First-class Dynamic Types

Michael Homer
School of Engineering
and Computer Science
Victoria University of Wellington
New Zealand
mwh@ecs.vuw.ac.nz

Timothy Jones
Montoux
New York, USA
tim@montoux.com

James Noble
School of Engineering
and Computer Science
Victoria University of Wellington
New Zealand
kjx@ecs.vuw.ac.nz

Abstract

Since LISP, dynamic languages have supported dynamically-checked type annotations. Even in dynamic languages, these annotations are typically static: tests are restricted to checking low-level features of objects and values, such as primitive types or membership of an explicit programmer-defined class.

We propose much more dynamic types for dynamic languages — first-class objects that programmers can customise, that can be composed with other types and depend on computed values — and to use these first-class type-like values *as* types. In this way programs can define their own conceptual models of types, extending both the kinds of tests programs can make via types, and the guarantees those tests can provide. Building on a comprehensive pattern-matching system and leveraging standard language syntax lets these types be created, composed, applied, and reused straightforwardly, so programmers can use these truly dynamic first-class types to make their programs easier to read, understand, and debug.

1 Introduction

Most dynamically-typed languages do not provide much support for programmers incorporating type checks in their programs. Object-oriented languages have dynamic dispatch and often explicit "instance of" reflective operations, gradual typing provides mixed-mode enforcement of a single static type system, and functional languages often include pattern-matching partial functions, all of which are relatively limited at one end of the spectrum. At the other end, languages like Racket support higher-order dynamic contract systems, wrapping objects to verify contracts lazily, and with concomitant power, complexity, and overheads [10, 46].

Programmers may wish for a greater level of run-time checking than provided by the language, or merely *different* checking. In this paper we explore a design in between the extremes: first-class dynamic types, building on syntactic support for type annotations and a framework for first-class pattern-matching, and extend the Grace language leveraging its existing support for both [2, 22]. The key change is that run-time expressions may be used as type annotations, and evaluate to first-class pattern objects: any value that flows through a type annotation is dynamically checked by the pattern, and types are constructed in a principled, compositional, and fully-extensible fashion. From most programmers' perspectives, these first-class annotations simply *are* types; more-advanced programmers can assemble new types either from reusable components or by writing custom patterns from scratch, drawing on the clear semantics of pattern-matching.

First-class dynamic types can encapsulate any check that can be expressed in code, can make use of or be parameterised by run-time values, can produce warnings or errors, and can manipulate dynamic program state. A pattern already created for use in pattern-matching can be used as a type, and vice-versa, rather than each being a distinct program element, and a library of patterns can provide a wealth of different sorts of type checks within the same language or even the same program. Our system requires no additional annotations, extra sub-languages, or new semantic or syntactic categories; rather, by adopting existing type-annotation syntax and pattern matching, first-class dynamic types fit seamlessly into the existing language. A program can be shorter, clearer, and more precise by abstracting away checks that would be repeated redundantly or omitted negligently, while more directly expressing the intention of the programmer simultaneously with providing immediate diagnostics.

The resulting system sits at an interesting point in the design space: less explicit than assertions or Eiffel-style pre- and post-conditions, less powerful than full-scale Racket-style higher-order contracts, more flexible than transient dynamic type tests, while retaining immediate, straightforward, pattern-matching semantics.

1.1 Contributions

The contributions of this paper are:

- The design of a system of dynamic type-checking based on a pattern-matching framework, with user-defined patterns.
- An implementation of this system on top of an existing implementation of the Grace language.
- A set of small case studies illustrating different styles of checking that can be introduced.

DLS'19, October 20–25, 2019, Athens, Greece
2019.

2 Patterns in Grace

We build on an object-oriented language called Grace [6], which already supports an object-oriented form of pattern matching [22]. We will extend this pattern-matching system to more general applications, but keep the underlying design intact.

There are three core elements of Grace pattern matching:

- a “pattern” is an object with a method match returning a MatchResult object indicating success or failure, where match may have arbitrary user code;
- a “lambda pattern” syntactic construct, where a unary block (an anonymous single-parameter function) connects a pattern, a name, and the ability to execute code in the context of the name when the pattern matches; and
- a match-case statement, which combines many lambda patterns together to provide a typecase-like construct.

In more detail, a pattern object has a match method accepting a single parameter, the object to be scrutinised, and returning a MatchResult object, which is a Boolean with a result property. A successful match acts like true, and m.result is bound to the matched object, while a failed match acts like false. For example, we can write a pattern to check if its argument is negative:

```
def negativePattern = object {
  method match(o) → MatchResult {
    if (o < 0) then {
      return successfulMatch(o)
    } else {
      return failedMatch(o)
    }}}
```

A successful match result behaves like a Boolean true, and a failure like a Boolean false, so patterns can be used directly in if statements:

```
if (negativePattern.match(obj)) then {
  ...
} else {
  ...
}
```

Any pattern can be used in the match-case construct:

```
match (obj)
  case { x : EvenNumber → "even" }
  case { x : Number → "other number" }
  case { x : Green → "green" }
  case { _ → "not numeric nor green" }
```

The key idea here is that Grace's blocks (akin to Smalltalk or Ruby blocks, or lambda expressions) model *partial* functions, rather than *total* functions as in most other languages. This is because single-parameter blocks also implement the match interface — which is why we also call blocks “lambda patterns” when they are used in the context of the wider pattern subsystem.

When a lambda pattern is asked to match another object, the lambda pattern checks that object against the type or pattern annotation on the corresponding parameter — in the match-case example, the first block will test the obj against the EvenNumber annotation. If the annotation matches, the whole lambda pattern executes its body and returns an instance of successfulMatch; if that annotation doesn't match, the whole lambda pattern returns a failedMatch.

Each case clause in a match-case construct is a single lambda pattern. The match-case method asks each lambda to match in turn until one succeeds, and the body of that block (only) executes. The overall match-case returns the result of the executed block, or raises an error if no pattern matched.

Patterns also support operators & and |, which combine two patterns together with conjunctive or disjunctive semantics. These operators parallel the operators used with Grace types, and in fact all types in Grace are reified as patterns at run time. The type Employee & Dog represents objects that are simultaneously Employees and Dogs (according to standard structural subtyping rules), and the type reified as the *pattern* Employee & Dog will succeed at matching the same objects.

A MatchResult object has a method result containing the object that has been matched, but it is possible for this object to differ from the one originally provided. Notably, the result of block used as a lambda pattern is the value returned by the body of the block. Grace also uses this feature to support type casts, but it can be used for specialised patterns that manipulate their targets as well. For example, a pattern like the following:

```
def halfPattern = object {
  method match(o) → MatchResult {
    if (Number.match(o)) then {
      return successfulMatch(o / 2)
    } else {
      return failedMatch(o)
    }}}
```

matches objects that are numbers, but yields *half* the value of the number as its result, so that:

```
match (8)
  case { x : halfPattern → print(x) }
```

will print “4” because x is bound to the result value of the successfulMatch (in this case 4) returned from the halfPattern's match method.

3 Patterns as types

Grace's original pattern-matching framework promotes types to reified pattern objects at run time, but the equivalence is not bidirectional: programmers can only annotate declarations with patterns (rather than types) in lambda patterns. A programmer is not permitted to annotate a method parameter, local variable, field, or method return value with a pattern, only a true type. In this work we generalise the system to permit patterns in all of these places, and augment the system to dynamically enforce that they are satisfied by the values passing through the annotations.

In particular, we let a general pattern be used as:

- A method parameter type
- A local variable (var) or constant's (def) type
- An object field's type
- The return type annotation on a method

Anywhere that a value can be given an expected *type* in traditional code, we now allow a dynamically-enforced *pattern* to take that place, and enforce validation of any value that passes through that annotation as soon as it reaches that point. A pattern is matched using Grace's standard infrastructure: its match method is called by the runtime system and given the assigned, passed, or returned value to inspect. A successful match allows the program to continue, and binds the result value of the match to the declared name (or uses it as the return value). A failed match triggers an immediate run-time type error.

3.1 Semantics

To support dynamic first-class types, a Grace runtime system must invoke the pattern's match methods implicitly, as they are reached in the normal course of execution. To make the intended semantics clear, we show how they could be implemented by a source-to-source rewriting of a program's source code (Vitousek et al. [51] describe Reticulated Python's semantics in a similar way). Given a method:

```
method foo(x : T) → R {
  ...
  return x.name
}
```

there are two elements to deal with:

- The type annotation on the parameter x, T.
- The return type annotation, R.

We will address these in turn. First, for the parameter:

1. *Rename* the parameter to x′arg, so that we can create a new local variable called x later for the method body to use.
2. Create a new local variable def x′pattern = T, to evaluate the type annotation and store it.
3. Create a new local variable

   ```
   def x′matchResult = x′pattern.match(x′arg)
   ```

   and invoke the pattern-match infrastructure to initialise it.
4. Check that the match succeeded, and raise an error if not:

   ```
   if (!x′matchResult) then {
     reportTypeError "x did not meet type T"
   }
   ```

5. Finally, create our new local variable with the original parameter name, def x = x′matchResult.result, so that the remaining code can continue in terms of x.

The outcome at this point is:

```
method foo(x′arg : T) → R {
  def x′pattern = T
  def x′matchResult = x′pattern.match(x′arg)
  if (!x′matchResult) then {
    reportTypeError "x did not meet type T"
  }
  def x = x′matchResult.result
  ...
  return x.name
}
```

The same process applies for any additional parameters as well. The pattern expressions are re-evaluated each time the method is called, but the implementation is free to optimise these evaluations away where the expression is known to be static, or to memoise when it is known to be pure. While it is possible to restrict the allowable pattern expressions further, we are permitting fully dynamic behaviour to explore the widest range of possible applications.

For the return type, the pattern must be evaluated in the same way, but any return x.name statement must be rewritten in the following way:

```
def return′matchResult = return′pattern.match(x.name)
if (!return′matchResult) then {
  reportTypeError "return value did not meet type R"
}
return return′matchResult.result
```

The final outcome is then a method evaluating patterns at the top, checking each argument or return value against the corresponding pattern when it is available, and otherwise proceeding exactly as it would have normally in the body.

Because Grace's var and def declarations are syntactic sugar for accessor methods, the same transformation applies.

3.2 Type declarations and interfaces

Grace allows type declarations of the form

```
type Foo = interface {
  x(_ : Number) → String
  y → Boolean
}
```

These interfaces naturally represent structural types, and have a run-time existence as patterns: Foo.match(o) succeeds, and has a Foo-typed result, if o satisfies that structural type. If a programmer wishes to implement some other type system, we permit these interfaces to be "wrapped" dynamically, so that user-level types can continue to use the interface syntax. The wrapper can produce a pattern that makes use of the method signatures with any desired semantics. We also allow run-time values to be assigned to type declarations.

Different styles of check may even be wrapped around different interfaces in the same program, to allow different systems to be applied at once in different areas of the program (including the baseline structural system). The wrapping must occur explicitly — we do not automatically wrap all interfaces, which means a small amount of additional code for each declaration. In practice, we do not find this requirement onerous, in particular because the type declaration need only appear once, and then refers to the pattern object forever after. It would be possible for a more aggressive rewrite to convert the interface expressions automatically, at the cost of some potential behavioural changes and limits.

3.3 Static type checking

Our approach operates on top of a purely untyped run-time system. Static checking is also useful, however, and a user or library author can use Grace's dialects [21] and its pre-execution checker methods to create a static checker consistent with their dynamic checks if they desire. We do not explore this idea further here: Homer et al. [21] includes examples of dialects that perform static type checks and require explicit type annotations, along with others. Dialects have enabled Grace to support different "pluggable" static type checkers: this work aims to give similar flexibility to Grace's dynamic type checking.

4 Case studies

We now present a series of small case studies illustrating some of the breadth of different dynamic type systems that can be implemented on this framework. All of these systems exist in other languages: here we show that they can all be built on the *same* principled framework.

4.1 Shallow structural type checks

We can obtain structural checking — replicating the default behaviour of Grace reified types — by reflectively checking whether the necessary methods are present on a scrutinised object. We first define a simple pattern that does so literally, and then a more advanced pattern that lifts and wraps an interface.

Our first pattern can be used as:

```
method greet(p : Methods ["firstName", "lastName"]) {
    print "Hello " ++ p.firstName ++ " " ++ p.lastName
}
```

The p parameter of greet has been given a type annotation Methods ["firstName", "lastName"]: a call to the Methods class, passing as argument a list literal containing two method names as strings. The resulting pattern will check that corresponding methods exist, so the method body can invoke expressions such as p.firstName safely.

The Methods pattern needs only to check reflectively that the methods are on the target object, and returns a failedMatch as soon as it finds one that is absent:

```
import "platform/mirrors" as mirrors
class Methods(l) {
  method match(obj) {
    def mirror = mirrors.reflect(obj)
    for (l) do { methodName →
      if (!mirror.respondsTo(methodName)) then {
        return failedMatch(obj)
      }
    }
    return successfulMatch(obj)
  }
}
```

With the above class available, our greet method would work as written, and would report a type error if its argument did not have the necessary methods.

We can also define a wrapper class that can be given an interface expression as an argument, and checks that each method from the interface is present on the object:

```
class wrapInterface[[I]] {
  method match(obj) {
    def mirror = mirrors.reflect(obj)
    for (I.signatures) do { meth →
      def methodName = meth.name
      ...
  }
}
```

The wrapper corresponds almost exactly to the string-literal version above, but accepts a type parameter I instead of the list of strings. With the wrapper in scope, a structural type can be created as follows:

```
def HasFoo = wrapInterface[[interface { foo }]]
method useFoo(x : HasFoo) { ... }
```

In this way the syntactic form afforded by the interface expression of the language can be leveraged to allow conveniently writing method signatures to use. From the end-user perspective, the types created by either Methods or wrapInterface are interchangeable, and HasFoo could be defined with either and subsequently used identically.

A pattern could go further and implement a different type system while retaining the syntactic form if desired. For example, both these definitions implicitly permit subtyping: an object will be matched if it has at least the methods named by the pattern. but it will also match if it has a superset of these methods. We can extend this pattern to support exact type matching, so that an object must have only those methods, quite straightforwardly using inheritance:

```
class ExactMethods(l) {
  inherit Methods(l)
     alias match(_) as matchMethods(_)
  method match(obj) {
    def mirror = mirrors.reflect(obj)
    if (mirror.size == l.size) then {
      return matchMethods(obj)
    } else {
      return failedMatch(obj)
    }}}
```

4.2 Branded nominal types

As an alternative to structural types, we can create a nominal type system. Brands are an approach to adding nominal types on top of a non-nominal type system (such as Grace's structural system). Objects marked with a brand acquire an additional type, which can distinguish them in type checks and typecase from objects with the same shape, but a different (or no) brand. For example, in a structural system, the two types:

```
  type Graphic = interface { draw → Done }
  type Gunslinger = interface { draw → Done }
```

are indistinguishable — an inherent drawback of structural systems — but brands can introduce a finer distinction when required by marking some objects with a distinct "brand" marker. Here we will replicate the brand system proposed for Grace [24], but without any special-purpose language extensions.

This brand system separates the *brand object*, which is used to brand (mark) an object, and the *brand pattern*, or brand Type, which is used to confirm that an object has the correct brand. These two objects can be separated, so that external users can only use the type, but not create new branded objects themselves [55].

```
import "platform/mirrors" as mirrors
class brand {
  def Type = brandPattern(self)
  method brandObject(obj) {
    mirrors.mutable(obj).addMetadata("brand", self)
  }
}
class brandPattern(needle) {
  method match(obj) {
    def mirror = mirrors.reflect(obj)
    if (mirror.getMetadata("brand").contains(needle)) then {
      return successfulMatch(obj)
    }
    return failedMatch(obj)
  }
}
```

Instantiating a brand object creates a new brand: requesting the Type of the brand returns the brand pattern that tests for that brand.

We can now create a brand and apply it to an object, and use the brand's Type to annotate a parameter:

```
def myBrand = brand
def myObj = object { }
myBrand.brandObject(myObj)
method foo(x : myBrand.Type) { }
foo(myObj)
foo(object { })
```

The first call to foo will succeed, because that object has been branded so the pattern check will succeed, while the second call will be a dynamic type error because the pattern will fail.

4.3 Restricted subtypes

It is often useful to have parameters restricted to a particular range or subset of possible values, and some languages (such as Pascal [23]) allow defining restricted types in this way: for example, defining a restricted integer subtype to represent a Unicode codepoint with range 0–1,114,112. In our system a simple RangeType pattern can replicate this ability:

```
method printCodepoint(b : RangeType(0,1114112)) { ... }
```

Integer subranges are particularly useful in Grace, which otherwise has only a single Number type. We can use subranges to get the effect of more specific integral types, such as a Byte type for integers from 0 to 255:

```
type Byte = RangeType(0, 255)
```

and then use it on a method parameter:

```
method printByte(b : Byte) { ... }
```

This RangeType pattern is fairly straightforward:

```
class RangeType(min, max) {
  method match(obj) {
    def int = obj.asInteger
    if ((int ≥ min) && (int ≤ max)) then {
      return successfulMatch()
    }
    return failedMatch(obj)
```

```
    }
}
```

Grace's standard library includes a built-in range object, created by the ".." operator (e.g. 0..255). By adding the match method from RangeType into the library's ranges, those objects can be used directly as integer subrange types:

```
method printPercentage(e : 0..100) { ... }
type Byte = 0..255
```

As in Smalltalk, Grace ranges are also collections. We can lift any collection to a pattern with an ElementPattern which implements match by checking whether a collection contains the object being matched. This allows collections to model (dynamic, if desired) enumeration types:

```
def legoColours = set("red", "yellow", "blue", "green")
type LegoColour = ElementPattern(legoColours)
method findBrick(col : LegoColour) { ... }
```

We have implemented lifting collections to patterns, but have avoided any further language extensions here.

4.4 Argument-dependent patterns

We can introduce patterns that can use the values of *other* arguments to determine whether an argument is satisfactory. For example, a method can require that its second argument is greater than its first:

```
method foo(x, y : be > x) { }
```

or that a list index is within the range of the list:

```
method getItem(list, index : be > 0 & be ≤ list.size) {
    return list.get(index)
}
```

The "be" term represents the value of the current argument, and its operators create patterns themselves to match the requested limit — effectively currying the left-hand argument. Those resulting patterns are what the argument itself is validated against. For example, be > 0 above evaluates to a pattern that requires its target to be positive. Because parameter patterns are evaluated in order, the expression can refer to arguments to the left of the current one: be ≤ list.size works because list has already been bound at the time of execution, and is available for use like any other name in scope. The pattern expression could also use self, or any other declarations in scope, such as fields.

This system can be implemented in our framework, although that implementation is quite subtle:

```
def be = object {
    method >(other) { relativePattern({t → t > other}) }
    method <(other) { relativePattern({t → t < other}) }
    method ≤(other) { relativePattern({t → t ≤ other}) }
    method ≥(other) { relativePattern({t → t ≥ other}) }
}
class relativePattern(lambda) {
    method match(obj) {
        if (lambda.apply(obj)) then {successfulMatch(obj)}
            else {failedMatch(obj)} } }
method foo(x, y : be > x) { }
method getItem(list, index : be > 0 & be ≤ list.size) {
    return list.get(index)
}
```

The relativePattern lifts a block returning a Boolean into a pattern, succeeding when the block returns true. The operators on the "be" object return a relativePattern parameterised by a lambda block that computes the relevant test. In fact, relativePattern works for *any* predicate: for example, RangeType could be implemented with:

```
relativePattern({x → (x ≥ min) && (x ≤ max) })
```

The same behaviour can be replicated by the user using the & pattern combinator:

```
type Byte = (be ≥ 0) & (be ≤ 255)
```

4.4.1 Higher-order dependent types

Because Grace patterns are first-class objects, they can easily be passed to other patterns, letting Grace model higher-order dependent types. For example, a matrix library could define a pattern Matrix with two arguments for the dimensions of the matrix. Methods can use that pattern alongside the value of an argument to give concise and explicit tests of compatible dimensions:

```
class matrix {
    method *(other : Matrix(self.width, Number))
              → Matrix(self.height, other.width) {
        ...
    }}
```

Both self and the parameter other are used within the pattern expression, with the higher-order pattern Matrix given two patterns at both sites. Each subpattern is either a number value, computed from one of the matrices, or the Number type pattern itself for the "free" dimension in the parameter. This style makes clear what is required and when it will be checked, without introducing explicit code in the method body to test the dimensions on either input or output, and an error message can be reported automatically in clear terms. The Matrix pattern to provide this functionality need only be:

```
class Matrix(heightPat, widthPat) {
    method match(o) {
        if (!heightPat.match(o.height)
                || !widthPat.match(o.width)) then {
            failedMatch(o)
        } else {successfulMatch(o)}
```

```
    }}
```

4.5 Coercion and clamping

Many languages implicitly coerce values between types. A pattern can simulate this behaviour: because the result value of a successful match need not be the original input object, the pattern can perform whatever conversion is required and allow the code to carry on as-is. For example, a type that automatically "stringifies" values is as simple as:

```
def Stringify = object {
  method match(obj) {
    return successfulMatch(obj.asString)
  }}
```

This Stringify type can be used wherever another type is required, and e.g. methods with Stringify-typed arguments invoked with any type, but the body of the method will always see the string it expects:

```
method foo(x : Stringify) {
  print("x": " ++ x)
}
foo "hello"
foo 1
foo(widget.button "close")
```

Coercing types can also be applied to variables or fields, for example to clamp of a value to a range:

```
class Clamped(min, max) {
  method match(obj) {
    if (obj < min) then {successfulMatch(min)}
      elseif {obj > max} then {successfulMatch(max)}
      else { successfulMatch(obj) }
  }}
var rating : Clamped(1, 10)
rating := 999
print(rating)
```

will print 10, not 999.

These patterns explicitly have a run-time effect when applied: they change the values of variables, and can affect the eventual result of the program. Automated coercion in particular is often regarded as a misfeature in languages that contain it, and while we allow the programmer to "opt in" to any such behaviours, they (or their library or dialect author) might prefer to avoid them entirely.

4.6 Decorators

Similar to coercions, we can apply a decorator [16] to the argument object, perhaps to provide additional checking further along in the program, or diagnostics. If this decorator has the same shape as the original object (a true decorator), the client code can continue essentially without noticing a difference.

For example, given a conventional dynamic-language List type

```
type List = interface {add(_); get(_); set(_,_)}
```

with no constraints on its elements, we can write a decorator that ensures our code only stores and retrieves Strings from it:

```
def stringList = object {
  method match(o) {
    def mr = List.match(o)
    if (!mr) then { return mr }
    return successfulMatch(decorateStringList(o))
  }
}
class decorateStringList(l) {
  method add(s : String) { l.add(s) }
  method get(i) → String { l.get(i) }
  method set(i, s : String) { l.set(i, s) }
}
```

If we apply this pattern to our parameter or a variable, within the method we will have a typed list of strings: if we try to add anything that is not a string, we will receive the usual error due to the argument checks on add and get, and if we try to retrieve an existing item that is not a string an error will also be reported when the return value of get doesn't match String:

```
def myList : stringList = list(1,2,3)
myList.add("Hello") // OK
myList.add(1) // Dynamic error
myList.at(1) // Dynamic error
```

In a similar fashion, we can restrict our code to using only read operations, or transform values along the way (perhaps with our Stringify pattern from Section 4.5). A generic decorator lifting a type could provide something approaching the guarded gradual-typing semantics as in Typed Racket [50] or Reticulated Python [51] semi-automatically. We can both detect issues in our code, and alter how it behaves or interacts with the rest of the program through an annotation. In particular, decorating a return type means that the rest of the program will see the decorated result.

These decorators are not perfect replacements for the original object: because they are separate objects, they have different object identities and can be distinguished in some ways (for example, it may be possible for a hash table to contain both objects simultaneously) [32]. The underlying issues here are not due to the dynamic pattern system, but the extent of the support for transparent decorators in the underlying language runtime [34], but an implementation such as chaperones [44] would enable clean replacement.

4.7 General pre- and post-conditions

While the previous case studies have all examined or manipulated the inspected value, a pattern's match method can do anything at all to decide whether to succeed or not. For example, a pattern could check that a class invariant holds at the end of a method by being placed as the return annotation. The following class requires that "hp ≤ maxHP" at the end of each method, while allowing it to be violated within a single method.

```
class rabbit {
  var hp : Number := 10
  var maxHP : Number := 15
  def MyInvariant = Confirm { hp ≤ maxHP }
  method tick → MyInvariant { ... }
  method consume(item : Food) → MyInvariant { ... }
}
```

The Confirm pattern can be defined as follows:

```
class Confirm(predicate) {
  use basePattern
  method match(o) {
     if (predicate.apply) then { succeed(o) }
       else { fail(o) }
  }}
```

The return value itself is never scrutinised — only the predicate block is evaluated to determine whether to match or not — and the value is passed along unchanged. Nonetheless, a violation of the invariant will be detected and reported on the appropriate method. Multiple conditions could be chained together within the block, or combined with another pattern using & to conjoin them or ▹ to compose them, so a method checking *both* the invariant and that the result is a Number could be annotated → MyInvariant & Number.

A precondition can be checked by a similar pattern on any parameter to the method, while attaching the pattern to a field definition would validate it immediately each time the field value changed. With suitable definitions, this level of checking could be enabled during development and removed for production by eliminating the actual checking from the patterns and ignoring their combination with others.

5 Implementation

We have extended the Kernan implementation of the Grace language to support all of the functionality of our system. All of the code from the case studies is executable on our version of Kernan, and the complete code files including sample cases are included in the distribution.

Our extension uses a mix of code-rewriting and modifications of the run-time system. Because Kernan is a tree-walking interpreter, some aspects (particularly around variable names) are much simpler as rewrites, while other aspects (notably field and variable assignments) are simpler within the runtime. Regardless, the patterns in use are executed in the same way, and all of the checks, transformations, and reporting occurs in user code. The transformations do not rely on the interpreted nature of the system and should be equally applicable to a compiled object-oriented target. All pattern code executes as ordinary user code as though from a standard method call in that site.

The implementation is available from http://ecs.vuw.ac.nz/~mwh/fcdt-kernan.zip, which includes a pre-built binary distribution (which runs on all platforms with Mono or the .NET runtime) along with the source code.

6 Discussion

6.1 Error locations

There are three basic approaches to generating type errors when an argument (or assigned) value does not meet the annotation:

1. Generate a conditional as part of the rewrite that reports an error:

   ```
   if (!x′matchResult) then {
     reportTypeError "x does not satisfy type T"
   }
   ```

   The message format is fixed by the rewriting or runtime system, and not customisable by the type author, but all static information is available for use in it, including the parameter name and type annotation.
2. Call an additional method on the match result object, which can report the error with any phrasing or detail required:

   ```
   x′matchResult.assert "x"
   ```

   In this case, the parameter name must be passed as an argument in order for the error message to be able to include it.
3. Add no additional code and proceed unconditionally to the final assignment, but have the result method on *failed* matches throw an error:

   ```
   // Throws an error if x′matchResult is a failure
   def x = x′matchResult.result
   ```

   In this case, the reported error is *not* able to include the parameter or variable name, though it is able to use the non-matching value and any information given to the pattern. However, no extension to the existing pattern interface is required and sensible default behaviour can be provided.

   Wrapping the check in a try-catch and reporting a combination of static information and the dynamic message raised by result would support both needs, but introduce complexity to the implementation and explanation.

All three options have both positive and negative elements, and there is no clear winner for all circumstances. For the time being, we have elected to use the conditional approach because it is simplest and supports the rewriting presentation from Section 3.1, but the other approaches also have merit.

6.2 Destructuring patterns

Many pattern-matching systems, including the original proposal for Grace, support "destructuring" patterns that extract some values from the matched object and do one or both of matching further patterns against them and binding them to names. While this can be powerful, binding names in particular is a significant complexifier for the language and neither base Kernan nor our extension support it.

It is however possible to create patterns parameterised by "sub-patterns" they match against some exposed values of the match target, and to have the pattern require those sub-patterns to match in order to match itself. For example, this Point pattern allows subpatterns for the x and y coordinates:

```
class Point(xPat, yPat) {
  method match(o) {
    if (!xPat.match(o.x) || !yPat.match(o.y)) then {
      failedMatch(o)
    } else {successfulMatch(o)}
  }
}
```

A refined pattern LRPoint matching only points in the lower-right quadrant could then be defined as:

```
def LRPoint = Point(PositiveNumber, NegativeNumber)
```

along with specialisations for any other desired constraints, perhaps using the "be" construct from Section 4.4. Patterns like Point supporting this behaviour can be easily defined manually, reducing the value of automated destructuring and allowing flexibility in exactly *what* the sub-patterns address. We leave name-binding and its semantics in this context for future work. For the very simple single-value case, in our pattern library we have introduced a chain combinator x ▹ y that composes two patterns such that y must match the computed result of x.

6.3 Early and late binding

When pattern expressions are used as types, our implementation evaluates them as late as possible: when the method is called. Doing so enables the especially-dynamic behaviours of some of our case studies: a pattern expression can use the value of another parameter to construct the pattern, or what it resolves to can change entirely over the course of the program's execution. This also requires every pattern expression to be evaluated each time a method is called.

An alternative would be to evaluate type annotations early, at object construction time, and remember the resulting pattern objects to use for each subsequent check. The more dynamic behaviours are then not possible, but the type annotations need only be evaluated once for any object, and the resulting pattern objects may be made available for examination by *other* patterns performing deeper checks (for example, structural checks of parameter and return types), a functional advantage not available in our late-bound model.

For the present work we are most interested in the dynamic end of the spectrum, but an early-bound variant could still support many useful pattern types and is also worth exploring.

6.4 Optimisation

Our current implementation is built on top of the straightforward (naive) Kernan interpreter, which does not focus on performance, and adding additional dynamic evaluations as we have can only slow it further, as we would also expect for other implementations of the approach. We hope that judicious application of dynamic compilation should be able to greatly increase the performance, using similar techniques to Marr et al. [28] applied to more traditional meta-object protocols, or Roberts et al. [40], Richards et al. [39], and Vitousek et al. [52] to more dynamic type checking as part of gradual type systems. We expect that some explicit notion of purity to restrict or detect handling access and modification of mutable state [1, 19] will be an important part of such optimisations. For example, a type test that depends *only* on the type object and the structure of the receiver, reading no other state, and making no externally-visible assignments, offers a clear opportunity for caching and inlining.

6.5 Static Checking

Type systems for static languages must obviously be checked statically. Most type systems for dynamic languages are also designed to be checked statically — whether that's "optional" or "pluggable" types [4] which are ignored or erased before execution or "gradual" or "hybrid" type systems that incorporate both static and dynamic checking into the same language [8]. The approach we are advocating here is purely dynamic. We are interested in extending the benefits of this approach to incorporate static type checking where practical: in the limit this could require dependent type checking or full functional verification, but we expect that there will be a number of tractable cases that are also useful.

We see the same benefits to dynamically checking these types as, for example, with dynamic assertion checking as in Eiffel (or now most imperative languages, including Grace). We can write types that are effectively dynamic assertions on a single parameter value or return value: our system lets them be expressed "where they belong" — with the parameter or return declaration, like any other type — rather than incorporating them into general method pre- and post-conditions.

For example, rather than expressing a crypto-currency transfer [11] using only assertions in the body of the method:

```
method deposit(amt, src) {
  assert {amt > 0}
  assert {Purse.match(src)}
  assert {amt ≤ src.balance}
  ...
}
```

we can express the same conditions, and undergo the same dynamic checks, by using our dynamic dynamic checks:

```
method deposit(amt : Number & be ≥ 0,
               src : Purse & be.balance ≥ amt)
```

Of course, Grace's flexible match/case syntax then enables those "dynamic type patterns" to be used to direct control flow, rather than just check arguments or return values as in Pascal or Eiffel.

7 Related work

Racket's run-time contracts system [47, 50] permits dynamic user-defined checking of constraints on argument and return types. Racket contracts were originally designed to be layered on top of existing purely dynamically typed code [15]. Special forms for contracts allow Racket functions to be declared along with contract annotations; contracts themselves are first-class values and annotations are expressions of those values, so contracts can be defined with the full power of the language. Racket also supports various kinds of dependent contracts, contracts for class definitions, and contracts that manage interactions between multiple objects [10, 43].

Crucially, Racket contracts are higher-order: contracts e.g. may be applied to formal parameters that expect functions, and will check the behaviour of the actual functional arguments when those functions are applied. The same is true for methods when a higher-order contract is applied to an object. Higher-order contracts can have both a first-order predicate to test on application, and a *projection* to specify how the contract wraps the value it guards. If the contract is an *impersonator* [44] then the result of its projection can be anything, so it is possible to build a contract that returns arbitrary values based on what it wraps. For example, the following defines a contract add1/c that increments the number it guards:

```
(define add1/c
  (make-contract
    #:name 'add1/c
    #:first-order number?
    #:projection (const add1)))
```

Consider a procedure spy that acts as identity except that it also displays the given value before returning it:

```
(define (spy x) (displayln x) x)
```

(spy 1) prints 1 and returns 1. Without changing the body of the procedure, both the input and the output can be modified by applying a contract to the definition:

```
(define/contract (spy x)
  (→ add1/c add1/c)
  (displayln x)
  x)
```

Now (spy 1) prints 2 and returns 3.

Higher-order contracts, impersonators, and special contract defining forms mean that Racket contracts are strictly more powerful than Grace's dynamic pattern types — Grace's ambitions are rather more modest. Compared with Grace's design, Racket provides additional syntax on top of the base language for constructing definitions that use contracts, instead of integrating directly with the existing syntax. This is required because Racket's base syntax does not include type annotations, while Grace was designed to include type annotations from the start. As contracts are intended more for *enforcing* properties than the more general matching made available by patterns, Racket's existing pattern matching constructs do not naturally interact with contracts either.

Racket's implementation also makes a significant effort to assign blame correctly — that is, to indicate the *underlying cause* of an error, rather than raising an error only when a *presenting symptom* occurs [9, 17, 54]. Correct blame tracking in the presence of higher-order contracts contributes to the execution overhead of gradual typing in Racket [8, 20]. Grace's transient typechecks, on the other hand, can have a very low overhead given a suitable implementation [41]. Experience or empirical studies may be able to investigate the advantages and disadvantages of each approach in practice.

Many languages, particularly functional languages, have some degree of pattern-matching support, and some allow user-defined patterns. Grace patterns draw from Scala [12] and Newspeak [18], with the object model closer to Newspeak and the surface syntax closer to Scala. A key behavioural contrast with Newspeak is that in Grace matching always begins with the *pattern*, while Newspeak arguably does the "right" thing (or at least the pure object-oriented thing) in that the target of the match is always in control of the protocol and receives the first message. This difference crystallises a fundamental tension: irrefutable patterns, or unmatchable objects? Any system can have only one of the two, and it is key to our system in this work that a pattern can decide its answer without interference from the target object: regardless of the object under examination, a pattern can decide to succeed or fail based on outside conditions, or can decorate the object without examining it at all, which is not possible in a Newspeak-style model.

In many functional languages pattern-matching is a core element and can be used in the vicinity of parameters. For example, in Haskell, a piecewise function on an algebraic data type can be defined as:

```
show (Operator op l r) = (show l) ++ op ++ (show r)
show (Value v) = show v
```

with the types in parameter position. Haskell *view patterns* go further, and allow some transformational code to execute during matching [37]; these grew from Erwig and Peyton Jones's proposed "transformational patterns" [14], and ultimately from Wadler's introduction of views [53]. View patterns provide a dynamic value output from the input value, which may be further matched, and *partial views* encode the possibility of failure, by use of Maybe. An interesting element is the ability of view patterns to use earlier arguments as patterns themselves:

```
example :: (String → Int) → String → Int
example f (f → 4) = 0
example f (f → v) = v ∗ 2
```

In the above, the function accepts two arguments, a function from Strings to Integers, and a String, and returns a Boolean. The argument f is itself evaluated — and given the next (string) argument — and its return value can then be checked against further patterns (4) or bound to a name (v). In this way the function argument is in effect itself a pattern describing the following argument. While not by design, our system does allow the same:

```
method example(f, v : f)
```

would allow a pattern f as argument to be applied to the next argument. However, we do not currently permit the direct further matching of other patterns, which would relate to the destructuring feature discussed in Section 6.2, though it is again possible to simulate with a higher-order pattern, and in the simple case above with our ▹ combinator: `method example(f, v : f ▹ 4)`.

F♯'s *active patterns* [45] can fill a similar niche, but are somewhat more restricted in what they can use; TypeScript's type guards are also similar [29]. Both of these are more special-purpose mechanisms than Grace's general pattern-matching framework.

Multimethod systems in languages with pattern matching also have similarities. Thorn [3], Fortress [42], and OOMatch [38] all allow patterns in a wide range of positions, permitting piecewise or partial functions to be defined using them. These patterns can have a range of effects, but largely do not fill the "type checking" niche so much as destructured piecewise processing. Perl 6's Signatures [36] can have similar elements.

Some languages with static pre- and post-condition annotations on functions, such as Whiley [35], defer their execution to run time when they cannot be definitively verified or falsified statically, but these systems have quite different goals and presentation to our approach.

Languages in the Pascal tradition [23] at least as far as Modula-3 [31] have traditionally included integer and enumeration subranges that must be checked at runtime. C abandoned such exotica, and most subsequent "curly-bracket" languages followed that example.

The E programming language [30, 55] supports *guards* on field declarations and method arguments. Guards are similar to our first-class patterns in that they can accept a value, coerce it to a substitute value, or raise an exception. E includes syntax for brands ("trademarks"), reflecting over source code ("auditors"), and numeric relations.

Predicate Dispatching [13] generalises object-oriented multiple dispatch, incorporating predicates to control method selection, in a manner very similar to guards on patterns in functional languages. Grace is resolutely a single-dispatch language, however patterns, types, etc, can be used within match-case constructs to simulate multiple dispatch if necessary.

X10 supports Constrained Types [33] and Constrained Kinds [49] to allow methods to depend upon immutable properties of their arguments. Where possible, X10 can discharge the proof obligations flowing from the constraints: although where necessary, X10 will compile checks for constrained types and kinds and defer checks to runtime.

Redefining the behaviour of fundamental language constructs such as types is often considered a reflexive operation [27]. Grace's patterns do not have to involve reflexive programming (e.g.! the negativePattern from section 2) just uses a comparison) although they can when necessary, typically to scrutinise the object they are matching (e.g.! Methods from section 4.1). Grace's patterns thus straddle the boundary between meta and base levels; this is true of other parts of Grace's design, including control structures, which are simply defined by methods accepting closures as arguments.

The quintessence of reflexive programming is the Common Lisp Object System's Metaobject Protocol (MOP) [25]: the MOP allows programmers to control, extend, or replace the way the base object system works, customising object storage, generic invocations, and the way objects respond to invocations via method combinations. CLOS's closest analogue to Grace's patterns are *specializers* that take the place of Common Lisp's optional dynamic type annotations and are used to select methods. The standard MOP supports only classes or equality checks on individual objects as specializers: the MOP core would need to be extended to support the kinds of Bracha and Ungar's taxonomy of reflection discusses structural and behavioural reflection in depth, and using types to limit access to reflexive facilities, but does not consider reflecting on types themselves [5].

CLOS's spiritual successor, aspect-oriented programming as embodied by AspectJ [26], doesn't support the same level of customisation, but is able to achieve some of the same goals as Grace's patterns. While programmers cannot redefine types in AspectJ, they can wrap new code around existing method calls or field accesses. The method or fields can be chosen based upon the types of each argument position, and the base program values can be accessed, modified, and replaced as part of the wrapping code. Other aspect-oriented langauges such as Reflex [48] have similar limitations.

Finally, there has been significant work on dependent types within statically typed functional languages, work which is now extending to object-oriented languages [7]. The design tradeoffs of such systems are well known: increasing the complexity of the type system can lead to better error detection and better error messages, and the surety provided by the type system means that redundant code paths or error handling code can be eliminated. This work in some sense explores a complementary point in the language design space: dependent, first-order types, in a dynamic, imperative, object-oriented setting. We find some of the same tradeoffs certainly apply: types are more complex, errors can be detected sooner (albeit at runtime) and error reports can be better — e.g. an integer argument out of range detected at an object's interface, rather than a subsequent indexing error in a collection deep inside the object. What we did not expect initially is that employing patterns as first-class dynamic types would offer opportunities to simplify and shorten method code, and could provide these benefits even for mutable properties in imperative programs. A dependent type requiring a mutable list to have a length of at least three at the entrance to a method, say, still offers programmers significant value — at least as much as a method precondition or assertion about the length of the list. We find type annotations are generally easier to understand than assertions: because annotations are specific to a particular parameter type, return type, or field declaration, their scope is clearer and they can be shorter than the corresponding assertions. Type annotations are easier to capture in interface specifications and documentation, and programmers coming from static languages will already know how to write them.

8 Conclusion

Extending an advanced dynamic pattern-matching system to support first-class dynamic types allows a programmer to express — and enforce — exactly the style of checks they want, in the time, place, and manner of their choosing. From simple conventional type systems applied only where wanted, to advanced dependent checks or coercions, a wide range of checks are available and limited only by what the programmer can implement. By leveraging the language's existing type annotations, the checks are unintrusive and stay out of the way on the client side, rather than introducing complex pre- or post-conditions at the method site.

Truly first-class dynamic types introduce great power and flexibility into the language, coupled with concision, and allow the full power of the host language to come to bear in designing types and defining their semantics.

Acknowledgments

This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden.

References

[1] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Mathieu Suen. 2010. Read-Only Execution for Dynamic Languages. In *TOOLS*. 117–136.

[2] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software*. ACM, New York, NY, 85–98. http://doi.acm.org/10.1145/2384592.2384601

[3] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. 2009. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *OOPSLA*.

[4] Gilad Bracha. 2004. Pluggable Type Systems. OOPSLA Workshop on Revival of Dynamic Languages.

[5] Gilad Bracha and David Ungar. 2004. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA*. 331–344.

[6] Kim Bruce, Andrew Black, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. 2013. Seeking Grace: a new object-oriented language for novices. In *SIGCSE*.

[7] Joana Campos and Vasco T. Vasconcelos. 2018. Dependent Types for Class-based Mutable Objects. In *ECOOP*.

[8] Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: Gradual Typing for Objects. In *ECOOP*.

[9] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *POPL*.

[10] Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, Please Don't Let Contracts Be Misunderstood (Functional Pearl). In *ICFP*.

[11] Sophia Drossopoulou, James Noble, and Mark Miller. 2015. Swapsies on the Internet: First Steps towards Reasoning about Risk and Trust in an Open World. In *(PLAS)*.

[12] Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *ECOOP*. 273–298.

[13] Michael D. Ernst, Craig Kaplan, and Craig Chambers. 1998. Predicate Dispatching: A Unified Theory of Dispatch. In *ECOOP Proceedings*.

[14] Martin Erwig and Simon Peyton Jones. 2000. Pattern Guards and Transformational Patterns. In *Haskell Workshop*.

[15] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ICFP*.

[16] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1994. *Design Patterns*. Addison-Wesley.

[17] Ronald Garcia. 2013. Calculating Threesomes, with Blame. In *Proceedings of the 18th International Conference on Functional Programming (ICFP'13)*. 417–428. https://doi.org/10.1145/2500365.2500603

[18] Felix Geller, Robert Hirschfeld, and Gilad Bracha. 2010. *Pattern Matching for an Object-Oriented and Dynamically Typed Programming Language*. Technical Report 36. Hasso-Plattner-Instituts für Sofwaresystemtechnik an der Universität Potsdam.

[19] Donald Gordon and James Noble. 2007. Dynamic Ownership in a Dynamic Language. In *Dynamic Languages Symposium (DLS)*.

[20] Ben Greenman and Matthias Felleisen. 2018. A spectrum of type soundness and performance. *PACMPL* 2, ICFP (2018), 71:1–71:32. https://doi.org/10.1145/3236766

[21] Michael Homer, Timothy Jones, James Noble, Kim B Bruce, and Andrew P Black. 2014. Graceful dialects. In *ECOOP (LNCS)*, Richard Jones (Ed.), Vol. 8586. Springer, 131–156.

[22] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns as objects in Grace. 17–28. https://doi.org/10.1145/2384577.2384581

[23] Kathleen Jensen and Niklaus Wirth. 1974. *PASCAL user manual and report.* Springer-Verlag.

[24] Timothy Jones, Michael Homer, and James Noble. 2015. Brand Objects for Nominal Typing.

[25] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol.* MIT Press.

[26] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP*.

[27] Pattie Maes. 1987. Concepts and Experiments in Computational Reflection. In *OOPSLA*.

[28] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-overhead metaprogramming: reflection and metaobject protocols fast and without compromises. In *PLDI*.

[29] Microsoft Corp. 2019. TypeScript Handbook. www.typescriptlang.org/docs.

[30] Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Baltimore, Maryland.

[31] Greg Nelson. 1991. *Systems programming with Modula-3.* Prentice-Hall.

[32] James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Mark S. Miller. 2016. The Left Hand of Equals. In *ONWARD! Essays*.

[33] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. 2008. Constrained types for object-oriented languages. In *OOPSLA*.

[34] Geoffrey A. Pascoe. 1986. Encapsulators: A New Software Paradigm in Smalltalk-80. In *OOPSLA*.

[35] David J. Pearce. 2017. Rewriting for Sound and Complete Union, Intersection and Negation Types. In *GPCE*. 14.

[36] Perl 6. [n. d.]. class Signature, Perl 6 documentation. https://docs.perl6.org/type/Signature, accessed 13 May 2019.

[37] Simon Peyton Jones. 2007. View patterns: lightweight views for Haskell. (2007). http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns.

[38] Adam Richard and Ondřej Lhoták. 2008. OOMatch: Pattern Matching as Dispatch in Java. In *FOOL*.

[39] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing. In *OOPSLA*.

[40] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks are (Almost) Free. In *ECOOP*.

[41] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks Are (Almost) Free. In *ECOOP*. 5:1–5:28.

[42] Sukyoung Ryu, Changhee Park, and Guy L. Steele Jr. 2010. Adding Pattern Matching to Existing Object-Oriented Languages. In *FOOL*.

[43] T. Stephen Strickland and Matthias Felleisen. 2010. Contracts for First-Class Classes. In *DLS*.

[44] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*.

[45] Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible Pattern Matching Via a Lightweight Language Extension. In *ICFP*.

[46] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16).* ACM, 456–468.

[47] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-class Classes. In *OOPSLA*.

[48] Éric Tanter, Rodolfo Toledo, Guillaume Pothier, and Jacques Noyé. 2008. Flexible metaprogramming and AOP in Java. *Science of Computer Programming* 72 (2008), 22–30.

[49] Olivier Tardieu, Nathaniel Nystrom, Igor Peshansky, and Vijay Saraswat. 2012. Constrained Kinds. In *OOPSLA*.

[50] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *POPL*.

[51] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *DLS*.

[52] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. *CoRR* (2019). arXiv:1902.07808

[53] P. Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *POPL*.

[54] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *ESOP*. 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

[55] Ka-Ping Yee and Mark S. Miller. 2003. Auditors: An Extensible, Dynamic Code Verification Mechanism. http://www.erights.org/elang/kernel/auditors/index.html