Grace: the absence of (inessential) difficulty

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

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

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

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

Grace is the absence of everything that indicates pain or difficulty, hesitation or incongruity.
William Hazlitt, *The Round Table (1817)*

Abstract

We are engaged in the design of a small, simple programming language for teaching novices object-oriented programming. This turns out to be far from a small, simple task. We focus on three of the problems that we encountered, and how we believe we have solved them. The problems are (1) gracefully combining object initialization, inheritance, and immutable objects, (2) reconciling apparently irreconcilable views on type-checking, and (3) providing a family of languages, each suitable for students at different levels of mastery, while ensuring conceptual integrity of their designs. In each case our solutions are based on existing research; our contribution is, by design, consolidation rather than innovation.

1. Introduction

Although object-oriented programming is widely taught in introductory computer science courses, no existing object-oriented programming language is the obvious choice for a teaching language. This makes it harder to transfer skills, techniques, and teaching materials between courses and between institutions. During ECOOP 2010, a group of language researchers and educators concluded that the time was ripe for an effort to design a language focussed on teaching.

A "design manifesto" was presented at SPLASH 2010 [5], in which we attempted to lay out design principles for such a language; as we did so, it became clear that the principles were often in conflict and that resolving these conflicts would be challenging. Since then three of us (Black, Bruce and Noble) have been meeting weekly to pursue the design of the language, which we have named "Grace", in honor of Admiral Grace Hopper, and in the hope that the name would serve as an admonition not to settle for less-than-graceful solutions. Homer has built a preliminary implementation of the core of Grace. which we review in Section 6.

From the beginning, our high-level goal has been to integrate proven newer ideas in programming languages into a simple teaching language whose features represent the key concepts of object oriented-programming directly, simply and gracefully. We feel that this is important because we want to focus the attention of our students on the *essential*, rather than the *accidental*, complexities of programming and modeling. We hope that the design sketched here meets this goal.

The preliminary design for Grace has been discussed on our blog[1], and a draft specification is available at the same site. The purpose of this paper is not primarily to review the state of Grace, but rather to focus on three problem areas where we believe we have found Graceful resolutions to what seemed to be difficult problems: Section 3 discusses how we construct objects, Section 4 explains how we treat types, and Section 5 summarizes our approach to providing different "levels" of Grace for teaching students at different stages of mastery. Because a language is an ecosystem in which everything is connected to everything else, these sections cannot be independent. We therefore start with an overview of Grace.

2. Grace in a Nutshell

Grace is an imperative object-oriented language with block structure, single dispatch, and many familiar features. Our design choices have been guided by the desire to make Grace look as familiar as possible to instructors who know

[1] http://www.gracelang.org

other object-oriented languages, and by the need to give instructors and text-book authors the freedom to choose their own teaching sequence. Thus, in Grace it is possible to start using types from the first, or to introduce them later, or not at all. It is also possible to start with objects, or with classes, or with functions. Most importantly, instructors can move from one approach to another while staying within the same language.

Grace can be regarded as either a class-based or an object-based language, with single inheritance. A Grace class is an object with a single factory method that returns an object:

```
class aCat.named(n : String) {
        def name = n
        method meow { print "Meow" }
}
var theFirstCat:Cat := aCat.named "Timothy"
```

Here the class is called aCat and the factory method named(). After executing this code sequence, theFirstCat is bound to an object with two attributes: a constant field (name), and a method meow. The expression c.name answers the string object "Timothy" and c.meow has the effect of printing *Meow*.

An object can also be constructed using an object literal — a particular form of Grace expression that creates a new object when it is executed. In addition to fields and methods, an object literal can also contain code, which is executed when the object literal is evaluated. For example:

```
var theSecondCat := object {
        def name = "Timothy"
        method meow { print "Meow" }
        print "Timothy now exists!"
}
```

This code has the effect of printing "Timothy now exists!", and binding the variable theSecondCat to a newly-created object, which happens to be operationally equivalent to theFirstCat.

As we will see in Section 3, a class is equivalent to an object with a factory method that contains an object literal. Thus, an instructor who wishes to start teaching with objects need not talk about classes at all until later. Classes are in the language because we felt that they were important for convenience, and to help make the connection between Grace and existing languages.

Mutable and immutable bindings are distinguished by keyword: **var** defines a name with a variable binding, which can be changed using the := operator, whereas **def** defines a constant binding, initialized using =, as shown here.

```
var currentWord := "hello"
def world = "world"
...
currentWord := "new"
```

The keywords **var** and **def** are used to declare both local bindings and fields inside objects.

An object's methods are immutable, in the sense that once an object is created, the code of its methods cannot be changed. A field that is declared with **def** is constant; the binding between the field name and the object cannot be changed, although the object, if mutable, may change its state. Each constant field declaration creates an accessor method on the object. For example, the object club defined by

```
def club = object {
        def members = MutableSet.empty
}
```

has a *method* called members that returns the current set of members. The value of this set may change over time, for example, after executing club.members.add(anApplicant).

Declaring a field with **var** creates two accessor methods, one for fetching the currently bound object and one for changing it. So, after the declaration

```
def car = object {
        def numberOfSeats = 4
        var speed: Number := 0.
}
```

the object car will have three methods called numberOfSeats, speed, and speed:=(). When we use () in the name of a method, it indicates the need to supply arguments. So, the last method might be used by writing car.speed := 30. Variable fields have no value until they are initialized. Because Grace does not define a universal "nil" object, there is no default value that could be used to initialize all variables. We expect that most variable fields will be initialized when they are declared, as in the example above. Attempting to access an uninitialized field is an error that the implementation will detect, and that will cause program termination.

Grace will support visibility annotations that allow the programmer to restrict access to fields and methods from outside an object by marking them as *public* or *confidential*. For simplicity, we do not discuss this further here, and omit visibility annotations in all the sample code in this paper.

In Grace we say that a method is invoked using a "method request". We introduce this terminology to distinguish the operation — fundamental to object-orientation — of *asking* an object to do something, where the choice of what to do is made by the object itself, from procedure or function call, where the choice of operation is made by "the caller". This distinction is also conveyed by Smalltalk's "message send" terminology, but now that networks and distributed systems are ubiquitous, "sending a message" has become an ambiguous term.

All the attributes of an object (methods, variable fields, and constant fields) exist in the same namespace and with the same lookup rules. There is thus a potential ambiguity in the interpretation of a name n. To resolve this, we disallow "shadowing" of variable names in enclosing scopes; one or

other of the variables must be renamed. Thus, if n is defined in the local scope, it cannot also be imported from an enclosing scope. It may be inherited, in which case the local definition overrides the inherited one. If there is no local declaration of n, you might wonder whether n is a reference to a variable (or constant) declared in an enclosing (static) scope, or a request **self**.n of an inherited method. To remove this ambiguity, we disallow this situation too. We do not allow n to be declared in a statically enclosing scope if it is also inherited. If that is the case, the programmer must rename the variable n in the enclosing scope.

Bracha has compared various options in this space [7]. Our choices might be too restrictive for an industrial-strength programming language, but for teaching purposes we believe them to be appropriate.

Making field access syntactically identical to a self method request is deliberate, following the lead of Eiffel [25] and Self [32]. This allows the programmer to override a field's implicitly-defined accessor method with a custom method, and thus allows the programmer to handle interface changes, implement bounds checking, or perform some other similar task without affecting the interface of the object.

An object containing no mutable state is by default equal to any other object with the same structure and values for its fields. This follows Baker's Egal predicate [2]. The equality method that implements this test is generated automatically.

Grace method names may consist of multiple parts ("mix-fix" notation) as in Smalltalk [16]. Separate lists of arguments are interleaved between the parts of the name, allowing them to be clearly labelled with their purpose. Thus we might define on Number objects

```
method between (l:Number) and (u:Number) {
    return (l < self) && (self < u)
}
```

The syntax of a method request is similar to that used in Java, C++, and many other object-oriented languages: obj.meth(arg1, arg2), but extended to allow meth to have multiple parts. We could request the above method between ()and() on 7 by writing

```
7.between(5) and(9)
```

Single arguments that are literals do not require parentheses, so alternatively we could write

```
7.between 5 and 9
```

Following many other languages, the receiver **self** can be omitted. We have already seen several messages requested of an implicit receiver; for example, print "Meow" is short for **self**.print "Meow".

Grace also allows operator symbols (and sequences of operator symbols) to be used to name methods. A method name composed of operator symbols is used as a binary infix operator, unless it is defined using the **prefix** keyword, in which case it is a unary prefix operator. There is no ambiguity because the receiver must be explicit when requesting operator methods. Thus, a − b is a request of the binary minus method on object a, while − b is a request of the prefix negation method on object b.

Grace includes first-class blocks (lambda expressions). A block is written between braces and contains some piece of code for deferred execution. A block may have arguments, which are separated from the code by −>, so the successor function is {x −> 1+x}. A block can refer to names bound in its surrounding lexical scope, and returns the value of the last-evaluated expression in its body.

Control structures in Grace are methods. The built-in structures are defined in the basic library, but an instructor or library designer may replace or add to them. Control structures are designed to look familiar to users of other languages:

```
if (x > 5) then {
    print "Greater than five"
} else {
    print "Too small"
}

for (node.children) do { child ->
    process(child)
}
```

Notice that the use of braces and parentheses is not arbitrary: parenthesized expressions will always be evaluated exactly once, whereas expressions in braces are blocks, and may thus be evaluated zero, one, or many times. A **return** statement inside a block terminates the *method* that lexically encloses the block, so it is possible to program *quick exits* from a method by returning from the *then* block of an if()then() or the *do* block of a while()do().

Instructors could provide their own for()do() methods with debugging enhancements or additional restrictions, either to replace the built-in version or to sit alongside it. A particular use of this facility is to define iterators that require the student to specify loop invariants.

String literals, written between double quotes, support interpolation, using a syntax similar to that of Ruby. Code inside braces within a literal is evaluated when the string is; the asString method is requested on the resulting object, and the answer is inserted into the string literal at that point.

```
print "1 + 2 = {1 + 2}" // Prints "1 + 2 = 3"
```

While Grace uses braces to delimit blocks and other literals, it also enforces correct indentation. Braces and indentation may not be inconsistent with one another: the body of a method, for example, must be indented. Enforcing this in the language ensures that students will learn good practice, and avoids the common problem of not being able to find a mismatched brace because of the tendency of one's eye to believe the indentation rather than the braces.

Grace code can also be written in "script" form, without object or class definitions. We imagine such code as being enclosed in a top-level object literal. Thus, methods can be defined at the top level, and any code written at the top level will be executed immediately. This supports imperative-first teaching styles.

We recognize the growing importance of parallelism, and recognize that Grace needs to support the teaching of parallel and concurrent programming. Unfortunately, there is as yet no consensus on the best way to teach these concepts. We intend to use libraries to provide a variety of language mechanisms for parallelism and concurrency. These efforts are very much a work-in-progress, and are not discussed further here.

3. Constructing Objects

Having completed our overview of Grace, let us turn to the first problem area: object construction and initialization. This is often quite tricky, especially for objects that are defined using inheritance. Many languages that are currently used for teaching novices impose a lot of overhead on the process of creating even the simplest objects.

3.1 The Problem

Let's look at making a simple AddressCard object in Smalltalk and in Java. In Smalltalk, we need to define a class AddressCard:

```
AddressCard (subclass of Object)
    instance variable names: name address
    methods:
        name
            ↑ name
        address
            ↑ address
        email
            EmailApp openComposerWindowTo: name
    private methods:
        setName: nm address: adr
            name := nm.
            address := add.
            ↑ self
```

and a metaclass AddressCard class:

```
AddressCard class
    methods:
        for: aName address: anAddress
            ↑ self basicNew
                  setName: aName address: anAddress
```

Then we can use an AddressCard object like this

```
| ww |
ww := AddressCard for: 'Wackiki Wabbit'
          address: 'ww@WarnerBros.com'.
ww email
```

The weakness of the Smalltalk design, from the point of view of teaching novices, is that there is quite a lot to explain here, and we have to explain all of it before our students can define and create the very simplest object. AddressCard defines the instance variables and methods of every AddressCard object. In contrast, AddressCard class defines the behavior of the factory object that is used to make new AddressCard objects; it has a method for:address: that exists for this purpose. Although AddressCard objects are intended to be immutable, there has to be a method (here called setName:address:) whose only purpose is to allow the for:address: method in AddressCard class to initialize the instance variables of the newly created object.

Although we don't have to explain the whole idea of the class–metaclass hierarchy before students construct their first objects, it is still hanging there as a teaser to the smart students: if every object has a class, and every class is an object, then what's the class of the class's class? And what of that class?

The strength of the Smalltalk design is that once these things have been understood, the students can apply what they have learned, and discover that the same principles are at work everywhere in the Smalltalk system. For example, basicNew isn't magic: students can look at its implementation and see how it is implemented. Neither are classes magic: they are just objects that ultimately inherit from Behavior. The instance-variable-setting method setName:address: may be inconvenient, but it has to exist because an object's instance variables are accessible only by that object; there is no "special exception" for the object's class, which is, after all, just another object.

Java, following C++ and other languages, attempts to simplify the situation by combining the class and the metaclass into a single syntactic construct, the Java Class, and by replacing object encapsulation by class encapsulation. The corresponding Java class looks like this:

```
class AddressCard {
    private String nameField;
    private String addressField;
    // constructor method //
    public AddressCard(String nm, String adr) {
        nameField = nm;
        addressField = adr;     }
    public String address() {
        return addressField;    }
    public String name() {
        return nameField;       }
    public void email() {
        EmailApp.openComposerWindowTo(nameField);
    }
}
```

The strength of the Java design is that, because Java classes are not objects, they don't themselves have classes, so there is no class–metaclass hierarchy and no danger of students having their heads explode trying to figure it out. The Java design has its own weaknesses, however. Jettisoning the idea that classes are objects means that inheritance doesn't work

on static methods. Because of this, instead of each class being equipped with a static method basicNew, Java needed — and instructors must explain — two language primitives: the operation new, which creates an object, and "constructor methods", the parameters of which come from new, but which side-effect the newly constructed object. All of this is, from the perspective of the student, "magic". It cannot be deduced from a few underlying principles: it is "just the way that the language works", and must be memorized. Moreover, one of the overarching principles of object-orientation — that information hiding applies to objects, not classes — has been lost as an object can access the private instance variables of other objects of the same class.

The process of object creation becomes even more complicated when we add inheritance to the picture, although it may be possible to postpone that for a few weeks, depending on the teaching sequence. Once inheritance is introduced, we have to explain how the inherited instance variables are initialized, and how the initial values are passed from the constructor of the inheriting class to its parent. Then we have to teach programming patterns that make that reuse of the parent robust to change.

3.2 Object Construction in Grace

Grace seeks to play on the strengths of both Smalltalk and Java by giving the student a small number of underlying principles that can be applied uniformly, and by simplifying the syntax so that everything about an object can be specified in a single construct. As a result, we believe that it presents a much simpler picture of objects to the student.

Grace objects are self-contained: conceptually, each object owns its *own* fields and methods. As in Emerald, O'Caml, and JavaScript, objects are created by executing a language expression, which in Grace is called an object literal. There is no need to introduce classes. Here is how the above address card object might be created in Grace:

```
object {
    def name = "Wackiki Wabbit"
    def address = "ww@WarnerBros.com"
    method email
        { EmailApp.openComposerWindowTo(name) }
}
```

Execution of an object literal results in a new object with the attributes between the braces. Object literals are naturally first-class: they can be passed as arguments, returned as results, and bound to variables, so we might give a name to our object using a **var** or **def** declaration:

```
def ww = object {
    def name = "Wackiki Wabbit"
    def address = "ww@WarnerBros.com"
    method email
        { EmailApp.openComposerWindowTo(name) }
}
```

Grace's information hiding rules tell us that, from the outside, the object ww above and the object ww' that follows

```
def ww' = object {
    method name
        { "Wackiki Wabbit" }
    method address
        { "ww@WarnerBros.com" }
    method email
        { EmailApp.openComposerWindowTo(name) }
}
```

are indistinguishable: both appear to their clients to have *methods* called name, address, and email. The fact that some of these methods are implemented by field access and others by executing code is no object's business but ww and ww' themselves.

Of course, it is quite common to want to create many address cards, which will need to be initialized with different values. To achieve this, we need to parameterize over the values of the fields. In Grace, we do this in the same way that we parameterize over anything else: we write a method with parameters.

```
method for (nm) address (adr) {
    object {
        def name = nm
        def address = adr
        method email {
          EmailApp.openComposerWindowTo(name) }
    }
}
```

Such a method is conventionally called a *factory method*, because it manufactures an object, but in Grace there is nothing special about a factory method: it is just a method that returns the result of executing an object literal, and is not fundamentally different from any other code that makes an object.

It's usually convenient to put such a method in a named object, so that it can be re-used easily:

```
def anAddressCard = object {
    method for (nm) address (adr) {
        object {
            def name = nm
            def address = adr
            method email {
              EmailApp
                .openComposerWindowTo(name) }
        }
    }
}
```

Now the student can write:

```
def ww = anAddressCard.for "Wackiki Wabbit"
     address "ww@WarnerBros.com"
def fjl =  anAddressCard.for "Foghorn Leghorn"
     address "fjl@WarnerBros.com"
```

We expect that this pattern — an object containing a single factory method — will be common. Although it's quite simple, it does depend on nesting (an object literal in a method in an object literal), and that can be hard for novice students to understand at first. For instructors who want to delay a discussion of nesting, or who just prefer a more concise or more familiar syntax, the following class declaration accomplishes the same thing:

```
class anAddressCard.for (nm) address (adr) {
    def name = nm
    def address = adr
    method email
        { EmailApp.openComposerWindowTo(name) }
}
```

Thus, a Grace class is really just a factory object. In other languages, classes play a multitude of rôles; Borning lists no less than eight [6]. Grace classes are not meta-objects that describe their instances; nor are they types. We agree that meta-objects and types are important. Indeed, we think that they are sufficiently important to have their own representations in Grace, leaving classes to be generators of new objects.

3.3 Adding Inheritance

Students of language design will see that there is nothing new here: Grace's object literals are clearly based on Emerald's object constructors [4]. Emerald did not support inheritance, and Grace must. How can we add inheritance to the picture?

Our solution draws from the original use of prefix classes in Simula, which defined the meaning of class prefixing in term of concatenation [27], and the work of Taivalsaari on delegation [30]. In a conventional class-based language, derived objects *share* the methods of their base class, but have their *own* independent set of fields. Taivalsaari observed that in an object-based language, the effect of inheritance could be obtained by concatenating a conceptual *copy* of the base object, both methods and fields, to the methods and fields of the derived object. It's vitally important that each object has its own set of fields; whether the methods are actually copied or shared is semantically irrelevant, since they are immutable.

In Grace, one inherits from an object, as shown in the following example.

```
def wwPhoneCard =
    object {
        inherits anAddressCard.for "Wackiki Wabbit"
            address "ww@WarnerBros.com"
        def phoneNumber = "866−373−4389"
        method call {
            PhoneDialer.dial(phoneNumber) }
    }
```

The object wwPhoneCard inherits all of the (public and confidential) methods and fields of anAddressCard.for "Wackiki Wabbit" address "ww@WarnerBros.com". To these it adds a new field phoneNumber and a new method call. It's also possible for the derived object to override inherited methods, and to make super-calls to overridden methods. Inheritance can also be used with the class syntax; the meaning is that the object constructed by the factory method inherits from the object that follows the **inherits** keyword.

```
class aPhoneCard.for(nm)address(adr)phone(nbr) {
    inherits anAddressCard.for (nm) address (adr)
    def phoneNumber = nbr
    method call {
        PhoneDialer.dial(phoneNumber)
    }
}
```

Grace restricts the expression that appears after **inherits** to be "definitively static". The normal use case is that this expression is a method request on a factory object. This restriction means that it is always possible to ascertain statically which attributes (fields and methods) are being inherited, and allows the compiler or the IDE to warn the programmer if methods are being overridden without override annotations, or *vice versa*.

Notice that the attributes of an object never change over that object's lifetime. Here Grace stands in contrast to languages like JavaScript, Python, and Ruby, where fields can be added to objects at any time. While this may make Grace less dynamic than these languages, we also think it makes Grace programs easier to design and to understand than programs in more dynamic languages. Crucially, this also means that Grace programs can be type-checked with a straightforward type system.

4. Type Checking

To separate the concepts of class and type, and to allow different styles of teaching that introduce these concepts in different orders, a Grace class is not a type, nor does a Grace class or object implicitly define a type. When programmers need types they must define them explicitly.

Most Grace types are structural — sets of method names and signatures. The keyword **method** is omitted in type literals, since only methods can be in a type. Thus, the object ww from Section 3.2

```
def ww = object {
    def name = "Wackiki Wabbit"
    def address = "ww@WarnerBros.com"
    method email
        { EmailApp.openComposerWindowTo(name) }
}
```

has the (anonymous) type

```
{
    name −> String
    address −> String
    email −> Nothing
}
```

Recall that, from the outside, fields are indistinguishable from methods.

A **type** declaration can be used to name this type:

```
type AddressCard = {
    name -> String
    address -> String
    email -> Nothing
}
```

The key relation between types is conformance. We write B <: A to mean B conforms to A; we say that the "subtype" B conforms to the "supertype" A. Grace's conformance relationship is standard: a subtype must contain all the methods of a supertype, result types are covariant, and argument types contravariant.

Because Grace types are structural, any object that has methods with conforming method signatures conforms to the type: no inheritance relationships or implements declarations are necessary. The various address card and phone card objects from Section 3 all conform to the AddressCard type declared above, but they will also conform to the smaller type { name -> String }, which requires a conforming object to have a name method that returns a String.

Grace method declarations can include the types for their arguments and results. For example, the method doubleUpNames below takes as argument any object that has a name method that returns a String.

```
method doubleUpNames(
  namedObject : { name -> String } ) -> String
    { namedObject.name ++ namedObject.name }
```

Classes can be tied into the type system simply by giving their factory method a return type:

```
class anAddressCard.for (nm) address (adr)
  -> AddressCard { ... }

class aPhoneCard.for(nm)address(adr)phone(nbr)
  -> PhoneCard { ... }
```

This shows the power of a simple mechanism used consistently.

As examples in earlier sections have shown, local variables, methods and fields are not required to declare their types. Grace is gradually typed: omitted types of local variables and constants, and method results are inferred; but omitted argument types are treated as the predefined type Dynamic. This ensures that any requests are dynamically checked, as in C# [3]. In this way, Grace supports both statically and dynamically-typed code; indeed, programmers can choose at the level of an individual declaration.

Within dynamically-typed code, types need not be mentioned at all, and so their introduction can be delayed until late in the teaching sequence. When instructors do introduce types, they may do so in the language they are already using, as opposed to, for example, starting teaching in Python and then transitioning to Java. A simple static type checker will support instructors who wish to require that all student programs are fully typed.

Grace defines a type Any that defines no methods and to which any object conforms, and a type None that defines all methods and to which no object conforms. The type Nothing is a placeholder for methods that do not return a result. These types help ensure that every expression has a principal type, and every type can be denoted in the language. We hope this will give us a chance of explaining every type in students' programs in terms that they can understand.

As well as type literals, Grace provides a small selection of type operators. T1 & T2 is the type of all objects that contain the methods in T1 as well as the methods in T2. In contrast, T1 + T2 is the type of objects that have all the methods common to *both* T1 and T2.

These type operators allow more complex types to be built up from simpler types. As a consequence, Grace's type system needs no special support for inheritance, because & types are all we need. For example, the type of a phone card object can be defined by "anding" the type of an address card and the type of the methods added by the subclass:

```
type PhoneCard = AddressCard & {
    phoneNumber -> Number
    call -> Nothing
}
```

Grace's type system has two non-structural features to support multi-way branches and pattern matching: variant types and singleton types. Variant types [19], which are written T1 | T2, are similar to structural sum types T1 + T2 in terms of the requests they permit on objects of the type. In both cases, only those requests common to both T1 and T2 may be made. The difference is that, whereas any object supporting just these common requests would conform to T1+T2, only subtypes of T1 and subtypes of T2 conform to the variant type T1 | T2.

For example, an object that responds to nothing but an email method conforms to the structural sum type

AddressCard + {email -> Nothing; url -> String}

because that sum type is equivalent to the structural type

{email -> Nothing}

Such an object would not conform to the variant type

AddressCard | {email -> Nothing; url -> String}

because neither

{email -> Nothing} <: AddressCard

nor

{email -> Nothing} <:
  {email -> Nothing; url -> String}

even though the only request that can be made via a reference of the variant type is precisely {email -> Nothing}.

Variant types let the typechecker guarantee that multi-way branches are exhaustive. Inspired by Scala, a Grace match statement (itself defined as a library method) takes an object and selects one of a series of single-argument blocks:

```
method handle(x :
    AddressCard | {email −> Nothing; url −> String})
{
 match (x)
   case { a : AddressCard −>
          print "Name: {a.name}" }
   case { u : {email −> Nothing; url −> String} −>
          print "URL: {u.url}" }
}
```

By checking that each leg of the variant type is handled by one of the cases, the typechecker can ensure that this match is exhaustive, so that no catch-all clause needs to be supplied, and a missing case exception cannot be generated.

Similar to Scala, singleton types denote individual objects (although unlike Scala, Grace's singleton types do not admit null). Once we have defined a "missing value" object that will act as a singleton

```
def missing = object {
    method asString { "Missing Value" }
}
```

we can use this in other types to represent potentially missing data:

```
class databaseRecord.new(id', name') {
    var id : Number | singleton(missing) := id'
    var name : String | singleton(missing) := name'
}
```

Some simple objects, notably strings and numbers, can act as their own singleton type, supporting very concise multi-way branches

```
method howMany( n : Number ) −> String {
  match (n)
    case { 1 −> "One" }
    case { 2 −> "Two" }
    case { _ −> "Lots of" }
}
```

and to implement simple enumerations:

```
type Colour = "red" | "blue" | "green" |
              "cyan" | "magenta" | "yellow" | "puce"
```

Types are gradual: dynamically-typed and statically-typed code can co-exist in the same program. When objects move from dynamic to static code, a run-time type check is performed, and an error may be given. Types can be introduced into an existing program gradually, without needing to add types everywhere before the program can be executed.

There are several options for interpreting types in a gradual system. In a safe static type system, the compiler guarantees in advance that no type errors will occur at run-time. In a safe dynamically typed system, the run-time system also guarantees (though this time by inserting dynamic checks) that no type errors will occur at run-time.

What is the meaning of a type annotation in a dynamically-typed system? As suggested above, one can set up the system so that dynamic checks are inserted when objects move from static to dynamic code. What should happen when such a test fails? Certainly, at a minimum, the system should report that failure. It could at that point terminate the execution of the program; however, it could be the annotation that is wrong. In that case it might make more sense to continue execution after the error report in order to determine if there is a later catastrophic error (e.g., a "method request not understood" error) that will result from that earlier failure. We intend to design run-time systems that will allow programmers to choose from these options.

5. Language Levels

Grace will include "language levels" similar to those found in DrRacket [31] (formerly DrScheme). DrRacket includes five different language levels designed for teaching. Moving up from the basic level, successive languages introduce list constructs, local bindings and higher-order functions, anonymous functions, and mutable state.

Findler and colleagues have argued persuasively for the value of language levels [15]. Starting teaching in a restricted language allows the compiler or IDE to help the student by catching mistakes that might otherwise be interpreted as esoteric advanced features. It also allows error reports and suggestions for corrections to be more specific and helpful. DrRacket supports its language levels using a module system and by using macros to translate higher-level features into core-Racket. We intend to use library modules to support language levels in Grace.

Each Grace module will specify the level in which it is written with a **using** clause, the target of which is a library that defines a language level.

```
using BasicGrace
def o = object {
      method m(...) {... basicMethod ...}
}
```

In the above, basicMethod represents a method imported from object BasicGrace. The **using** construct imports all *public* names in the specified object. Because basicMethod is a public method of BasicGrace, it may be accessed via a method request to **self** just as if it were defined at the top-level of this example. Because **self** as the target of a method request can be left implicit, we can write basicMethod rather than **self**.basicMethod. This is especially useful for methods that implement new control structures.

The keyword **using** invokes a simple variant of inheritance that imports only the public attributes of the "used" object. It is essentially the same mechanism that Dahl and Nygaard used to incorporate the simulation language Simula I into

the general-purpose language Simula 67, about which Dahl wrote:

> One way of using a class, which appeared important to us, was to collect concepts in the form of classes, procedures, *etc.*, under a single hat. The resulting construct could be understood as a kind of "application language" defined on top of the basic one. It would typically be used as a prefix to an in-line block making up the application program. [13]

Because Grace incorporates closures and multi-part method names, what in other languages would be built-in control structures, such as while ... do ..., are defined as methods. The **using** construct lets Grace provide different control structures in different language levels. What **using** does not do is restrict the use of basic language features, like **var** declarations. If we find such restrictions necessary, we plan to implement them using annotations that are understood by the compiler.

While we have not yet specified the language levels for Grace, we imagine a lattice of languages rather than a sequence, so "sublanguages" is a more accurate term than language levels. For example, one instructor might start teaching with simple immutable objects operating on numbers, while another might start with a library of graphical objects that are updated in response to user interaction.

One possible lattice of sub-languages is shown in Figure 1. There might even be higher languages not shown in the figure, such as a language that loosens Grace's restrictions on shadowing variable names, a restriction that is appropriate for introductory teaching, but which might prove inconvenient for large programs. We expect that several such hierarchies might be necessary to meet the needs of different instructors.

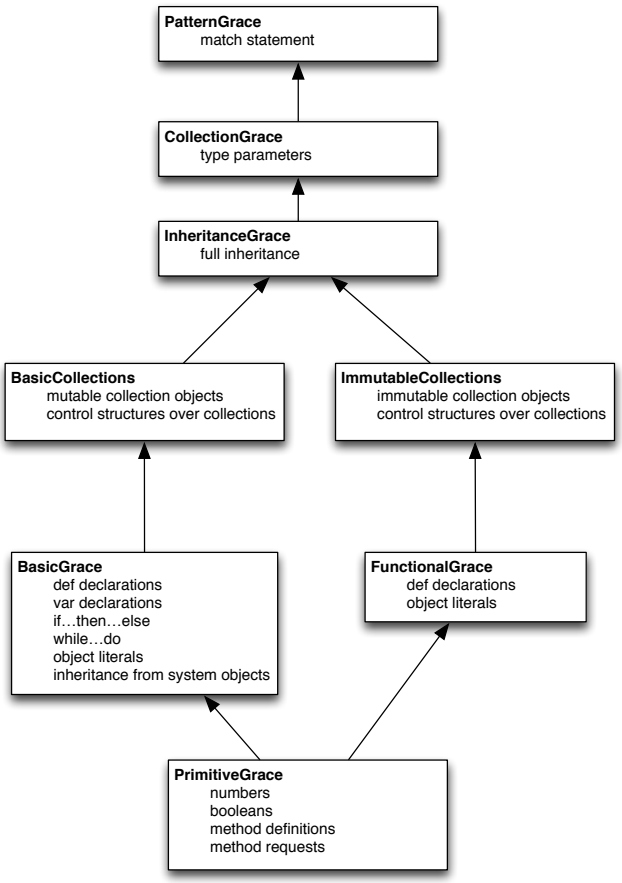

Figure 1. A possible lattice of sublanguages; the arrows indicate containment. Each sublanguage includes the features listed in its box and the union of all of the features of the sublanguages that it contains.

6. Implementation

Homer has written a prototype self-hosted compiler for Grace, known as Minigrace. Minigrace is able to generate both C, which can then be compiled to native code, and JavaScript, which can be run in a web browser, and recently has gained experimental support for generating Java. Minigrace currently accepts almost all of the specified parts of Grace, and includes a static structural type-checker. The same compiler front-end supports all the backends; the C backend is designed to work on any POSIX-compatible system that is equipped with *gcc*, such as Linux, NetBSD, and Mac OS X.

Minigrace comprises about 10,000 lines of Grace, distributed as shown in Table 1, along with a small runtime library in each of the target languages. The compiler compiles itself, and exercises most of the features of Grace.

To bootstrap the compiler, Homer built a simple prototype for a small subset of Grace which used the Parrot Compiler Toolkit and ran on on the Parrot virtual machine; the current compiler was compiled by that prototype until it was able to compile itself. The implementation process raised some language design issues, which in turn influenced the language specification.

The C backend was preceded by an LLVM backend, but debugging was overly difficult and the runtime library, in C, expanded such that it became clear that a higher level would be a better target. Both used the same runtime library, and implementation of the new backend progressed much faster than the original. C is also more portable, allowing source tarballs to be pregenerated and distributed. The JavaScript backend was originally a toy developed while other work had lapsed, but became fully functional. The C backend is the primary development target, supporting all of the features, while the JavaScript backend is updated subsequently to include new functionality subject to the limitations of the platform.

Efficiency is not a goal of Minigrace, beyond practicality of execution. Small- to mid-sized programs compile and run quickly. The native compiler incorporates garbage collection

Component	Lines
Compiler driver	100
Utility and interface functions	300
Lexer	700
Parser	1700
AST	600
Typechecker	1600
C backend	1300
JavaScript backend	700
Java backend	1000
LLVM backend	1500

Table 1. Distribution of Minigrace source lines

and (optionally) optimizes tail-calls. Error reporting is currently inadequate, and at times invalid code may misbehave rather than reporting an error. Nevertheless, Minigrace is able to compile itself, its test suite, and various sample programs in reasonable time and operates correctly on these programs.

Implementation of the language gave rise to some challenges. One difficulty was Grace's semantics for non-local returns — that a return statement inside a block returns from the method in which the block is defined. Because of this, some, but not all, executions of `return` in a block must jump up the call stack. In C we are able to use `setjmp` and store the return point in the block object, with special handling for method dispatch on blocks. In JavaScript we did not have this option, and instead used the only form of non-local control-flow available: throwing an exception. This greatly complicates the generated code: every dynamic execution of a method body must have a unique identifier to distinguish its own returns from others, which we implement as a counter, and the method must be prepared to catch a *return* exception and either perform a local return or rethrow the exception.

The implementation process also gave us some insights into the language design. For example, at first we used the type `None` to indicate a command with no return value. Because `None` is uninhabited, this made the result of such commands unassignable, thus enabling the implementation to alert students to a common error. However, this typing made it impossible to write code that was polymorphic over whether or not an object was returned. For example, it was impossible to apply a generic block and store the result. In response, we changed the language specification so that commands return `Nothing`. On another occasion we found that the method request syntax was ambiguous, and so we changed it so that it has a single clear interpretation.

Minigrace can be compiled into JavaScript, and run in a web page (see Figure 2). A simple client-side web implementation is available [18], and is a reasonable way of trying out small programs, but not suitable for wider use: it provides no way of saving your code, and performance, correctness, and error reporting are very limited compared to the stand-alone compiler. The web interface also provides access to the test suite, which demonstrates the implemented features, and to various modes showing the parsing and subtyping determined for the code.

Tarballs of the compiler source can be downloaded from `http://ecs.vuw.ac.nz/~mwh/minigrace/dist/` and built using `./configure ; make`. A Grace program may be executed like a script with `minigrace --run` *program*`.grace`. Other modes, options, and limitations are described in the documentation.

7. Discussion and Related Work

Arguably harking back at least to Algol-60, educational programming languages form a long and illustrious lineage. Certainly Pascal has left its marks on Grace: we've consciously reverted back to Pascal's rational and explicable type syntax, rather than C's "Clockwise Spiral Rule" [1]; Grace's variable and constant field declarations are basically the same as in Modula-2 — except that we replaced "const" with "def", because we wanted constant declarations to be no longer than variable declarations. Scala's design choices are also a clear influence on Grace's syntax [28].

In terms of object-oriented teaching languages — or at least languages explicitly designed for teaching — Eiffel [23, 24] is the most well known, and a customized first-year course tailored to Eiffel has been designed recently [25]. Compared with Eiffel, and other efforts such as Object-Oriented Turing [17] and Blue [20, 21], we have intentionally designed Grace to look like a "curly bracket language" (as Ward Cunningham once put it). This is not simply a matter of choice. Rather, we recognize that students will need to transition to other languages — particularly Java and C++, but also Python, and Ruby, and perhaps Scala, Ceylon, and Dart. Grace's operator syntax, including both prefix operators and square brackets as an operator to access collections, has been designed so that Grace expressions look like Java or C expressions. Unfortunately, everything has a cost, and much of the complexity of Grace's syntax comes from this stylistic compatibility.

In terms of recent educational languages, Racket has also strongly influenced Grace [14]. The idea of a language tailored to education that is supported by a programming environment, where both the language and the environment support a series of language levels, comes directly from Racket. Racket is also gradually typed. Nevertheless, although our aims are similar, many of Grace's design decisions are very different from Racket's.

As we've described above, Grace's syntax is intentionally close to Java and C, while Racket is based on Scheme and adopts S-expression syntax. Racket's language levels are implemented using its own powerful macro facilities [31], whereas Grace relies on careful syntax design to admit as much C-like syntax as possible, on a language substrate closer to Self [32] or Newspeak [8]. On the one hand, this means that some features of Grace — notably the "class"

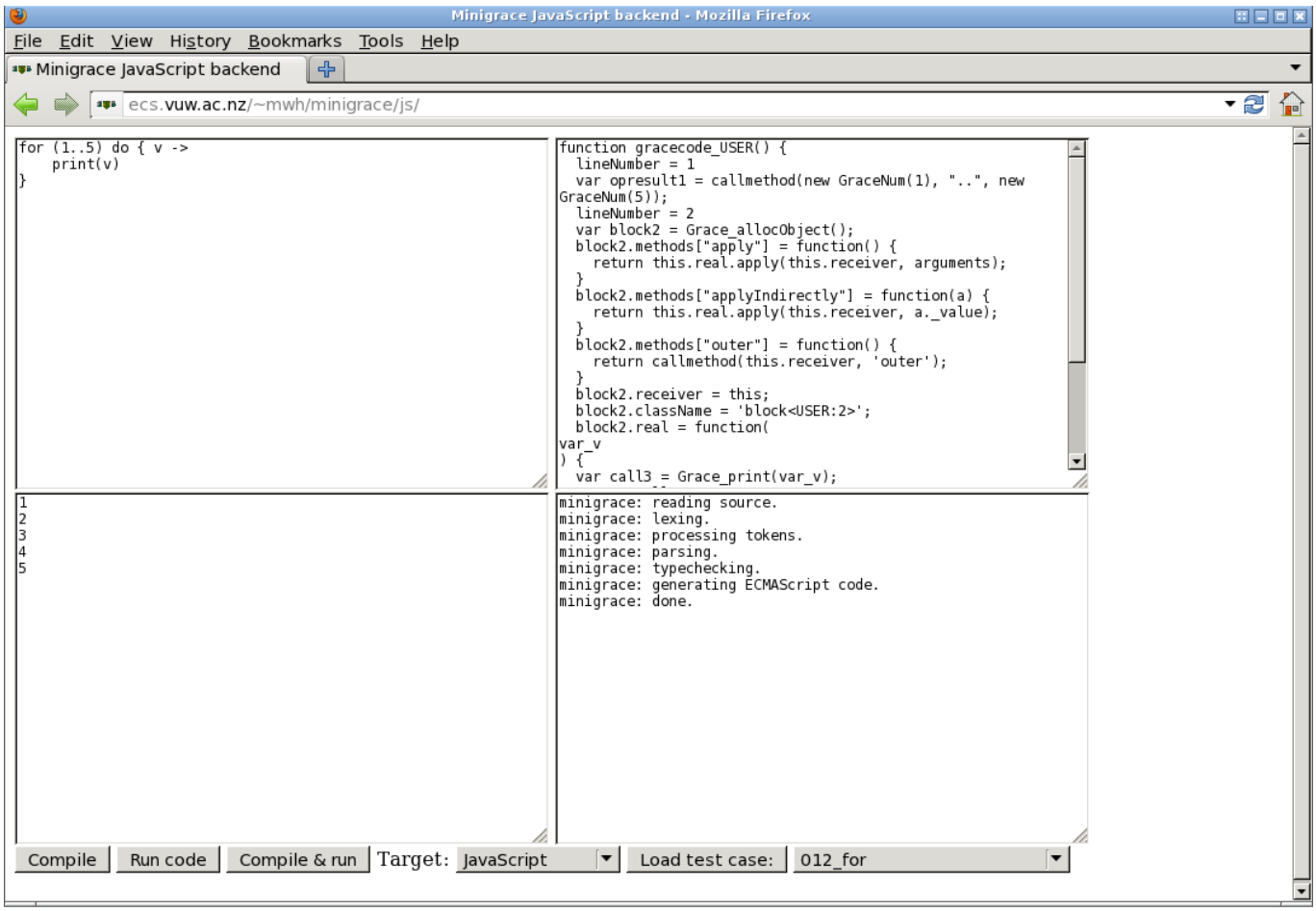

Figure 2. The Minigrace JavaScript frontend running in a web browser

syntax — cannot be implemented as extensions in Grace itself, but must be special-cased in the compiler. On the other hand, Grace's multi-part method names and operator syntax supports significant flexibility and extensibility, including the ability to define basic control-flow constructs and providing support for parser combinators in the base language. Thus, we have not felt a compelling need to introduce macros as a language construct. In this context, we note that Scala is also introducing support for macros [9].

Grace shares features of both class-based languages and prototype-based languages [26]. On the one hand, like class-based languages, Grace has an explicit class construct; moreover, Grace's object literals behave much more like classes than prototypes. In Self or Scala, each "evaluation" of an object literal returns the same object, whereas in Grace, as in O'Caml, and Emerald, each evaluation returns a distinct object (assuming the objects are designed and initialized so they they are distinguishable) [22, 32]. On the other hand, Grace objects can be created without writing any class declarations, and each Grace object conceptually stands alone without depending on a class. This is much closer to the prototype-based style.

Finally, as far as we can tell, Grace seems to be the first purely object-oriented, structurally typed language that has been designed since the late 1980s (since Trellis/OWL in fact [29]). O'CAML and Modula-3 (along with Java and its vast legacy) are what used to be called"hybrid" object-oriented languages, and while Eiffel is pure, it is nominally (and covariantly) typed with a subtle multiple-inheritance scheme [10, 22, 24]. In some respects, Grace feels to us like the "road not taken": connected at least as much to Pascal, Emerald, Self, or Eiffel, as it is to Java, C♯, or Scala. In many ways, much of Grace could well have been designed back in 1990 — or perhaps in 1995 for gradual typing. That it has taken until now says more about the many factors that influence programming language choice than says about any technical features of the language design.

8. Conclusion

The main aspects of Grace's design have been pounded out between three academic researchers with very different views on both language design and teaching programming, so few decisions were uncontroversial. We have been inspired by

Barry Commoner's Five Laws of Ecology, the first four of which appeared in his book "The Closing Circle" [11]:

> Everything is connected to everything else.
> Everything must go somewhere.
> Nature knows best.
> There is no such thing as a free lunch.
> If you don't put something in the ecology, it's not there.

The fifth law, which was added later, seems most obviously appropriate to language design: "If you don't put something in the language, it's not there". As with pollutants, once a feature is in the language, it's close to impossible to get rid of it, so we have tried to reduce the features of Grace to the bare minimum, knowing that it will be much easier to add a feature later than it will be to remove one.

One example of this is pattern matching, a powerful facility from ML and Haskell that has been popularized by hybrid languages like Scala. Some would argue that pattern matching is fundamentally at odds with object-oriented programming, since it interrogates the structure of an object rather than requesting that the object perform some method.

However, "Everything is connected to everything else". Once we admit types to the language, and decide that types do not automatically include a null value, we are led to include variant types, as described in Section 4. Pattern matching is certainly not the only way of handling variants, but it is a way that at least some instructors will wish to teach, especially as it is included in the strawman ACM curriculum [12]. Even those who feel that a pure-object-oriented approach is superior might want to compare and contrast the two approaches. Moreover, most of pattern matching can be defined in library objects, so, strictly, it doesn't rank as a language feature at all. However, to get the full benefit of pattern matching, patterns need to bind variables, and that cannot be done in a library. It is still not clear to us how to resolve these conflicting forces.

It has also become apparent that "There is no such thing as a free lunch". Every feature that we include in Grace adds a cost, in terms of complexity of the language description and the teaching sequence. There is also an implementation cost, although we have not let that become a major concern. The design of Grace's sublanguages is incomplete. While it is certainly true that "everything must go somewhere", instructors will have different views on where that should be, since they will wish to introduce features in different sequences, or perhaps not at all.

We are still searching for an analog for Commoner's "Nature Knows Best". We are certainly not under the illusion that the self-appointed Grace design committee knows best. For this reason we invite comments, criticism and participation. Go to http://www.gracelang.org to find out how to get involved. Perhaps the analogous law of language design will turn out to be "Experience is the best teacher". Whatever we choose as good ingredients for a teaching language, only experience using Grace to teach programming will tell us whether we have chosen wisely.

References

[1] D. Anderson. The Clockwise/Spiral Rule. Posted to `comp.lang.c`, May 1994. `http://c-faq.com/decl/spiral.anderson.html`.

[2] H. G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), Oct. 1993.

[3] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C♯. In *OOPSLA*, 2007.

[4] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the First ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, volume 21, pages 78–86, Portland, Oregon, October 1986. ACM. Published as SIGPLAN Notices **21**(11), November 1986.

[5] A. P. Black, K. B. Bruce, and J. Noble. Panel: designing the next educational programming language. In *SPLASH/OOPSLA Companion*, pages 201–204. ACM, 2010.

[6] A. Borning. Classes versus prototypes in object-oriented languages. In *FJCC*, pages 36–40. IEEE Computer Society, 1986.

[7] G. Bracha. On the interaction of method lookup and scope with inheritance and nesting. In *3rd ECOOP Workshop on Dynamic Languages and Applications (DYLA)*, 2007.

[8] G. Bracha. Newspeak programming language draft specification version 0.0. Technical report, Ministry of Truth, 2009.

[9] E. Burmako, M. Odersky, C. Vogt, S. Zeiger, and A. Moors. Scala macros. `scalamacros.org`, Apr. 2012.

[10] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 reference manual. Technical Report Research Report 53, DEC Systems Research Center (SRC), 1995.

[11] B. Commoner. *The closing circle: nature, man, and technology*. Knopf, New York, NY, 1971.

[12] CS2013 Steering Committee. Computer Science Curricula 2013 (Strawman Draft), Feb. 2012.

[13] O.-J. Dahl. The roots of object-oriented programming: the Simula language. In M. Broy and E. Denert, editors, *Software Pioneers: Contributions to Software Engineering*, pages 79–90. Springer-Verlag, Berlin, Heidelberg, 2002.

[14] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How To Design Programs*. MIT Press, 2001.

[15] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

[16] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation*. Addison Wesley, 1983.

[17] R. Holt and T. West. OBJECT ORIENTED TURING REFERENCE MANUAL seventh edition version 1.0. Technical report, Holt Software Associates Inc., 1999.

[18] M. Homer. Minigrace to JavaScript compiler. http://ecs.vuw.ac.nz/~mwh/minigrace/js/, 2011. Most recently accessed April 2012.

[19] A. Igarashi and H. Nagira. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1435–1441, New York, NY, USA, 2006. ACM.

[20] M. Kölling, B. Koch, and J. Rosenberg. Requirements for a first year object-oriented teaching language. In *ACM Conference on Computer Science Education (SIGCSE)*, 1995.

[21] M. Kölling and J. Rosenberg. Blue — a language for teaching object-oriented programming. In *ACM Conference on Computer Science Education (SIGCSE)*, 1996.

[22] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 3.12 documentation and user's manual, July 2011.

[23] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[24] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[25] B. Meyer. *Touch of Class: Learning to Program Well with Object and Contracts*. Springer-Verlag, 2009.

[26] J. Noble, A. Taivalsaari, and I. Moore, editors. *Prototype-Based Programming: Concepts, Languages, Applications*. Springer-Verlag, 1997.

[27] K. Nygaard and O.-J. Dahl. The development of the SIMULA languages. In R. L. Wexelblat, editor, *History of programming languages I*, chapter IX, pages 439–480. ACM, New York, NY, USA, 1981.

[28] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, 2005.

[29] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/OWL. In *OOPSLA*, 1986.

[30] A. Taivalsaari. Delegation versus concatenation, or cloning is inheritance too. *SIGPLAN OOPS Mess.*, 6:20–49, July 1995.

[31] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *PLDI*, 2011.

[32] D. Ungar and R. B. Smith. SELF: the Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.