_{1\ell}, \ldots\}$
and $T_2 = \{\ldots, \ell : T_{2\ell}, \ldots\}$
and $T_3 = \{\ldots, \ell : T_{3\ell}, \ldots\}$

$$\nu : T \overset{T}{\Rightarrow} T \mid \mu \longmapsto v \mid \mu$$

$$(\nu : T_1 \overset{T_2}{\Rightarrow} T_3) : T_3 \overset{T_4}{\Rightarrow} T_5 \mid \mu \longmapsto \nu : T_1 \overset{T_2 \sqcap T_4}{\Longrightarrow} T_5 \mid \mu$$

$$\nu : T_1 \overset{\bot}{\Rightarrow} T_3 \mid \mu \longmapsto \mathsf{error}$$

$$e : T_1 \Rightarrow T_2 \mid \mu \longmapsto e : T_1 \overset{T_1 \sqcap T_2}{\Longrightarrow} T_2 \mid \mu$$

Figure 4. Expression evaluation

This design for gradually typed objects ensures that no undetected type errors occur, and that multiple references with different types can refer to the same object through proxies. This, combined with the insights of Siek and Wadler with regard to the composition of casts, allows for a semantically correct and space-efficient design of gradually typed object casts.

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 while preventing type errors, and requires minimal space overhead on object casts after the initial cast. In combination with ongoing work on space efficient gradual function casts, and with techniques such as blame-tracking [1] and gradual type inference [5], this work increases the practicality and feasibility of integrating gradual typing into object-oriented dynamic languages such as JavaScript 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. ISBN 9781605585437.

[2] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#.

[3] Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid Types, Invariants, and Refinements For Imperative Objects. In *FOOL/WOOD '06*, 2006.

[4] Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *OOPSLA '11*, October 2011. ISBN 9781450309400. doi: 10.1145/2076021.2048114. URL `http://dl.acm.org/citation.cfm?doid=2076021.2048114`.

[5] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *POPL '12*, pages 481–494. ACM, 2012. ISBN 9781450310833. URL `http://dl.acm.org/citation.cfm?id=2103714`.

[6] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop '06*, pages 81–92. ACM, 2006. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.8890&rep=rep1&type=pdf`.

[7] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP '07*, pages 2–27. Springer, 2007. URL `http://www.springerlink.com/index/A052J27UM6829356.pdf`.

[8] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *POPL '10*, pages 365–376. ACM, 2010. ISBN 9781605584799. URL `http://dl.acm.org/citation.cfm?id=1706342`.

[9] Satish Thatte. Quasi-static typing. In *POPL '89*, volume 31, pages 367–381. ACM, July 1989. ISBN 0897913434. doi: 10.1016/j.msec.2011.02.016. URL `http://www.ncbi.nlm.nih.gov/pubmed/22003646http://portal.acm.org/citation.cfm?id=96747`.

[10] Sam Tobin-Hochstadt and Robert Bruce Findler. Cycles without pollution: a gradual typing poem. In *STOP '09*, pages 47–57. ACM, 2009. URL `http://dl.acm.org/citation.cfm?id=1570512`.