Beyond Types: Extending the Gradual Guarantee

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Michael Homer
Victoria University of Wellington
mwh@ecs.vuw.ac.nz

Timothy Jones
Victoria University of Wellington
tim@ecs.vuw.ac.nz

Sophia Drossopolou
Imperial College, London
scd@doc.ic.ac.uk

Andrew P. Black
Portland State University
black@cs.pdx.edu

Kim. B. Bruce
Pomona College, CA
kim@cs.pomona.edu

Abstract

The gradual guarantee lets us understand gradual typing: a system is gradually typed if removing a type annotation cannot change the semantics of a correct program. We extend the gradual guarantee beyond types: privacy annotations or inheritance restrictions, for example, may be gradual if changing them does not change the semantics of a correct program.

1. Introduction

Gradual typing is increasingly of interest in programming language design [1, 2]. Whether a type system is actually gradual or not can be determined by considering the **gradual guarantee**:

> *The gradual guarantee says that if a gradually typed program is well typed, then removing type annotations always produces a program that is still well typed. Further, if a gradually typed program evaluates to a value, then removing type annotations always produces a program that evaluates to an equivalent value.*
>
> Refined Criteria for Gradual Typing.
> Siek, Vitousek, Cimini, Tang Boyland [3].

There are more things in programming languages, however, than are dreamt of in this philosophy. Encapsulation, fixity, mutability, and incompleteness are typically addressed in programming languages using other kinds of annotations, just not type annotations. Can the gradual guarantee be generalised to these annotations also?

2. Example

Consider the Grace program shown in Fig. 1

```
class jeremy {
  method bolognese is confidential { print "harry lime" }
  var iable is public := 42
  method spaghetti { confusing }
  method confusing is private { print "confusing jeremy" }
}

method john {
  var jsPrivate := 23
  return object {
    inherit jeremy
    method confusing { print "confusing john" }
  }
}

var j := john
j.bolognese  // error. method not accessible
j.iable // returns 42 unless public annotation removed
j.confusing  // prints "confusing john"
j.spaghetti  // prints "confusing jeremy"
j.jsPrivate  // error. request not understood
```

Figure 1. A Grace program

The class jeremy declares a method bolognese that is annotated confidential (roughly equivalent to Java's protected, but per instance): removing this annotation will not change the semantics of a correct program, because that change makes the bolognese method more accessible throughout the program. (Note that methods are public by default in Grace, and are invoked without gratuitous BCPL-derived "()" syntax.) If the program is correct, then it cannot request bolognese other than on **self**. On the other hand, the **var** iable is annotated **is** public because variables are confidential by default in Grace: removing that annotation would cause a program that accessed the variable from outside a jeremy object to fail.

The method confusing is annotated private; assuming a Java-like semantics for private, this means the call of confusing inside the body of spaghetti is statically bound. When considering an object such as john which inherits from jeremy, removing the annotation would actually change the semantics of the program: with private, the request to confusing inside the spaghetti method always resolves to the definition in its defining jeremy class, and so that request can never be overridden, whereas if the annotation is removed, then the request to confusing is resolved via normal dynamic dispatch.

Finally, note that the method john effectively gets a private variable jsPrivate, due to lexical scope. This is how private variables are defined e.g. in Javascript and E: access restriction depends on the topology of the program's structure, not annotations enforcing encapsulation rules.

3. Gradualism Beyond Types

We argue gradualism can apply to more than just types: many other aspects of (or annotations on) programs can also be gradualised, and as such can participate in an extended version of the gradual guarantee. Our example shows three kinds of constructs:

restrictive constructs reduce the set of well formed programs and permissible executions ("confidential")

permissive constructs increase the set of well formed programs and permissible executions ("public")

semantic constructs alter programs' behaviour ("private")

Restrictive constructs can be gradualised in the same way as type annotations: from this perspective, type annotations are one kind of restrictive constructs. Permissive constructs can be gradualised "in reverse": adding a permissive annotation maintains the guarantee, while removing one does not. Semantic constructs cannot be gradualised because any change there will change the behaviour of the program other than by raising an error, or suppressing one. (This is one of the reasons why Grace doesn't have a private annotation).

We can express an extended gradual guarantee by a simple cut-and-paste of the gradual typing guarantee:

> *The **extended** gradual guarantee says that if a gradually **structured** program is well **formed**, then removing **restrictive** annotations **(or adding permissive annotations)** always produces a program that is still well **formed**. Further, if a gradually **structured** program evaluates to a value, then removing **restrictive** annotations **(or adding permissive annotations)** always produces a program that evaluates to an equivalent value.*

This approach is not limited to encapsulation. Java's final is a restrictive annotation that addresses mutability and inheritance; @Override is another restrictive annotation that also addresses inheritance. Grace is toying with a fixity annotation, manifest, that would require objects to be determinable at compile time. Being able to gradualise these (or other) annotations offers many of the same advantages as gradualising type annotations: we can ignore annotations and program in a "scripting" style, while being sure of how the semantics of the program will change if we start with a "script" and later convert it into a "program". As with gradual types, such a refactoring is typically in the opposite direction to the "arrow of gradualisation" which makes it easy to remove types (or restrictions) rather than add them.

This definition also implies that many of the practical difficulties related to gradual typing apply to other gradual constructs. If the error raised by a restrictive annotation can be caught within the program, then removing the error will cause a semantic change — similarly for the lack of an error caused by the lack of a permissive annotation. If the language includes constructs (such as reflection or try/catch or typecase) that can detect the presence or absence of annotations, then any detectable difference in the presence or absence of an annotation may indirectly cause a semantic change. Note that with this wording, the problem is not caused by constructs or annotations that *raise* errors, but constructs that *detect* that errors have been raised, or that otherwise introspect on the program.

This definition also raises questions for language designers: is restriction better than permission? Are annotations preferable to first-class constructs (e.g. Scala's var and val versus Java's restrictive final for mutability)? Are structural patterns (e.g. Javascript's lexical private) preferable to annotations or first-class constructs?

Acknowledgements

We thank the anonymous reviewers for their comments. This work was supported in part by a James Cook Fellowship and by the Royal Society of New Zealand Marsden Fund.

References

[1] John Tang Boyland. The problem of structural type tests in a gradual-typed language. In *FOOL*, 2014.

[2] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming*, 2006.

[3] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL*, 2015.