# Dotty Phantom Types

(Technical Report)

Nicolas Stucki Aggelos Biboudis Martin Odersky
École Polytechnique Fédérale de Lausanne
Switzerland

## Abstract

Phantom types are a well-known type-level, design pattern which is commonly used to express constraints encoded in types. We observe that in modern, multi-paradigm programming languages, these encodings are too restrictive or do not provide the guarantees that phantom types try to enforce. Furthermore, in some cases they introduce unwanted runtime overhead.

We propose a new design for phantom types as a language feature that: (a) solves the aforementioned issues and (b) makes the first step towards new programming paradigms such as proof-carrying code and static capabilities. This paper presents the design of phantom types for Scala, as implemented in the Dotty compiler.

## 1 Introduction

Parametric polymorphism has been one of the most popular features of type systems since it was first introduced in Standard-ML [Milner et al. 1975]. Under parametric polymorphism, functions and data-types can handle and store values of different types without loss of generality. As an example, in a multi-paradigm programming language like Scala, the generic data-type definition for a machine can be abstract over the type of items it produces and can be represented as `Machine[P <: Product]`. This variation is called bounded parametric polymorphism [Cardelli and Wegner 1985] because we constrain the type of products based on a subtype relationship. As a result, a method that inspects a product can be programmed in a type-safe manner. We rely on the type-checker to check our generic uses of values of type `P`.

However, apart from relying on the type-checker to verify the *uses* of values, sometimes the only thing we need is to enforce some kind of constraint. While run-time assertions can implement such behavior we can let the type system perform our checks at the type level. For example, assume we need a function that turns a machine on (`turnOn`), only if it is turned off. Without relying on run-time checks, we can encode such constraints at the type-level using *phantoms*.

Phantom types were introduced to create type-safe domain-specific languages (DSLs) in Haskell [Leijen and Meijer 1999], increase safety of tagless interpreters [Carette et al. 2009], encode type safe generators [Elsman and Larsen 2004], implement heterogeneous lists [Kiselyov et al. 2004], GADTs [Cheney and Hinze 2003], model subtyping [Fluet and Pucella 2006] and even a lightweight form of dependent types [Kiselyov and Shan 2007].

There are two ways of using phantoms as a design pattern: a) relying solely on type unification and b) relying on term-level evidences. According to the first way, phantoms are encoded with type parameters which must be satisfied at the use-site. The declaration of `turnOn` expresses the following constraint: if the state of the machine, `S`, is `Off` then `T` can be instantiated, otherwise the bounds of `T` are not satisfiable.

```
type State; type On <: State; type Off <: State
class Machine[S <: State] {
  def turnOn[T >: S <: Off] = new Machine[On]
}
new Machine[Off].turnOn
```

An alternative way to encode the same constraint is by using an *implicit* evidence term of type `=:=`, which is globally available. This encoding depends on the existence of such evidence terms on the use-site. These are encoded as terms with phantom types and again, they are not used at runtime. Similarly to the previous encoding, if `S` is `On` the evidence type `S =:= Off` is not inhabited, hence no value exists for it.

```
class Machine[S <: State] {
  def turnOn(implicit ev: S =:= Off) = new Machine[On]
}
```

These values can be either capturable by closures or not, depending on the intended scenario.

As shown, in multi-paradigm programming languages like Scala, phantoms can be used as a type-level design pattern for guarantee checking. Unfortunately, under the erasure semantics of generics, the combination of evidences and unchecked casts make the pattern starts to fall apart. The design pattern relies on well-behaved programmers that do not fake evidences through unchecked casts. Additionally, when using terms with phantom types we have to encode the type in a designated class–the instances of which are not used. The aforementioned has the consequence that the instances must be explicitly passed to all functions needing them, incurring runtime costs. The desired behavior is to have both runtime safety and zero runtime costs when using phantoms.

### *Contributions*

- we introduce a simple mechanism to create compile-time, type evidences with zero runtime cost.

- we introduce terms for these evidences with the same guarantees.
- we show that this can be achieved while retaining the whole power of the type system on abstract types.

The proposed design for phantom types aims to support two new programming paradigms. Following Curry-Howard, we can model propositions as types and terms as their proofs. If a proposition is represented by a phantom type, the proofs corresponding to it are erased at runtime [Sheard 2005]. We can also model *capabilities* as unforgeable values of types [Crary et al. 1999; Liu 2016]. If the types are phantom types, the capabilities are purely static. We have implemented our design in Scala as part of the Dotty compiler [Odersky 2017].

## 2 Overview

We begin with a simple state machine equipped with methods that can only be called in particular states, using Scala's `=:=` type constraint. As shown below, calling `turnOn` when the machine is on will fail at compile time.

```
type State; type On <: State; type Off <: State
class Machine[S <: State] private {
  def turnOn(implicit ev: S =:= Off) = new Machine[On]
}
object Machine { def newMachine = new Machine[Off] }
Machine.newMachine.turnOn.turnOn
// Error:                        ^
//   Cannot prove that On =:= Off.       (as expected)
```

In this example we have two types that are designed as phantom types. The first one is the `type State`, which is represented as an abstract type that is erased by the conventional Scala/Java erasure. The second one is the `=:=` type which is currently defined in `scala.Predef`. In contrast to the pervious, it is used for the term `ev`. Unfortunately `=:=` is just a sealed abstract class which means that the compiler lacks any guarantee about it never being used. It would always be possible to write `ev.hashCode` or other method of `Object`.

Without this guarantee it would be impossible to ensure binary compatibility if in some version the parameter might be used and in a subsequent one it is not. In practice, it is also hard to know if a method will use a parameter as it might be overridden in external code. Therefore the parameter must always be explicitly passed at runtime.

This design also introduces a safety shortcoming, as the evidence can easily be corrupted or faked. It is relatively easy to corrupt the state of a `Machine` using a simple bad cast, which would violate the contract that we wanted to enforce. The following code shows how this is used to call `turnOn` twice.

```
newMachine.turnOn.asInstanceOf[Machine[Off]].turnOn
```

As `X =:= Y` is a subtype of `Any` it can be either coerced to any other `A =:= B` or replaced by some term of type `Nothing` (such as `Predef.???`). This allows fake term evidences to be materialized out of thin air using unchecked casts or exploiting a term of bottom type (such as `null` and `Predef.???`).

```
def onMachine = newMachine.turnOn
onMachine.turnOn(implicitly[Off =:= Off].asInstanceOf[On =:= Off])
onMachine.turnOn(null.asInstanceOf[On =:= Off])
onMachine.turnOn(???)
```

The previous code demonstrates that casting will make the code compile and run disregarding the broken constraint. Specifically, the use of `Nothing` will compile but throw an exception when the code is reached, while the other one will actually call the method `turnOn` again.

### 2.1 Introducing Phantoms

To solve all the aforementioned issues we the `scala.Phantom` interface; a synthesized interface that enables creation of proper phantom types in the language. This design provides a simple and extensible way to add phantom types. We can now present phantoms through a modified version of the `Predef.=:=` that is a phantom and works on phantom states.[1]

We define all states as a subtype hierarchy. Note that the top of the type hierarchy is defined by the type `this.Any` and the terms are created with the protected method `this.assume`. We use the redundant `this` to explicitly show members defined in `Phantom`.

```
object Constraints extends Phantom {
  type State <: this.Any; type On <: State; type Off <: State
  type =:=[From <: State, To <: State] <: this.Any
  implicit def tpEquals[X <: State]: X =:= X = this.assume
}
```

This redefinition of `=:=` and `State` is all that is needed, we just need to import the `=:=` in `Machine` and the `tpEquals` in the scope of the calls to `turnOn`. With this change `=:=` evidence parameters are completely removed and the compiler will not allow runtime operations on the phantoms such as `asInstanceOf`. In fact the only term of `Constraints.Any` that can ever be materialized are `X =:= X` for any `X <: State`.

## 3 Implementation

This section describes the implementation of our prototype the introduces phantoms in Dotty [Odersky 2017]. We argue that our design is simple as we only needed around 200 LOC to define the phantom types and enforce their properties (sections 3.1 and 3.2), 100 LOC to erase phantom terms (section 3.3) and 120 LOC to implement functions/lambdas that receive phantom parameters (section 3.4).

### 3.1 Design of the Phantom Types

Phantom types are abstract types that have no effect on the runtime. By effect we not only describe IO, field mutation,

[1]We have also implemented a phantom version `Predef.=:=` that works as a source compatible replacement

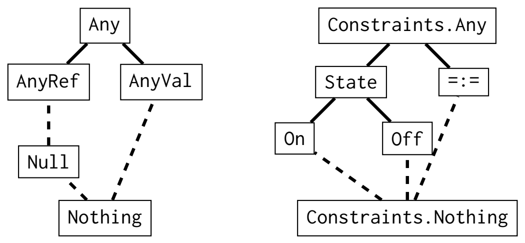

**Figure 1.** Scala Universe (left) and a Universe of Phantom Types (right).

exceptions and so on. We also imply that the result and execution of functions that receive a phantom remains unaffected by this argument.

Phantoms should not have a runtime representation so they must be not part of the `scala.Any` type hierarchy, which defines methods such as `hashCode`, `equals`, `getClass`, `asInstanceOf` and `isInstanceOf`. All these operations cannot exist on phantoms as there will not be an underlying object instance or value at runtime. At first glance this could look like a limitation, but in fact, not having `asInstanceOf` is the key to make constraints reliable as it will not be possible to downcast a phantom value to fake an evidence. Upcasting (or evidence weakening) is still possible through type ascription.

Furthermore we disallow unchecked casts of phantom type parameters/members of some object. This disallows `.asInstanceOf[Machine[Off]]` as `Off` is a phantom type. It would be possible to be allow casts on phantom that do not change the phantom. But, we can still redefine `Machine.S` as a type member to allow casts on `Machine` as shown bellow.

```
class Machine { type S <: State }
class SubMachine extends Machine
val m: Machine = new SubMachine{ type S = Off }
m.asInstanceOf[SubMachine] // cast OK, conserving state Off
```

Given that the phantom types are not in the `scala.Any` and `scala.Nothing` (top and bottom types) universe, they will be in their own *Phantom-based Universe*. Every phantom type must have upper and lower bounds only in the same universe. This also implies that `&` and `|` types must contain types in a single universe. In Figure 1 we show the Scala Universe and the Phantom Universe defined by `Constraints`. Other objects extending `Phantom` would define their own universe.

For each universe we materialized a term of that universe's `Phantom.Nothing` through `assume` which can be upcasted to any type in the universe, but never downcasted again. Provided that this all-powerful phantom term would have defeated the safety purpose of our proposal, we make it protected in `Phantom`. This raises the expressiveness of the proposed language feature as some universes could leak their own bottom term while safer ones would only provide specific phantom instances.

## 3.2 Universal Phantom Interface

Phantom types are defined by an object extending the synthetic `scala.Phantom` interface. This object will represent a universe of phantom types that is completely separated from types in `scala.Any` or other phantom universes.

```
package scala
// only an `object` can extend this trait
trait Phantom {
  // not a subtype of scala.Any
  protected final type Any
  // subtype of every subtype of this.Any
  protected final type Nothing
  // no implementation as this value/call does not exist at runtime
  protected final def assume: this.Nothing
}
```

The types in the phantom universe are defined by the top type `Phantom.Any` and bottom type `Phantom.Nothing`. These types are protected and can not be accessed from outside unless an alias is defined for them.

New phantom types can be defined using `type XYZ <: OtherPhantom`, where the correct bottom type will be inferred as lower bound. A phantom type can be aliased with `type MyAny = this.Any`. Higher-Kinded Types can be or have parameters of any phantom or non-phantom type.

Values of a phantom type can be created only by using the protected definition `assume`. This value can be used as any value of a phantom type as it's type is the bottom type of the universe. Usually it will be used to define an implicit definition that returns a phantom with a more precise type.

To ensure sanity of the types the compiler checks that all type bounds, intersection types and union types are in the same universe.

## 3.3 Erasing Phantom Types and Terms

Once the type-checker has used the phantom types to check the constraints, they can be removed. As phantom types are completely disjoint from runtime type we can trivially know which ones are phantoms[2] and always remove them.

Type parameters and type members are erased out of the box by the normal Scala/Java generic type erasure mechanism and therefore no change is needed. A language that did not adopt the same erasure semantics would have to erase them explicitly at compile time.

Any call to `assume` is replaced by a `UNIT` value (`ErasedPhantom`), known to be pure which can be optimized away at the call site. Any method returning `assume` should be marked as `inlined` explicitly to allow full erasure by the compiler[3].

We remove all phantom parameters and arguments from all methods and calls. We do still evaluate the argument expression to not lose any hidden side effects such as IO, exceptions and non termination. In the code below we demonstrate the source code and the target, after erasure. We observe the

[2]Any type for which its top type is some `Phantom.Any`

[3]If not marked as inline, the JVM JIT sill has a chance to optimize it away

translation of assume and the retainment of the potentially effectful myPhantom.

```
type MyPhantom <: this.Any
def myPhantom: MyPhantom = assume
def f(a: Int, b: MyPhantom): Int = a
f(getSomeInt(), myPhantom)
// after erasure
def myPhantom(): ErasedPhantom = ErasedPhantom.UNIT
def f(a: Int): Int = a
val a$ = getSomeInt()
val b$ = myPhantom // erased if myPhantom is inlined
f(a$)
```

## 3.4 Higher-Order Functions

It is also possible to use phantom parameters in higher-order functions. The following example shows a lambda that receives a phantom parameter.

```
type CanOpen <: MyPhantom
implicit def canOpen: CanOpen = ...
val lookInside: CanOpen => String = ev => "I'm in a safe box"
lookInside(canOpen)
```

Furthermore it is possible to receive the parameters implicitly using implicit function types [Odersky et al. 2017]. With implicit function types, values of lambdas that take implicit parameters are now represented on the type system. This way the overhead of explicitly passing constraints around can be abstracted away.

```
type Open[T] = implicit CanOpen => T // implicit function types
val lookInside: Open[String] = "I'm in a safe box"
lookInside
```

In Phantom, we define the FunctionN and ImplicitFunctionN traits (for any N > 0) similarly to the corresponding vanilla Scala functions. The difference is that both definitions are now aware of a Phantom-based Universe. All type parameters of the phantom function will be bounded by their Any and Nothing type. Having only phantom parameters implies that all parameters will be erased and hence all these functions will become Function0 at runtime.

With this design we force all parameters to be of the same type universe and all functions to have homogeneous names. To mix several kinds of parameters it is possible to curry the functions. It would also be possible to have functions that receive all kinds of combinations parameters but this would require heavy name mangling, which the user would need to know about.

In practice we do not use the FunctionN types directly, we use the lambda notation as shown in the CanOpen example. Therefore we also modify the lambda type inference to adapt their function type based on the parameter types.

## 4 Towards Capabilities: Non-Captured Evidences

A stronger kind of type evidences is one where the evidence must be provided by the context where the call is performed. This section introduces a first encoding towards capabilities. We build upon the CanOpen phantom as an example and turn it into a capability.

We want to enforce a constraint on lookInside to only be called in a context where there is a CanOpen value available. We require an additional constraint that disallows capturing a CanOpen in a method or closure. Without it the issue would be that a method could capture the evidence at the definition site. However, the evidence should originate for the use-site.

```
object Foo {
  def canOpen: CanOpen = ...
  def look = lookInside(canOpen) // should not be allowed
}
Foo.look // calls transitively lookInside without a CanOpen
```

By disallowing capturing, the sole way to have a CanOpen value is through a parameter which will be needed at the call site. Usually these parameters would be implicit parameters defined through implicit function types.

```
def look(implicit ev: CanOpen) = lookInside
// or simply
def look: Open[String] = lookInside
```

We tag the phantom type universes as being capturable or not. This makes it impossible to mix phantom types with & and | that have different capturing rules. Hence the multiple universe design make this distinction trivial and reliable.[4]

## 5 Related Work

F* is dependently typed, higher-order, call-by-value language that supports a lattice of primitive, monadic effects. A GHOST monad [Swamy et al. 2016] is part of this lattice and represents purely specificational computations that can be erased. Although F* is designed to offer stronger guarantees, the erasure model and guarantees are similar to our system. We currently lack purity constraints for functions retuning phantoms.

Rust also encodes phantoms on the term level as parameters of a special marker trait PhantomData. However, instances of PhantomData can be created anywhere for any type as opposed to our assume which is protected.

In fully dependently-typed programming languages such as Idris [Tejiščák and Brady 2015], equalities are witnessed by computational proofs and as a result dedicated erasure-analysis is performed. In our proposal, phantoms are user defined.

[4]As shown in a 30 LOC prototype implementation

## 6 Conclusion

Phantom types are a type-level encoding that is used to enforce guarantees about program constraints. Our work promotes the concept of phantom types from being merely a design pattern, to a fully-fledged feature. We presented the design choices behind them and took an initial step forward in supporting proof-carrying code and capabilities.

## References

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (Dec. 1985), 471–523.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543.

James Cheney and Ralf Hinze. 2003. *First-class Phantom Types.* Technical Report. Cornell University.

Karl Crary, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99).* ACM, New York, NY, USA, 262–275.

Martin Elsman and Ken Friis Larsen. 2004. Typing XHTML Web Applications in ML. In *Practical Aspects of Declarative Languages: 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004. Proceedings*, Bharat Jayaraman (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 224–238.

Matthew Fluet and Riccardo Pucella. 2006. Phantom Types and Subtyping. *Journal of Functional Programming* 16, 6 (2006), 751–791.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04).* ACM, New York, NY, USA, 96–107.

Oleg Kiselyov and Chung-chieh Shan. 2007. Lightweight Static Capabilities. *Electron. Notes Theor. Comput. Sci.* 174, 7 (June 2007), 79–104.

Daan Leijen and Erik Meijer. 1999. Domain Specific Embedded Compilers. In *Proceedings of the 2Nd Conference on Domain-specific Languages (DSL '99).* ACM, New York, NY, USA, 109–122.

Fengyun Liu. 2016. *A Study of Capability-Based Effect Systems.* Master's thesis. EPFL. (updated at 2017 Jan 16).

Robin Milner, Lockwood Morris, and Malcolm Newey. 1975. *A Logic for Computable Functions with Reflexive and Polymorphic Types.* University of Edinburgh. Department of Computer Science.

Martin Odersky. 2017. Dotty Compiler: A Next Generation Compiler for Scala. https://web.archive.org/web/20170325001401/http://dotty.epfl.ch/. (2017).

Martin Odersky, Aggelos Biboudis, Fengyun Liu, Olivier Blanvillain, and Heather Miller. 2017. *Simplicitly.* Technical Report.

Tim Sheard. 2005. Putting Curry-Howard to Work. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05).* ACM, New York, NY, USA, 74–85.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM, New York, NY, USA, 256–270.

Matúš Tejiščák and Edwin Brady. 2015. *Practical Erasure in Dependently Typed Languages.* Technical Report. University of St Andrews.