2013

Modules and Dialects as Objects in Grace

Michael Homer
Victoria University of Wellington

James Noble
Victoria University of Wellington

Kim B. Bruce
Pomona College

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

Modules and Dialects as Objects in Grace

Michael Homer[1], James Noble[1],
Kim B. Bruce[2], and Andrew P. Black[3]

[1] Victoria University of Wellington, New Zealand,
[2] Pomona College, CA, USA
[3] Portland State University, OR, USA

Abstract. Grace is a gradually typed, object-oriented language for use in education; consonant with that use, we have tried to keep Grace as simple and straightforward as possible. Grace needs a module system for several reasons: to teach students about modular program design, to organise large programs, especially its self-hosted implementation, to provide access to resources defined in other languages, and to support different "dialects" — language subsets, or domain specific languages, for particular parts of the curriculum. Grace already has several organising constructs; this paper describes how Grace uses two of them, objects and lexical scope, to provide modules and dialects.

1 Introduction

In object-oriented languages, objects and the classes that generate them are the primary unit of reuse. But objects and classes are typically too small a unit for software maintenance and distribution. Many languages therefore include some kind of package or module construct, which provides a namespace for the components that it contains, and a unit from which independently-written software components can obtain the components they wish to use.

1.1 The Grace Programming Language

We are engaged in the design of Grace, a new object-oriented programming language aimed at instructors and students in introductory programming courses [7]. To keep Grace small and easy to learn, we have relied on three principles:

1. omit from the Grace language itself anything that can be defined in a library;
2. design Grace around a small number of powerful mechanisms, each of which can be used to provide the effect of what might otherwise be several special-purpose features; and
3. structure Grace out of a number of language subsets, so that students can program in a language that closely matches their level of competence at any given time; these language subsets are called *dialects*.

Principle 1 implies the need for some kind of module facility, so the libraries can be defined in Grace and so that student programs can use those libraries. Principles 2 and 3, when taken together, strongly suggest that the same module facility be used to define

Grace's dialects. Principle 2 prompted us to try to build a module facility out of the more primitive concepts already included in Grace. We believe that we have succeeded, and present here as evidence a description of Grace's modules and dialects, showing how they are built from more basic language features.

1.2 What is a Module?

As an educational language, Grace does not need as elaborate a module system as might be required in an industrial-strength language. Grace *does* need a module system adequate to support the development of its own tools, which already include a self-hosting compiler, and will, we hope, eventually include a programming environment. Even more importantly, we would like to use the module system, or a variant of it, to implement the sub-languages mentioned in Principle 3 above.

The specific requirements for Grace's module system are as follows:

R1. Separate compilation: each module can be compiled separately.
R2. Foreign implementation: it should be possible to view packages implemented in other languages through the façade of a Grace module; the client code should not need to know that the implementation is foreign.
R3. Namespaces: each module should create its own namespace, so maintainers of a module need not be concerned with name clashes.
R4. Sharing: objects provided by a module should be sharable by client code.
R5. Type-independent: because Grace is gradually typed, the module system cannot depend on the type system, but the module system should support programmers who wish to use types.
R6. Controlled export: some mechanism should be available to hide the internal details of a module's implementation.
R7. Multiple implementations: it should be possible to replace one module by another that provides a similar interface, while making minimal changes to the client.
R8. Explicit dependencies: code that uses a module *depends* on that module: dependencies should be explicit.

Grace meets these requirements by representing modules as objects. Our design has been influenced by the module systems of Python and Newspeak, and takes advantage of the power of Grace's object model. Grace modules can be used to define dialects, which not only supply the sub-languages required for teaching purposes, but also make it possible to define a variety of domain-specific languages.

1.3 Contributions

The contributions of this paper are:

- the design of a module system in which modules are objects (Section 3);
- the design of a system for sub-languages and dialects obtained by combining the module system with lexical scope (Section 4);
- the rationales that led us to these designs (Sections 3.5 and 4.4);
- examples of dialects that can be defined in and for Grace (Section 4.3); and
- a description of the novel features of our implementation (Section 5).

To help the reader understand our design, the next section summarises the features of Grace's object system that relate to modules.

2 Objects in Grace

Grace is an imperative object-oriented language with block structure, single dispatch, and many familiar features [7]. Grace aims to be suitable for teaching introductory programming courses, to look familiar to instructors who know other object-oriented languages, and to give instructors and text-book authors the freedom to choose their own teaching sequence.

A single object is created by executing a particular kind of Grace expression called an object constructor:

```
object {
    def name is public, readable = "Joe the Box"
    var subBoxes := List.empty
    method topLeft { 100@200 }            // @ is an infix operator
    method bottomRight { 200@400 }        // that creates a Point
    method width { bottomRight.x − topLeft.x }
    method height { bottomRight.y − topLeft.y }
    method addComponent(w) { subBoxes.push(w) }
    print "Joe the Box lives!"
}
```

The object created by executing this expression has methods topLeft, bottomRight, width, height, and addComponent. It also has a method name that acts as an accessor for the constant field name; this method is generated automatically because of the annotation **is** public, readable. The variable field subBoxes could have been annotated as readable and writable, but since it was not, it is private to the object. Creating this object has the side effect of executing the code inside the object constructor, which in this case prints `"Joe the Box lives!"`.

Grace's class construct abbreviates the definition of an object with a single method that contains an object constructor, and thus plays the role of a factory method:

```
class aBox.named(n:String)origin(tl:Point)diagonal(d:Point) {
    def name is public, readable = n
    var subBoxes := List.empty
    method topLeft { tl }
    method bottomRight { tl + d }
    method width { bottomRight.x − topLeft.x }
    method height { bottomRight.y − topLeft.y }
    method addComponent(w) { subBoxes.push(w) }
    print "{name} the Box lives!"
}
var joeTheBox := aBox.named("Joe") origin(100@200) diagonal(100@200)
```

The class is called aBox, and its factory method is named()origin()diagonal(); Grace allows method names to have multiple parts — similar to Smalltalk, but without the colons. The declarations and statements between the braces (lines 2 to 9) describe the object created by this factory method; note how some of the methods capture state from the parameters of the factory method. Two other features of Grace are shown incidentally: method arguments that are already delimited, such as strings and numbers, need not be enclosed in parentheses, and strings can contain Grace code within braces.

Grace evaluates the code, requests the resulting object to convert itself to a string, and inserts that string in the place of the brace expression.

The last line illustrates the use of this class. Executing the expression on the right of the assignment will print `"Joe the Box lives"` and create a new object. The assignment will bind the variable joeTheBox to the new object.

Classes are completely separate from types: the class aBox is not a type and does not implicitly declare a type. The programmer may specify types if desired:

```
type Box {
    name −> String
    topLeft −> Point
    bottomRight −> Point
    width −> Number
    height −> Number
    addComponent(w:Widget) −> Done
}
var joeTheBox:Box := aBox.named("Joe") origin(100@200) diagonal(100@200)
```

The type Done indicates that a method does not return a useful result. Types are structural: an object has a type if it responds to all of the methods of the type, with the correct argument and result types. It is not necessary for the object to have been "branded" with that type when it was created. Along with methods, types may be included as components of objects.

Variable and constant 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, initialised using =, as shown here:

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

The keywords **var** and **def** are used to declare both local variables and fields inside objects. Visibility annotations allow the programmer to control access to methods from outside an object by marking them as public or confidential; the latter means accessible only to the object itself and objects that inherit from it. As we saw in the initial examples, annotations can also be used to create reader and writer methods on fields. Method requests without an explicit receiver are resolved either as requests on **self**, or as requests on **outer**, the object in the surrounding lexical scope, on **outer.outer**, *etc*. If a receiverless request is ambiguous, the programmer must resolve the ambiguity explicitly.

Grace includes first-class blocks (lambda expressions), which are written between braces and contain 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 variables bound in its surrounding lexical scope, and returns the value of the last-evaluated expression in its body.

Control structures in Grace are methods defined in libraries. The built-in structures are defined in the standard prelude, but an instructor or library designer may replace or add to them. Grace's syntax is designed so that control structures look familiar to users of other languages:

```
if (x > 0) then { x } else { -x }

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

The use of braces and parentheses is not arbitrary: parenthesised expressions will always be evaluated exactly once, whereas expressions in braces are blocks, and may be evaluated zero, one, or many times.

Grace code at the top-level (of a file, or of the read-eval-print loop) is treated as if it were enclosed in an implicit object constructor. This allows methods and types to be defined at the top level; moreover, any code written at the top level will be executed immediately. As a consequence, Grace programs can be written in "script" form, without any object or class definitions at all, so `print "Hello, world"` is a compete Grace program.

3 Modules as Objects

A Grace module is a piece of code that constructs an object. This *module object* behaves like any other object; in particular, it may have types and methods as attributes, and can have state. In the current implementation, a module is represented as a Grace source file; the module object is the one created by the implicit top-level object constructor. Executing the file creates an object with normal Grace object semantics. The effect of this is similar to modules in Python [4], but uses existing language features instead of adding a special construct. Modules as objects also provides a consistent story for interactive read-eval-print loops as well as scripts; the programmer defines an object progressively as they write. As we shall see, making modules objects interacts constructively with other aspects of the language, such as gradual typing.

3.1 Importing modules

To access another module, the programmer uses an import statement:

```
import "minigrace/parser" as parser
```

The string that follows the **import** keyword must be a literal; it identifies the module to be imported. From the perspective of the language this string is opaque; the current implementation treats it as a relative file path. The identifier following **as** is a local name that is bound to the module object created by executing that file.

Because of Grace's gradual typing, a variant of the import statement allows the programmer to specify the type that the imported module should meet:

```
import "minigrace/parser" as parser:BasicParser
```

Types in Grace specify interface, not implementation, so this asserts that the imported module object must satisfy the interface defined by the BasicParser type; if it does not, an error occurs (at compile-time or bind-time, depending on the implementation). The type could be imported from another module, or be defined by the client:

```
type StackType = {
    push(_:Object)
    pop −> Object
}

import "stackImpl" as StackImpl:StackType
```

An alternative implementation of the same type may be chosen by changing the import path string. When no type is specified in the import statement, the type of the module is inferred from its implementation. The Grace compiler saves enough information about a compiled module to avoid the need to re-parse it each time it is imported.

3.2 Gradual typing of modules

Through careful use of modules and imports, a system can be configured in a range of different ways:

- `import "x" as x` imports module `"x"` with its original types, whatever those may be. Parameters and return values with dynamic types in `"x"` will also be dynamic in the importing module, while uses of statically-typed parameters and return values will be statically checked in the importing module.
- `import "x" as x:Dynamic` means the the importing module will ignore any type information specified in the module `"x"`. This means that providing an argument to a method of x that does not match the declared type of a parameter, or requesting a method that is not defined, will not be reported as static errors, and will not prevent the importing code from being compiled and run. Grace's gradual typing means that such errors will be caught dynamically, if and when the offending code is executed. This is very useful when programming intentionally, coding test-first, and when writing client code to explore what the interface of `"x"` should be.
- We can define the interface that we want a module to support separately, that is, in another module. This allows for multiple implementations of the same interface; it can also be used to check that a module provides the features that we expect:

  ```
  import "xSpec" as xSpec
  import "xImpl" as x:xSpec.T
  ```

  Here, `"xSpec"` is a module defining the type T, like a Modula-2 definition module [32]. The type of the separate implementation module `"xImpl"` is required to conform to the type xSpec.T. A different implementation of xSpec.T can be selected by changing just the second import statement.
- A module that is intended to satisfy a type defined elsewhere can assert its own compliance with that type. Suppose that you are writing a module that you intend to satisfy the type T defined in `"xSpec"`. Then you could write:

  ```
  import "xSpec" as spec
  assertType<spec.T>(self)
  ```

  The library method assertType<T>(o) statically checks that o has type T, so this code asserts that the current module object has the type T imported from spec. If we are especially concerned with meeting a type specification, we can perform a similar check on our own module.

– We can perform "type ascription":

```
type ExpectedType = { ... }
import "x" as x:ExpectedType
```

Here we import a module, but are explicit about the type that our code assumes that module will satisfy — which will often be a supertype of the type of the object actually supplied by `"x"`. Future changes to the module `"x"` that invalidate that assumption will yield an error at the import site; this puts the "blame" in the right place. Type ascription also imposes the constraint `x:ExpectedType` on code that uses `x` in the importing module. So, if our importing code tries to use `x` in a way that conflicts with that constraint, it will receive a static error. This is true even if the implementation module `"x"` uses dynamic typing.

Every import of the same path within a program will access the *same* module object. Conceptually, importing a module returns a reference to a pre-existing object from the pool of module objects.

3.3 Recursive modules

A module `A` cannot import a module that directly or transitively imports `A`. This restriction is a deliberate choice. For our target audience — novice programmers — we believe that cyclic or recursive imports are most likely to indicate a program structuring problem. This choice also means that we can fully create and initialise all imported modules before the importing module. This avoids problems caused by partially-initialised modules, which novices are likely to have difficulty understanding.

While cyclic *imports* are prohibited, modules may recursively *use* one another, just like any other pair of objects. This can be accomplished by the programmer explicitly providing each of the modules with a reference to the other.

3.4 Extensions and future work

The module system design that we have presented is open to a number of extensions. In particular, it is possible to interpret the import string in various ways, without affecting the rest of the language. Moreover, because a module presents itself as an object, its internal implementation and behaviour are hidden. In this section we present some preliminary experiments and discuss future extensions that exploit these features.

Foreign objects. We can access code written in other languages, or behaving in unusual ways, by compiling it appropriately and then importing it in the ordinary way. Objects that have been imported from a source outside the universe of Grace code are called "foreign objects". From the perspective of client code, there is no difference between an import that returns a foreign object, and an import that returns an ordinary module object. Internally, a foreign object may construct new objects or classes "on the fly" to represent the resources it provides, and it may access other libraries available on the implementation platform.

We have written a fairly complete Grace binding to the GTK+ widget library [12] that demonstrates foreign objects. A Grace program can use this module as follows.

```
import "gtk" as gtk
def window = gtk.window(gtk.GTK_WINDOW_TOPLEVEL)
def button = gtk.button
button.label := "Hello, world!"
button.on "clicked" do { gtk.main_quit }
window.show_all
gtk.main
```

This code creates a window with a "Hello, world!" button that terminates the program, using a Grace transliteration of the underlying GTK+ interfaces. GTK+ is an object-oriented library, and its object features are mapped directly to Grace objects. Here, notwithstanding that the `"gtk"` module is not Grace, to the client code this is an ordinary module import. Because object implementations are always opaque the module object is indistinguishable from one defined in Grace code. The prototype compiler (see Section 5) understands how to find and load a module including these bindings, along with any metadata needed for compilation.

External Data. Because the source of an import is a string whose interpretation is left to the implementation, we can give certain strings special interpretations. One interpretation allows external data sources like web services, databases, and local metadata, to be reified as foreign objects. The overall effect is similar to F#'s "type providers" [25]. These foreign objects can be implemented either dynamically (as in the GTK bindings) or by code generation, as in F#. Our prototype implementation (described in section 5) supports both approaches by providing a general hook in the import system. For example, one prototype reifies the filesystem as a set of foreign objects. We can then import an external file as a Grace string object:

```
import "file://readme.txt" as helpText
```

This kind of foreign import is very useful, *e.g.*, for providing access to compile-time data such as help text, images, and sound files. The idea behind this feature was uncovered serendipitously as we developed the import facility for objects-as-modules. This shows the power of a simple mechanism used consistently.

Automatic Dependency Injection. Modules are normally imported by name, given as an explicit string; this has the benefit of making module dependencies explicit, Nevertheless, it is sometimes desirable to refer to modules *indirectly*, thus providing external control over which module is selected by a given import statement. For example, suppose that we want to choose a different implementation of a logging service for production and test code. This can be accomplished by writing

```
import "logger" as logger
```

in one module, while the programmer specifies how `"logger"` should be resolved elsewhere. This might be through a compile-time parameter or environment variable, or in the source of the main module of the program. This option may also be used to replace another module's dependency without its collaboration, for example when working with legacy code. The effect is similar to that of using a customised class loader, but without interfering with the semantics of the language itself.

3.5 Design Rationale

In Section 1.2, we laid out the requirements for modules in Grace. Of the concepts already in the language, objects satisfy many of these requirements. Top-level objects cannot capture variables, so they are separately compilable (requirement R1); they create a namespace accessible through "dot" notation (R3); the same object may be referred to by many other objects (R4); they are gradually typed (R5); they provide controlled export to clients (R6); and object use is explicit (R8). Through careful design of the import mechanism we were able to achieve the ability to use foreign implementations (R2), and to substitute one implementation for another (R7). Using objects as modules avoids introducing another concept into the language, supporting our design principle of building a small language.

We considered an alternative design in which modules were classes, as they are in Newspeak [8]. Using classes would offer some advantages: modules would be instantiable and could be parameterised over objects and types. In the style of Newspeak, we could have omitted the **import** construct in favour of providing an explicit platform parameter to the module containing its dependencies. We eventually rejected this approach because it requires boilerplate code to be repeated in every client, and would not have given a consistent interpretation across different forms of program. In our design, a module that will be imported by other code, a top-level "script" program consisting of statements to execute, and code destined for an interactive read-eval-print loop all have the same interpretation of gradually building an object. A class-based module system would not support this consistency. Another difficulty with using classes as modules is that multiple imports of the same module (say, by two different client modules) would obtain different instances. The actual dependencies between the client modules would therefore need to be constructed dynamically, so these dependencies would not be clear in the source. Explaining these nuances to novices would be undesirably complicated. With modules as objects and Grace's class semantics, a module can define a factory method, and therefore act as a class, if it so chooses. This allows for situations where class-like behaviour is desirable without requiring it at all times.

We also considered a variant of the current design in which a file containing a single top-level declaration of an object, type, or class was a module, while a file with multiple declarations was an object with the attributes thus declared. We rejected this option because of the extra complexity, lack of uniformity, and the need for special-case behaviour, all of which would need to be explained to students.

4 Dialects

Dialects are modules that provide supersets or subsets of the standard Grace language. Dialects can not only make extra definitions available to their users (remember that what would be statements or operators in other languages are just methods in Grace) but they can also restrict the language to detect new classes of error or to report errors differently. Dialects support both language subsets to aid novice programmers and domain-specific languages customised for a particular purpose.

Families of language subsets were introduced by DrScheme [27] (now DrRacket) under the name "language levels". Language subsets allow a programming course to in-

troduce new features gradually. Students program in a restricted language that includes only the features they have been taught so far, while instructors or library authors can use the unrestricted language. This means that the compiler or IDE can help the student by catching mistakes that might otherwise be interpreted as esoteric advanced features, and by wording error messages to match the students' knowledge. Being able to define such language subsets is important to Grace's goal of being a teaching language.

We use the term "language subsets" rather than "language levels" because we imagine a lattice of languages rather than a sequence. 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. Sublanguages would be defined for different useful combinations of features and instructors could choose their own path through them.

Dialects can also be used to implement domain-specific languages: languages where the vocabulary is taken from a particular application domain and that allow programming tasks within that domain to be more straightforward. Domain-specific dialects can add new control structures, maintain complex data structures implicitly, or report errors in a way appropriate to the task at hand. Potential dialects include:

- A set of dialects for students that define language subsets, as discussed above.
- A dialect that requires that all code be fully statically-typed. This would be useful to an instructor who wishes to begin with types and require that students always use them.
- A dialect to augment the built-in control structures to include assertions for reasoning about the program. For example, a dialect could permit or require the programmer to specify loop variants and invariants on every `for` and `while` loop. Some of these assertions could be checked statically while others would require dynamic checks; the dialect could support both versions.
- A dialect for writing test cases and test suites. Test suites often involve a significant amount of repeated boilerplate code, and have a distinguished class of errors associated with them. A unit-testing library could include a dialect to help users write and use tests and provide errors or warnings about common mistakes, or suggestions about missing cases.
- A domain-specific language for writing finite state machines, which are useful for solving many classes of problem. Implementations of finite state machines are usually custom-built and include a lot of boilerplate code, even when a library is used. A dialect built for the purpose can allow a domain expert to work on the design of the machine without needing a deep understanding of the implementation, or of programming in general.
- A dialect for writing documentation, in the style of literate programming. Good documentation is more than comments: it refers to features of the code, such as types and parameters, in specific ways. It takes part in refactorings and tests, and may be available at run-time in an interactive mode. A dialect could enforce good practices on the documentation, and would make it possible to build reference documentation automatically during compilation.

– A dialect for writing other dialects. Instructors may wish to construct specialised dialects for their students. A dialect can support them in doing so by providing abstractions of common functionality, such as prohibiting certain identifiers or constructs or overriding some error messages with others tailored to their course.

We present examples of some of these dialects in Section 4.3.

A module declares the dialect it is written in using a dialect declaration:

```
dialect "beginner"
```

This declaration loads the module named by the string just as if it were an import statement (see Section 3). Unlike an import statement, however, the dialect object is not bound to a name. Rather the dialect module is installed as the outermost lexically-surrounding scope. Any request for a method in that outer scope — most often a receiverless request — will access a method of the dialect object. This resolution rule is the same one used for any other receiverless request in a lexically nested scope, such as from inside an object constructor out to a surrounding object.

When no dialect is specified, the module is written in the full Grace language with the standard prelude as its dialect. An IDE for beginning students might dictate a particular dialect either in the source text or through compiler flags.

Dialects are defined in the same way as modules; it is possible for the same module to be usable both as a dialect and by ordinary import. All of the properties of modules that we have previously discussed also apply to dialects.

A dialect may itself be written in a dialect. This exposes a difference between dialectical nesting and other kinds of lexical nesting: dialectical nesting is not transitive. Method name resolution will search only as far as the first dialect, and not proceed into the dialect's dialect or beyond. This is because special-purpose dialects will commonly be less powerful than the language as a whole, and may be written in a dialect — or in the unrestricted language — that provides the dialect writer with useful features that should not be exposed to clients.

Dialects do not depend any kind of macro expansion: lexical nesting, combined with Grace's flexible method naming conventions, allow a variety of very different dialects to be defined using only standard language features. A Grace dialect is strictly less powerful than dialects in Racket [27] or other languages with strong macro functionality, but is able to satisfy our requirements for dialects, as outlined in Section 1. There are also advantages to restricting the expressive power of dialects: because dialects can't introduce new syntactic forms, code written in a dialect remains readable without knowledge of the dialect it is using. The parse of a Grace program does not depend on dialects, types, or operator definitions: syntactically, there are only method requests.

4.1 Checkers

As well as providing definitions, dialects may outlaw or control particular features of the language, or offer additional and more accurate error and warning messages. New students will need different error messages than more advanced programmers, as the scope of things they are trying to do is much more restricted.

These checks are implemented by the dialect module defining a checker method. This method is executed by the compiler when modules written in the dialect are compiled. This checker is passed the abstract syntax tree of the module and may do with it as it pleases, including indicating to the compiler whether it should proceed, or terminate with an error.

4.2 Auxiliary definitions

Although making dialect nesting intransitive ensures that a dialect does not expose external definitions from its own dialect to its clients, the implementation of a dialect may itself need auxiliary definitions, and it should be possible to hide these too. For example, a dialect might define some helper methods that abstract common functionality from multiple dialect methods, or it might declare some state variables required by the dialect; clients should not see these attributes of the dialect.

In ordinary modules such helper methods can be marked as confidential and not be exposed to clients of the module, and definitions of variables are always private to the module. However, because dialects use lexical nesting, making methods confidential does not stop client code — which is logically nested inside the scope of the dialect — from accessing them; local definitions of the dialect would similarly be exposed.

Here the dialect system can come to its own rescue: a dialect can define a checker method (see Section 4.1) that disallows access to certain features that the dialect would otherwise expose. Any code that would have access to these features must be in the scope of the dialect, and so be subject to the dialect's checker. A dialect for writing dialects could provide a generic checker method, and an accompanying local annotation for code written in it. This would make the prohibition of access to local attributes of a dialect behave like a language feature.

4.3 Examples of Dialects

In this section we present a number of example dialects. For reasons of space we include a complete implementation only of one. Sample code for all examples is included in the downloadable implementation (see Section 5). We sketch a simple "loop invariants" dialect, give the implementation of a dialect that requires static typing, show the design of a dialect for writing other dialects, and briefly touch on two other domain-specific languages.

Loop invariants. A loop invariant is a property that should be true for every iteration of the loop. Inspired by Eiffel [20], we might want programmers to specify these invariants inline on their loops:

```
while {x > 0} invariant { y > x } do {
  x := x − 1
  ...
}
```

A dialect could add support for specifying and checking invariants in code using the dialect — here's a very simple version of a "for" loop with a checked invariant.

```
// Loop invariant dialect
method for(it : Iterable)invariant(inv : Block<Boolean>)do(blk : Block) {
    for (it) do {i->
        if (! inv.apply) then {
            InvariantFailure.raise "Loop invariant not satisfied."
        }
        blk.apply(i)
    }
    if (! inv.apply) then {
        InvariantFailure.raise "Loop invariant not satisfied."
    }
}

method while(b)invariant(inv)do(blk) {
  ...
}
```

This example is simplified to show the general idea; a more realistic implementation would provide more arguments to the invariants (such as a history variable), and also support loop variants and pre- and post-conditions.

Although the dialect is able to use the primitive looping constructs of the language, it does not make them available to its users. A program written in the dialect will need to specify the loop invariant on every loop. In this example the dialect performs dynamic checks on the invariant; it could also try to verify the invariant statically using a checker.

Requiring Type Annotations. An instructor may require that for all or part of a course, all student code is fully annotated with types, and no dynamically-typed code is permitted. A StaticGrace dialect can allow access to all of the ordinary language features, while reporting compile-time errors to students who omit the types on their declarations. We assume here that the normal type-checking mechanisms will take care of situations in which a student's type annotations are wrong.

```
// Include all of the standard language features this time:
inherits BasicGrace.new
import "minigrace/ast" as ast
def staticVisitor = object {
  inherits ast.baseVisitor
  method visitDefDec(v) is public {
    if (v.decType == DynamicType) then {
      CheckerFailure.raiseWith("no type given to declaration"
          ++ " of def '{v.name.value}'", v.name)
    }
  }
  method visitVarDec(v) is public {
    if (v.decType == DynamicType) then {
      CheckerFailure.raiseWith("no type given to declaration"
          ++ " of var '{v.name.value}'", v.name)
    }
  }
  method visitMethod(v) is public {
```

```
    for (v.signature) do {s->
      for (s.params) do {p->
        if (p.decType == DynamicType) then {
          CheckerFailure.raiseWith("no type given to declaration"
            ++ " of parameter '{p.value}'", p)
        }
      }
    }
    if (v.returnType == DynamicType) then {
      CheckerFailure.raiseWith("no return type given to declaration"
        ++ " of method '{v.value.value}'", v.value)
    }
  }
  method visitBlock(v) is public {
    for (s.params) do {p->
      if (p.decType == DynamicType) then {
        CheckerFailure.raiseWith("no type given to declaration"
          ++ " of parameter '{p.value}'", p)
      }
    }
  }
}
method checker(code : List<ASTNode>) {
    for (code) do {n : ASTNode ->
        n.accept(staticVisitor)
    }
}
```

This implementation uses the Visitor Pattern [11], but it could also be implemented using Grace's pattern-matching system [13]. The checker is presented with the raw AST and is able to perform any computation it wants on it; here every variable, parameter, and method declaration is checked to ensure that it has a declared type.

Dialect for writing dialects. So far we have have focused on demonstrating the structure of the dialect system, but even from these few examples, it should be clear that programmers writing different dialects will have similar needs. These needs can be met by a dialect for writing other dialects, thus simplifying the process of writing a dialect.

It will be common for dialects with checkers to perform a test on a certain subset of the AST nodes. Rather than requiring the dialect author to build a Visitor, as we did in our StaticGrace example, a Dialect dialect could provide a declarative syntax, such as

```
eachVarDec { n : VarDecNode ->
    if (n.decType == DynamicType) then {
        fail "var declaration must have a static type"
    }
}
```

or even

```
fail "var declaration must have a static type"
    when { n:VarDec -> n.decType == DynamicType }
```

The block argument after the "when" keyword uses Grace's pattern-matching [13] to simplify the expression of the kind of node to which the check should apply. VarDec is a pattern that matches any variable declaration node; the fail()when() method above will produce a failure message only when the node is a VarDec and the equality holds. Multiple kinds of AST node can be checked at once:

```
fail "var and def declarations must have a type declared"
  when { n : VarDec | DefDec −> n.decType == DynamicType }
```

A declarative approach allows checkers to be expressed concisely, and to be understood without needing a deep understanding of the system.

Another common task will be to ban the use of certain identifiers by code written in the dialect, to encapsulate the dialect's own implementation. The Dialect dialect can provide a way of declaring this:

```
prohibit "myHelperMethod"
```

Even simpler, it can provide a "local" annotation that performs the check for these names automatically:

```
method myHelperMethod is local {
  ...
}
```

The dialect-writing dialect can collect and compose the various tasks to be performed on the tree so that they run more efficiently, and without the author of a dialect needing to consider any of the complexities. A library of pre-defined checkers can be used largely declaratively.

Finite state machines. We can define a dialect for expressing finite state machines, and for processing and computation on these machines. The dialect allows the machine to be described declaratively, and results in a module object that encapsulates the machine:

```
dialect "fsm"
def startState = state { print "Starting" }
def runState = state { print "Running" }
def endState = state { print "Done" }

in(startState) on("A") goto(runState)
in(runState)
  on("A") goto(runState)
  on("B") goto(endState)

method process(symbol : String) {
  // Forward handling of state transitions to the dialect
  transition(symbol)
}
```

This module performs the computation of a finite state machine and could be used by client code without either the designer of the machine or the author of the client needing to understand the other's role. All of the complexity of setting up the machine has been abstracted into the dialect; the goal is for the description of the machine to be intelligible to domain experts (or engineering students!) who do not know Grace. The

user's process method delegates the responsibility for handling state transitions to the dialect-provided transition method, which takes an input symbol (here `"A"` or `"B"`) and performs the appropriate state transition.

Relations. Relations, or associations, are representations of the semantic connections between entities in the program's model. UML supports the concept of relations, but Grace does not support them primitively, so relations must usually be put into a more concrete form in a Grace implementation. This has the disadvantage of losing some of the information in the model. A dialect could provide the "objects as associations" [21] design.

```
dialect "object-associations"
def Attends = Relationship<Student, Course>
def Teaches = Relationship<Course, Faculty>
def Prerequisites = ReflexiveRelationship<Course>
// Set up or obtain our data objects
def james = student(...)
...
Attends.add(james, cs102)
...
for (Attends.to(cs102)) do { each -> ... }
```

The dialect allows the programmer to manipulate relationships directly in a natural way, and without obscuring the purpose of the code. The dialect user can write code that is similar to what they might write in a relationship-based language.

4.4 Alternative Designs for the Dialect System

We considered three major alternative approaches to dialects: inheritance, macro expansion, and creating an additional kind of definition with special properties. We rejected all of these in favour of the approach described here, each for a different reason.

Inheritance. With an inheritance-based approach, the module using a dialect inherits from the dialect, and dialectical methods can be invoked using a receiverless request, since they would be available on **self** in the module scope, and on **outer** in any nested scopes. The dialect's methods could also be defined as confidential if required, and hence available only on **self**.

This approach was inspired by SIMULA, and envisaged in our early writing on Grace. However, as the language developed, and we began to consider the implementation, several problems revealed themselves. Any state in the dialect would become part of every module written in that dialect, and would not be shared. Because inheritance is transitive, dialects would be forced to be transitive. This would mean, for example, that the Dialect dialect would always be included in any module that used a dialect written in it. Inheriting from a dialect would make all of the dialect's methods available on the module object itself, thus enlarging that object. For these reasons we discounted the inheritance approach.

Macro expansion. The idea behind the macro-expansion approach to dialects is to translate a dialect statement into an import statement for the dialect module itself, along with a set of local definitions of methods, one for each of the public methods of the dialect. Each of these local methods would forward to the corresponding method of the dialect. In this way encapsulation of the dialect module is preserved. For example, given a dialect module containing:

```
method for(i)do(b) is public { ... }
method helper is confidential { ... }
```

and a module using it, the **dialect** keyword would be translated into:

```
import "someDialect" as secret
method for(a1)do(a2) is confidential {
    secret.for(a1) do(a2)
}
```

Dialect methods that are not public would not get local forwarding methods, so local definitions of the dialect would be hidden. The local forwarding methods would be marked confidential, so that they would not be available to clients of the module.

This approach would again make the dialect methods available as requests on **self** in the module scope. Many of the issues with the inheritance approach do not arise here. The dialect object is used compositionally, but new methods are defined in the client module. The concept of exposing only public methods seemed attractive, but did not allow for a method to be exposed to a client written in the dialect without also exposing that method to everyone. We were also hesitant to include a reflective macro-expansion system in the language, or to include the behaviour as a special case.

Explicit dialect support. The third option was to add an additional language mechanism, specific to dialects, enabling the behaviours we want from an idealised dialect system. We were reluctant to add another mechanism and complicate the language unless absolutely necessary; very particular specialisations of dispatch and the treatment of identifiers would be required to make this approach work.

When we found that lexical scoping gave us the required behaviour, combined only with the checkers framework (which would in any case be required for the negative components of the dialect), we discarded the separate-mechanism approach. The only special rule that we needed was the rule that lexical search stopped at the first dialect. This is different from ordinary nesting, but is a reasonable generalisation of the desired semantics of the standard prelude.

4.5 Future work

In this section we describe possible future extensions of the dialect system. These have not been included in the design or implementation at this time.

Lexical dialects. The current design permits dialects on the level of a whole module or file. Because dialects are designed to use lexical scoping, an obvious extension is to permit dialects to be applied to smaller lexical scopes, perhaps a block or object

constructor. In that case the surrounding dialect would be replaced for that scope, and only the new dialect's static checker would be run.

The interaction of ordinary lexical scoping rules with a lexical dialect needs careful thought. In many cases, code in the new dialect may well want to access identifiers from elsewhere in the module, but not from the outer dialect, while in other cases the desire may be to augment the existing dialect on a temporary basis. A hard rule in either direction will be wrong for many situations where local dialects may be used, but permitting an algebra of dialects will greatly complicate the system. Our current implementation allows only module-scoped dialects, but we hope to experiment with alternatives in future.

Mutating dialects. Dialects are able to check, but not modify, the AST of the module they apply to, because mutating the AST can lead to complicated dialect code that is not syntactically meaningful without the dialect. At times it may be worthwhile to allow limited modifications of the user's code, for example, to implement new syntactic sugar, or to make use of information that is discarded before run-time.

Such a "mutating dialect" could implement the single transformation to blocks required to support Grace's pattern-matching facilities, which was instead included in the compiler. Similar modifications could be made experimentally using a mutating dialect and later incorporated into the language if they proved useful. Providing a structured way of laying out the transformations that would not lead to the issues that macro systems often face would be key to a workable implementation of this extension.

Default methods. Some dialects, like the finite state machine dialect described in Section 4.3, will have simple methods that are very often defined in the modules using the dialect in order to provide access to a feature of the dialect itself. These methods usually forward to a method defined by the dialect. It might be helpful if a dialect could provide these methods itself and have them automatically included in the interface of the modules that use it (unless disabled), or to have the end programmer be able to "opt in" to them individually or collectively.

Such methods could be marked by an annotation in the dialect and recognised by the compiler. They could also be included using a mutating-dialect feature as described above, but direct support would be significantly easier to use for most dialect authors.

5 Implementation

Modules and dialects are implemented in minigrace, a compiler for Grace. The minigrace compiler is itself written in Grace and compiles Grace source code into C, for native execution, and into JavaScript, to run on the web. The native compiler is intended to run on any POSIX-compatible system with GCC, including Linux, Cygwin, and Mac OS X, although it is most robust under Linux. The compiler is distributed as

Grace source code through git[4], and as tarballs of pre-generated C code[5] which should compile on any suitable system. The JavaScript version of the compiler can be accessed without any installation using a public web frontend that runs entirely in the client[6]. However, this version of minigrace does not readily support the features described in this paper, because it provides the ability to write only one module at a time. The recommended way to obtain minigrace for casual use, or for experimenting with modules and dialects, is from a tarball.

While minigrace is a prototype, it supports almost all of the language specified to date, and is able to compile both itself and a variety of sample programs. The major limitations revolve around type checking, although basic structural type checking is supported and enforced; virtually all dynamically-typed code works. The compiler currently consists of approximately 11,000 lines of Grace code, 5,000 lines of hand-written C libraries, and 1,500 lines of JavaScript library. Limited documentation on its usage, build process, and extensions and deviations from the specification is included in the source distribution.

The minigrace compiler supports multiple modules with separate compilation, accessed through the `import "module/path" as localName` syntax described in Section 3. The compiler interprets the quoted import strings as file system paths rooted in the same location as the importing module, or in a distinguished directory provided with minigrace itself.

A module can specify the dialect it is written in using the **dialect** keyword; minigrace will locate the given dialect module in the same way as other modules. The definitions of the dialect are then inserted into scope prior to name resolution and typechecking. Next the checker given by the dialect, if any, is run over the abstract syntax tree. If the checker raises an error minigrace will report it using the ordinary mechanism for static errors, which includes an attempt to indicate the offending token.

The minigrace distribution includes a number of sample dialects implemented more fully than could be included in this paper, including the dialect for writing dialects and the StaticGrace dialect (see Section 4.3). Grace programs compiled with minigrace can use these dialects; detailed usage instructions are in the documentation in the source tree.

6 Related work

6.1 Classes and Objects as Modules

Python [4] supports for modules with separate compilation; modules become objects at run time. All top-level definitions in a source file are attributes of the module object,

[4] Available from https://github.com/mwh/minigrace. The suggested way to bootstrap minigrace is to run the `tools/tarball-bootstrap` script that can be found in the source repository and follow the instructions it gives to use one of the C tarballs to build the compiler for the first time.

[5] Available from http://ecs.vuw.ac.nz/~mwh/minigrace/dist/. Executing the commands `./configure && make` should be sufficient to build a minigrace executable.

[6] Available at http://ecs.vuw.ac.nz/~mwh/minigrace/js.

which may be referenced by another module using an import statement. The import statement includes the qualified name of the module as a sequence of dot-separated identifiers, which are mapped onto a filesystem path. The source file is loaded at runtime and the resulting singleton object bound to the imported name. This is the only way that a Python object can be created without a class.

Python objects are generally mutable, unlike in Grace, so when a module's qualified name used to import includes a dot, all modules along the chain are imported from left to right and a new field added to each parent module pointing to its new child, which is globally visible. The leftmost component of the name is bound in the local scope, so that the same qualified name used to import the module can then be used to access it. Multiple imports in the same namespace may result in several mutations to a module object higher in the hierarchy. This means that the name used to import is also the name used to access the module afterwards, but also that the behaviour of the parent module can change globally with the new field. As in Grace, the module object may be passed as a parameter and assigned, and otherwise treated as any other object, but a Python module is not an instance of a more general construct in the language. As Python is dynamically typed, there is no type information present, and Python does not enforce encapsulation other than by metaprogramming.

Bracha *et al.* [8] describe modules as classes in Newspeak. In this language a module definition is a top-level class, whose instances are termed "modules". Classes can be nested, and the code in a class can access external state in three ways: lexically, from an outer scope; from an argument provided at instantiation-time, or from a superclass. A module definition has no lexically-surrounding class and so must be passed all modules it will use encapsulated in a "platform" object, or obtain them through inheritance. Dependencies are not defined statically in the module source; instead, the dependencies of a module can depend on instantiation-time arguments, so a module may be provided a different implementation of a dependency at different times. Every module can have multiple instances and can be inherited from. In Grace we did not want to depend on a "platform" object: setting up such an object is a layer of complexity for novice programmers that will seem to be a magic incantation, where a small error can lead to disastrous effects. In contrast, Grace's practice of binding modules using **import** statements makes clear both which modules are being used, and how they are named.

Ungar and Smith [29] describe how the Self language can support modular behaviour by prototype-based object inheritance. Self does not include a distinguished "module" concept; rather, everything is an object and some objects may be used as modules. Like other objects, these module objects must be accessed by sending a message to another object or creating them locally. Any object may both inherit and be inherited from, and inheritance of environment objects subsumes the role of lexical scope, allowing an object to present a customised picture of the world to code defined inside it. The Self system as a whole works in terms of "worlds": whole ecosystems of living objects. To move (or copy) objects between these worlds, a "transporter" [28] is used, and this transporter does have a concept of a module, which it uses to produce the correct behaviour. The programmer annotates individual slots (fields or methods) with the module they belong to, and potentially with instructions for how each should be

treated. An entire module, spanning many living objects, may then be moved to another world to be used there.

In AmbientTalk [10,1] modules are written as explicit objects and loaded by asking a "namespace object" for them, which maps directly onto the filesystem according to a configuration set up earlier. Module import implicitly creates delegation methods when required, allowing modules to be imported into any object in the system.

Kang and Ryu [14] formally describe a proposed module system for JavaScript. JavaScript itself does not have any support for modules, but they are often simulated by objects or functions in the global scope, which can lead to naming conflicts. Kang and Ryu's module system extends the language with an explicit `module` declaration, creating a new namespace and binding a reference to it, which may be traversed to obtain explicitly-exported properties defined in the module. They show that their system safely isolates private state, but allows both nesting and mutual recursion. The view of a module presented to the outside is an object, although the implementation of the "module" declaration desugars to multiple objects and closures to give effect to encapsulation rules.

6.2 Packages

In contrast to the above languages, in which modules are first-class and have a run-time existence, we now look at some designs that provide what we call "packages": grouping of components that are less than first-class or which do not exist at runtime at all.

Modules in Modular Smalltalk mange the visibility and accessibility of names [30]. Modules are not objects and do not exist at runtime. Instead they define a set of bindings between names and objects. They can be used to group collaborating classes, and provide a local namespace for their own code to refer to while making only the module itself globally accessible.

Scala [22] includes the concept of "packages", which can contain classes, traits, and objects, but not any other definitions, and are imported by their qualified name. Scala also has "package objects", which are specialised objects that can be declared inside a package to augment it with other definitions. Definitions from another package can be imported directly into a scope or be accessed through a qualified name.

Java also includes "packages", which are weaker than those in Scala; they serve to subdivide the namespace of classes, as well as providing a level of visibility intermediate between public and private. Strniša *et al.* [24] propose and formally describe a module system for Java in which a module is an encapsulation boundary outside these namespace packages. A module is able to define the interface available to the outside world and to import other modules into scope with their interfaces. These modules also allow combining otherwise-incompatible components in separate areas of the program by enforcing a hierarchy on access.

Szyperski [26] argues that both modules and classes are essential components for structuring object-oriented programs. He defines a module as a statically-instantiated singleton capsule containing definitions of other items (objects, methods, types, classes) in the language, capable of exposing some, all, or none of its contents to the outside and providing unrestricted access to the contents internally. A class is a template for constructing objects, which may inherit from other classes and have many instances.

While class instances (objects) exist at run-time, a module is a purely static abstraction serving to separate code. One module may import another, obtaining access to the public contents of the other module through a qualified name. This mechanism is separate from both the inheritance and instantiation mechanisms supported by classes, but accurately reflects the intention of the programmers of both modules. Because modules have no run-time existence, however, it is not possible to parameterise code with them, and an additional language mechanism is required to define and import them. Szyperski argues that modules as he describes them are beneficial for program design because they enhance encapsulation and structure, and for programming practice because they allow separate compilation.

Modula-3 [9] includes both "interfaces" and "modules". An interface is a group of declarations without bodies; a module may export implementations of part of an interface, and may import an interface to gain access to its elements. A module may provide definitions of some, all, or none of the elements of the interfaces it exports, but will have unqualified access to definitions made elsewhere. A single interface may have its implementation spread across multiple separate modules. Only the interface needs to be referenced or understood by other code.

Standard ML [19] includes a module system built around functors. ML modules make heavy use of static types to achieve their goals. Modules bind environments, types, and functions together, and may have multiple different instantiations. The module system is powerful but includes many constructs with subtle interactions.

The BETA language itself does not include a module system, instead supporting modularisation through an integrated programming system [16]. Programmers can split their programs however they wish, because subtrees of the abstract syntax tree can be maintained and compiled separately, and then combined. Modules are not first-class entities in BETA, but modularisation of any arbitrarily complex component is possible.

In the Go language [2], a package may comprise several files, all of which declare themselves part of the same package. The definitions in these files are combined and may be accessed by clients using Go's **import** statement. Once imported, the module's public interface is available through a dotted name, but the module itself has no run-time existence. Like Grace, Go's import syntax uses opaque strings, which in practice are interpreted as filesystem paths. Associated tools are able to interpret import paths as URLs and use them to fetch and install modules from remote locations when required.

6.3 Dialects and DSLs

Andreae *et al.* [6] describe JavaCOP, a framework for implementing pluggable type systems in Java. This framework provides a declarative language for specifying new type rules and a system for enforcing those rules during compilation. JavaCOP rules may enforce, for example, that a parameter must not be null, or that a field is transitively read-only. A dialect can enforce these rules as well, but is also able to enforce broader constraints by extending or limiting the constructs available to the user of the dialect. The Checker Framework [23] provides similar functionality, with better support for overloading and some other Java language features in part by using an only-partially-declarative syntax. Imperative rules provide more power to the Checker Framework than JavaCOP at the expense of concision. Our system allows combining the two by

building dialects specifically for the purpose of writing other dialects and checkers, which may provide declarative syntax as well as allowing flexible imperative tests.

Tobin-Hochstadt *et al.* [27] describe languages as libraries in Racket, a Lisp-based language with an accompanying IDE designed for teaching. Racket supports multiple language definitions through powerful macros, which may add new functionality or replace the language syntax altogether, while translating the input source text down to core Racket. Racket includes multiple "language levels" for teaching, intended to be moved through in sequence with gradually increasing power. Earlier language levels restrict functionality that novices will not need to use, and provide more informative error messages and suggestions based on their knowledge of what the programmer can write. Other languages can also be defined that depart more radically from the basic Racket language, including a statically-typed varient of Racket, a documentation-writing language, and an implementation of ALGOL-60 with the standard syntax [5]. In contrast to DrRacket's extensive use of macros, Grace dialects are object-oriented.

Scala [22] includes powerful macro features integrating the compiler and runtime. These features have similar ability to those in Racket, including the ability to fully rewrite code and to defer some processing until runtime.

Cedalion [18] is a language for defining domain-specific languages; it aims to promote "language-oriented programming". This is a programming style in which many DSLs are used in combination, with a new language defined for each subdomain spanned by the program. Cedalion encourages the definition and use of multiple languages within the same program and provides tools to make this easy. Lorenz and Rosenan, Cedalion's designers, define four kinds of language-oriented programming system: internal DSLs, where a DSL is implemented within a host language (as in a Grace dialect), external DSLs, where the DSL is a separate language with its own compiler or interpreter, language workbenches, which combine tools and an IDE to present external DSLs as though they were internal, and language-oriented programming languages, like Cedalion. Cedalion allows the programmer to define an internal DSL and then use it like an external DSL.

Kats and Visser [15] describe the creation of domain-specific languages (DSLs) using the Spoofax workbench. Spoofax provides a declarative syntax for defining domain-specific languages and accompanying IDEs. A programmer may describe the constructs of their language and their intended behaviours, and then write their application logic within the program domain using a customised IDE with autocompletion and syntax checking as they type. A Spoofax DSL definition includes a grammar and transformation description, with additional information for errors and warnings, optimisation, and further IDE integration. A DSL defined in this way consists only what the language defines, rather than being a part of a broader language.

6.4 Foreign objects

F# accesses external data sources with "type providers" [25]. A type provider defines a way of accessing a data source outside the program — like a database or web service — and integrating it with the program as though it were an integral part of the system. Type providers are fully integrated with the IDE: when accessing a web service, for example, auto-complete menus will appear with the different actions or sub-fields available from

that service in that particular context. The F# compiler statically generates binding code according to the definitions in the type provider and what is obtained from the external source, and ensures that type information from the remote source is fully propagated into the program.

Newspeak's foreign function interface defines "alien" objects [3], which come from outside Newspeak code but appear to their clients exactly like ordinary Newspeak objects. The implementation of an alien object is unknown and undefined, but it understands and responds to messages sent to it, and knows how to use its own representation to implement its own behaviour.

7 Conclusion

Module systems of one kind or another have been part of programming languages at least since Alphard [33] in 1976 and CLU [17] and Modula in 1977 [31]. In type-centred languages, modules have been built out of types; in class-centred languages, modules have been built out of classes; in function-centred languages, modules have been built out of functions. The exact goals of modules have varied, but they generally provide an additional structuring mechanism, interacting, crosscutting, and (hopefully) modularising the core language features.

In this paper we have presented our design for a module system for Grace, which is based around objects. In this design, a module *is* an object; more precisely, a module *creates* an object when it is loaded. Because classes and types can be declared within Grace objects, modules can contain classes and types. Because objects are encapsulated, modules hide information. Because objects are gradually typed, Grace modules can be typed statically or dynamically. Because Grace is structurally typed, the features contributed by a module can be described by types defined within that module, by types within the program using the module, or by types defined in an entirely separate module. This design meets the requirements we set out in Section 1.2.

Dialects are similar to modules, except that they *surround* the code written using the dialect, rather than being included by it. Dialects can both add language features by defining methods, and remove language features, or enhance error reporting, using checkers. Dialects enable Grace to support both multiple teaching languages for novices and domain specific languages for advanced students.

It might seem that a module system that added all these features into a language would be rather heavyweight. The key idea behind Grace's module system is that a module is, essentially, just a Grace object. From this, all the other features follow.

References

1. Ambienttalk documentation: Modular programming. http://soft.vub.ac.be/amop/at/tutorial/modular, last accessed December 17 2012.
2. Go language website. http://golang.org/, last accessed 17 December 2012.
3. Newspeak foreign function interface user guide. http://wiki.squeak.org/squeak/uploads/6100/Alien%20FFI.pdf, last accessed 17 December 2012.
4. Python website. http://python.org/, last accessed 17 December 2012.

5. Racket ALGOL-60 documentation. http://docs.racket-lang.org/algol60/index.html, last accessed 17 December 2012.
6. C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. OOPSLA '06, pages 57–74. ACM, 2006.
7. A. P. Black, K. B. Bruce, M. Homer, and J. Noble. Grace: the absence of (inessential) difficulty. In *Onward!*, 2012.
8. G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in Newspeak. ECOOP'10, pages 405–428. Springer-Verlag, 2010.
9. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *SIGPLAN Not.*, 27(8):15–42, Aug. 1992.
10. J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter. Ambient-oriented programming in AmbientTalk. In *ECOOP*, pages 230–254, 2006.
11. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.
12. M. Homer. Grace-GTK source repository. https://github.com/mwh/grace-gtk, last accessed 17 December 2012.
13. M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In A. Warth, editor, *DLS*, pages 17–28. ACM, 2012.
14. S. Kang and S. Ryu. Formal specification of a JavaScript module system. *SIGPLAN Not.*, 47(10):621–638, Oct. 2012.
15. L. C. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and ides. OOPSLA '10, pages 444–463. ACM, 2010.
16. B. B. Kristensen, O. L. Madsen, B. Möller-Pedersen, and K. Nygaard. Syntax-directed program modularization. In P. Degano and E. Sandewall, editors, *Integrated Interactive Computing Systems*, pages 207–219. North-Holland, Amsterdam, 1983.
17. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Comm. ACM*, 20(8):564–576, Aug. 1977.
18. D. H. Lorenz and B. Rosenan. Cedalion: a language for language oriented programming. *SIGPLAN Not.*, 46(10):733–752, Oct. 2011.
19. D. MacQueen. Modules for Standard ML. LFP '84, pages 198–207. ACM, 1984.
20. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
21. J. Noble and J. Grundy. Explicit relationships in object oriented development. In *TOOLS 18*. Prentice Hall, 1995.
22. M. Odersky. The Scala language specification. Technical report, Programming Methods Laboratory, EPFL, 2011.
23. M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. ISSTA '08, pages 201–212. ACM, 2008.
24. R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. *SIGPLAN Not.*, 42(10):499–514, Oct. 2007.
25. D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. Strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.
26. C. A. Szyperski. Import is not inheritance - why we need both: Modules and classes. In *ECOOP '92*, pages 19–32.
27. S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *PLDI*, 2011.
28. D. Ungar. Annotating objects for transport to other worlds. *SIGPLAN Not.*, 30(10):73–87, Oct. 1995.
29. D. Ungar and R. B. Smith. Self: The power of simplicity. OOPSLA '87, pages 227–242. ACM, 1987.
30. A. Wirfs-Brock and B. Wilkerson. A overview of Modular Smalltalk. OOPSLA '88, pages 123–134. ACM, 1988.
31. N. Wirth. Modula: A language for modular multiprogramming. *Software — Practice and Experience*, 7:3–35, 1977.
32. N. Wirth. *Programming in Modula-2*. Springer Verlag, 1985. ISBN 0-387-15078-1.
33. W. A. Wulf, R. L. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Softw. Eng.*, SE-2(4):253–265, 1976.