Cast Insertion Strategies for Gradually-Typed Objects *

Esteban Allende † Johan Fabry Éric Tanter

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
{eallende,jfabry,etanter}@dcc.uchile.cl

Abstract

Gradual typing enables a smooth and progressive integration of static and dynamic typing. The semantics of a gradually-typed program is given by translation to an intermediate language with casts: runtime type checks that control the boundaries between statically- and dynamically-typed portions of a program. This paper studies the performance of different cast insertion strategies in the context of Gradualtalk, a gradually-typed Smalltalk. We first implement the strategy specified by Siek and Taha, which inserts casts at call sites. We then study the dual approach, which consists in performing casts in callees. Based on the observation that both strategies perform well in different scenarios, we design a hybrid strategy that combines the best of each approach. We evaluate these three strategies using both micro- and macro-benchmarks. We also discuss the impact of these strategies on memory, modularity, and inheritance. The hybrid strategy constitutes a promising cast insertion strategy for adding gradual types to existing dynamically-typed languages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance

Keywords gradual typing, casts, Gradualtalk

1. Introduction

The popularity of dynamic languages and their use in the construction of large and complex software systems makes the possibility to fortify grown prototypes or scripts using the guarantees of a static type system appealing. While research in combining static and dynamic typing started more than twenty years ago, recent years have seen a lot of proposals of either static type systems for dynamic languages, or partial type systems that allow a combination of both approaches [2–5, 10, 12, 14, 20].

Gradual typing [15, 16] is a partial typing technique proposed by Siek and Taha that allows developers to define which sections of code are statically typed and which are dynamically typed, at a very fine level of granularity, by selectively placing type annotations where desired. The type system ensures that dynamic code does not violate the assumptions made in statically-typed code. This makes it possible to choose between the flexibility provided by a dynamic type system, and the robustness of a static type system.

The semantics of a gradually-typed language is typically given by translation to an intermediate language with casts, *i.e.* runtime type checks that control the boundaries between typed and untyped code. A major challenge in the adoption of gradually-typed languages is the cost of these casts, especially in a higher-order setting. Theoretical approaches have been developed to tackle the space dimension [11, 17], but execution time is also an issue. This has led certain languages to favor a coarse-grained integration of typed and untyped code [22] or to consider a weaker form of integration that avoids costly casts [24]. Other approaches include the work of Rastogi et al. [14], using local type inference to significantly reduce the number of casts that are required.

In developing Gradualtalk[1], a gradually-typed Smalltalk, our first concern was the design of the gradual type system, with its various features [1]. In the current stage of this work, we are concerned with the efficiency of casts, especially those related to method invocations. This is because method invocations are naturally very frequent in object-oriented programs, especially in pure object-oriented languages like Smalltalk. Casts incur a runtime cost, and we are interested in their efficiency so as to achieve an acceptable level of performance without losing the features of gradual typing. In the foundational paper on gradually-typed objects [16], Siek and Taha describe the semantics of cast insertion using a caller-side strategy—which we term the *call strategy*. Due to implementation issues (which have since been resolved), our very first implementation of cast insertion, before implementing the Siek-Taha approach, was however based on a different approach, which we name the *execution strategy*. Here, casts are inserted on the callee side, at the beginning of each typed method. Studying the performance of both approaches revealed that they have complementary strengths, and that a third approach, which we call the *hybrid strategy*, could combine the best of both approaches.

This paper reports on the study of these three cast insertion strategies in Gradualtalk. We present the experimental setting for microbenchmarks in Section 2. We then describe all three strategies

* This work is partially funded by FONDECYT Project 1110051.

† Esteban Allende is funded by a CONICYT-Chile Ph.D. Scholarship.

DLS '13, October 28, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2433-5/13/10. . . $15.00.
http://dx.doi.org/10.1145/2508168.2508171

[1] `http://www.pleiad.cl/gradualtalk`

```
MyCollection ≫ (Self) addElement: (Integer)x
  collection addLast: x.

MyCollection ≫ (Integer) at: (Integer)index
  ↑collection at: index.

MyCollection class ≫ (Self instance) new: (Integer)size
  ↑super new
    collection: (OrderedCollection new: size);
    yourself.
```

Listing 1. MyCollection class.

```
Client ≫ untypedClient: stop withSize: size
  |col|
  col := MyCollection new: size.

  1 to: size do: [:i|
    col addElement: i.
  ].
  1 to: size do: [:i|
    ((col at: i) > stop) ifTrue: [ ↑i ]
  ].
  ↑−1
```

Listing 2. Untyped client.

```
Client ≫ (Integer) typedClient: (Integer)stop withSize: (Integer)
      size
  |(MyCollection)col|
  col := MyCollection new: size.

  1 to: size do: [:(Integer)i|
    col addElement: i.
  ].
  1 to: size do: [:(Integer)i|
    ((col at: i) > stop) ifTrue: [ ↑i ]
  ].
  ↑−1
```

Listing 3. Typed client.

in turn in Sections 3, 4, and 5. In each of these sections, we informally describe the approach and report on microbenchmarks. We then report on macrobenchmarks in Section 6. We discuss memory consumption and other considerations in Section 7. Section 8 discusses related work and Section 9 concludes.

2. Experimental Setting: Microbenchmarks

In this work, we evaluate cast insertion strategies by implementing them for Gradualtalk [1]. Gradualtalk is a gradually-typed Smalltalk that features a combination of nominal and structural types, self types, parametric polymorphism, and union types. In Gradualtalk code, omitting type annotations is equal to specifying the unknown type, which we denote Dyn. Gradualtalk is currently implemented in Pharo Smalltalk version 2.0. Note that Pharo uses the Cog VM, which features a JIT compiler.

Because there is no standard benchmark suite for Smalltalk, we designed both micro- and macrobenchmarks. We describe microbenchmarks in this section; they will be used in the explanation of the different strategies to give a first assessment of their performance. The macrobenchmarks are described in Section 6 to provide an evaluation of the performance of cast insertion strategies on larger-scale, real-world scenarios. To favor reproducibility, the Smalltalk image used to perform all our experiments can be downloaded from:
`http://pleiad.cl/gradualtalk/strategies`

Microbenchmarks

We designed the microbenchmark setting to study the specific cost of each cast insertion strategy in both their best and worst cases.

To do so, we start with a typed collection, MyCollection, shown in Listing 1. Note that the addElement: method returns Self, following the convention for side-effecting methods in Smalltalk. The class method new: has Self instance as the return type, specifying that it returns an instance of itself.

We then use two versions of a client: untyped and typed, that repeatedly inserts elements and then looks for them. The code for both versions is shown in Listing 2 and Listing 3, respectively. The goal of the microbenchmarks is to measure the cost of each strategy for the calls that perform element insertion (#addElement:) and the calls that perform lookup of an element (#at:). Beyond these calls, the code run in each version is the same; in particular, methods #to:do:, #ifTrue: and #> are primitives, *i.e.* they are not subject to cast insertion.

We execute the microbenchmarks for different sizes between 1,000,000 and 10,000,000 elements. For each size, the experiment is repeated ten times and the average time is calculated. In each case, the value of stop is half that of size. The benchmarks were run on a machine with an Intel Core i7 3.20 GHz CPU, 4 GB RAM and 250 GB SSD disk, running Ubuntu 12.10.

3. Call strategy

The call strategy is the direct implementation of the specification of Siek and Taha [16].

Before explaining the informal description of the call strategy, we need to introduce the concept of consistent subtyping. Gradual typing extends traditional subtyping to *consistent subtyping* [16]. Consistency, denoted $\sim$, is a relation that accounts for the presence of Dyn: Dyn is consistent with any other type and any type is consistent with itself. The consistency relation is not transitive in order to avoid collapsing the type relation [15]. A type σ is a consistent subtype of τ, noted $\sigma \lesssim \tau$, iff either $\sigma <: \sigma'$ and $\sigma' \sim \tau$ for some σ', or $\sigma \sim \sigma''$ and $\sigma'' <: \tau$ for some σ''.

3.1 Description

Essentially, the call strategy inserts casts at call sites whenever needed. A typed callee can therefore rely upon the fact that all callers have been checked previously, and hence assume that its arguments are of the proper types.

The call strategy has two different scenarios for inserting casts, depending on whether the receiver type in the call site is known at compile time. For call sites where the receiver type is known at compile time (*i.e.* non-Dyn), the compiler compares the type of the passed arguments in the invocation with the type of the parameters in the method declaration. If the argument type is not a subtype of the parameter type but it is a consistent subtype, then a cast to the parameter type is inserted.

For call sites where the receiver type is unknown at compile time (*i.e.* Dyn), the compiler inserts code that, at runtime, looks up the method type information and casts each argument to the expected parameter type. This lookup must take into account that methods can be overridden, and that the required type information is obtained from the appropriate overriding method. This lookup procedure is similar to the lookup used to retrieve a method when it is invoked. That means that when realizing method lookup, the runtime could also retrieve the type information of that method for the usage of the cast strategy. However, this would require to modify the VM.

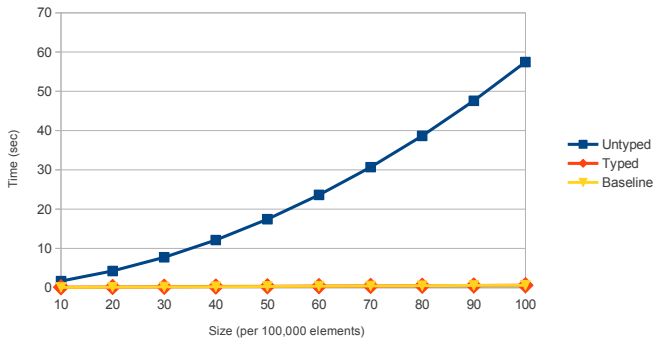

Figure 1. Running times of the call strategy for untyped and typed code, as well as of baseline Smalltalk.

For example, in the case of typedClient, because col is statically known to be of type MyCollection and the argument i of type Integer, there is no cast inserted when adding an element to the collection:

```
...
col addElement: i.
...
```

Similarly, there is no cast needed when accessing the collection with at:.

In the case of untypedClient, however, there is no static type information available, therefore the call strategy inserts code that will, at runtime, retrieve the actual type of the receiver, and assuming it is statically typed (as MyCollection is), perform runtime checks on the arguments to ensure they match.

The code below shows the transformation of the call strategy in the untypedClient when addElement: is invoked (optimized for the single parameter case):

```
...
__rcv := col.
__typeParam := GTRuntime getTypeParamOf: #addElement:
                                    in: __rcv.
__rcv addElement: (<__typeParam>i).
...
```

Note that retrieving the method type information dynamically is costly; our current implementation maintains a cache per class that associates each selector with its argument types. As a matter of fact, if gradual typing were integrated at the virtual machine level, we could extend the existing infrastructure of polymorphic inline caches to deal with the type information, and hence further reduce the associated cost.[2]

3.2 Microbenchmarks

Figure 1 shows the results of the microbenchmark for the call strategy. Detailed numbers for all microbenchmarks are in Tables 1 and 2. We can observe that with a typed client, the call strategy exhibits almost identical performance as base Smalltalk, *e.g.* 0.62 seconds versus 0.63 seconds for 10M elements. With the untyped code, the call strategy however takes up to 90 times the time of base Smalltalk: 57.49 seconds versus 0.63 seconds for 10M elements. This result reflects the fact that with a typed client, the call strategy does not insert any cast, making the resulting bytecode exactly the same as that of base Smalltalk. Conversely, with an untyped client, the call strategy inserts costly casts. The results show that even with the cache, the incurred overhead is substantial.

[2] For simplicity, in this paper we focus on argument types only; in the three strategies, casts on return types are performed in the callee, following [16].

4. Execution strategy

As we have seen, the call strategy performs very well when a typed client calls a typed library, but it does not perform well when the client is untyped and the library is typed. This is unfortunate, because the scenario of a typed library that is used from untyped code is a predictably frequent scenario in Gradualtalk. Indeed, only a handful of libraries have been typed so far [1]. If using a typed library incurs a high performance overhead, this is likely to discourage the adoption of static types.

As it turns out, when first implementing Gradualtalk, a limitation of the compiler (which was subsequently resolved) prevented us from adopting the call strategy for cast insertion at first. To address this, we developed another approach, called the *execution strategy*, which turns out to perform well in the case of dynamically-typed receivers.

4.1 Description

The idea of the execution strategy is to insert casts on arguments of a statically-typed method directly at the beginning of the method. The interesting characteristic of this strategy is that, in the case of a dynamically-typed receiver, there is no need to retrieve its type information at runtime.

In the execution strategy, the compiler inserts one cast per parameter at the start of the method. Each cast checks that the type of the value bound to the parameter corresponds to the declared type in the method signature.

For example, at the start of the method MyCollection >> #addElement:, the execution strategy inserts casts to the parameters of the method:

```
MyCollection ≫ addElement: x
  (<Integer>) x.
  collection addLast: x.
```

At call sites, regardless of whether the receiver is statically or dynamically typed, the execution strategy does not perform any transformation:

```
...
col addElement: i.
...
```

Note that this strategy is only meaningful in safe languages like Smalltalk, in which the runtime type of an object can be retrieved directly, *e.g.* through its class pointer. In the core semantics of gradually-typed objects of Siek and Taha, two-position casts play the role of tagging values with their type, so that injecting a value into Dyn does not lose its original type. This being said, all existing dynamic object-oriented languages that we are aware of are safe, and therefore the execution strategy is a meaningful option in all these languages.

Finally, observe that if casts are not implemented as primitives of the VM, then care must be taken to not create an infinite loop when the casting method uses methods of the system. This infinite loop can be produced if the casting procedure calls a method with at least one argument. In that case, because of how the execution strategy works, a cast would be done again in that argument when the method is invoken, creating the infinite loop. To avoid this, we have chosen to disable casts while a cast method is being executed, as it is the most straightforward approach.

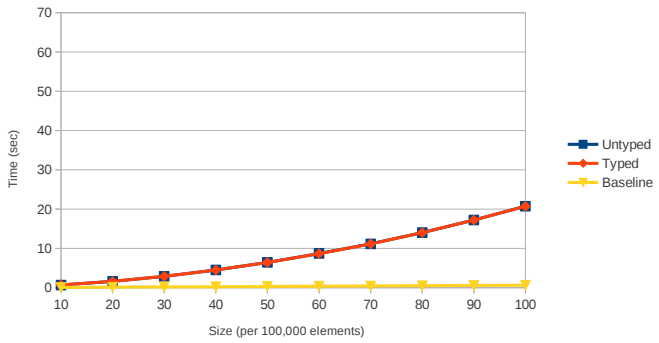

Figure 2. Running times of the execution strategy for untyped and typed code, as well as of baseline Smalltalk.

Size (1M)	Call (sec)	Exec (sec)	Hybrid (sec)	Base ST (sec)	Exec vs Call (%)	Hybrid vs Exec (%)
1	1.666	0.671	0.626	0.063	-59.75	-6.71
2	4.256	1.609	1.529	0.127	-62.19	-4.97
3	7.740	2.899	2.785	0.190	-62.55	-3.93
4	12.129	4.504	4.357	0.252	-62.87	-3.26
5	17.425	6.433	6.252	0.315	-63.08	-2.82
6	23.611	8.705	8.473	0.378	-63.13	-2.67
7	30.693	11.158	11.010	0.441	-63.65	-1.32
8	38.656	14.033	13.870	0.504	-63.70	-1.16
9	47.592	17.222	17.054	0.566	-63.81	-0.98
10	57.464	20.748	20.528	0.629	-63.89	-1.06

Table 1. Running times of the untyped client microbenchmark.

4.2 Microbenchmarks

Figure 2 shows the results of the microbenchmark for the execution strategy. We can observe that the execution strategy is unaffected if the client is typed or not: for 10M elements, it takes 20.75 seconds with the untyped client, and 20.73 seconds with the typed client. This is because argument casts are inserted at the start of the methods of MyCollection, and are therefore always executed.

As a result, the call strategy is much faster than the execution strategy in the case of the typed client (around 93%). Recall that with the typed client, the call strategy does not insert any cast at all. Conversely, the execution strategy is considerably faster than the call strategy when using an untyped client (around 63%). While both approaches incur the cost of argument casts, the call strategy is slower because it first needs to retrieve the method type information (from the cache) to determine the actual cast to perform. In the execution strategy, the expected argument type is statically known, so only the proper cast happens.

5. Hybrid strategy

The comparison of the call and execution strategies shows that they have complementary benefits. The call strategy performs well with typed clients, and the execution strategy performs well with untyped clients. We now present a novel strategy that combines the best of both strategies.

5.1 Description

The idea of the hybrid strategy is to trade space for speed. This is done by duplicating each method: one version is the original method that does not cast its arguments—called the *unguarded* method; and the other method starts by casting its arguments, as in the execution strategy—called the *guarded* method. Then, as in the call strategy, it has two different scenarios for inserting casts, depending on whether the receiver type in the call site is known at compile time or not. For statically-typed receivers, the compiler modifies the call site to invoke the unguarded method. If the argument type is not a subtype of the parameter type of the method declaration, the compiler inserts casts to the argument types. For dynamically-typed receivers, the compiler leaves the call site intact, as it will call the guarded method. If casts are also not implemented as primitives, then the same precaution with respect to the infinite loops made in the execution strategy should be taken in the hybrid strategy.

For instance, the addElement: method of MyCollection is replaced with the two methods:

```
MyCollection ≫ addElement: x      "guarded"
  (<Integer>)x.
  collection ___addLast: x.

MyCollection ≫ ___addElement: x   "unguarded"
  collection ___addLast: x.
```

When sending a message, if the type of the receiver is known, the code is modified to invoke directly the unguarded method:

```
...
col ___addElement: i.
...
```

Otherwise, no transformation occurs, and hence the guarded method is called:

```
...
col addElement: i.
...
```

Size (1M)	Call (sec)	Exec (sec)	Hybrid (sec)	Base ST (sec)	Call vs Exec (%)	Hybrid vs Call (%)
1	0.126	0.663	0.125	0.063	-80.95	-0.79
2	0.180	1.606	0.180	0.127	-88.81	-0.11
3	0.235	2.898	0.234	0.190	-91.89	-0.34
4	0.289	4.500	0.292	0.252	-93.59	1.18
5	0.344	6.430	0.346	0.315	-94.65	0.70
6	0.397	8.667	0.402	0.378	-95.42	1.31
7	0.452	11.146	0.455	0.441	-95.95	0.80
8	0.504	14.018	0.514	0.504	-96.40	1.94
9	0.565	17.212	0.564	0.566	-96.72	-0.11
10	0.620	20.725	0.620	0.629	-97.01	-0.13

Table 2. Running times of the typed client microbenchmark.

5.2 Microbenchmarks

Figure 3 shows the results of the microbenchmark for the hybrid strategy. We can observe that the typed client runs faster than the untyped one: for 10M elements, it takes 20.53 seconds with the untyped client, and 0.62 seconds with the typed client. The difference reflects the code of the casts that are done repeatedly in the case of the untyped client.

We now compare the three strategies first on the untyped client and then on the typed client. In Figure 4 we show the execution times of the three cast insertion strategies for the microbenchmark using the untyped client. The figure shows that for untyped code the call strategy is by far the slowest strategy, taking 57.46 seconds for 10M elements. The execution and hybrid strategies have what amounts to the same level of performance: for 10M elements 20.75

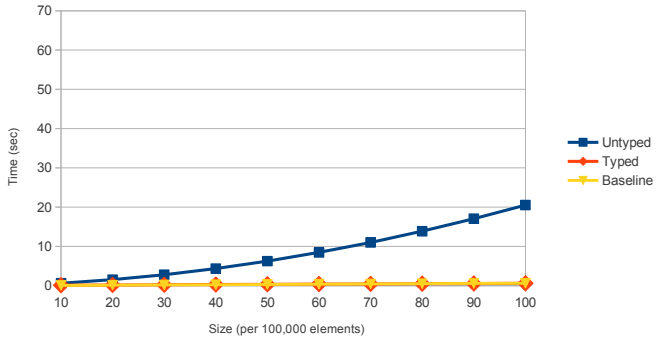

Figure 3. Running times of the hybrid strategy for untyped and typed code, as well as of baseline Smalltalk.

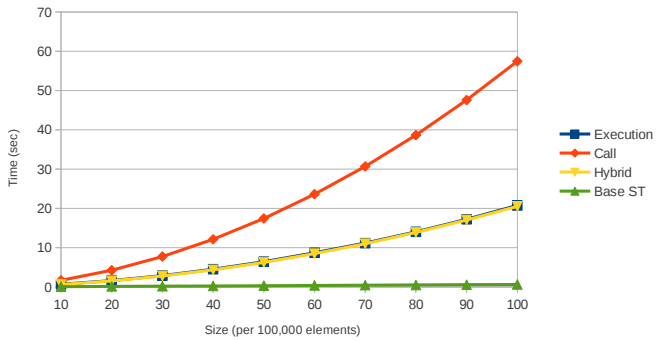

Figure 4. Running times of different cast strategies on the microbenchmark using the untyped client.

seconds and 20.53 seconds, respectively. These however still take up to 33 times the time of base Smalltalk, which takes 0.63 seconds for 10M elements. The reason for this slowdown is that each strategy needs to perform some runtime casts, while the standard Smalltalk does not.

In Figure 5 we show the execution times of the three cast insertion strategies for the microbenchmark using the typed client. As can be expected, the execution strategy is the slowest, taking 20.73 seconds for 10M elements. The call and hybrid strategies have the same performance, and are nominally as fast as base Smalltalk (0.63 seconds for base Smalltalk and 0.62 seconds for both the hybrid and call strategy). This remarkable similarity is because none of these strategies needs to do any cast. The execution strategy does perform such casts, which causes it to have a lower performance.

To conclude, the microbenchmarks confirm that the hybrid strategy performs as good as its best competitor in all cases.

6. Macrobenchmarks

In order to get an indication of whether the microbenchmark results carry over to larger-scale and real-world scenarios, we designed and performed some initial macrobenchmarks.

We have seen in the microbenchmarks that the call strategy is two orders of magnitude slower with untyped code. Considering that Smalltalk code is mostly untyped, the call strategy will be

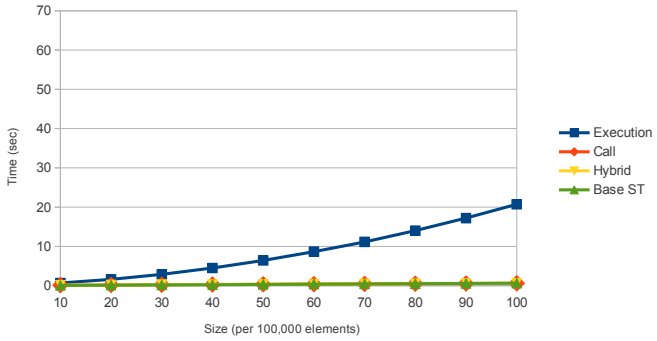

Figure 5. Running times of different cast strategies on the microbenchmark using the typed client.

	GZip-T	SAX-T1	SAX-T2
typed classes	1	2	3
methods	23	62	116
fully typed	2	31	51
untyped	21	29	69
partially typed	0	2	3

Table 3. Characterization of the partially-typed libraries.

clearly outperformed in orders of magnitude by the execution and hybrid strategies on macrobenchmarks. Therefore, in this section, we only compare the performance of the execution and hybrid strategies.

6.1 Experimental Setup

We start with two scenarios that use untyped libraries:

- GZip: Compressing a 32MB XML file (adapted from a game project) to a gzip file. This is a writing I/O intensive operation.
- SAX: Parsing the same XML file with a SAX parser. This is a reading I/O intensive operation.

In order to measure how both strategies perform when type annotations are added, we developed partially-typed variants of these libraries. GZip-T adds some annotations to the GZip library. And for SAX, we developed two partially-typed versions, SAX-T1 with few type annotations, and SAX-T2 with more annotations. Note that even though the code in the image has no type annotations, some types are implicitly deduced: self and literal values.

To characterize the extent to which each library is typed, we classify methods in three categories: typed methods, partially-typed methods and untyped methods. A fully-typed method specifies a static type for its return value and all its arguments. An untyped method leaves all return and argument types unspecified. A method that is neither typed nor untyped is classified as partially typed. Table 3 shows the numbers of elements in each category, for the different versions of the GZip and SAX libraries.

More precisely, for GZip we profiled the execution of the benchmark and typed the two methods whose interaction was causing most overhead: methods nextPutAll: and next:putAll:startingAt: of the DeflateStream class. For SAX-T1, we typed the interface between the driver SAXDriver and the handler SAXHandler. In SAX-T2, we also typed the interface between the driver and the tokenizer XMLTokenizer.

Every macrobenchmark is repeated ten times and the average time is calculated. The benchmarks were run on a machine with an

Intel Core i7 3.20 GHz CPU, 4 GB RAM and 250 GB SSD disk, running Ubuntu 12.10.

	Base ST (sec)	Exec (sec)	Hybrid (sec)	Hybrid vs Exec (%)
GZip	1.774	1.776	1.804	+1.58
GZip-T	–	1.804	1.790	-0.80
SAX	7.681	9.003	9.075	+0.80
SAX-T1	–	14.468	13.184	-8.87
SAX-T2	–	35.667	22.528	-36.84

Table 4. Running times of the macrobenchmarks.

6.2 Results

Table 4 shows the results of the macrobenchmarks. For the GZip benchmark, we see that both strategies perform similarly in all scenarios. With the untyped library, the hybrid strategy is slightly slower (2.98%), and it is marginally faster with a version of the library with a few type annotations (0.80%). These factors are however negligible, and hence we did not pursue more advanced typing for this scenario. The reason why GZip is fairly stable irrespective of typing is that much of the time is spent in I/O and other primitives, which are out of reach of the type system.

In the case of the untyped SAX benchmark, both approaches also exhibit the same performance. Interestingly, in this case, as more type annotations are added to the SAX library, the hybrid strategy becomes noticeably more competitive: it is nearly 9% faster with SAX-T1, and almost 37% faster with SAX-T2.

Overall, these results are consistent with the microbenchmarks: available type information allows the hybrid strategy to use unguarded methods, which are faster than the cast-first methods used by the execution strategy.

Finally, we see that as type annotations are added to a library, the performance tends to degrade, irrespective of the chosen cast insertion strategy. While this phenomenon is not perceptible in the case of GZip, it is substantial for SAX. The reason of the degradation is that, by adding type information, we create boundaries with dynamically-typed sections of the program. At these boundaries, casts have to be inserted to ensure that the static type assumptions are not violated when this code is called from the dynamically typed world. Since casts are performed eagerly, the entailed verification is costly compared to the default behavior of Smalltalk, which does no eager verification and only raises an exception when a method is not found. Reflecting on this result and the results of the microbenchmarks for the hybrid strategy (discussed in Section 5.2) suggest that predicting the performance impact of adding static type information is not trivial. Using only the results of the microbenchmark, we could assume that typing more code in a language with a gradual type system would always improve performance. However, this is generally not so. It remains to be studied whether there is a tipping point beyond which adding more type annotations enables better absolute performance, or at least does not degrade it. If we stick to a relative comparison between the execution and hybrid strategies, the results confirm that hybrid is progressively more advantageous as more type annotations are added.

7. Comparing Strategies beyond Performance

Until now we have focused on the performance of cast insertion strategies. In this section we discuss the impact of these strategies on memory consumption, modularity, and the interaction with inheritance.

Strategy	Size (MB)	Overhead vs. Base ST
Call	6.08	+56.8%
Execution	4.68	+21.3%
Hybrid	6.81	+75.9%
Hybrid-fwd	5.93	+53.9%
Base ST	3.88	

Table 5. Memory footprint of Smalltalk images compiled using the different strategies.

7.1 Memory

As stated in Section 5.1, the hybrid strategy trades memory for speed: for each method in the source program, it generates both an unguarded version of the method—whose body is the same as the original method—and a guarded version, which additionally performs argument casts. Table 5 shows the memory usage of the Smalltalk image—which includes 7314 classes and 67066 methods—compiled with all three strategies, compared to the memory usage of the standard image. Note that we also report on an alternative implementation of the hybrid strategy (Hybrid-fwd), discussed below.

The results confirm that the hybrid strategy uses substantially more memory than the other strategies. This is unsurprising: the high memory overhead comes from the duplication of all methods. The overhead of the call strategy, on the other hand, is entirely due to call site transformations (casting arguments and type information retrieval at runtime), which turn out to be quite space consuming.

Different implementations of the hybrid strategy are possible, each with a different tradeoff between performance and memory consumption. First, instead of duplicating all methods, it is possible to make each guarded method call the corresponding unguarded method. This approach, called Hybrid-fwd in Table 5, avoids duplicating method bodies and clearly reduces the overhead, down to +53.9%, which is slightly better than the memory overhead of the call strategy. The relatively high memory overhead of this particular implementation of the hybrid strategy compared to the execution strategy (around 30% more when compared to the baseline) can be explained by the fact that many methods are small. For a small method, introducing an extra forwarder method (which is also small) is as consuming as duplicating it.

Of course, the Hybrid-fwd approach saves some space at the expense of an extra method call in each guarded method. Because Smalltalk does not support statically-bound private methods, the added calls turn out to have a noticeable overhead on the SAX macrobenchmarks (Table 6). In any case, the results show that it is a viable alternative if memory consumption becomes an issue. We conjecture that this overhead would be negligible in a language like Java where private method calls can be aggressively optimized.

	Hybrid (sec)	Hybrid-fwd (sec)	H-fwd vs H (%)
GZip	1.804	1.839	+1.94
GZip-T	1.790	1.824	+1.90
SAX	9.075	10.980	+20.99
SAX-T1	13.184	14.174	+7.51
SAX-T2	22.528	25.423	+12.85

Table 6. Macrobenchmark running times of the duplication-based vs. forward-based implementations of the hybrid strategy.

Other implementations can also be considered. For example, an implementation could use only one method and pass an additional boolean parameter that determines whether casts should be performed or not. This would be fairly efficient space-wise, but comes

at the cost of passing an additional argument and adding a branch in the code. The optimal version would be to allow a single method to have two entry points: one entry at the start of the method where the arguments are cast, and another entry just after these casts. The translation would then insert calls to the second entry point when casts can be safely skipped. This would however require support at the level of the virtual machine. We have not fully explored these different implementations so far.

7.2 Modularity

Suppose that we modify MyCollection (Listing 1) so that the type of elements stored changes from Integer to Date, however we do not recompile Client. In the case of the untyped client, all of the strategies would detect the mismatch and their casts would fail. However, in the case of the typed client, only the casts in the execution strategy would fail.

The reason for this is that the call strategy relies on the possibility to analyze all call sites of a given method in order to introduce the casts of arguments, if needed. If Client is not recompiled, then its implementation still assumes, wrongly, that the argument to addElement: has to be of type Integer. Conversely, the execution strategy casts arguments in the callee, and therefore does not need to re-check callers after such a change. Gradualtalk addresses the need for analyzing all the callers of a method when using the call strategy through a dependency tracking mechanism [1]. It triggers recompilation of all call sites of a given method when needed, causing required casts to be inserted accordingly.

To be able to introduce efficient calls to unguarded methods, the hybrid strategy also needs to analyze all callers of a given method and may need to introduce casts in the callers. This being said, if the choice is to favor modular recompilation instead of performance, it would be possible to configure the hybrid strategy so that certain modules are not transformed (and therefore call guarded methods).

Note that the dependency between callers and callees that manifests itself when using typed clients with the call or hybrid strategies is the same as the dependencies between typed components in any typed language[3]. Put succinctly: when shared assumptions are changed, both parties need to be rechecked. In contrast, the execution strategy is inherently modular (though less performant) in such scenarios, performing such rechecking dynamically.

7.3 Interaction with inheritance

The calculus of gradually-typed objects of Siek and Taha does not include any form of inheritance, and therefore issues related to overriding are not considered. In their work on gradual typing for first-class classes, Takikawa *et al.* observe that a standard subtyping approach "would fail because a subclass may override a method with a different type" [19]. Consequently, they use use row polymorphism instead of standard subtype polymorphism.

More precisely, if overriding a method is valid whenever the overriding method is a *consistent* subtype of the overriden method, the call cast insertion strategy is unsound. Consider the following example:

```
A ≫ m: x          "superclass, untyped"

B ≫ m: (Integer) x   "subclass overrides with typed argument"
```

and the following client:

```
|(A) a|
a = B new.
a m: 'hi'
```

[3] This is what Gilad Bracha calls the "anti-modularity" of types: `http://gbracha.blogspot.com/2011/06/types-are-anti-modular.html`

The static type of a is A and the signature of m: in A does not specify any argument type. Therefore the call strategy accepts the invocation of m: without inserting any cast to Integer. Hence the use of the call strategy would result in an unsound execution as the body of B.m executes with an argument of an invalid type.

A possibility to retain soundness is to simply restrict valid overridings to proper subtyping, and not consistent subtyping. This however means that typed and untyped hierarchies cannot be mixed: a typed method can never be overridden by a untyped method, and vice versa. Remarkably, the execution strategy does not suffer from this soundness issue at all: because casts are inserted in the callees, the first thing B.m does is to cast its argument to Integer, which fails as expected. Therefore it is possible to define valid overriding based on consistent subtyping, but at the cost of sacrificing the efficiency benefits of the call strategy.

Again, the hybrid strategy provides the opportunity to achieve the best of both worlds: retaining soundness while exploiting opportunities for optimizations. To properly deal with the case above, the hybrid strategy needs to refrain from using the unguarded method call in case the expected type of an argument is Dyn. Nonetheless, it may still use the unguarded method when it is safe to do so. Consider the following client:

```
|(B) a|
b = B new.
b m: 1
```

The invocation of m: can be performed efficiently without any cast, because the static type of b specifies that the argument must be an Integer, which it is. In order to deal with the dual case of overriding—*i.e.* a typed method is overriden by a dynamically-typed one—the guarded method must perform specific checks to ensure that the dynamic method is used only in ways that are compatible with the subtyping relation. More precisely, the arguments to the dynamic method should be either supertypes or subtypes of the declared argument types in the overriden method (supertypes are valid because of the contravariance in argument types)[4]. Also, the value returned by the dynamic method should be a subtype of the declared return type. Consider the following:

```
C ≫ m: x          "subclass of B, untyped"
```

Then the untyped client code:

```
c = C new.
c m: 'hi'
```

This invocation raises a cast error at runtime because C.m is used in a way that is incompatible with any typed overriding of B.m: subtyping-wise, String is unrelated to Integer. We leave the detailed, formal treatment of this approach to future work.

8. Related work

Gradualtalk follows the line of work on gradual typing developed by Siek and Taha [15, 16]. A key ingredient of gradual typing is the consistency relation, which statically allows untyped values to flow in positions where values of specific types are expected. Casts are used to dynamically ensure that actual values are compatible with their expected type. In the original work of Siek and Taha, the objective is to show how a fully-annotated gradual program directly corresponds to a statically-typed program. Therefore, the runtime semantics do not assume that values have built-in runtime type tags for safety; casts are used to tag values. For instance when a newly-created string is bound to an untyped variable, it is cast as

[4] Interestingly, this corresponds to the valid assignment relationship $\Longleftrightarrow$ in Dart [7, §15.4].

$\langle \mathsf{Dyn} \Leftarrow \mathsf{String} \rangle$. The source type of the cast plays the role of the runtime type tag; in other words, a casted value is a boxed value. In Smalltalk, as in most dynamically-typed object-oriented languages, all values already carry along their runtime type, so this specific use of cast is not needed. This is why we use single-position casts as in Featherweight Java.

As a value flows in a program, casts can pile up. Different strategies exist to deal with chains of casts. They can be reduced as eagerly or as lazily as possible, yielding different flexibility/strictness tradeoffs [18]. Even in an eager approach, higher-order casts, *i.e.* casts on function types, cannot be fully resolved eagerly and typically imply wrapping functions in proxies that perform casts upon entry and exit. As noted by Herman *et al.*, this approach can result in unbounded growth in the number of proxies, affecting both space efficiency and tail call optimization [11]. They propose to use coercions instead of proxies so as to be able to combine adjacent coercions in order to limit space consumption. Going a step further, Siek and Wadler develop threesomes as a data structure and algorithm to represent and normalize coercions [17]. A threesome is a cast with three positions: source, target, and an intermediate lowest type. Combining a sequence of threesomes is done by taking the greatest lower bound of the intermediate types. In this work, we have focused on nominal object casts, which are not higher-order: they can be resolved immediately. In the future, we are going to investigate different implementation strategies for supporting structural object casts and casts on first-class closures.

A transversal issue when dealing with casts is whether or not blame tracking is done in order to report the guilty party whenever a cast fails [9, 18, 23]. There are different strategies for blame assignment, that may lead to blaming different parties for the same example [18]. Adding blame tracking complicates matters, both theoretically and practically. For instance, threesomes with blame is considerably more complex to understand than threesomes without blame [17]. In this paper, we have chosen to focus on the cast insertion strategies without blame first. The practical impact of blame tracking, in terms of both performance and actual help in debugging programs, is still an open question.

The expected overhead of fine-grained integration between typed and untyped code that gradual typing supports has led several researchers to develop alternative ways to do the integration. In Typed Racket, the granularity is at the module level: a module is either typed or untyped, but it cannot mix both disciplines internally [20]. This reduces the flexibility of the integration somewhat, but also reduces the cases of interaction, while proposing a reasonable engineering tradeoff. Interaction between typed and untyped code in Racket is mediated through contracts [21], with blame tracking.

Wrigstad *et al.* propose another approach to alleviate the performance issue of gradual types, integrated in Thorn [2, 24]. Instead of relying on the type consistency relation, they introduce a novel intermediate point between dynamic and static types: like types. When a method argument is declared to be of a like type, its uses in the method body are statically checked. But clients of the method can pass any value, just as if there was no declared static type. Conformance is checked dynamically. The Thorn compiler is able to aggressively optimize concrete types, and the authors report speedups of 2 to 4x between an untyped Thorn program and a fully typed one (with untyped libraries). In our case, in which we are retrofitting a gradual system on top of an existing dynamic language with the aim of being backwards compatible, it seems hard to obtain such speedups without working at the virtual machine level.

Chang *et al.* report on JIT-level optimizations based on optional type information [6]. Their work is in ActionScript, which is not gradual in the sense of Siek and Taha in that it does not rely on the consistency relation and does not support higher-order casts.

Also in the context of ActionScript, Rastogi *et al.* use local type inference to eliminate occurrences of the dynamic type and therefore augment the "static-ness" of programs [14]. They report very encouraging results: on average, they observe a 1.6x improvement with inferred types, and up to 5x in certain cases. We expect that developing a local type inference algorithm for Gradualtalk could significantly reduce the number of inserted casts.

Finally, Takikawa *et al.* develop a gradual type system for first-class classes in Racket [19]. They notice that the calculus of Siek and Taha does not deal with inheritance, and that a direct adaptation would be unsound (as discussed in Section 7.3). Furthermore, the inheritance semantics they support is rich in that it deals with accidental overridings. In order to address both requirements, they adopt row polymorphism instead of standard subtype polymorphism. To match these static typing features, on the dynamic checking side they propose opaque and sealed contracts. Consequently, mixins and other higher-order programming patterns with first-class classes can be checked soundly. A contribution of our work in this regard is to suggest that gradual typing with consistent subtyping is not necessarily unsound in presence of overriding: it depends on the cast insertion strategy. This being said, dealing with accidental overriding does require types to express missing members, a scenario for which row polymorphism seems necessary.

9. Conclusion and Perspectives

This paper studies different cast insertion strategies for a gradually-typed language with objects. Experiments are carried out in Gradualtalk, a gradually-typed Smalltalk, and focus mainly on performance. Starting from the direct implementation of the semantics specified by Siek and Taha [16], which we term the call strategy, we present two alternative strategies: one that inserts casts at the callee side, termed the execution strategy, and a hybrid strategy that combines ideas of the call and execution strategies. The execution and hybrid strategies build upon the fact that the language runtime is already safe, and that therefore we can discharge casts from their safety-bearing role in [16]. Microbenchmarks exhibit the best and worst cases of the call and execution strategy, and show that the hybrid strategy is effectively a best-of-both-worlds approach, always exhibiting a performance similar to that of the fastest strategy. A set of macrobenchmarks help us to further characterize the benefits of the hybrid strategy, which manifest themselves more clearly as more type annotations as added. We also compare the three strategies on different accounts than execution time. We report on the extra memory cost of two different versions of the hybrid strategy, which we believe are reasonable in view of the associated benefits. Considering modular compilation, the execution strategy is better, although the hybrid strategy allows to fine-tune the modularity/efficiency tradeoff. Finally, we discuss the interaction of these strategies with inheritance: the call strategy is unsound, the execution strategy is sound but sacrifices performance. We informally describe a way to adapt the hybrid strategy to retain soundness and still exploit static type information for optimization. Overall, this work suggests that the hybrid strategy is a promising approach to implement gradual typing in an existing dynamic language with a safe runtime.

There are many venues for future work. First, the hybrid strategy needs to be fully formalized in a context with overriding, in order to prove that it effectively retains soundness. On the implementation side, we are interested in exploring approaches other than source-level transformation: exploiting the intermediate representation of the compiler, or directly modifying the virtual machine. Language-wise, we need to tackle the case of first-class closures, and study the impact of adding blame tracking, especially considering the different semantics that have been proposed [18].

References

[1] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, Aug. 2013. Available online: http://dx.doi.org/10.1016/j.scico.2013.06.006

[2] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2009)*, pages 117–136, Orlando, Florida, USA, Oct. 2009. ACM Press.

[3] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, pages 1–6, 2004.

[4] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, pages 215–230, Washington, D.C., USA, Oct. 1993. ACM Press. ACM SIGPLAN Notices, 28(10).

[5] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 278–292, Toronto, Ontario, Canada, 1991.

[6] M. Chang, B. Mathiske, E. Smith, A. Chaudhuri, A. Gal, M. Bebenita, C. Wimmer, and M. Franz. The impact of optional type information on JIT compilation of dynamically-typed languages. In *Proceedings of the ACM Dynamic Languages Symposium (DLS 2007)*, pages 13–24, Montreal, Canada, Oct. 2007. ACM Press.

[7] Dart Team. Dart programming language specification, May 2013. Version 0.41.

[8] ESOP 2009. *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, York, UK, 2009. Springer-Verlag.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, Pittsburgh, PA, USA, 2002. ACM Press.

[10] M. Furr. *Combining Static and Dynamic Typing in Ruby*. PhD thesis, University of Maryland, 2009.

[11] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher-Order and Sympolic Computation*, 23(2):167–189, June 2010.

[12] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2):Article n.6, Jan. 2010.

[13] POPL 2010. *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2010)*, Madrid, Spain, Jan. 2010. ACM Press.

[14] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2012)*, pages 481–494, Philadelphia, USA, Jan. 2012. ACM Press.

[15] J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

[16] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in Lecture Notes in Computer Science, pages 2–27, Berlin, Germany, july/august 2007. Springer-Verlag.

[17] J. Siek and P. Wadler. Threesomes, with and without blame. In POPL 2010 [13], pages 365–376.

[18] J. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In ESOP 2009 [8], pages 17–31.

[19] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*, pages 793–810, Tucson, AZ, USA, Oct. 2012. ACM Press.

[20] S. Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs*. PhD thesis, Northeastern University, Jan. 2010.

[21] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *Proceedings of the ACM Dynamic Languages Symposium (DLS 2006)*, pages 964–974, Portland, Oregon, USA, Oct. 2006. ACM Press.

[22] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press.

[23] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In ESOP 2009 [8], pages 1–16.

[24] T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In POPL 2010 [13], pages 377–388.