Modules as Gradually-Typed Objects

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

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

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

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

ABSTRACT

Grace is a gradually typed, object-oriented language for use in education. Grace needs a module system for several reasons: to teach students about modular program design, to organise large programs, especially its self-hosted implementation, and to provide access to resources defined in other languages. Grace uses its basic organising construct, objects, to provide modules, and is then able to use its gradual structural typing to obtain a number of interesting features without any additional mechanisms.

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 [4]. Two principles have helped us to keep Grace small and easy to learn:

P1. omit from the Grace language itself anything that can be defined in a library; and

P2. 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.

Principle P1 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. Principle P2 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, showing how they are built from more basic language features yet permit powerful modular functionality.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
DYLA '13, July 01 2013, Montpellier, France
Copyright is held by the authors. Publication rights licensed to ACM.
ACM 978-1-4503-2041-2/13/07 ...$15.00.

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. We want the module system to support the different applications of modules that students may need to learn, and so arrived at a set of requirements an idealised module system should meet.

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: clients should be able to share the objects provided by a module.

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: these dependencies should be explicit so that a reader may follow the dependency chain and flow of execution.

Grace meets these requirements by representing modules as objects. Combining these module objects with Grace's gradual structural typing provides a wide range of functionality. This approach to the design of module systems has been influenced by Python and Newspeak.

1.3 Contributions

The contributions of this paper are:

- the design of a module system in which modules are objects with optional gradual structural typing (Section 3); and
- the rationale that led us to this design (Section 4);

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 [4]. 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. In Grace it is possible to start using types from the beginning, 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.

A Grace object is created by executing an object constructor expression; each time the object constructor is executed, it creates a new object:

```
def fido = object {
    def name = "Fido"
    var age := 2
    method say(phrase : String) {
        print "{name} says: {phrase}"
    }
    print "{name} was born."
}
fido.say("Hello")
```

Teaching curricula using Grace may start with objects or with classes. In Grace's conceptual model, objects are primary, and own their own methods. Classes are "factory" objects that create "instance" objects.

Objects can contain methods and fields. A constant is declared with **def** and defined with =, while a variable uses **var** and is assigned (and re-assigned) with :=. The object fido has a constant field name and a variable field age, and a method called say whose parameter has type String. Grace uses "curly bracket" syntax and the form obj.m to request that object obj execute its method m — an action that we call *method request*. To prevent a method being accessed from outside the object, the programmer may annotate it with **is** confidential.

When an object is constructed, any inline code in the definition is executed. Strings can include expressions enclosed in braces: the expression is evaluated, converted to a String, and inserted in place of the brace expression. Thus, the above program prints "Fido was born." and then "Fido says: Hello".

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 dog.named(n : String) {
    def name = n
    var age := 0
    method say(phrase : String) {
        print "{name} says: {phrase}"
    }
    print "{name} was born."
}
def fido = dog.named "Fido"
```

The above class is a factory for constructing objects with the same structure and effects as the object constructor earlier. The class definition creates an object called "dog", with a factory method called "named". Instantiating a class is requesting a method on an object.

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

```
type Speaker {
    say(_ : String) -> Done
}
def fido : Speaker = dog.named "Fido"
```

The type Done indicates that a method does not return a useful result.

Types are gradual and structural. They are *gradual* because if no types are specified, code is dynamically typed, but when statically- and dynamically-typed code are combined, automatic runtime type checks preserve safety. Types are *structural* because an object has a type exactly 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. All of our definitions of fido result in an object belonging to the type Speaker. Along with methods, types may be included as components of objects. Types have a runtime representation as a *pattern* [10] object.

3. MODULES AS OBJECTS

A Grace module is a piece of code that constructs an object, and the object then constructed. This *module object* behaves like any other object; in particular, it may have types and methods as attributes, and can have state. A module corresponds to a source file: the module object is created as if the entire file were inside an **object** {...} constructor. Alternative implementations may allow other representations of a module's source code. Here is a complete, if simple, module:

```
def person = "reader"
method greet(name) {
    print "Hello, {name}!"
}
greet(person)
```

Executing this module will print "Hello, reader!" and construct a module object with the greet method in it. That module object has normal Grace object semantics, and so may contain many methods, objects, classes, fields, and types, or none. The effect of this is similar to modules in Python [16] (discussed in Section 7.1, but uses existing language features instead of adding a new concept that must be explained and has its own idiosyncratic behaviours.

Modules as objects provide a consistent story for interactive read-eval-print loops as well as scripts: programmers define an object progressively as they write. As we shall see, making modules objects, rather than introducing a new feature, 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 "examples/greeter" as doorman
```

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. If we assume that `"examples/greeter"` refers to the simple module shown above, then a name doorman is introduced in the local scope, bound to an object with a greet method.

To take advantage of static type checking, a variant of the import statement allows the programmer to specify the type that the imported module should meet:

```
import "examples/greeter" as doorman : BasicGreeter
```

Types in Grace specify an interface, not an implementation, so this asserts that the imported module object must satisfy the interface defined by the BasicGreeter 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 BasicGreeter = {
    greet(n : String) -> Done
}

import "examples/greeter" as doorman : BasicGreeter
```

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. A Grace implementation should save enough information about a module to avoid the need to process the source code again each time it is imported.

3.2 Gradual typing of modules

Through careful use of modules, imports, and type specifications, a system can be configured in a range of different ways covering many of the common purposes of modules.

- We can import a module with its original types, whatever those may be, using **import** `"x"` **as** x. 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.

- We can ignore any type information specified in the module `"x"` using **import** `"x"` **as** x : Dynamic. 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 [25]. The type of the 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)
  ```

  Using the library method assertType<T>(o) causes the compiler to statically check that o has type T, so this code asserts that the current module object has the type T imported from spec.

- 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; we may develop a module gradually starting from dynamic code and either moving to static or not, while using a consistent target interface in client code.

In a given program, a module is executed only once. Every import of the same path within a program will access the *same* module object.

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 pro-

viding one of the modules with a reference to the other as an ordinary method parameter.

4. DESIGN RATIONALE

In Section 1.2, we laid out the requirements for modules in Grace. Grace objects already satisfy many of these requirements. Top-level objects cannot capture variables, so they can be compiled separately (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 and our existing gradual structural typing 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 the design principle of building a small language (P2).

We considered an alternative design in which modules were classes, as they are in Newspeak [5]. 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 the module with an explicit `platform` parameter containing its dependencies. We eventually rejected this approach because we wished to make dependencies as explicit as possible in the code using them, because we wanted to avoid formulaic incantations, and because Grace is primarily based on objects (with classes derived from objects), rather than on classes (instantiating objects), as is Newspeak.

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 could 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) could 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 complicated.

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, the lack of uniformity, and the need for special-case behaviour, all of which would need to be explained to students.

5. IMPLEMENTATION

Modules 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 in a web browser. 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[1], and as tarballs of pre-generated C code[2] that 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[3]. The JavaScript version of `minigrace` has limited support for the features described in this paper because it does not support editing multiple modules at once, but it is possible to define and use multiple modules sequentially. The recommended way to obtain `minigrace` for casual use, or for experimenting with modules, is from a tarball.

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. The compiler also supports the foreign objects extension from Section 6.1 by default on both target platforms, and the import hooks used to implement external data (described in Section 6.2) by a compiler flag.

6. 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.

6.1 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. Because these foreign objects present themselves as ordinary objects to Grace code, all of the ordinary facilities of objects and modules are available for use with them: a foreign (perhaps optimised) module may be substituted for a Grace implementation used with a type specification (as described in Section 3.2) in exactly the same way that another Grace implementation could be so substituted.

We have written a fairly complete Grace binding to the GTK+ widget library [8] 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)
```

[1]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.

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

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

```
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.

Because these foreign objects appear as ordinary Grace objects, all of the gradual typing functionality discussed in Section 3.2 will work unchanged. While the actual implementation is unknown, the public interface of the object is subject to the same strictures as for any other object. In the case where the interface of the foreign module or other objects returned from its methods is unknown or subject to addition at runtime, the gradual enforcement can be based on the information currently available.

6.2 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. In particular, we can interpret some strings as references to 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" [18]. 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. As for other potentially-foreign objects, gradual types can enforce that we do get what we expected when using an external data source. The source itself can provide types for us to use dynamically. 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. We hope to extend our prototypes to support a wider range of external data sources and investigate dynamically-provided types further in the future.

7. RELATED WORK

7.1 Classes and Objects as Modules

Python [16] supports modules with separate compilation, which become objects at run time. All top-level definitions in a source file are attributes of the module object. Python's **import** statement includes the qualified name of the module as a sequence of dot-separated identifiers, which are mapped onto a filesystem path: **import** x.y.z resolves to a file z.py inside the directory x/y. 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. The **import** statement may include an optional as clause providing a name to bind, exactly as in our system.

When the qualified name of a module includes multiple levels, as above, each intermediate layer is itself a module. The module x is defined in x/__init__.py. When a module's qualified name includes a dot all modules along the chain are imported from left to right, and the leftmost component of the name is by default bound in the local scope so that the same qualified name used to import the module can be used to access it. To make that name available, each intermediate module has a new field added referring to the next in the chain: after **import** x.y.z x has a y field, and x.y a z, and these changes are globally visible. This mutation is necessary for the design to work, and possible because Python objects are by default mutable. Our objects are not mutable in this way, and we did not want this complication in Grace. Because multiple imports of the same module give the same object in Python (as in Grace), these mutations also obscure the actual dependencies of client code. 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 in the way that ours are instances of object constructors. As Python is dynamically typed, there is no type information present, and Python does not enforce encapsulation other than by metaprogramming.

Bracha *et al.* [5] 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 in different programs. 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 or even passing along such an object is another point of complexity for novice programmers that will seem to be a magic incantation, while the actual dependencies of a module are implicit and may change depending on what is provided from the outside in the platform object. 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 [22] 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 itself. 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" [21] 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. We did not wish to commit to a world-based approach, but did want to have a concept of modules that did not encompass every object.

In AmbientTalk [7, 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. AmbientTalk's approach contrasts with ours in writing module objects explicitly, and in permitting imports to happen anywhere. We chose to ensure that all dependencies were clearly visible by permitting them only at the beginning of a module.

Kang and Ryu [11] 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. The semantics of modules are much more complicated than in our system, largely because of JavaScript's idiosyncratic scoping and visibility rules.

7.2 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, unlike in Grace.

Modules in Modular Smalltalk mange the visibility and accessibility of names [23]. 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. Our modules provide a superset of this functionality, as objects provide visibility, accessibility, and namespace behaviours.

Scala [15] 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. Package objects are not required in our design, as our modules are already objects. We do not permit unqualified import because it masks the origin of a method, but some applications of that behaviour are possible using our dialect mechanism [9].

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.* [17] 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.

Modula-3 [6] 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. Our system does not allow these partial implementations, other than by inheritance, but the interface (a type) and the implementation (a module object) may be defined and used separately.

Standard ML [14] 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. Because Grace is gradually-typed, the module system cannot involve any behaviours that depend on the static type information in a program. Our system binds types and functions (as methods) together, with state, but does not support the more complicated type-dependent behaviour ML modules may have.

Racket [20] includes modules packaging multiple functions into a unit of distribution. Modules are imported using the require function, providing it a string generally interpreted as a relative file path, and defined either explicitly using the module form or implicitly by the variant of the language a file is written in. Typed Racket includes a require/typed function, which assigns local types to the functions and structures imported from the module. These types are enforced as contracts. The module itself does not have any run-time existence as it would in Grace; instead require simply makes its contents available to the importing code. Our design does not include an explicit module construct as in Racket; we have instead opted for consistency in providing one way of creating a module from Grace code, while abstracting away the actual behaviour behind the veil of an object. We were able to build Grace's "dialects" [9] system, our version of Racket's sublanguages concept, using our module system and without having the dialect participate in the actual construction of the final module, so a special module form was an unnecessary addition.

Alphard [26] incorporates a module system aimed at enabling program verification. Implementation and interface are separated, so that verification needs only to confirm that the implementation behaves in accordance with the specification. Our system allows types to be defined separately

from the implemenation, but does not incorporate other verification, which we consider should be only enabled, but not required, by the language semantics. A dialect [9] can perform any additional verification desired on a module-by-module basis.

CLU [13] includes modules that encapsulate the implementation of a desired behaviour. These modules may interact with each other through public interfaces, which fully characterise the module. A module may be replaced by another implementation with the same abstract interface. A special language construct permits collecting together operations into an abstract interface. Our modules also encapsulate implementation and may communicate only through public interfaces, but do not use a special construct to collect together different operations. Grace code is written in-place to construct an object that has the behaviours and state defined in the module.

Szyperski [19] 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. Modules in our system are singleton capsules with all of the traits described as desirable, except that they also have run-time existence. Because modules are objects, they may be passed as arguments to other code exactly as any other object, while the gradual structural typing also allows asserting the programmer's expectations about the imported modules as well as their intention to use them.

The BETA language itself does not include a module system, instead supporting modularisation through an integrated programming system [12]. 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. Grace does not depend on such an integrated editor, although it does not preclude using one; our modules are first-class top-level entities that may be defined in a file.

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. Our design does not include this multiple-file definition semantics, although it is not prohibited either: the interpretation of the import string is up to the implementation. The static appearance of modules is similar in Grace and Go, using dotted qualified-name notation and a similar import statement, but our modules do have runtime existence. We do not currently have tools to interpret the import paths as URLs, but were influenced by Go in allowing the possibility.

7.3 Foreign objects

F# accesses external data sources with "type providers" [18]. 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. Our system allows this behaviour, but we do not have the tooling support necessary necessary to provide the live access F# allows.

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. These alien objects are similar to our foreign objects, but not necessarily integrated with the module system.

8. CONCLUSION

Module systems of one kind or another have been part of programming languages at least since Alphard [26] in 1976 and CLU [13] and Modula in 1977 [24]. 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.

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.

9. 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\FFI.pdf, last accessed 17 December 2012.

[4] A. P. Black, K. B. Bruce, M. Homer, and J. Noble. Grace: the absence of (inessential) difficulty. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! '12, pages 85–98, New York, NY, USA, 2012. ACM.

[5] 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.

[6] 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.

[7] 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.

[8] M. Homer. Grace-GTK source repository. https://github.com/mwh/grace-gtk, last accessed 17 December 2012.

[9] M. Homer, J. Noble, K. B. Bruce, and A. P. Black. Modules and dialects as objects in Grace. Technical report, School of Engineering and Computer Science, Victoria University of Wellington, 2013.

[10] M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 17–28, New York, NY, USA, 2012. ACM.

[11] S. Kang and S. Ryu. Formal specification of a JavaScript module system. *SIGPLAN Not.*, 47(10):621–638, Oct. 2012.

[12] 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.

[13] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Comm. ACM*, 20(8):564–576, Aug. 1977.

[14] D. MacQueen. Modules for Standard ML. LFP '84, pages 198–207. ACM, 1984.

[15] M. Odersky. The Scala language specification. Technical report, Programming Methods Laboratory, EPFL, 2011.

[16] Python Software Foundation. Python website. http://python.org/, last accessed 17 December 2012.

[17] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. *SIGPLAN Not.*, 42(10):499–514, Oct. 2007.

[18] 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.

[19] C. A. Szyperski. Import is not inheritance - why we need both: Modules and classes. In *ECOOP '92*, pages 19–32.

[20] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.

[21] D. Ungar. Annotating objects for transport to other worlds. *SIGPLAN Not.*, 30(10):73–87, Oct. 1995.

[22] D. Ungar and R. B. Smith. Self: The power of simplicity. OOPSLA '87, pages 227–242. ACM, 1987.

[23] A. Wirfs-Brock and B. Wilkerson. A overview of Modular Smalltalk. OOPSLA '88, pages 123–134. ACM, 1988.

[24] N. Wirth. Modula: A language for modular multiprogramming. *Software — Practice and Experience*, 7:3–35, 1977.

[25] N. Wirth. *Programming in Modula-2*. Springer Verlag, 1985. isbn 0-387-15078-1.

[26] 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.