Science of Computer Programming 96 (2014) 52–69

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Gradual typing for Smalltalk ☆

Esteban Allende [a,1], Oscar Callaú [a,*], Johan Fabry [a], Éric Tanter [a], Marcus Denker [b]

[a] PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Chile
[b] RMoD, Inria Lille – Nord Europe, France

HIGHLIGHTS

- A practical gradual type system for Smalltalk that supports a smooth path from untyped to typed code.
- A novel combination of several typing features, with some interesting interactions.
- A discussion of some implementation tradeoffs and challenges for a gradual type system in a "live" programming environment like that of Smalltalk.
- An initial validation of the type system through typing several Smalltalk projects.

ARTICLE INFO

Article history:
Received 12 November 2012
Received in revised form 3 May 2013
Accepted 14 June 2013
Available online 6 August 2013

Keywords:
Type systems
Gradual typing
Smalltalk

ABSTRACT

Being able to combine static and dynamic typing within the same language has clear benefits in order to support the evolution of prototypes or scripts into mature robust programs. While being an emblematic dynamic object-oriented language, Smalltalk is lagging behind in this regard. We report on the design, implementation and application of Gradualtalk, a gradually-typed Smalltalk meant to enable incremental typing of existing programs. The main design goal of the type system is to support the features of the Smalltalk language, like metaclasses and blocks, live programming, and to accommodate the programming idioms used in practice. We studied a number of existing projects in order to determine the features to include in the type system. As a result, Gradualtalk is a practical approach to gradual types in Smalltalk, with a novel blend of type system features that accommodate most programming idioms.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

The popularity of dynamic languages has led programs in these languages to become larger and more complex. A number of frameworks like Ruby on Rails, the Google JavaScript library suite, Seaside (Smalltalk), Django (Python), etc. are regularly used to build large and complex systems. In this context, the possibility to consolidate grown prototypes or scripts with the guarantees of a static type system is 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 [1–7].

Gradual typing [8,9] is a partial typing technique described by Siek and Taha. It proposes a type system that allows developers to define which sections of code are statically typed and which are dynamically typed, at a very fine level of

☆ This work is partially funded by FONDECYT Project 1110051, and the INRIA Associated Team PLOMO.
* Corresponding author.
E-mail addresses: eallende@dcc.uchile.cl (E. Allende), oalvarez@dcc.uchile.cl (O. Callaú), jfabry@dcc.uchile.cl (J. Fabry), etanter@dcc.uchile.cl (É. Tanter), marcus.denker@inria.fr (M. Denker).
[1] Esteban Allende is funded by a CONICYT–Chile Ph.D. Scholarship.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.06.006

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. Gradual typing has a strong theoretical basis and an ample design space. Industrially, it has been introduced in the ActionScript language [7].

Smalltalk [10] is the emblematic dynamic object-oriented language and has served as inspiration for many recent languages. Smalltalk is still used in industry, and the appeal of partial typing is attractive to many. The language with the most developed static type system for Smalltalk, Strongtalk [2] is not a gradual type system, but an *optional* one [3]. An optional type system does not influence the runtime semantics of the language and therefore does not enforce any guarantee about the type of values at runtime. This is very different from a gradual type system which does ensure that assumptions made by statically-typed code are not violated, and if they are, the faulty dynamic code is blamed accordingly [11]. The key to these guarantees is the insertion of runtime casts at the static/dynamic boundaries [8].[2]

Designing and implementing a gradual type system for Smalltalk is a challenging task because of the highly-dynamic nature of the language and the "live" programming environment approach. Indeed, Smalltalk is a reflective language that cannot be, in general, easily typed. Moreover, incremental programming in Smalltalk implies accepting partially-defined methods by the type system and to dynamically (at runtime) react to class updates. Additionally, as in any language, programmers rely on various programming idioms, some of which are challenging to type properly. These Smalltalk particularities make the design and implementation of a gradual type system a challenge in itself.

We report on the design, implementation and initial application of Gradualtalk,[3] a gradually-typed Smalltalk, which is fully compatible with existing code. Following the philosophy of Typed Racket [5], a major design goal of Gradualtalk is that the type system should accommodate existing programming idioms in order to allow for an easy, incremental path from untyped to typed code. The design of Gradualtalk was guided by a study of existing Smalltalk projects, incrementally typing them in Gradualtalk.

The type system of Gradualtalk supports Smalltalk idioms as much as possible through a number of features: a combination of nominal and structural types, union types, self types, parametric polymorphism, and blame tracking, amongst others. While there is no groundbreaking type system feature in Gradualtalk (and hence no formal description in this paper), the combination is quite novel, and the choice of these features—as well as their interactions—is carefully discussed across the paper. Furthermore, as an initial validation of the practicality of Gradualtalk, we report our findings on typing several existing Smalltalk projects.

Contributions. To summarize, this paper makes the following contributions:

- A practical gradual type system for Smalltalk that supports a smooth path from untyped to typed code: any Smalltalk program is a valid Gradualtalk program and type annotations can be added selectively per expressions.
- A novel combination of typing features, with some interesting interactions.
- A discussion of some implementation tradeoffs and challenges for a gradual type system in a "live" programming environment like that of Smalltalk.
- An initial validation of Gradualtalk through typing several Smalltalk projects.

Structure of the paper. We introduce Gradualtalk by examples in Section 2. We then refine it with several typing features in Section 3. Later, in Section 4, we explore subtyping and runtime coercion semantics. Section 5 discusses implementation challenges. We report on the practical experience of typing Smalltalk code in Section 6. We finally discuss related work (Section 7) and conclude (Section 8).

2. Gradual typing for Smalltalk

In this section, we present the Gradualtalk language, which is a Smalltalk dialect with gradual typing support. We now showcase the features of the language using as an example code snippets from a geometric calculation module.

2.1. From dynamically typed to gradually typed code

A developer is trusted with the development of the geometric calculation module for a graphics application. She starts writing dynamically-typed code. The following code snippets are the implementation of two example methods: euclidean distance and a class method for creating points.

```
Point≫ distanceTo: p
  |dx dy|
  dx := self x – p x.
  dy := self y – p y.
  ↑ (dx squared + dy squared) sqrt
```

[2] The guarantees of gradual typing do incur a runtime performance overhead. Gradualtalk makes it possible to deactivate runtime casts insertion, thereby turning Gradualtalk into an optional type system.

[3] Available online at http://www.pleiad.cl/gradualtalk.

```
Point class≫ x: aNumber1 y: aNumber2
  ↑self new x: aNumber1; y: aNumber2
```

After development and testing, the developer wants to increase robustness and provide basic (checked) documentation for these methods. For that purpose, she needs to type the method declarations of those methods. The following example is the typed version of the method distanceTo:.

```
Point≫ (Number) distanceTo: (Point) p
  |dx dy|
  dx := self x − p x.
  dy := self y − p y.
  ↑ (dx squared + dy squared) sqrt
```

The method declaration of this method specifies that the type of the parameter p is Point, while the return value type is Number. Because the local variables dx and dy are not annotated, they are treated as being of type Dyn, *i.e.* the type of any object in a dynamically-typed language.

Note that the Dyn type is also very helpful to type methods that cannot be otherwise be typed precisely, either because of a limitation of the type system, or because of inherent dynamicity. The typical example of the latter is reflective method invocation, done in Smalltalk with the perform: method:

```
Object >> (Dyn) perform: (Symbol)aSymbol
```

The argument to perform is a Symbol, which denotes the name of the method (selector) that must be invoked on the receiver object. In general, the return type cannot be statically determined. Declaring it as Dyn instead of Object means that clients of this method can then conveniently use the return value at any type, instead of having to manually coerce it.

2.2. Closures

The next method to type in our example is perimeter:. This method takes as parameter a closure that computes the distance between two points, and returns the value of the perimeter of the polygon, using the provided closure. Closures, also known as blocks, are a basic feature in Smalltalk, so the type system supports them. The following code is the typed version of the perimeter: method declaration:

```
Polygon ≫ (Number) perimeter: (Point Point → Number) metricBlock
  ...
```

In the example, the parameter metricBlock is a closure; its type annotation specifies that it receives two Points and returns a Number.

2.3. Self and metaclasses

The next method to type is y:. This method is a setter for the instance variable y. Its return value is self, the receiver of the message. The following code corresponds to its typed method implementation:

```
Point≫ (Self) y: (Number) aNumber
  y := aNumber.
```

Self is the type of self, as in the work of Saito et al. [12]. Declaring the return type to be Point would not be satisfactory: calling y: on an instance of a subclass of Point would lose type information and forbid chained invocations of subclass-specific methods.

We now consider the class method x:y:, which acts as a constructor:

```
Point class≫ (Self instance) x: (Number) aNumber1 y: (Number) aNumber2
  ↑self new x: aNumber1; y: aNumber2
```

Self instance is the type of objects instantiated by self. Self instance is therefore only applicable when self is a class or metaclass. This was inspired by the type declaration "Instance" in Strongtalk [2]. Using Self instance instead of Point brings the same benefits as explained above. Constructor methods are inherited, and Self instance ensures that the returned object is seen as an object of the most precise type. The dual situation, where an object returns the class that instantiated it, is dealt with using Self class, which is also inspired by Strongtalk.

Self instance in Gradualtalk and Strongtalk Instance are similar, but subtly different. The difference shows up when looking at the Class class, and related classes. Recall that in Smalltalk, classes are objects, instance of their respective metaclass, which derive from the Class class. The problem is that in Strongtalk, inside that class, the type Instance is a synonym of Self. This means that all methods defined in Class—and its superclasses ClassDescription and Behavior—lack a way to refer to the type of their instances. This limitation can be observed in several places. For example, the return type of Behavior ≫ \#new is Object in Strongtalk, which is imprecise, while it is Self instance in Gradualtalk. To type the method new correctly, Strongtalk needs to redefine new in the subclass Object class (the metaclass of Object), and change its return type to Instance. Another example of this problem is in the following method from Behavior:

```
Behavior≫ (Self) allInstancesDo: (Self instance → Object)aBlock
  "Evaluate the argument, aBlock, for each of the current instances of the  receiver."
```

Using Self instance above as the argument type of the block denotes any possible instance of a Behavior object. Properly typing this method is not possible in Strongtalk: as a consequence, it has been moved down the hierarchy to the Object class class. Self types in Gradualtalk are strictly more expressive than in Strongtalk.

2.4. Casts

The following code is the method perimeter, which computes the perimeter using the euclidean metric:

```
Polygon ≫ perimeter
  ↑ self perimeter: [:x :y| x distanceTo: y]
```

This dynamically-typed method invokes the perimeter: method with a (Dyn Dyn → Dyn) closure, yet this method expects a (Point Point → Number) closure. In the type system of Gradualtalk, the former closure type is *implicitly* cast to the latter. As a result, the developer does not need to write any type annotation.

The language also gives the programmer the option of explicitly coercing from one type to another type. An explicit cast is shown in the following:

```
Polygon ≫ perimeter
  ↑ self perimeter: [:x :y| (<Integer> x distanceTo: y)]
```

The return value of the expression "x distanceTo: y" is cast to an Integer. If it is not an Integer at runtime, a runtime exception is raised.

2.5. Blame tracking

Casts can fail. However, a higher-order cast (*i.e.* a cast that involves function types or, by extension, structural object types) cannot be verified immediately and therefore this check must be delayed [13,14]. This means that the point where an error is detected can be spatially and temporally decoupled from the actual source of the error. The general solution to this issue is to perform *blame tracking* [11]. When a check is delayed, the type system remembers information that will then allow blame to be properly assigned, pointing at the expression that is responsible for the error.

Consider the following:

```
PolygonTest ≫ testPolygon: b
  |block (Polygon)pol|
  block := (<Point Point → Integer> b).
  ...
  pol perimeter: block.
```

The cast of b to the proper function type cannot be checked immediately. Hence the type system ensures that clients of block use it properly by providing two Point arguments (otherwise they are to blame), and checks that the block effectively always return an Integer (otherwise the cast itself is to blame, because it would have failed if we were able to check it immediately).

3. Refining the type system

In this section, we extend the gradual type system introduced in Section 2. These extensions are a series of features that we found necessary while typing several Smalltalk projects. We conclude that these are necessary for a smooth migration of Smalltalk projects to a typed version that is as precise as possible. Indeed, since no type annotation means Dyn, any Smalltalk program is already a Gradualtalk program with only Dyn types. The objective is therefore to introduce some features that help minimize the number of required Dyn annotations in the code. As we show in this section, many Smalltalk programming idioms suggest a specific type system mechanism in order to avoid relying on Dyn.

Supporting programmer idioms is important for backward compatibility and seamless integration. The first simply is to maintain support for the legacy packages that are the core of today's Smalltalk programs. The second is because programmers should not have to refactor code to satisfy the type system.

In order to support these idioms, there are several typing features that can be useful. However adding features to the type system can significantly increase its complexity. Consequently, each feature we added was justified by our experience studying existing Smalltalk projects and typing them in Gradualtalk. The corpus of typed code we produced is discussed in more detail in Section 6. For now we solely highlight that it consists of 137 classes with a total of 3382 methods.

3.1. Parametric polymorphism

Consider the following piece of code, where an array of Dyn objects is defined:

```
|(Array) points|
...                    "filled with points"
(points at: 1) x       "potentially unsafe"
```

The programmer knows that any element of the array is a Point., and invokes the method x of class Point. Sadly, the type system cannot guarantee a safe method call at compile time, consequently a coercion is introduced by the type system. Here, the type information is lost, forcing the programmer to either use casts or the Dyn type. Casts need to be manually inserted, which is cumbersome and error prone.

To solve this problem, Gradualtalk supports parametric polymorphism [15]. Adding parametric polymorphism to gradually typed languages is not new: Ina and Igarashi [16] presented a formalization and initial implementation of generics for gradual typing in the context of Featherweight Java. We adopt their approach in Gradualtalk. As of now, generics are implemented using type erasure as in Java.

Gradualtalk includes a generically-typed version of the Collection library. For instance, the next piece of code solves the above problem by introducing Point as a type argument to the generic Array type:

```
|(Array<Point>) points|
...
(points at: 1) x       "safe call"
```

Below is an example of a generic method definition:

```
Collection<e> ≫ (a) add: (a <: e) newObject
```

This method inserts an object in a collection. Interestingly, in Smalltalk, the return value of this method is the added object. Therefore, in order to not lose type information, we use a bounded type variable a, subtype of the collection element type e, and specify a as the return type. Note that by convention, in Gradualtalk, type variables are single lowercase characters, similar to Haskell and ML.

Along with generics the type system also supports polymorphic functions (blocks in Smalltalk), which is useful in several cases, *e.g.* higher-order functions in collections:

```
Collection<e> ≫ (Self<f>) collect: (e → f) op
Collection<e> ≫ (Self<e>) select: (e → Boolean) pred
Collection<e> ≫ (f) inject: (f) init into: (f e → f) op
```

Note that the two first methods above use parametric self types to precisely type their return values.

Combining parametric types with some other typing features may produce new and interesting properties. For instance, the interactions between the Dyn type and generics, called bounded dynamic types [16], permits flexible bounded parametric types. Gradualtalk does not include this feature as of now, because we have not found conclusive evidence in practice that justifies it yet. Another interesting interaction occurs between self types and generics, called self type constructors [12], allowing programmers to parametrize self types. Self type constructors are required to properly type collections in Gradualtalk, and are therefore supported.

Occurrences. In the corpus, we declared 14 classes as being parametric, and in these classes the type parameters are used 246 times in their methods. These parametric types are used in 16 classes, 33 times in the methods of these classes.

3.2. Union types

The following piece of code is a polymorphic implementation of the ifTrue:ifFalse: method, where we use RT as a placeholder for the return type:

```
Boolean≫ (RT) ifTrue: (→ a) trueBlock ifFalse: (→ b) falseBlock
 ...
```

The method receives two block arguments, one for the true case named trueBlock, and one for the false case, named falseBlock. In this example each block can evaluate to a result with a different type. Because of this the trueBlock has return type a and the falseBlock has b, and a=b is not always the case. Consequently, at this moment there are two possible values for RT:

Object. The type Object does not provide any information to programmers. Even if we consider the lowest common ancestor between types A and B, still some type information is lost. Therefore, programmers are forced to insert a cast to get the real type.

Dyn. We get the flexibility that we need, but again type information is lost.

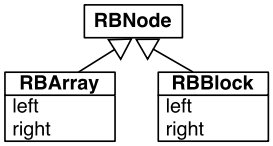

Fig. 1. A common structural protocol.

While this is a simple example, there are several places in the corpus where examples like this can be found. To solve this problem, we use *union types* [15]. These allow programmers to join several types in a single one, via disjunction. Union types are represented by | in Gradualtalk. A union between types a and b solves the problem of the example, letting the programmer specify that only one of these is possible.

```
Boolean≫ (a | b) ifTrue: (→ a) trueBlock ifFalse: (→ b) falseBlock
```

Another interesting example is the following method:

```
Collection<e> ≫ (Self | a) ifEmpty: (→ a) aBlock
  ↑ self isEmpty ifTrue: [ ↑aBlock value ] ifFalse: [ self ]
```

The method returns the result of the invocation of aBlock (of type a) if the collection is empty, or self otherwise. To type this precisely, a union type Self | a is used.

When using a variable typed with a union type a | b, the programmer can safely call common methods in a and b. Calling specific methods of a or b requires explicit disambiguation, for instance using isKindOf: to perform a runtime type check and then using a coercion. We are currently considering several flow-sensitive typing mechanisms to avoid having to explicit coerce values after a runtime type check [5,17,18].

Occurrences. In the corpus, union types were used in 19 classes, in 82 methods.

3.3. Structural types

Fig. 1 describes the RBNode hierarchy (RB is shorthand for Refactoring Browser) that represents abstract syntax tree nodes in a Smalltalk program. In the example, only the classes RBArray and RBBlock understand the selectors left and right.

Consider the following code that is added to handle brackets in the parser, where we use AT as a placeholder for the argument type:

```
RBParser ≫ bracketsOfNode: ( AT ) node
... node left.
... node right.
```

Consequently, there are three possible values for AT:

RBNode. RBNode is the common ancestor of RBArray and RBBlock. However any call to the methods left and right will be rejected by the type system, because RBNode does not define these methods. Even a cast will not help, because the type system cannot statically determine if either RBArray or RBBlock will be the correct type.

RBArray | RBBlock A union type could be a good solution. However it is not scalable if more nodes include brackets later on in development.

Dyn. The code will be accepted by the type system, but again type information is lost.

This problem appears because RBArray and RBBlock have no relation between them except from being nodes, and not all nodes have brackets. But RBArray and RBBlock also share a set of common methods used in the method bracketsOfNode:. Therefore, objects of type AT will understand this set of methods, *i.e.* the selectors left and right. A type with this structural representation, *i.e.* set of method types, is called a *structural type* [15]. This means that the type system permits as argument any object that understands the selectors in the structural definition. Using structural types, the solution is as follows:

```
RBParser ≫ bracketsOfNode: ({left (→ Integer) . right (→ Integer)}) node
```

The use of structural types allows programmers to explicitly specify a set of methods that an object must implement. These methods are the only available methods for the structurally-typed variable (node, in the above example) and therefore any call to another method will be invalid, unless a cast is used.

Type alias. The verbosity of structural types could be a problem for programmers, and even worse it can lead to an agglomeration of anonymous protocols. To solve this, Gradualtalk permits the use of a *type alias* [15], where programmers can give names to arbitrary types, in order to enhance readability. Note that the use of a type alias is not only restricted to structural types, for example Nil is a type alias for the UndefinedObject type.

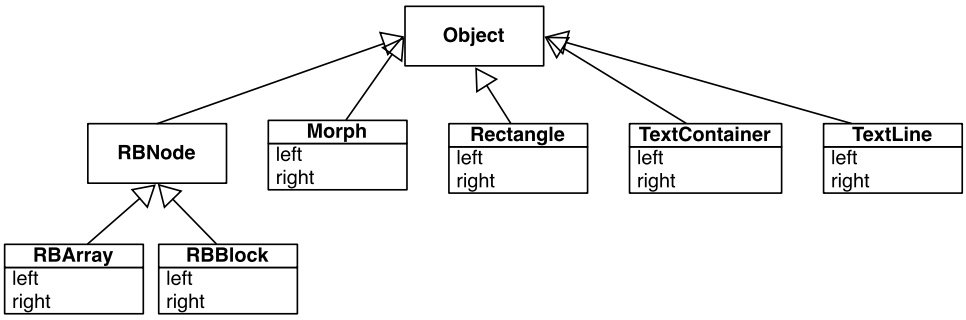

Fig. 2. A common structural protocol across projects.

Named protocols. Smalltalk does not support explicit interfaces or protocols. Instead, programmers rely on their understanding of what a given protocol is, and provide the necessary methods. For example consider the *pseudo* protocol "property", where the methods that handle properties in an object are listed:

```
propertyAt:
propertyAt:Put:
propertyAt:ifAbsent:
propertyAt:ifAbsentPut:
removeProperty:
removeProperty:ifAbsent:
```

Not making this protocol explicit is fragile, because it may evolve over time.

By combining a structural type and a type alias, programmers are able to define *named protocols*, which are similar to nominal interface types, except that they are checked structurally. With this, protocols are explicitly documented, and programmers can explicitly require them *e.g.* as an argument type of a method, without losing the flexibility of structural typing.

Note that a named protocol can serve to give a type to a trait [19]. However, traits come with a specific implementation, while named protocols are pure interface specifications. The same protocol can be implemented by different traits.

Occurrences. In the corpus, structural types are used in 12 classes, in 42 of their methods.

3.4. Nominal types

Nominal types [15] are the types that are induced by classes, *e.g.* an instance of class String is of type String. One of the primary advantages of nominal types is to help programmers to easily express and enforce design intent. Because of this, most mainstream statically-typed object-oriented languages support nominal types rather than other alternatives, such as structural types.

Nevertheless, structural types offer their own advantages [20,21]. For instance, structural types are flexible and compositional, providing better support for unanticipated reuse. This is because they imply a more flexible subtyping relationship compared to nominal subtyping, allowing unrelated classes in the class hierarchy to be subtypes. Taking this into account, some type systems [9,22,20] use structural types. In fact, a nominal type also can be considered in terms of its structural representation. This means that instances of class String have a structural type: the set of all methods that a string understands. Using such a type alias, programmers can benefit of the advantages of structural types.

In the case of Smalltalk, considering class-induced types as their structural representation is, however, not suitable. This is because Smalltalk classes tend to have a large amount of methods, which makes it impractical to comply with subtyping outside of the inheritance hierarchy. For instance, consider the SequenceableCollection class, which has hundreds of methods. If the programmer wants to define a subtype that is not a subclass she must implement all methods in the SequenceableCollection class. A solution is to combine structural and nominal types, as discussed next.

3.5. Reconciling nominal and structural types

Fig. 2 describes the hierarchy of some classes in Smalltalk that define the selectors left and right with type signature (→ Integer). With this new set of classes, the solution presented in Section 3.3 is not complete. This is because the type system will accept calls to the method bracketsOfNode: with a parameter that complies with the protocol, *e.g.* a Morph object, but which is not a node.

Table 1
Types in Gradualtalk.

τ	$::= \gamma \mid \overline{\tau} \to \tau \mid \tau + \tau \mid \gamma\langle\overline{\tau}\rangle \mid \gamma\sigma$	Type
γ	$::=$ ν | ϵ | x | Dyn	Ground type
ν	$::=$ C | C class |	Nominal type
ϵ	$::=$ Self | Self instance | Self class	Self type
σ	$::= \{\mathsf{m}\ \overline{\tau} \to \tau\}$	Structural type

Gradualtalk supports the combination of nominal and structural types, similar to Unity [20] and Scala [23].[4] A type combines both a nominal part and a structural part, as in A{m1...mn}. For instance, consider the following modification in the parameter type that takes structural and nominal types into account:

```
RBParser ≫ bracketsOfNode: (RBNode{left (→ Integer) . right (→ Integer)}) node
```

Here the type system is requesting an explicit RBNode object that has selectors left and right. Now a call with a Morph object as argument is rejected because it is not an RBNode.

Note that a nominal type A is a syntactic shortcut for the combined type A{} (empty structural component), while a structural type {m1, m2, ...} is the equivalent of Object{m1, m2, ...}.

Flexible protocols. Interestingly, the combination of the Dyn type with a structural type produces a *flexible protocol*, of the form Dyn {m1, m2, ...}. A flexible protocol represents objects that must comply with a protocol (structural part), but can otherwise be used with an implicit coercion (Dyn part). Consider the following piece of code:

```
Canvas≫ (Self) drawPoint:(Dyn {x(→ Integer). y(→ Integer)}) point
... point x. "safe call"
... point y. "safe call"
... point z. "not an error, considering point as Dyn"
```

The last statement does not raise a type error, because point has been typed with a flexible protocol. However, calling drawPoint: with an argument that does not support the {x, y} protocol is a static type error. Since Unity and Scala are not gradually-typed, flexible protocols are a novel feature of Gradualtalk.

Occurrences. In the corpus, we found 3 classes that use a nominal-structural type, in one method for each of these classes.

3.6. Safety and type soundness

Gradualtalk is based on Smalltalk, which is a *safe* language: sending unknown a message to an object is a trapped error that results in a MessageNotUnderstood exception, instead of producing unspecified result or system crash. Gradualtalk inherits this safety property.

With respect to type soundness, Gradualtalk follows the foundational work on gradual typing by Siek and Taha [9], with the blame assignment mechanism of Wadler and Findler [11]. The result is that Gradualtalk guarantees that, if a runtime type error occurs (that is, a MessageNotUnderstood exception is thrown), it is either due to an explicit cast that failed, or the consequence of passing an inappropriate untyped value to typed code. In the latter case, the error may occur anywhere in the code, but blame assignment will necessarily point to a faulty dynamically-typed expression that caused the error to occur later.

Note the practical value of blame assignment: without it, the programmer is left with the current call stack to investigate; however the root cause may be long gone and therefore not appear in the call stack.

3.7. Summary

Table 1 presents the grammar of types in Gradualtalk. C ranges over class names in the system, x ranges over type variables and m ranges over selector names. A bar over a type term denotes zero or more occurrences of the term.

A type τ is a either a ground type γ, a function type, a union type, a generic type, or a combined type with a structural component σ. A ground type is either a nominal type ν, a self type ϵ, a type variable or Dyn. A structural type σ is a list of selector types, including a selector name and a function type.

4. Type system semantics

We now describe three important aspects of the type system of Gradualtalk: self types, subtyping and runtime coercions.

[4] Note that because Scala compiles to the JVM, structural invocations introduce an extra performance penalty due to reflection. Gradualtalk, on the other hand, does not penalize structural invocations.

$\mathrm{instance}(\mathsf{Nil})=\mathsf{Nil}$
$\mathrm{instance}(\mathsf{C\ class})=\mathsf{C}$
$\mathrm{instance}(\mathsf{Metaclass})=\mathsf{Class}$
$\mathrm{instance}(\mathsf{A})=\mathsf{Object}$, if $\mathsf{Class} <: \mathsf{A} <: \mathsf{Behavior}$
$\mathrm{instance}(\mathsf{Self_C\ class})=\mathsf{Self_C}$
$\mathrm{instance}(\mathsf{Self_C})=\mathsf{Self_C\ instance}$, if $\mathsf{C} <: \mathsf{Behavior}$
$\mathrm{instance}(\mathsf{Self_C\ instance})=\mathrm{instance}(\mathrm{instance}(\mathsf{C}))$
$\mathrm{instance}(\gamma\sigma)=\mathrm{instance}(\gamma)$
$\mathrm{instance}(\tau_1+\tau_2)=\mathrm{instance}(\tau_1)+\mathrm{instance}(\tau_2)$
$\mathrm{instance}(\mathsf{x})=\mathrm{instance}(\mathrm{upperbound}(\mathsf{x}))$
$\mathrm{instance}(\gamma<\overline{\tau}>)=\mathrm{instance}(\gamma)$
$\mathrm{instance}(\mathsf{Dyn})=\mathsf{Dyn}$

Fig. 3. Definition of the instance relation on types.

$\mathrm{class}(\mathsf{Nil})=\mathsf{Nil}$
$\mathrm{class}(\mathsf{Object})=\mathsf{Behavior}$
$\mathrm{class}(\mathsf{C})=\mathsf{C\ class}$, if $\mathsf{C} \not<: \mathsf{Behavior} \wedge \mathsf{C} \neq \mathsf{Object}$
$\mathrm{class}(\mathsf{A})=\mathsf{A}$, if $\mathsf{A} \in \{\mathsf{Behavior},\mathsf{ClassDescription}\}$
$\mathrm{class}(\mathsf{Class})=\mathsf{Metaclass}$
$\mathrm{class}(\mathsf{Metaclass})=\mathsf{Metaclass\ class}$
$\mathrm{class}(\mathsf{Self_C\ class})=\mathrm{class}(\mathrm{class}(\mathsf{C}))$
$\mathrm{class}(\mathsf{Self_C})=\mathsf{Self_C\ class}$
$\mathrm{class}(\mathsf{Self_C\ instance})=\mathsf{Self_C}$
$\mathrm{class}(\gamma\sigma)=\mathrm{class}(\gamma)$
$\mathrm{class}(\tau_1+\tau_2)=\mathrm{class}(\tau_1)+\mathrm{class}(\tau_2)$
$\mathrm{class}(\overline{\tau}\to\tau)=\mathsf{BlockClosure\ class}$
$\mathrm{class}(\mathsf{x})=\mathrm{class}(\mathrm{upperbound}(\mathsf{x}))$
$\mathrm{class}(\gamma<\overline{\tau}>)=\mathrm{class}(\gamma)$
$\mathrm{class}(\mathsf{Dyn})=\mathsf{Dyn}$

Fig. 4. Definition of the class relation on types.

4.1. Self types

Although the semantics of the type Self are well known, this is not the case for the Self instance and Self class types. To define them properly, we define the concepts of instance types and class types. The instance type of τ is the type of objects instantiated by objects of type τ. If an object of type τ cannot have instances, then the instance type of τ is undefined. The class type of τ is the type of the class object that produces objects of type τ. Fig. 3 and Fig. 4 define the rules for instance types and class types respectively. Note that a key challenge in Smalltalk is to properly take into account the core classes that describe classes and metaclasses: Behavior, its subclass ClassDescription, and its subclasses Class and Metaclass.

A self type can be found in a calling context, as the type of a parameter or return value of an invoked method, or in called context, as the type of a variable or of the return value. In a calling context, self types are replaced by the type of the receiver. If the type of the receiver of the invoked method is τ, Self, Self instance and Self class are replaced by τ, instance(τ) and class(τ) respectively. In a called context, the type Self is represented in the type system as $\mathsf{Self_C}$, where C is the current class. Self instance and Self class are represented in the same manner.

4.2. Subtyping

One important feature in object-oriented languages is subtyping, by which an object of a given type can also be considered as being of any of its supertypes. The presence of several kind of types in Gradualtalk makes the subtyping relationship nontrivial. We next explain how it is treated.

Basic forms of subtyping. Lambda types, self types, union types and parametric types have well-known subtyping relationships [15,24,12,16]. Gradualtalk follows these rules. However, because of our extension to self types, there are two additional subtyping rules concerning self types:

$$(\text{Self instance})\frac{}{\mathsf{Self_C\ instance} <: \mathrm{instance}(\mathsf{C})} \qquad (\text{Self class})\frac{}{\mathsf{Self_C\ class} <: \mathrm{class}(\mathsf{C})}$$

Bottom type. In Gradualtalk Nil (which is an alias for UndefinedObject) serves as the bottom type. Since this type is a subtype of any other type, the programmer can use either nil or raising exceptions in any place where a typed object is expected.

Nominal and structural subtyping. As explained in Section 3.4, Gradualtalk supports the combination of nominal and structural subtyping as in Scala. First, note that Gradualtalk (as most mainstream languages) equates nominal subtyping with the inheritance relationship. Subtyping of mixed types is described by the following rule:

$$(\text{Mixed})\frac{\gamma_1 <: \gamma_2 \quad \mathrm{structural}(\gamma_1) \cup \sigma_1 <: \sigma_2}{\gamma_1\sigma_1 <: \gamma_2\sigma_2}$$

This rule states that a mixed type A {n1,...} is subtype of B {m1,...} if and only if A is a nominal subtype of B and the *union* of {n1,...} and all the methods of A (*i.e.* the structural view of A) is a structural subtype of {m1,...}. The definition of structural(.) is direct and omitted here for brevity.

Consistent subtyping. Gradual typing extends traditional subtyping to consistent subtyping *consistent subtyping* [9]. 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 [8]. A type τ_1 is a consistent subtype of τ_2, noted $\tau_1 \lesssim \tau_2$, iff either $\tau_1 <: \sigma$ and $\sigma \sim \tau_2$ for some σ, or $\tau_1 \sim \sigma$ and $\sigma <: \tau_2$ for some σ. The type system of Gradualtalk therefore operates based on the consistent subtyping relation.

Consistent subtyping does not present any specific challenge with respect to the different kinds of types in Gradualtalk. An interesting case to mention though is that of flexible protocols, since these are a novelty of Gradualtalk. Recall that a flexible protocol is a type of the form Dyn {m1,...}, *i.e.* a type that combines Dyn with a structural type. The consistent subtyping relation for flexible protocols is defined by rules Mixed-Dyn1 and Mixed-Dyn2:

$$\text{(Mixed-Dyn1)}\frac{\tau \lesssim \sigma}{\tau \lesssim \mathsf{Dyn}\ \sigma} \qquad \text{(Mixed-Dyn2)}\frac{}{\mathsf{Dyn}\ \sigma \lesssim \tau}$$

Mixed-Dyn1 states that τ is a consistent subtype of Dyn σ, if τ is a consistent subtype of σ. This rule makes explicit that τ must comply with the structural part of the flexible protocol. Mixed-Dyn2 states that Dyn σ is a consistent subtype of any τ. Indeed, it is valid to pass a value of type Dyn σ anywhere, since this is already the case with Dyn alone. Interestingly, both rules Mixed-Dyn1 and Mixed-Dyn2 correspond to two of the basic rules of consistent subtyping, $\tau \lesssim$ Dyn and Dyn $\lesssim \tau$, generalized to mixed types. Both of these basic rules are obtained when σ is the empty structure.

Note that flexible protocols also enjoy a direct subtyping relation as defined by the following rule Mixed-Dyn-sub:

$$\text{(Mixed-Dyn-sub)}\frac{\sigma_1 <: \sigma_2}{\mathsf{Dyn}\ \sigma_1 <: \mathsf{Dyn}\ \sigma_2}$$

Mixed-Dyn-sub states that Dyn σ_1 is a subtype of Dyn σ_2, if σ_1 is a subtype of σ_2. This rule is the generalization of the reflexive rule Dyn <: Dyn to mixed types; that rule can be recovered by considering both σ_1 and σ_2 empty.

4.3. Runtime coercion

Coercions are expression-level checkers introduced by the type system at compile time, see Section 2.4. There are two kinds of coercions: explicit and implicit. Explicit coercions are written by the programmer, while implicit coercions are introduced by the type system to guarantee soundness. Coercions are performed at runtime either nominally or structurally through subtyping.

Method invocation. Implicit coercions on method arguments can be performed either on the call site when sending the message, or at the beginning of the execution of the called method. We have chosen the latter option. This makes the transformation to insert implicit coercions more modular, which is an important criteria in a live programming environment like Smalltalk.

Union types. To reduce the penalty of performing a coercion at runtime, in the presence of union types these are simplified as follows: An object declared as a union type is implicitly coerced only in the calls to valid methods (methods in the union) that are not present in the intersection between the structural representation of the types in the union. Consider the following simplified example:

```
|({m1,m2} | {m2,m3}) obj|
obj m2. "no coercion needed"
(<{m3}>obj) m3. "manually coercion is needed"
```

In this example, the call to m2 is safe, however the call to m3 needs a manual coercion.

Structural types. Coercions to structural types can be performed in two ways:

Eager. These are performed when the code is executed, checking that the object with the structural type understands all messages that are defined in the type. A benefit of this is that it aims to express the users intention, because objects are forced to understand all methods specified by the programmer. However, it could reject some code that will never fail.

Lazy. These are performed on demand, checking only the used methods when they are invoked. This is more flexible, because it is not necessary to comply with the full set of methods, only a subset. However, it may be harmful, because some objects can accidentally match this subset of methods.

The choice between these options is hard to make and may depend on the context where these objects or types are used. We have chosen to use eager coercions in order to maximize the benefits of using static types in the first place. If practical experience ends up requiring the possibility to use lazy coercions, it can be integrated in the language, offering the choice to the programmer.

5. Implementation

The implementation of Gradualtalk extends Pharo Smalltalk, by adding a gradual type system. This extension consists primarily of three components: the core, the typechecker and the type dictionary. The core allows for the representation of

types and their relationships in Smalltalk. The typechecker is a pluggable extension to the Smalltalk compiler that verifies the correct typing of methods before compilation. The type dictionary is where type information is stored, *i.e.* the typing of instance variables and methods for each class. The current implementation of Gradualtalk is focused on expressiveness and correctness; performance is not (yet) a priority.

In this section, we present the problems we encountered while developing Gradualtalk that we believe are important to consider when implementing a type system in Smalltalk or a language with similar features.

5.1. Live system

Nearly every Smalltalk environment is a live system. This means that the developer writes the code, runs it and debugs it in the same execution environment. To support this live environment, individual methods can be compiled and added to an existing class. This is in contrast to other languages where the smallest compilation unit is a class. This feature of a live environment raises three problems for a typechecker.

The first problem is the granularity of the compiling process. In Smalltalk, the compilation process is done per method, instead of per class. Traditionally, a type checker prevents compilation when type errors are found. But with such a fine-grained compilation process, the traditional approach does not work. For example, if a programmer needs to define two mutually-dependent methods, when the first method is defined, the typechecker cannot know if the second method referenced is going to be defined later. The error should however not block the programmer from keeping this as-yet-buggy method and then define the second method. The same situation happens when loading code, since code loading in Smalltalk is just a script adding definitions one-by-one. In order to address this issue, we decouple the typechecking process from the compiling process: Gradualtalk can compile methods with type errors. Errors are collected in a separate typing report window.

The second problem is that the work done by the typechecker can become obsolete when new methods are introduced or an old method is modified. For example, if the return type of a method is changed from Integer to String, all methods that invoke it can potentially become ill-typed. To solve this problem, we introduce a dependency tracking system based on Ghosts [25], which allows the type system to properly support partially-defined classes and circular dependencies. Undefined classes and methods that are referenced are considered as ghost entities, about which type information is gathered. This allows the type system to check for consistent usage of as-yet-undefined entities. Dependency tracking considers both defined entities and ghosts. Each time the programmer updates or deletes definitions, the dependency tracker notifies the type system of which methods must be checked again. In case the type system detects some type errors, it reports the exact points of failure. More precisely, the dependency tracking system records bi-directional references between *dependents* and *dependees*. These dependencies are updated whenever a method is type-checked, and whenever the format of a class definition (variables) changes. The result of this process is a dependency graph of dependent and dependee nodes. A dependent node is either a pair (*class*, *selector*), for dependent methods, or a pair (*class*, *variable*), for dependent instance or class variables. A dependee node is either a class, for type related dependencies, or a pair (*class*, *selector*) for method invocation dependencies. Whenever a dependee node is updated, all dependents are re-checked and re-compiled (necessary because implicit cast insertion may have to change).

The third problem occurs when compiling typed system code that is critical. It is common that programmers commit errors when typing code, specially if it was not developed by them. In normal code, it is not a problem that a method fails when compiling, or cast errors are raised when they are executed. However, in critical code, having cast errors is fatal. For example, if the default error handler raises a cast error, an infinite loop is produced and the system is irresponsive, making it impossible to use the debugger. To address this problem, in Gradualtalk runtime casts insertion and checking can be disabled or enabled at will. To gradually type important and critical system parts, we used this feature to first focus on debugging the cause of typecheck errors at compile time, then progress to runtime cast errors. Also, disabling runtime casts after a cast error is raised allows us to use the debugger without further interference of the type system.

Gradual or optional? Disabling runtime casts insertion was built in Gradualtalk to address the problem discussed above. Interestingly, it can also be used to make the type system of Gradualtalk an *optional* type system, just like that of Strongtalk. Moreover, because code instrumentation can be enabled or disabled at will, Gradualtalk allows optionally-typed and gradually-typed code to co-exist in the same system; a combination which, to the best of our knowledge, has not been explored so far.

5.2. The untouchables

Smalltalk permits a programmer to modify existing classes, including the structure of their instance variables, with a few exceptions. The classes whose object instance structure cannot be touched are Class, CompiledMethod and any of their ancestors. The reason for this is fundamental: the VM needs to know statically how they are composed so it can realize method lookup and execute Smalltalk code without relying on the method lookup being implemented as Smalltalk code. Because of this restriction, we need to put the type information of methods and instance variable outside of the class, in the type dictionary.

There also is a set of virtually untouchable methods, *e.g.* whileFalse: in BlockClosure. These methods can be modified by a developer, however, the VM ignores those changes. This is because the compiler optimizes methods that invoke them with a

set of label and jump bytecodes that implement the original behavior of these methods. Because of this, we cannot modify them to realize coercions on their parameters. Instead, we insert coercions at their call sites.

5.3. Fragile classes

The type system uses nominal subtyping, as described in Section 4.2. For testing if a class is a subclass of another one, we use the tools provided by the Smalltalk environment. However, there is a time where using those tools can be detrimental: when the structure of the class is changing. For example, when changing the structure of the class Point, the environment in that moment can either answer that ColorPoint is or is not one of its subclasses.

The problem is that when changing the class structure, a typecheck is *required* to be performed on all the methods of the class and its subclasses. This is needed to verify that when an instance variable is removed, it is unused in the class or subclasses.

The above is not the only part where classes are fragile. Interrupting the process of modification of a class can have far-reaching consequences. For example, if the typechecker throws a type error, ColorPoint can become a subclass of a pseudo-Point that appears exactly like the class Point, has the same name, but nonetheless is a different class. This makes ColorPoint effectively not a subtype of Point. The developer has no easy way to be aware of this without the knowledge that the class is corrupted.

Both of these problems can be solved with two actions: separating the typechecker from the compilation process when a class is being modified, and recording the structure of the actual class hierarchy. This allows to simulate the change on the copy of the structure, and perform the typecheck on that copy without changing the current environment. If the typecheck fails the actual hierarchy remains unchanged.

6. Practical experience

In this section, we present the result of the validation of Gradualtalk using a corpus of seven existing projects. First, in Section 6.1 we present the quantitative results. Afterward, we present the qualitative results of the validation, in the form of bugs and optional refactoring (Section 6.2), interesting typed methods (Section 6.3) and challenging methods to type (Section 6.4).

6.1. Corpus and overview of findings

Corpus. The corpus we study is composed of seven projects: Kernel, Collections, Gradualtalk, Ghosts, AST-Core, Zinc and Spec. Kernel and Collections are both sub-projects of Pharo Smalltalk. The first provides the basic classes of Smalltalk, *e.g.* Object, Class, Integer, ClassDescription, Behavior, etc. The second set of classes we typed are the fundamental classes of the Collections framework in Smalltalk: Collection and SequenceableCollection. Gradualtalk is the implementation of the type system described in this paper. Ghosts [25] is an IDE tool for supporting incremental programming through automatic and non-intrusive generation of code entities based on their usage. AST-Core is a set of classes that allows to produce abstract syntax trees of Smalltalk methods. Zinc is a framework that implements the HTTP networking protocol. Finally, Spec is the new standard framework to declaratively specify user interface components in Pharo.

Kernel, Collections and AST-Core were included in the corpus because of their maturity. Moreover, Kernel is a challenging package because it contains the core classes of the system and Collections are a typical benchmark for type systems. Gradualtalk, Zinc, Spec and Ghosts were included because these are libraries and tools that we are familiar with.

The corpus is composed of 137 classes, with 3382 methods that have been typed (18 780 LOC). Although the classes in the corpus are a fraction of all classes in these projects, we believe that this corpus has a sufficient size for validating the practicality of the type system in different scenarios and kinds of projects.

Overview. Table 2 presents a quantification of the corpus composition and how much dynamically typed they are. The measure we use to calculate how much dynamically typed are the classes is the percent of Dyn types present in type annotations in the classes compared with the total. Using this measure, the more statically-typed project is Collections, with only a 5.46% of Dyn types. In contrast, the project more dynamically typed is AST-Core, with 25.49% of Dyn types. The difference between these numbers reflects two different stages of a migrating untyped code to typed one. The Collection framework is a well-studied case that is mostly typable when parametric polymorphism is supported. AST-Core has a significant portion of Dyn several AST-Core classes use classes and methods in the Smalltalk image that are not typed yet.

Table 3 presents the usage of the kinds of types in the projects of the corpus. It is included to provide more details for the numbers we presented in Section 3.

6.2. Bugs and refactoring

Bugs found. When typing the corpus, we found three bugs in the Kernel using Gradualtalk. The three bugs are present in Pharo Smalltalk 1.4 version #14438, and thanks to the feedback provided by our typing effort have since been fixed. Although this is a small quantity of bugs, the code being typed is very mature and hence can be expected to have a low number of bugs. That we still encountered bugs illustrates the advantages of typing code using Gradualtalk.

Table 2
Projects typed with Gradualtalk, * indicates these are not all classes of the project.

Projects	Classes	Methods	Dyn types	Non-Dyn types	Percentage Dyn types	Typed LOC	Total LOC
Gradualtalk	14*	294	69	657	9.50%	1116	3006
Ghosts	9	112	15	203	6.88%	338	338
AST-Core	17	579	402	1175	25.49%	2335	2339
Zinc	41*	452	192	749	20.40%	1733	1833
Collections	2*	305	113	1955	5.46%	2596	16292
Kernel	16*	1290	652	3439	15.94%	9319	24161
Spec	38*	350	174	776	18.31%	1343	2913
Total	137	3382	1617	8954	15.30%	18780	50882

Table 3
Usage of types in methods and classes.

Projects	Parametric type		Union type		Structural type		Structural-nominal	
	Classes	Methods	Classes	Methods	Classes	Methods	Classes	Methods
Gradualtalk	0	0	0	0	0	0	0	0
Ghosts	0	0	2	3	1	1	0	0
AST-Core	4	8	0	0	1	2	1	1
Zinc	1	3	4	8	3	20	0	0
Collections	2	197	2	39	1	1	1	1
Kernel	6	36	4	15	5	17	1	1
Spec	4	115	7	17	1	1	0	0
Total	18	359	19	65	12	42	3	3

The first bug was found in the method silentlyValue in BlockClosure. The bug consists in the call to a method that does not exist, leading to an error at runtime. The bug was introduced because a method in Object was recently removed. However, there are classes in the system which still implement this method, making it difficult to statically detect this error without a typechecker.

The second bug was found in the method putOn: in Magnitude. The bug consists in that certain types of objects cannot be sent on a binary stream. The problem is that this method requires that all subclasses of Magnitude would implement the method asByteArray, while this is the case only for one subclass.

The last bug was found in the method organization in ClassDescription. The problem is that it tries to invoke a recovery method when it finds a certain type of exception. However, that recovery method does not exist anywhere.

Refactoring. One of the main goals of Gradualtalk is to adapt itself to Smalltalk programming idioms. The typechecker does not require changes to the source code for typing, and many idiomatic uses of Smalltalk can be statically typed. Yet this still means that the programmer may need to resort to the dynamic type in some cases.

To make an existing code base more suitable for defining static types, a number of additional changes may be performed, ranging from small changes up to a complete redesign. Here we present three simple, optional changes to increase the amount of code that can be straightforwardly statically typed.

The first change consists of always adding a return to methods that raise an exception. The following code snippet shows an example of this case:

```
ClassDescription ≫ definition
  self subclassResponsibility
```

Because the method subclassResponsibility never returns normally, it does not matter what the method does after the invocation. However, following the Smalltalk semantics, for the typechecker this method returns self, because it does not have any information to know that the last statement never returns. The solution is to make the code explicitly return self subclassResponsability and annotate the return type of this method to be the expected return type of the concrete implementations in subclasses. This typechecks because the return type of self subclassResponsability is the bottom type.

A second, related change is adding abstract methods to classes when there is an implicit common selector between subclasses. This change is recommended because it indicates which methods need to be implemented in subclasses. However, if the developer does not want to implement this abstract method, she can use a Union type.

The third change is not to use "#()" to instantiate empty ordered collections, as this instantiates an Array object instead. Although this expression is shorter to write, and quite common throughout the code we typed, array has one important difference with ordered collections: its size cannot be changed. This already raises issues in Smalltalk, since when the developer tries to add an element to this object it throws an exception. The result is that there are potential errors hidden throughout the code. This is currently treated by, when adding to a collection, first invoking asOrderedCollection to obtain

a guaranteed expandable collection. We believe that enforcing this change of idiom would make having these guarantees more explicit and make typing these values easier. The latter is because it would no longer require the use of a Union type of OrderedCollection and Array.

6.3. Interesting illustrations of Gradualtalk

We now describe a couple of methods from the corpus that showcase the combination of features of Gradualtalk.

Object ≫ #at:modify:.

```
(a) at: (Integer)index modify: (Dyn→a)aBlock
    "Replace the element of the collection with itself transformed by the block"
    ↑ self at: index put: (aBlock value: (self at: index))
```

This method shows the usefulness of the Dyn type. The parameter aBlock is a closure that receives as a parameter the i-th element in the collection and returns the new element to be stored instead. However, the type of the originally stored element is unknown in Object. Declaring aBlock as (Object→a) would require the use of casts every time this method is used. This is the reason aBlock is typed as (Dyn→a).

Object ≫ #caseOf:otherwise:.

```
(a|b) caseOf: (Collection<Association<→Object, →a>>)aBlockAssociationCollection otherwise: (→b)aBlock
    "The elements of aBlockAssociationCollection are associations between blocks.
    Answer the evaluated value of the first association in aBlockAssociationCollection
    whose evaluated key equals the receiver.  If no match is found, answer the result
    of evaluating aBlock."
  aBlockAssociationCollection associationsDo:
      [:(Association<→Object, →a>)assoc | (assoc key value = self) ifTrue: [↑assoc value value]].
  ↑ aBlock value
```

This method is the Smalltalk version of the switch statement of Java or C++. This interesting method shows various features of Gradualtalk being used, like type parameters (a), union types (a|b), function types for blocks (→a) and generic types (Association<→Object, →a>).

6.4. Typing challenges

When typing the corpus, we found some challenging methods to type using the current features of Gradualtalk. We now discuss some of them.

BlockClosure ≫ #whileFalse:

```
(Nil) whileFalse: (→Object) aBlock
    ...
```

This is one of the basic control methods in Smalltalk.[5] This method is problematic for Gradualtalk, because it has conditions for its invocation, namely that the receiver must be a block of type (→Boolean) to be valid. In any other kind of block, invoking this method raises an exception. However, in Gradualtalk we cannot declare that a method in a given class can only be invoked on a subset of its instances. A possibility is support a form of typestate checking [26] but there is not enough evidence so far that such a feature would be sufficiently useful to warrant the added complexity.

Number ≫ #+

```
(Dyn) + (Number)aNumber
    "Refer to the comment in Number + "
    aNumber isInteger ifTrue:
        [self negative == aNumber negative
            ifTrue: [↑ (self digitAdd: (<Integer>aNumber) normalize)]
            ifFalse: [↑ self digitSubtract: (<Integer>aNumber)]].
    ↑ aNumber adaptToInteger: self andSend: #+
```

The method + is particularly challenging to type. The ideal type for this method would be one that could represent the type relation between receiver, argument and return shown in Table 4.

However, this ideal is not expressible in Gradualtalk. Consider the case where the receiver is an Integer. We could type Integer ≫ + with four different types: Number→Dyn, Integer→Integer, Number→Number or (a <: Number)→a. The first type works, but it loses type information. The second type would reject the expression 2 + 3.5, requiring the manual coercion

[5] The curious reader may wonder why the method returns Nil instead of Self. The reason is that it is a special method that is inlined at call sites, where self is not bound to the block object.

Table 4
Type relation for Number ≫ #+. Rows correspond to the receiver type, columns correspond to the argument type and each cell value is the corresponding return type.

	Float	Fraction	SmallInteger	LargeInteger
Float	Float	Float	Float	Float
Fraction	Float	Fraction|Integer[1]	Fraction	Fraction
SmallInteger	Float	Fraction	Integer[2]	Integer[2]
LargeInteger	Float	Fraction	Integer[2]	Integer[2]

[1] Fractions are automatically simplified to Integer if applicable.
[2] The size of the integer is not guaranteed to be maintained.

of 2 to a Float. The third type would require to add an explicit cast whenever the program does an arithmetic operation on Integers and stores it in a variable of type Integer (something which, not surprisingly, happens very often). Typing Integer≫+ as (a <: Number)→a is not correct either. Consider:

```
|(SmallInteger)x (LargeInteger)y|
y := (2 raisedTo: 10000).
x := y + 0.
```

In the expression y+0, y is the receiver, and the argument, 0, is a SmallInteger. So the type of y+0 would be SmallInteger, whereas numerically it clearly is not.

Number ≫ #to:by:do:

```
(Nil) to: (Number)stop by: (Number)step do: (Dyn→Object) aBlock
   "Normally compiled in-line, and therefore not overridable.
   Evaluate aBlock for each element of the interval (self to: stop by:
step)."
   | (Number)nextValue |
   nextValue := self.
   step = 0 ifTrue: [self error: 'step must be non-zero'].
   step < 0
      ifTrue: [[stop <= nextValue]
            whileTrue:
               [aBlock value: nextValue.
               nextValue := nextValue + step]]
      ifFalse: [[stop >= nextValue]
            whileTrue:
               [aBlock value: nextValue.
               nextValue := nextValue + step]].
      ↑nil.
```

This method is the Smalltalk "for" statement with steps. This method is problematic to type because the method Number ≫ #+ is difficult to type. The ideal type of this method is:

$$\text{Number} \quad (a <: \text{Number}) \quad ((\text{Self}|x) \rightarrow \text{Object}) \quad \rightarrow \text{Nil}$$

with x being the type of (self+step). Typing aBlock as (Number→Object) would force the Smalltalk programmer to either use only (Number→Object) or (Object→Object) closures, or add a cast to (Number→Object) in the closure. Our actual typing of aBlock (Dyn→Object) does not enforce correctness in this argument in the usage of the method, but does preserve usability.

Object ≫ #as:

```
(Dyn) as: (Object class)aSimilarClass
   "Create an object of class aSimilarClass that has similar contents to the receiver."
   ↑ aSimilarClass newFrom: self
```

This method creates a new object using the provided class and based on the contents of the receiver. This method is problematic to type in Gradualtalk, because the return type depends directly on the argument. However, this dependency relationship is instantiation. The ideal type of this method is (a→x), where x is the type of an instance of a. Typing this method as (Object class→Object), results in the programmer being required to add a cast to this expression, even if it would be obvious, making the code more verbose. The following code snippet is an example of this "obvious" cast:

```
|(ColorPoint)cp (Point)p|
cp := (<ColorPoint> p as: ColorPoint).
```

Currently, this method is typed as (Object class → Dyn), which removes the need for explicit casts, but it does not prevent misuse of the return value. A proper precise typing would be (a → a instance), but this is not (yet) supported in Gradualtalk.

7. Related work

There has been a lot of research on migrating dynamic languages to a typed equivalent. In this section, we compare Gradualtalk with the most representative practical partial type systems available today. At the end of this section, we mention other interesting type systems and partial typing techniques.

Strongtalk. Strongtalk [2] is a well-known statically-typed Smalltalk dialect that incorporates several typing features. The Strongtalk type system is optional: it does not guarantee that the assumptions made in statically-typed code are respected at runtime. There are two major versions of Strongtalk. The first one relies on a structural type system using brands (named structural protocols). The second version abandons brands and uses declared relations to determine subtyping. The main reason reported by Bracha for this change is the fact that structural types do not appropriately express the intent of the programmer, and are difficult to read, especially when debugging [27]. Strongtalk supports parametric polymorphism and self types. In contrast to Gradualtalk, *"Strongtalk is not designed to type Smalltalk code without modifications"* [2]. Pegon [28] is a recent optional type system for Smalltalk, inspired by Strongtalk. It includes all typing features of Strongtalk (including the conflation of Instance and Self in Class and related classes that we discussed in Section 2.3), and adds type inference.

The objective of Gradualtalk is to be a gradually-typed language, while both Strongtalk and Pegon are optionally-typed: they do not enforce any guarantees at runtime about the types of values [3]. However, as we discussed earlier, Gradualtalk can also be used like an optional type system, by deactivating the runtime casts insertion and checking. In addition, there are subtle differences: in Gradualtalk nominal subtyping is equated to inheritance (Section 4.2), the semantics of self types is refined (Section 2.3), and protocols are implicit (Section 3.3).

Typed Racket. Typed Racket [29] is an extension to Racket in order to support statically-typed Racket programs. Typed Racket provides smooth and sound interoperability with untyped Racket [5], by using contracts at the boundaries. Gradualtalk was directly inspired by Typed Racket in the sense the type system should be flexible enough to support existing programmer idioms. Typed Racket includes several interesting features, such as union types, occurrence typing, first-class polymorphic functions and local type inference. It is designed for the functional core of the Racket language.

The most notable difference between Gradualtalk and Typed Racket is the granularity of typed and untyped code boundaries. In Typed Racket, the boundary is at the module level: a whole module is either entirely typed or not at all. Expression-level boundaries are more costly but more flexible, in that it is possible to statically type portions of a class while leaving a few difficult expressions typed dynamically. Accordingly to this design philosophy, Typed Racket does not support explicit type casts. The limitations of the per-module approach have been reported by Figueroa et al. in an experiment to implement a monadic aspect weaver in Typed Racket; they had to resort to the top type Any everywhere in their system [30].

DRuby. DRuby [6] is an optional static type system for Ruby, which uses type inference. Programmers can annotate their code, such as methods, and DRuby will checks these annotations using runtime contracts on suspicious code, *i.e.* DRuby infers type to discard well-typed code. DRuby includes union and intersection types, structural types (called object types), parametric polymorphism and self types, among others. Furthermore, DRuby introduces a novel dynamic analysis to infer types in highly-dynamic language constructors, *i.e.* the use of eval, send and missing_method functions.

Although the type systems of Gradualtalk and DRuby are very similar, despite the language differences, there are some notable differences. Gradualtalk does not infer types (yet). Dynamic language operators, such as send (perform in Smalltalk), are dealt with using Dyn in Gradualtalk. While Ruby has proper classes as objects and class methods, DRuby does not support the notions of Self instance and Self class. This means that constructor (class) methods cannot be precisely typed, nor can the uses of class. Finally, DRuby is not a partial type system, so not all Ruby programs are valid DRuby programs.

Other type systems. The previous type systems are just a few examples of types systems for dynamically-typed languages. There are several other proposals for Smalltalk [31–35], although Strongtalk is the most representative and complete. Some [31–33] are prior to Strongtalk, and they do not include all necessary features to type Smalltalk programs, as Strongtalk does. Haldiman et al. [34] present a practical approach to pluggable types implemented in Smalltalk where only code that contains partial type annotations are type checked using type inference and traditional static type checking. Finally, Pluquet et al. [35], present RoelTyper, a type reconstruction tool for Smalltalk that infer possible nominal types for variables (instance, temporary and argument variables, as well as return method) with an accuracy of 75%.

In addition, type systems have been developed for other dynamic languages, for instance for Python, JavaScript and ActionScript. RPython (Restricted Python) [36] is a statically-typed *subset* of Python, in which some dynamic features (*e.g.* dynamic modifications of classes and methods) have been removed. JS_0 [37] is a statically-typed version of JavaScript with inference, where both dynamic addition of fields and method updating are supported. ActionScript is one of the first languages used in the industry to embrace gradual typing, and efforts have been made to optimize it using local type inference [7]. Gradualtalk could certainly benefit from this technique.

Other partial typing techniques. Integrating static and dynamic type checking is a highly active area of research. Gradual typing, defined by Siek et al. [8,9], is just one of several partial typing techniques developed in the literature. Gradualtalk is based on Siek and Taha's gradual typing approach. Some other popular partial typing techniques are soft typing [1], pluggable types [3], hybrid typing [4] and like types [38]. Soft typing is a static type system that does not reject potentially

erroneous programs, but inserts casts to ensure safety. Hybrid typing combines static typing with refinement types. An automated theorem prover is used to check consistency, and run-time checks are inserted where static inconsistencies are detected. Wrigstad et al. introduce like types in the scripting language Thorn, as an intermediate ground between the Dyn type and standard static types.

8. Conclusion

Providing a practical gradual type system that supports programmer idioms in a highly-dynamic language such as Smalltalk is a complex task. This requires meeting several goals: a type system design that properly supports common programmer idioms without unduly increasing its complexity; dealing with implementation tradeoffs for such a type system in a live environment; and a validation of the system by typing a nontrivial corpus of code.

In this paper, we have introduced Gradualtalk, a practical gradually-typed Smalltalk that successfully meets the above challenges. The type system of Gradualtalk combines several state-of-the-art features, such as gradual typing, unified nominal and structural subtyping, self type constructors for metaclasses, and blame tracking. Gradualtalk is designed to ease the migration of existing, untyped Smalltalk to typed Gradualtalk code. We have motivated the existence of the different features in the type system by showing their use in a corpus of seven Smalltalk projects, and provided an initial validation of the usefulness of Gradualtalk in general by typing 3382 methods in 137 classes in this corpus.

While we will continue to add types to more existing libraries ourselves, the most interesting feedback on the usefulness of Gradualtalk will come from the user community, which will allow us to refine the selection of type system features and deepen our understanding of how certain features are used (or not). We will study if other mechanisms, such as flow-sensitive typing, are required.

The most pressing challenge to ensure the wide adoption of Gradualtalk is performance. Based on our experience so far, gradually-typed applications run significantly slower than their dynamically-typed version. We are currently closely investigating performance issues. While certain techniques to optimize gradual typing have been proposed [39,40], it is unclear yet which techniques are most effective in the specific context of Smalltalk, and if it is necessary to devise new optimization strategies for this context.

References

[1] R. Cartwright, M. Fagan, Soft typing, in: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91, 1991, pp. 278–292.
[2] G. Bracha, 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 93, ACM, Washington, DC, USA, ACM SIGPLAN Not. 28 (10) (1993) 215–230.
[3] G. Bracha, Pluggable type systems, in: OOPSLA Workshop on Revival of Dynamic Languages, 2004, pp. 1–6.
[4] K. Knowles, C. Flanagan, Hybrid type checking, ACM Transactions on Programming Languages and Systems 32 (2) (2010), Article no. 6.
[5] S. Tobin-Hochstadt, Typed scheme: From scripts to programs, Ph.D. thesis, Northeastern University, Jan. 2010.
[6] M. Furr, Combining static and dynamic typing in Ruby, Ph.D. thesis, University of Maryland, 2009.
[7] A. Rastogi, A. Chaudhuri, 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, ACM Press, Philadelphia, USA, 2012, pp. 481–494.
[8] J. Siek, W. Taha, Gradual typing for functional languages, in: Proceedings of the Scheme and Functional Programming Workshop, 2006, pp. 81–92.
[9] J. Siek, W. Taha, Gradual typing for objects, in: E. Ernst (Ed.), Proceedings of the 21st European Conference on Object-Oriented Programming, ECOOP 2007, in: Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, 2007, pp. 2–27.
[10] A. Goldberg, D. Robson, Smalltalk-80: The Language and Its Implementation, Addison–Wesley, 1983.
[11] P. Wadler, R.B. Findler, Well-typed programs can't be blamed, in: ESOP 2009 [41], pp. 1–16.
[12] C. Saito, A. Igarashi, Self type constructors, in: Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, OOPSLA '09, ACM, New York, NY, USA, 2009, pp. 263–282.
[13] R.B. Findler, M. Felleisen, Contracts for higher-order functions, in: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, ACM Press, Pittsburgh, PA, USA, 2002, pp. 48–59.
[14] J. Siek, R. Garcia, W. Taha, Exploring the design space of higher-order casts, in: ESOP 2009 [41], pp. 17–31.
[15] B.C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, USA, 2002.
[16] L. Ina, A. Igarashi, Gradual typing for generics, in: Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2011, ACM Press, Portland, Oregon, USA, 2011, pp. 609–624.
[17] S. Tobin-Hochstadt, M. Felleisen, Logical types for untyped languages, in: Proceedings of the 15th ACM SIGPLAN Conference on Functional Programming, ICFP 2010, ACM Press, Baltimore, Maryland, USA, 2010, pp. 117–128.
[18] A. Guha, C. Saftoiu, S. Krishnamurthi, Typing local control and state using flow analysis, in: G. Barthe (Ed.), Proceedings of the 20th European Symposium on Programming, ESOP 2011, in: Lecture Notes in Computer Science, vol. 6602, Springer-Verlag, 2011, pp. 256–275.
[19] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: L. Cardelli (Ed.), Proceedings of the 17th European Conference on Object-Oriented Programming, ECOOP 2003, in: Lecture Notes in Computer Science, Springer-Verlag, Darmstadt, Germany, 2003, pp. 248–274.
[20] D. Malayeri, J. Aldrich, Integrating nominal and structural subtyping, in: J. Vitek (Ed.), Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP 2008, in: Lecture Notes in Computer Science, Springer-Verlag, Paphos, Cyprus, 2008, pp. 260–284.
[21] D. Malayeri, J. Aldrich, Is structural subtyping useful? An empirical study, in: ESOP 2009 [41], pp. 95–111.
[22] D. Doligez, A. Frisch, J. Garrigue, D. Rémy, J. Vouillon, The OCaml system release 3.12, Institut National de Recherche en Informatique et Automatique, Jul. 2011, http://caml.inria.fr/pub/docs/manual-ocaml/index.html.
[23] É.P.F. de Lausanne (EPFL), Scala, http://www.scala-lang.org.
[24] L. Cardelli, Type systems, in: A.B. Tucker (Ed.), The Computer Science and Engineering Handbook, CRC Press, 1997, pp. 2208–2236 (Ch. 103).
[25] O. Callaú, É. Tanter, Programming with ghosts, IEEE Software 30 (1) (2013) 74–80.
[26] R. Wolff, R. Garcia, É. Tanter, J. Aldrich, Gradual typestate, in: M. Mezini (Ed.), Proceedings of the 25th European Conference on Object-Oriented Programming, ECOOP 2011, in: Lecture Notes in Computer Science, vol. 6813, Springer-Verlag, Lancaster, UK, 2011, pp. 459–483.

[27] G. Bracha, The Strongtalk type system for Smalltalk, http://www.bracha.org/nwst.html.
[28] R. Smit, Pegon, http://sourceforge.net/projects/pegon/.
[29] S. Tobin-Hochstadt, V. St-Amour, The typed racket guide, http://docs.racket-lang.org/ts-guide/.
[30] I. Figueroa, É. Tanter, N. Tabareau, A practical monadic aspect weaver, in: Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages, FOAL 2012, ACM Press, Potsdam, Germany, 2012, pp. 21–26.
[31] R.E. Johnson, Type-checking Smalltalk, SIGPLAN Notices 21 (11) (1986) 315–321.
[32] R.E. Johnson, J.O. Graver, L.W. Zurawski, Ts: an optimizing compiler for Smalltalk, SIGPLAN Notices 23 (11) (1988) 18–26.
[33] J.O. Graver, R.E. Johnson, A type system for Smalltalk, in: POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1990, pp. 136–150.
[34] N. Haldiman, M. Denker, O. Nierstrasz, Practical, pluggable types for a dynamic language, Computer Languages, Systems and Structures 35 (1) (2009) 48–62.
[35] F. Pluquet, A. Marot, R. Wuyts, Fast type reconstruction for dynamically typed programming languages, in: DLS '09: Proceedings of the 5th Symposium on Dynamic Languages, 2009, pp. 69–78.
[36] D. Ancona, M. Ancona, A. Cuni, N.D. Matsakis, RPython: A step towards reconciling dynamically and statically typed OO languages, in: Proceedings of the 2007 symposium on Dynamic languages, DLS '07, 2007, pp. 53–64.
[37] C. Anderson, S. Drossopoulou, P. Giannini, Towards type inference for JavaScript, in: A.P. Black (Ed.), Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP 2005, in: Lecture Notes in Computer Science, Springer-Verlag, Glasgow, UK, 2005, pp. 428–452.
[38] T. Wrigstad, F. Zappa Nardelli, S. Lebresne, J. Östlund, J. Vitek, Integrating typed and untyped code in a scripting language, in: POPL 2010 [42], pp. 377–388.
[39] D. Herman, A. Tomb, C. Flanagan, Space-efficient gradual typing, Higher-Order and Symbolic Computation 23 (2) (2010) 167–189.
[40] J. Siek, P. Wadler, Threesomes, with and without blame, in: POPL 2010 [42], pp. 365–376.
[41] Proceedings of the 18th European Symposium on Programming Languages and Systems, ESOP 2009, Lecture Notes in Computer Science, vol. 5502, Springer-Verlag, York, UK, 2009.
[42] Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, ACM Press, Madrid, Spain, 2010.