Graceful Dialects

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

[1] Victoria University of Wellington, Wellington, New Zealand
{mwh,tim,kjx}@ecs.vuw.ac.nz
[2] Pomona College, Claremont, California, USA
kim@cs.pomona.edu
[3] Portland State University, Portland, Oregon, USA
black@cs.pdx.edu

Abstract. Programming languages are enormously diverse, both in their essential concepts and in their accidental aspects. This creates a problem when teaching programming. To let students experience the diversity of essential concepts, the students must also be exposed to an overwhelming variety of accidental and irrelevant detail: the accidental differences between the languages are likely to obscure the teaching point.
The dialect system of the Grace programming language allows instructors to tailor and vary the language to suit their courses, while staying within the same stylistic, syntactic and semantic framework, as well as permitting authors to define advanced internal domain-specific languages. The dialect system achieves this power though a combination of well-known language features: lexical nesting, lambda expressions, multi-part method names, optional typing, and pluggable checkers. Grace's approach to dialects is validated by a series of case studies, including both extensions and restrictions of the base language.

Keywords: Grace · language variants · domain-specific languages · pluggable checkers · graphical microworlds · error reporting · object-oriented programming

1 Introduction

Grace is an imperative, gradually typed, object-oriented language designed for use in education, particularly for introductory programming courses [3,4]. The goals of Grace are similar to those of Pascal, of which Wirth wrote (in 1971!)

> The development of the language . . . is based on two principal aims. The first is to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language [27].

In the intervening forty-plus years, object-orientation has evolved into the dominant style of programming, and thus *one* of the styles to which students should be exposed if they are to receive a well-rounded education in computing. The design of Grace was intended to take advantage of recent research in programming language design to create a syntactically and conceptually simple language that could be used to teach the fundamental concepts of object-oriented programming. The focus of the language design was on consolidation of known features, rather than on innovation.

Grace supports a variety of approaches to teaching programming, including objects-early, objects-late, graphics-early, and functional-first. Grace makes it possible to teach courses using dynamic types or static types, to start with dynamic typing and then gradually move to static typing, or to do the reverse, without having to change to another language with a different syntax, semantics, IDE, and libraries.

This paper describes Grace's dialect system, which we introduced to support this variety. A *dialects* is package of extensions to and restrictions on the core Grace language that can be used for all or part of a program. A dialect can restrict access to some language features, replace existing functionality, and create new constructs and control structures. We show how we have built this system for dialects entirely out of existing well-known features, found both in Grace and in other languages; both the semantics of a dialect and the code that implements it are defined in core Grace.

In adding dialects to Grace, we intend to allow an instructor to use a succession of language variants tailored to the students' stage of learning and to the instructor's course design. The idea of a succession of teaching languages was introduced in SP/k [12], and revived in DrScheme [8] (now Racket) as "language levels". Language levels demonstrated the benefits of limiting the student programming language to the concepts that they have already learned, and excluding the features that they don't yet know about.

Because we expect different courses using Grace to use different approaches to teaching programming, we do not want to provide only a single sequence of dialects, as in Racket. Rather, we envisage a directed graph of such dialects — indeed, we hope that instructors, tutors, and course designers will be able to create custom dialects to suit their individual approaches to teaching. To make this hope realistic, we took care to ensure that defining a dialect requires nothing more than programming with ordinary Grace constructs. Instructors do not have to learn a new macro language, or generate code, or engage with the whole panoply of "professional" language building tools, like lexers, parsers, typecheckers, interpreters, and compilers. To keep the Grace language itself small and simple, we also tried to minimise both the number and the complexity of the features that we added to support dialects. As a consequence, Grace's dialect mechanism is limited in power: Grace dialects cannot implement a language with a completely different syntax or underlying semantic model. For our intended audience, we see this as an advantage.

The contributions of this paper are a description (in Sect. 3) of Grace's dialect mechanism: a system that extends (through libraries) and restricts (using pluggable checkers) the language available to a program module. The dialect system is made possible by some key features of Grace — lexical nesting, lambda-expressions, multi-part method names, and optional typing, which are described in Sect. 2. We demonstrate the power of this approach to dialects by presenting a range of case studies: dialects that define a graphical micro-world inspired by Logo (§ 4.1), implement assertions and design by contract (§ 4.2), require explicit type annotations (§ 4.4), suggest fixes to students who make simple errors (§ 4.6), support the writing of other dialects (§ 4.3), and perform static type-checking (§ 4.5). Section 5 discusses alternatives and extensions to our design; Section 6 compares Graceful dialects to a range of related work.

2 Grace in a Nutshell

This section introduces the core of the Grace programming language; it provided a basis for Sect. 3, which describes the elements that we added to support dialects.

Objects. A Grace object is created using an object constructor expression; each time the object constructor is executed, it creates a new object. Here is an example

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

This object contains a method (say) and two fields; **def** name defines a constant (using "="), while **var** age declares a variable, whose initial value is assigned with :=. Variables can be re-assigned, also with :=. When an object constructor is executed, any code inside its body is also executed, so the above object constructor will have the side effect of printing “Fido has been born.” when the object is created. This example also shows that strings can include expressions enclosed in braces: the expression is evaluated, converted to a string, and inserted in place of the brace expression.

Of course, to be useful, the object created by executing an object constructor typically needs to be bound to an identifier, or returned from an enclosing method. For example,

```
def fido = object {
   ... code from above example ...
}
fido.say "Hello"
```

will create an object and bind it to the name fido, and then *request* the say method on that object. This will print “Fido was born.” and then “Fido says: Hello”. Grace uses the term “method request” in preference to “message send”, because “sending a message” might easily be misinterpreted as referring to a network message. We prefer “request” over “call” to recognise that the receiver must cooperate in responding to the request.

Program Structure. Every file containing a Grace program is considered to be surrounded by **object** { ... }. This means that top-level declarations actually declare fields and methods in an anonymous object, which we call the *module object*. When a file is run, it is the constructor of this module object that is executed; this has the effect of running any code written at the top level of the file. Module objects have access to Grace’s standard prelude, which defines the language’s basic objects (numbers, strings, booleans, etc.), control structures, and methods.

Object constructors can be nested inside other objects or methods. Method requests without an explicit receiver are resolved in the lexical scope, finding a valid receiver in one of the surrounding objects. In the case of ambiguity, the programmer must resolve the receiver explicitly to **self** or **outer**. Classes in Grace are syntactic sugar for methods that return the result of an object constructor; classes do not serve as types. We do not use classes in this paper, and so we omit further details.

Visibility. By default, methods are publicly accessible. This default can be changed using an *annotation*: a name attached to a declaration using the keyword **is**. If a method is annotated confidential, it can be requested inside the object itself, and by any object that inherits from it, but not from outside objects. In contrast, fields are confidential[4] by default; they can be made public by an annotation. Regardless of visibility, code can access names defined in any of the surrounding scopes. This includes not just requests of methods and fields, but also parameters and temporaries. The implementation creates closures when necessary.

When visible, a variable or constant can be requested using *exactly* the same syntax as a parameterless method, as demanded by Ross's Uniform Referent Principle [24]. A variable can be assigned in a similar way, using a special request syntax that is syntactically identical to an assignment. Thus, the clients of an object need not know whether an attribute is implemented as a constant, as a variable, or as a method.

Types. Variables, definitions, method parameters and method return values can optionally be annotated with types. Grace supports a special annotation syntax for types, using : for variables, definitions and parameters, and −> for method results, for example:

```
method square(n : Number) −> Number { n * n }
```

A type annotation is an assertion on the part of the programmer that no attempt will be made to bind a value with a non-conforming type to the annotated program element — in this case, the programmer is asserting that arguments and the return value of the method square will conform to Number. The implementation will emit a warning or error if a type assertion does not hold; sometimes the warning will be produced at run time, and sometimes at compile time, depending on how the erroneous value is generated and on how many type annotations are present.

Types in Grace are structural. A type is a set of methods, where each method is decorated with the types of its parameters and the type of its result. Type a conforms to type b if it obeys the usual contravariant rule: a must support all of the methods of b, and for each common method m, the result type of m in a must conform to the result type of m in b, and the arguments types of m in b must conform to the argument types of m in a. Types can be given names for convenience, but the name of the type plays no role in checking type conformance. Types and objects can have attributes that define names for types.

[4] Grace uses the term *confidential* rather than *private* or *protected* because Grace's *confidential* is incomparable with the meaning of private and protected in Java and C++.

Grace supports gradual typing. Identifiers without type annotations are considered to have type Unknown, which is compatible with all types, but which may allow type errors at runtime.

Blocks. Grace *blocks* provide a concise syntax for lambda expressions (first-class functions). Grace's blocks are written between braces; if the block has parameters, the names of the parameters are written after the opening brace, separated from the body of the block by an arrow, so { x −> x + 1} defines the successor function. A block creates an object with an apply method with the same number of parameters as the block; requesting that a block apply itself evaluates the body of the block, and returns the value of the final expression in the body. Thus, { x −> x + 1}.apply(3) returns 4.

Patterns. Grace supports an object-oriented form of pattern matching [15]. A unary block with a type annotation on its parameter can be interpreted as a partial function — the block will execute if the argument matches the type annotation, otherwise the block will fail to match. The following code will add one to obj if it is a numeric object, suffix "one" to obj if it is a string object, and otherwise raise an error:

```
match (obj)
    case { x : Number −> x + 1 }
    case { s : String −> s ++ "one" }
    case { _ −> Error.raise "no match: {o} is neither a Number nor a
        String" }
```

In this example, the type acts as a pattern, but types are not the only patterns. In general, any object that responds to a match request with a MatchResult can be used as a pattern. For example, pattern.match(datum) tests if datum is matched by pattern. Type patterns match when the argument object has the methods of the type, but user-defined patterns can define their own criteria for matching. Primitive objects like numbers and strings match when they are equal to their argument.

Exceptions. Grace supports exception handling through an extension of the pattern system. Exceptions are raised using the raise method and caught by a special construct, using the exception as a pattern.

```
try { IndexOutOfBounds.raise "index {i} exceeds upper bound {u}" }
    catch { e : IndexOutOfBounds −> ... }
    catch { e : RuntimeError −> ... }
    finally { ... }
```

When an exception is raised, it is handled by the first block that matches the exception object. The raiseWith method permits the user to attach additional data to the exception packet.

Multi-part Method Names. Grace method names may contain multiple parts, making Grace method requests similar to Smalltalk message sends. For example, we can write

$2 < x < 5$ as x.isBetween (2) and (5). The combination of blocks and multi-part names allows control structures to be defined as methods:[5]

```
method if (cond : Boolean) then (body : Block) {
  cond.ifTrue(body)
}
method while (cond : Block) do (body : Block) {
  if (cond.apply) then {
    body.apply
    while (cond) do (body)
  }
}
while { x > 0 } do {
  print "{x} bottles of beer on the wall"
  x := x − 1
}
```

Because "control structures" are method requests, the placement of braces and parentheses is not arbitrary. The condition of an if statement is parenthesised, because the if's condition is a boolean expression that is evaluated exactly once. In contrast, the condition in a while may be re-evaluated many times, and must therefore be a block, which means that it is surrounded by braces. This is a departure from most other curly-brace languages, but represents semantic consistency. Of course, these two condition arguments have different types, so errors can be detected statically or dynamically.

Multi-part names do not cause a syntactic ambiguity like Algol 60's "dangling else" problem. This is because method arguments must always be delimited — either with parentheses, or, in the case of string and block literals, by the literals' own delimiters. A Grace program's layout must be consistent with its parse: a method request terminates at the end of the line, unless the next line is indented to indicate a continuation.

We included multi-part method names, along with blocks, to allow objects that represent data structures to provide methods that implement internal iterators and other control structures to look much as they do in other "curly bracket" languages. Combined with implicit receivers, multi-part names also make it easy to provide the "statements" of a dialect, as we show in this paper.

Modules. Any file containing Grace code can be treated as a module [14]. To access another module, the programmer uses an import statement, such as

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

The string that follows the **import** keyword must be a string literal (not an expression) that identifies (in an implementation-dependent fashion) the module to be imported. The effect of the import statement is to bind the name that follows **as** to the imported module object.

As mentioned earlier, the code in every file is treated as the body of an object constructor. The *module object* — the object generated by this constructor — behaves like

[5] Implementations may make these structures primitive for efficiency, as does our current prototype. The code shown here illustrates how they could be defined in Grace.

any other object. In particular, a module object may have types and methods as attributes, and can have state. Here is a complete, if simple, module:

```
def person = "reader"
type Greeter = { greet(name : String)−>Done }
method greet(name) {
  print "Hello, {name}!"
}
greet(person)
```

Executing this module will print “Hello, reader!” and construct a module object containing the type Greeter and the method greet.

If we assume that "examples/greeter" refers to the module shown above, then **import** "examples/greeter" **as** doorman introduces the name doorman into the local scope, bound to the module object. Every import of the same string within a program will access the *same* module object, although each import may bind it to a different name.

Implementation. We added the dialect functionality to Minigrace, our prototype Grace compiler. Minigrace is available from http://www.gracelang.org/ and includes the case studies described in this paper along with others. Minigrace is expected to function on POSIX-compatible systems with GCC. A web-based version of the compiler, running in JavaScript in the client’s browser, is available at http://ecs.vuw.ac.nz/~mwh/minigrace/js/. This version includes all of the case studies described in this paper as loadable samples.

3 Dialects

Dialects are modules that can both extend and restrict the standard Grace language. Dialects can not only make extra definitions available to their users; they can also restrict the language by defining and reporting new kinds of errors, and can change the way in which existing errors are reported. Dialects support the definition of language subsets to aid novice programmers, and of domain-specific languages.

3.1 Structure

A module declares the dialect in which it is written with a dialect declaration, like **dialect** "beginner", which loads the module named by the string, just as if it were imported. However, unlike an import statement, the dialect declaration does not bind the imported object to a name: instead, the dialect object is installed as the lexically-surrounding scope of the module that uses it, as shown in Fig. 1. Any request in the client module for a method defined in that outer scope — most often a receiverless request — will access a method of the dialect. This resolution rule is the same rule used for any other receiverless request in a lexically nested scope. Thus, if diaMeth is a method defined in the dialect, then, in a module (such as ModuleC) that is written in the dialect and does not contain a new definition of diaMeth, a receiverless request for diaMeth will invoke the method defined in the dialect.

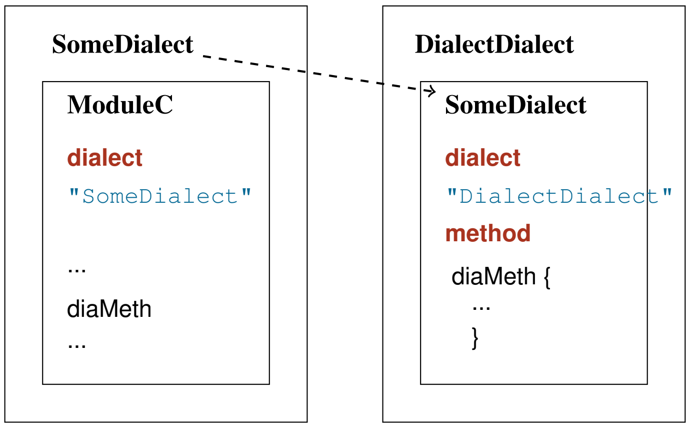


Fig. 1. Object nesting with dialects. The declaration **dialect** `"d"` logically nests the current module inside the module d. Notice that dialect use is not transitive: ModuleC is inside SomeDialect, and SomeDialect is inside DialectDialect, but ModuleC is not in DialectDialect.

When no dialect is specified, the module is assumed to be written in the standard Grace language, which uses the standard prelude as its dialect. The dialect mechanism thus provides a coherent explanation of how Grace's standard prelude works: a program in standard Grace generates a module object nested inside the standard prelude object. Because a dialect *replaces* this nesting, the author of a dialect can choose whether or not to expose the standard prelude's methods to their clients. If they wish, they can write

```
inherits StandardPrelude.methods
```

at the top of their dialect, and expose all of the methods of standard Grace.

A module that defines a dialect may itself be written in a dialect. This reveals a difference between dialectical nesting and other kinds of lexical nesting: the dialect is the *outermost* lexical scope, so dialectical nesting is not transitive, as shown in Fig. 1. The reason for this design decision is that special-purpose dialects, particularly those defining educational subsets, will commonly be less powerful than the language as a whole. These dialects will thus typically be written in a dialect — such as standard Grace — that provides the dialect writer with features that should not be exposed to clients.

3.2 Pluggable Checkers

As well as providing new definitions, dialects may restrict access to particular features of the language, or offer additional and more specific error and warning messages. The latter are useful because novice students can benefit from error messages that are tailored to the more restricted things that they are trying to do, compared to more advanced programmers.

Restrictions and new error messages are implemented by the dialect module defining a *checker method*, which is executed when modules written in the dialect are compiled. The checker method is passed as argument the abstract syntax tree of the module, and typically traverses that tree using one or more visitors. The visitors cannot change the tree, but can implement any checks the dialect needs, and can also indicate to the compiler whether it should proceed, or terminate with an error.

Checkers have the same ability to find and report errors as the compiler itself. They can perform any analysis they require: for example, a dialect may wish to perform a flow analysis to ensure that method parameters are used. If an error is found, the checker can report that error to the user, including whatever information the dialect author thinks is relevant, and either carry on to find more errors or stop at that point. Several modules that provide varied degrees of checking can be used within the same program, so a student's code can be subjected to strict constraints, while still being able to use a module provided by their instructor written in a more powerful dialect.

While a checker can examine the code of its client module using any technique the programmer wishes, we provide two mechanisms to make dialect-creation easier. One is support for the Visitor pattern [10] on the AST nodes, which we illustrate in Sect. 4.4; the other is a dialect to support largely-declarative definitions of checkers, which is presented in Sect. 4.3.

3.3 Run-time Protocol

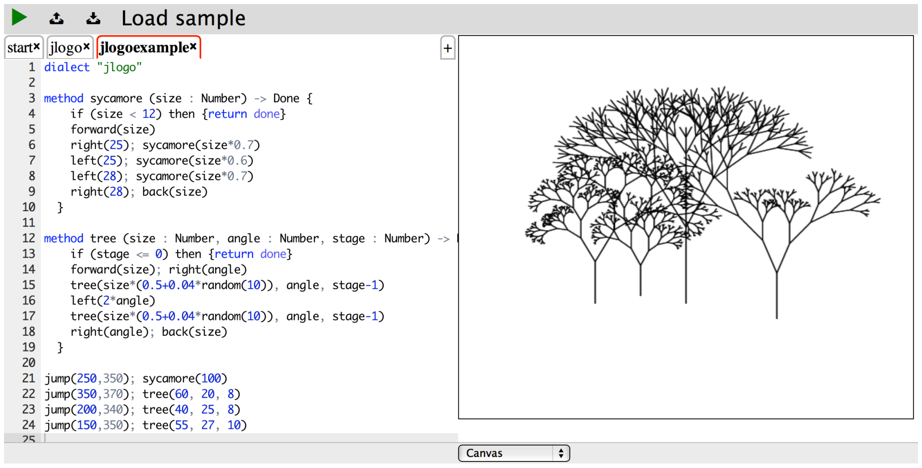
A dialect may wish to run code immediately before or after a module using it, perhaps for logging, initialising data structures, or launching a user interface. To enable this, the dialect protocol includes two further methods the dialect can define: `atModuleStart` and `atModuleEnd`. The method `atModuleStart` is requested, if it exists, immediately before the module written in the dialect is executed, and receives a single argument: a string containing the name of the module using the dialect. In Fig. 1, the string `"ModuleC"` would be provided to `SomeDialect.atModuleStart(...)`. At this point in execution, the module object does not yet exist, so it cannot be passed to the dialect. The method `atModuleEnd` is requested immediately after the code of the module completes, and is passed a reference to the module object itself. The dialect can use this reference in the same way as any other object, including storing it for future use, requesting methods on it, and passing it to other methods.

4 Case Studies of Dialects

To illustrate the power of our design, this section presents six case studies of dialects and their implementations. Further case studies are reported in the first author's thesis [13]. Sample code for the case studies is included in the downloadable implementation and artifact, and is also accessible (and runnable) in the web-browser-based implementation.

4.1 Logo-Like Turtle Graphics

Our first case study is a simple dialect that supports procedural turtle graphics, inspired by Logo. This dialect is designed to be used by beginning students to learn geometry and basic control structures with as little overhead as possible — in particular, without



```
dialect "jlogo"

method sycamore (size : Number) -> Done {
    if (size < 12) then {return done}
    forward(size)
    right(25); sycamore(size*0.7)
    left(25); sycamore(size*0.6)
    left(28); sycamore(size*0.7)
    right(28); back(size)
  }

method tree (size : Number, angle : Number, stage : Number) ->
    if (stage <= 0) then {return done}
    forward(size); right(angle)
    tree(size*(0.5+0.04*random(10)), angle, stage-1)
    left(2*angle)
    tree(size*(0.5+0.04*random(10)), angle, stage-1)
    right(angle); back(size)
  }

jump(250,350); sycamore(100)
jump(350,370); tree(60, 20, 8)
jump(200,340); tree(40, 25, 8)
jump(150,350); tree(55, 27, 10)
```

Fig. 2. A simple program in our Logo-like dialect and its output.

the syntactic and semantic overhead of a more object-oriented style. We define a dialect giving access to simple movement primitives and presenting what amounts to a procedural language. Figure 2 shows a simple program in this dialect, and its output, in the web implementation.

This dialect is straightforward to implement. First, variables to hold the turtle's state can be declared at the top level of the dialect module:

```
var x : Number := 250
var y : Number := 250
var heading : Number := 270
var nib : Boolean := true
```

Then, the commands to move the turtle and draw can be written straightforwardly as Grace methods, e.g:

```
import "simplegraphics" as sg
...
method left(deg : Number) −> Done { heading := (heading − deg) % 360 }
method right(deg : Number) −> Done { heading := (heading + deg) % 360 }
method forward(n : Number) {
  def nx = x + math.cos(heading / 180 ∗ π) ∗ n
  def ny = y + math.sin(heading / 180 ∗ π) ∗ n
  if (nib) then {sg.drawLineFrom (x,y) to (nx,ny) in (ink)}
  x := nx; y := ny
}
```

This is basically the same way turtle graphics would be implemented in any procedural or scripting language. Note that the drawLineFrom()to()in() method is requested on the sg object — this delegates drawing to Grace's "simple graphics" library.

Grace's support for blocks also allows us to implement new control structures as methods that take blocks as arguments. For example, the dialect can provide a Logo-style repeat loop as a method that declares a counter variable and delegates to Grace's while()do() loop.

```
method repeat (n : Number) times (b : Block) −> Done {
  var counter := 1
  while {counter <= n} do {
     b.apply
   counter := counter + 1 } }
```

4.2 Design by Contract

Courses taking a formal approach to software engineering may wish to teach programming disciplines such as Design by Contract, using pre- and post-conditions, and loop variants and invariants, as in Eiffel [19]. A dialect can provide these facilities in Grace. Our approach here is reminiscent of Scala, but based on dialects rather than traits [20].

The simplest support is for assertions — for example, asserting that the arrays used to store keys and values in a hash table have the same size:

```
assert {hashTable.keyArray.size == hashTable.valueArray.size}
```

This assert "statement" is defined in a dialect as a method that accepts a Predicate (a parameterless block that returns a Boolean when evaluated). If the value of the predicate is false, the assertion has failed, so we raise an appropriate exception:

```
method assert( condition : Predicate ) {
    if ( ! condition.apply) then { InvariantFailure.raise }
}
```

We can extend this technique to support pre- and post-conditions on methods, inspired by Eiffel's "require", "do", and "ensure" clauses:

```
method setHours ( hours' : Number ) {
     require { (0 <= hours') && (hours' <= 23) }
         do { hours := hours'; hours }
         ensure { result −> (result == hours') && (hours == hours') }
}
```

The identifier result in the ensure clause refers to the value returned by the method.

This construct can be defined straightforwardly in a dialect, using multi-part method names for the syntax. As in Eiffel, pre- and post-conditions are checked dynamically.

```
method require( precondition : Predicate )
         do ( body : Block )
         ensure ( postcondition : Predicate ) {
    if ( ! precondition.apply )
        then { InvariantFailure.raise "Precondition Failure" }
    var result
    try { result := body.apply }
        catch { _ −> InvariantFailure.raise "Unexpected Exception" }
        finally {
            if ( ! postcondition.apply(result) )
                then { InvariantFailure.raise "Postcondition Failure" }
        }
    return result
}
```

Going still further towards Eiffel, we can add support to the dialect for specifying and checking loop variants and invariants:

```
loop {
    print(letters[i])
    i := i+1
  }
  invariant { i <= (letters.size + 1) }
  until { i > letters.size }
  variant { letters.size − i + 1 }
```

Once again, expressions defining variants and invariants, as well as the code for the loop body and the termination condition, are supplied as blocks, which are evaluated as required by the implementation of the loop()invariant()... method.

4.3 Dialect for Writing Dialects

Programmers writing different dialects tend to have similar needs. In particular, writing a checker requires inspecting the user's code and determining whether or not it is acceptable; the *form* of this inspection will be the same in many dialects. We have abstracted these repeated tasks into a dialect of their own. Our **dialect** dialect makes it easy to declare rules to test different parts of the source code and to report errors; these rules are used in the static dialect described in Sect. 4.4. The dialect dialect can also maintain state; we demonstrate how this is used for type checking in Sect. 4.5. The dialect dialect hides the details of the checking process and allows programmers to write dialect definitions that are largely declarative.

Fundamentally, Grace checkers are methods that examine the nodes of the program's abstract syntax tree at compile time. A checker either accepts a node, or raises an exception to report an error. The AST nodes support the Visitor Pattern to assist in this examination. Although quite efficient, this kind of code is too low-level to be written by most instructors, who may nevertheless need to write dialects for use in their teaching.

The dialect for writing dialects simplifies the process of writing a checker by implementing a generic visitor that applies *rules* defined by the dialect-writer. The dialect maintains a list of rules to apply in a module-level object rules. The rule method takes an ASTBlock (a block that accepts an AST node) as an argument, and adds it to the list of rules.

```
method rule(block : ASTBlock) −> Done {
  rules.push(block)
}
```

More complex kinds of rule, such as when()error() rules, are defined in terms of the basic rule method:

```
method when(pred : UnaryPredicate) error(msg : String) {
  rule { node −>
    def matches = pred.match(node)
    if (matches.andAlso {matches.result}) then { fail(msg) }
  }
}
```

The first argument to when()error() is a UnaryPredicate, that is, a block that takes a single argument and returns a Boolean. The body of the method declares a primitive rule that accepts an AST node, and then applies pred as a partial function to that node. If the function is applicable, *and* the result of invoking the the predicate is true, then an error is raised using fail.

The dialect dialect defines a single visitor over the AST, which runs all the rules over every node:

```
method visitDefDec(node) −> Boolean {
    runRules(node)
}

method visitVarDec(node) −> Boolean {
    runRules(node)
}
```

It is sometimes useful to examine a node from a perspective that is different from the way that the AST is defined. For example, parameters appear within method, block, and class definition nodes, but the dialect-writer may wish to treat them all in the same way. To simplify the matching of all parameters, regardless of their location in the AST, the dialect constructs special parameter nodes against which to run the rules:

```
method visitMethod(node) −> Boolean {
    runRules(node)

    for(node.signature) do { part −>
        for(part.params) do { param −>
            runRules(aParameter.fromNode(param))
        }
    }

    for(node.body) do { stmt −>
        stmt.accept(self)
    }

    return false
}
```

The dialect also defines a pattern Parameter to match these nodes; this allows the dialect author to write a rule against all parameters, rather than having to write separate rules to deal with each place in which a parameter may appear. The static dialect in Sect. 4.4 uses this pattern to ensure that all parameters are annotated with types. Similarly, specialised patterns While and For match while and for loops, common cases that a dialect may want to examine. A dialect author can easily create similar patterns for their own constructs using the aRequestPattern.forName(...) method provided by the dialect dialect.

The pattern-matching approach trades off some efficiency for ease of programming, but efficiency is not a primary goal of Grace. Moreover, we expect most programs, especially in beginner dialects (which are likely to have the most additional checks) to be quite small. A declarative approach allows checkers to be expressed concisely, and to be understood without a deep understanding of the whole of the implementation.

```
import "ast" as ast
def CheckerFailure = Exception.refine "CheckerFailure"
def staticVisitor = object {
  inherits ast.baseVisitor
  method visitDefDec(v) {
    if (v.decType.value == "Unknown") then {
      CheckerFailure.raiseWith("no type on '{v.name.value}'", v.name)
  } }
  method visitMethod(v) {
    for (v.signature) do {s−>
      for (s.params) do {p−>
        if (p.decType.value == "Unknown") then {
          CheckerFailure.raiseWith("no type on '{p.value}'", p)
    } } }
    if (v.returnType.value == "Unknown") then {
      CheckerFailure.raiseWith("no return type on '{v.value.value}'", v.
        value)
  } }
}
method checker(code : List<ASTNode>) {
    for (code) do {n −> n.accept(staticVisitor) }
}
```

Fig. 3. Requiring static types implemented as a Visitor. Similar code for **var** declarations and blocks is omitted for space.

4.4 Requiring Explicit Type Annotations

An instructor can require that, for all or part of a course, all student code is fully annotated with types, so that no dynamically-typed code is permitted. The static dialect allows access to all of the ordinary language features, while reporting compile-time errors to students who omit the types on their declarations. The definition of this dialect is relatively straightforward. We can use a visitor, as shown in Fig. 3, or the dialect-writing dialect to express it more concisely:

```
dialect "dialect"
inherits StandardPrelude.methods
when { d : Def | Var −> d.decType.value == "Unknown" }
    error "declarations must have a static type"
when { m : Method −> m.returnType.value == "Unknown" }
    error "methods must have a static return type"
when { p : Parameter −> p.decType.value == "Unknown" }
    error "parameters must have a static type"
method checker(code : Code) {
    check(code)
}
```

The first two rules provide a particular error message to display, specify what kind of node they care about — **var**, **def**, and **method** declarations — and what should trigger the error message. Here, the error appears when the declaration type is Unknown (which is the type of an un-annotated declaration). The last when()error() clause matches against the Parameter pattern from the dialect dialect, which was described in Sect. 4.3 The checker method in the static dialect delegates to check from the dialect dialect; check applies all of the declarative rules we have given.

4.5 Type Checking

Because dialects can perform checks over the whole of a module, various static checks that would typically be built into the compiler can be moved into a dialect. The Minigrace compiler does not perform any compile-time type checking, instead deferring type checks until runtime. However, if a module is written in the structural dialect, the dialect will perform structural subtyping checks before the compiler generates code for the module.

```
dialect "structural"
type Foo = { bar −> String }
method takesFoo(foo : Foo) {
  print(foo.bar)
}
takesFoo("foo") // Fails: argument does not satisfy parameter type
```

Type checking is implemented by extending the **dialect** dialect with the typeOf method, which takes an AST node and executes the rule defined for it, returning the type of the execution. Rules written in the structural dialect ensure that the typing of a node is correct, and return the static type information of nodes that represent expressions.

The **dialect** mechanisms are entirely agnostic to the nature of the type information used, allowing different forms of type checking to be implemented in the same dialect. The structural dialect includes several classes for describing types, providing a basis for building and testing type information in the rules that follow. The anObjectType class provides the isSubtypeOf method, which determines if one type is a subtype of another. For instance, the type error generated above comes from the request typing rule, given in Fig. 4, that ensures that the method exists in the receiver and the parameters

```
rule { req : Request −>
  match(typeOf(req.in).getMethod(req.name))
    case { _ : NoSuchMethod −> fail "no such method" }
    case { mt : MethodType −>
      for (mt.signature) and(req.with) do { s, w −>
        for (s.params) and(w.args) do { p, a −>
          if (!typeOf(a).isSubtypeOf(p.decType)) then {
            fail "argument does not satisfy parameter type" }
      } }
      mt.returnType // A request for typeOf(req) will receive this value
} }
```

Fig. 4. Request typing rule in the structural dialect

are correctly typed before producing a type for the result of the request. The nested requests to the method for(aCollection)and(anotherCollection) do(aBinaryBlock) iterate through the signature parts and parameters, testing that each argument is a subtype of its corresponding parameter.

The extension to the **dialect** dialect also supplies tools for managing information about what variables, methods and types are available in the current scope. Rules can enter into new scopes, introduce new values, and retrieve them again with identifiers. The two rules below ensure that a block like { x : Number –> x } produces the appropriate type.

```
rule { blk : BlockLiteral –>
  scope.enter {
    for(blk.params) do { param –>
      def pType = anObjectType.fromDecType(param.decType)
      scope.variables.at(param.name) put(pType)
    }
    typeOf(blk.body.last) } }
rule { idnt : Identifier –> scope.variables.find(idnt.value) }
```

Although variables, methods, and types all inhabit the same namespace in Grace, keeping them separated in the scope management makes it easier to distinguish between run-time and compile-time information. Essentially, every type declaration introduces new type information *and* a new runtime object into the local scope bound to that information.

Structural type checking is compatible with other checkers. To complete the implementation of a fully static variant of Grace, the structural and static dialects can be combined.

```
import "static" as static
import "structural" as structural
inherits StandardPrelude.methods
method checker(code : Code) {
    static.checker(code)
    structural.checker(code)
}
```

Because two imports of the same module access the same module object, multiple checkers written in the **dialect** dialect are able to share type information with one another. This allows a checker to extend the typing of another by providing extra type information, and type checking rules that operate in tandem with the existing rules. The following dialect adds basic type inference to definitions. It uses the type information provided by structural, and adds extra information into the type environments of the shared scope object.

```
dialect "dialect"
import "structural" as structural
inherits StandardPrelude.methods
rule { d : Def –>
    if (d.decType.value == "Unknown") then {
        scope.at(d.name) put(typeOf(d.value))
}  }
method checker(code : Code) { structural.checker(code) }
```

4.6 Literal Blocks

Because control structures in Grace are simply methods with the same semantics as other parts of the language, a programmer (particularly one familiar with other languages) may make mistakes that are not syntactically invalid, but lead to errors they find difficult to understand. In particular, the condition of a while loop is a block, as it may be executed repeatedly, and so is written in braces. If the programmer writes the condition in parentheses instead, or writes some other expression in place of the condition, they will receive a type error they may find difficult to understand.

This dialect ensures that the condition of a while loop is written in braces, as a literal block, and will not permit passing a reference to a block defined elsewhere. The dialect dialect provides checking rules and a special While pattern that allows us to write the body of the dialect very briefly:

```
rule { req : While(cond, _) −>
    if (cond.kind != "block") then {
        reportWhile(req)
    }
}
method reportWhile(req) {
    // Report an explicit error to the user and suggest what they may have intended.
}
```

The reportWhile method uses the **dialect** dialect's error-reporting and the compiler's suggestions infrastructure to tell the user what they did wrong, and what they might have intended to write. In a simple case like the following, the error is reported as ranging from the first parenthesis to the last, and the user will be prompted as follows:

```
literal_test.grace[4:7-14]: Syntax error: The condition of
a while loop must be written in {}.
  3: var x := 0
  4: while (x < 10) do {
-----------^^^^^^^^
  5:     print "Counted to {x}."

Did you mean:
  4: while {x < 10} do {
```

A user interface can present this suggestion as an action to be taken, as the web-based IDE does.

5 Discussion

We considered three major alternative approaches to dialects: inheritance, delegation, and special-purpose macros. We rejected all of these in favour of the approach described here, each for a different reason.

5.1 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 through **outer** in any nested scopes. The dialect's methods could also be defined as confidential if required.

This approach was inspired by SIMULA, and envisaged in the early descriptions of Grace. As the language developed, several problems with this approach revealed themselves. Most of these problems arise because inheritance in Grace (as in most other languages) is transitive, so dialects implemented via inheritance would also be transitive. What this means is that a module that inherits from a dialect will have all of the dialect's methods available on the module object itself. For example, if a dialect were itself defined by a dialect (as in Sect. 4.3) then all the features of the dialect-defining dialect would also be included in any module that uses that dialect. For these reasons we discounted the inheritance approach.

5.2 Delegation

We also considered supporting dialects by delegation. In particular, we considered translating a dialect statement into an import statement for the dialect module, along with a set of local (re)declarations 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; the effect is similar to unqualified imports in other languages. 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)
}
```

Only public dialect methods would 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 all other code.

There were two reasons why we rejected this design. The first is that it added another mechanism — delegation — into the language. Grace already has three relationships between objects: simple references, inheritance, and lexical nesting: delegation would add a fourth. The second reason is that the proposed semantics for delegation were very similar to the existing semantics for lexical nesting. Nesting makes outer objects' methods available to the objects nested inside them, but not to those objects' clients; those methods can be involved via implicit requests, or explicitly via **outer** (rather than **self**); self-requests in the outer object go to that object, not back to the original **self**. Given these similarities, it seemed simpler overall to extend nesting to encompass dialects, rather than introduce another separate mechanism.

5.3 Macros

The third option was to add macros, an additional language mechanism, allowing a dialect to define their own syntax and semantics from scratch. This is the approach taken in Racket [25], discussed in more detail in section 6.1 below. Macros provide vastly more power than Grace's dialects: they may reorder or prevent the evaluation of arguments, introduce new bindings not mentioned in the source code, or transform the program in arbitrary ways.

For example, an SQL-style select macro in Racket could share an iteration variable across several expressions:

```
(for n (numbers)
  (where (< n 5))
  (select (* 3 n)))
```

In contrast, an equivalent form in Grace would make the sub-expressions (arguments to where and select clauses) blocks, with the value of the current number being provided as an argument to each block in turn:

```
for (numbers)
  where { n −> n < 5 }
  select { n −> n * 3 }
```

(C#'s lambdas have the same limitations as Grace's blocks, which is why C# has a built-in "macro" that re-writes its select statement into expression using multiple lambdas. [2]).

There are a number of reasons why we chose not to use macros to implement dialects in Grace. The first is that, without macros, dialects can't introduce new syntactic forms; this means that code written in a dialect remains readable without knowledge of the dialect it is using. Thus, the parse of a Grace program does not depend on dialects, types, or operator definitions: syntactically, there are only method requests. A novice can understand that control passes to a given method on a given receiver, with the arguments written in the source, without needing to understand what that method does or how it does it.

The second reason is that, without macros, Grace code that *implements* a dialect uses essentially the same language features as code that *uses* a dialect. Instructors do not have to learn a powerful new feature (macros) to write dialects, and don't have to understand a new feature to be able to debug code using dialects.

The final reason is that macros are an additional feature that have not (so far) been required in Grace. Because we want to keep Grace minimal, and hopefully easy to learn and easy to use, we didn't want to add complex and powerful additional features unless we could not find any simpler alternatives.

5.4 Local Dialects

In the current design, dialects are chosen for the whole of a module. Because dialects rely on lexical scope, an obvious extension is to permit dialects to be applied to smaller "local" lexical scopes, perhaps for the extent of a block, an object constructor, or a class. For example, we could shift into the turtle graphics dialect in the middle of a for loop to draw the bars of a histogram.

```
...
def histogram = source.getData
for (histogram) do { datum ->
  dialect "turtle" do {
    forward(datum * 10)
    right(90); forward(10); right(90)
    forward(datum * 10)
    left(90); forward(10); left(90)
  }
}
```

We have not pursued this extension for several reasons. Local dialects do not seem to be necessary to support teaching — the primary purpose of Grace dialects. Local lexically scoped dialects may indeed be useful for domain specific languages used to support modelling, such as the relationship and finite state machine dialects described in the thesis [13], but for pedagogical purposes, students will typically write a single module in a single dialect.

The interaction of dialect scoping and ordinary lexical scoping 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 programmers may want to augment the existing dialect on a temporary basis.

Pragmatically, we can generally do without lexical dialects at the cost of extra modules. The above code example could be refactored so that the body of the for loop becomes a method in a separate module that is written in the turtle dialect; the loop would then request that method from the other module.

6 Related work

6.1 Racket

Tobin-Hochstadt *et al.* [25] describe languages as libraries in Racket, a Scheme-based language with an accompanying IDE designed for teaching. Racket supports multiple language definitions through the use of avowedly *"Advanced Macrology"* [6] to translate the input source text down to core Racket, adding new functionality, or even replacing the language syntax and semantics along the way.

Racket (then DrScheme) reintroduced the concept of using multiple "language levels" for teaching [8], originally from SP/k [12]: Grace's dialects were inspired by Racket's language levels. Racket's levels are intended to be moved through in sequence with gradually increasing power: earlier 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.

Racket languages are strictly more powerful than our dialects, because Racket macros are full Scheme procedures that manipulate syntax trees. This is particularly useful when creating new defining forms, allowing their arguments to span multiple scopes.

A Racket language also has the ability to provide information to the Racket integrated development environment. This information can aid syntax highlighting and error reporting when the language has been modified. Because Grace dialects do not make such modifications, this tight coupling with the editing environment is not required: all programs are in standard Grace syntax. The dialect's checker can provide error reporting to whatever level of detail is required.

Racket also offers significant support for defining new languages from scratch. A Racket language definition can entirely replace the Racket "reader", and parse the source text itself, allowing arbitrary input. A Racket implementation of Algol-60 is included in the Racket distribution, and programs need only declare `#lang algol60` in order for the rest of the source to be treated as Algol. Our system does not support this; while a dialect may, by the combination of multi-part methods, operators, and predefined objects, present a language with a similar feel to another, programs written in that dialect must still conform to the overarching Grace syntax. This limitation is both a blessing and a curse. A programmer who already knows the other language may not be immediately at home, but working within a single consistent syntax allows integrating code from different paradigms and gradually moving from one to another.

Compared with Racket, the author of a Grace dialect does not need to embark upon full-scale metaprogramming (nor do they have the opportunity). To define a dialect without a checker, programmers define the methods, classes, variables, and types they want to have available to users exactly as they would in any other program. To provide dialect checkers, programmers need to understand the visitor pattern, or use the "dialect" dialect to write a largely declarative specification of a visitor, within Grace's standard syntax and semantics.

All Grace dialects have the same semantics as any other Grace program — method requests with arguments passed by value. Grace's parse depends only upon syntax, not on types or other implicit operations, so programmers can always determine the flow of execution from a program's surface syntax. By avoiding macros we avoid code that does not do what it appears to do: arguments are always evaluated before methods are requested, new bindings are never introduced implicitly, and parse or type errors can stem only from what was actually written in the input source code. A macro-based system cannot guarantee any of these points.

6.2 Scala

Scala [21,23] includes several features supporting domain-specific languages. The language syntax permits methods acting like built-in structures and operators with many levels of precedence and associativity. Scala implicit parameters allow an argument to be passed without naming it, determined by the type. In combination these allow domain-specific languages that are aware of the context in which they are used. Scala's treatment of syntax and semantics is determined by the static type information it has available. By contrast, Grace programs have the same semantics with or without type definitions, and Grace's syntax, while flexible, does not admit ambiguities that need to be resolved by static types.

Scala also includes powerful macro features [7,5] integrating the compiler and run-time. There is no formal "dialect" system in Scala, although similar functionality can be built using other constructs of the language. Scala mirrors have the ability to perform both run-time and compile-time reflection, and these can be used to implement domain-specific languages with similar ability to those in Racket, including the ability to defer some processing until run time, although with the same fundamental syntax. Compile-time execution in Grace dialects is limited to reading and proscribing: they cannot modify or specialise code, and the run-time behaviour of dialects is exactly Grace method execution.

6.3 Ruby

In Ruby internal domain-specific languages (DSLs) are common, supported by particular language features [9]. Two common strategies for Ruby DSLs involve using the language's open classes, and using per-instance dynamically-bound evaluation.

Open classes permit modifying third-party classes — including built-in objects — to add new methods, enabling users of the DSL to write, for example, `3.years.ago` to represent a time. These modifications are globally visible, and work only so long as they don't conflict with other modifications.

The second strategy depends on dynamically-bound block evaluation using the method `instance_eval`. This method allows one to execute a block of code inside the context of another object L as though the block were written inside L's definition, and thus with access to methods defined in L. The language syntax permits reasonably fluid code to be written in this way. Moreover, different DSLs may be used at different points by evaluating code inside different objects.

Grace's dialects are more static than Ruby's. Whereas Ruby uses dynamic metaprogramming to modify existing classes or modules, Grace uses nesting to make definitions available where they are needed; in Grace, the bindings seen by a block depend on where it is defined, and not on where it is evaluated.

6.4 Haskell

Haskell is also used to define domain specific languages [1,16]. Haskell DSLs typically use the language's type classes to embed themselves in the language. Existing functions and operators become part of the language by defining type-class instances for the language representation — whether that representation is the data the DSL consumes, or a reflexive representation of the program itself. Static type information directs which functions are actually executed for a particular expression, often based upon the calling context (i.e. the expected return type). A programmer can temporarily enter the domain of a DSL simply by declaring the return type of their function.

Static type information is crucial to the semantics of Haskell DSLs (as it is in Haskell programs generally). A semantics relying on static types is undesirable for a gradually-typed language like Grace. Haskell's available syntax is more constrained than Grace's dialects, and the scope for extension is more constrained by what already exists in the language. A Haskell DSL will have difficulty relying on some subset of the functions or operators from a Haskell type class, while Grace dialects may define exactly the methods and operators they need.

6.5 Cedalion

Cedalion [17] is a language for defining domain-specific languages. Cedalion aims to promote "language-oriented programming", a programming style in which many DSLs are used in combination, with a new language defined for each subdomain spanned by the program. 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.

All Cedalion languages are interoperable because they share the same host language. In this respect they resemble Grace dialects: within the same fundamental semantics, many different variants may coexist simultaneously. On the other hand, Cedalion uses a special "projectional editor" [26] to edit code: the abstract syntax tree is edited, rather than textual source. A Cedalion language defines a display grammar for that syntax tree, rather than a parsing grammar for text. This approach contrasts with Grace, where the same surface syntax persists in every dialect, but where the syntax itself is quite flexible. A reader of one Cedalion language has no more benefit in understanding another than an outsider, while an author in the language needs not conform to any other overriding syntax. In both cases, Cedalion takes the opposite position to Grace.

6.6 Pluggable Checkers

JavaCOP [18] is 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 [22] is a mature library that provides similar functionality to JavaCOP, 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.

7 Conclusion

The language designer should be familiar with many alternative features designed by others, and should have excellent judgment in choosing the best
— Tony Hoare,
Hints on Programming Language Design [11].

We have described how a novel combination of language features — lexically nested objects, syntax for blocks and multi-part method names, optional typing, and pluggable checkers — supports dialects in Grace. Because Grace's dialects are based on these standard language features, programmers can write dialects much as they write any other Grace program — by defining objects and methods — without having to learn additional macro systems, define lexers, parsers, and semantic rules, or use metaprogramming to modify class definitions on the fly. To illustrate the power of Grace's dialect mechanism, we have presented a number of case studies of dialects of varying complexity. These range from a Logo-style turtle graphics microworld, through an Eiffel-style design by contract dialect, to a dialect that ensures that programs are statically typed, and a dialect that helps instructors to write dialects.

A more mature implementation (compiler and IDE) will enable us to begin empirical evaluations of Grace in use in teaching. We hope to begin these evaluations in October 2014, and expect to refine Grace's design based on this experience. Much work remains to be completed with Grace in general and dialects in particular. Grace's implementation, although sufficient to host the compiler, and to support small assignments in programming-language classes, is still a proof-of-concept prototype.

Acknowledgements. We thank Matthias Felleisen and the other (anonymous) reviewers for their comments on a previous versions of this paper.

References

1. Augustsson, L., Mansell, H., Sittampalam, G.: Paradise: a two-stage DSL embedded in Haskell. In: ICFP. pp. 225–228. ICFP '08, ACM, New York, NY, USA (2008)
2. Bierman, G.M., Meijer, E., Torgersen, M.: Lost in translation: formalizing proposed extensions to C#. In: OOPSLA (2007)
3. Black, A.P., Bruce, K.B., Homer, M., Noble, J.: Grace: the absence of (inessential) difficulty. In: Onward! pp. 85–98. ACM, New York, NY, USA (2012)
4. Black, A.P., Bruce, K.B., Homer, M., Noble, J., Ruskin, A., Yannow, R.: Seeking Grace: a new object-oriented language for novices. In: SIGCSE (2013)
5. Burmako, E., Odersky, M., Vogt, C., Zeiger, S., Moors, A.: Scala macros. http://scalamacros.org (Apr 2012)
6. Culpepper, R., Tobin-Hochstadt, S., Flatt, M.: Advanced macrology and the implementation of Typed Scheme. In: ICFP workshop on Scheme and Functional Programming (2007)
7. EPFL: Environment, universes, and mirrors - Scala documentation. http://docs.scala-lang.org/overviews/reflection/environment-universes-mirrors.html (2013)
8. Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: a programming environment for Scheme. J. Funct. Program. 12(2), 159–182 (3 2002)
9. Fowler, M.: Domain Specific Languages. AW (2011)

10. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns. AW (1994)
11. Hoare, C.: Hints on programming language design. Tech. Rep. AIM-224, Stanford Artificial Intelligence Laboratory (1973)
12. Holt, R.C., Wortman, D.B.: A sequence of structured subsets of PL/I. SIGCSE Bull. 6(1), 129–132 (Jan 1974), http://doi.acm.org/10.1145/953057.810456
13. Homer, M.: Graceful Language Features and Interfaces. Ph.D. thesis, Victoria University of Wellington (2014)
14. Homer, M., Bruce, K.B., Noble, J., Black, A.P.: Modules as gradually-typed objects. In: Proceedings of the 7th Workshop on Dynamic Languages and Applications. pp. 1:1–1:8. DYLA '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2489798.2489799
15. Homer, M., Noble, J., Bruce, K.B., Black, A.P., Pearce, D.J.: Patterns as objects in Grace. In: Dynamic Language Symposium. ACM, New York, NY, USA (2012)
16. Jones, M.P.: Experience report: playing the DSL card. In: ICFP (2008)
17. Lorenz, D.H., Rosenan, B.: Cedalion: a language for language oriented programming. OOPSLA 46 (Oct 2011)
18. Markstrum, S., Marino, D., Esquivel, M., Millstein, T.D., Andreae, C., Noble, J.: JavaCOP: Declarative pluggable types for Java. ACM Trans. Program. Lang. Syst. 32(2) (2010)
19. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
20. Odersky, M.: Contracts for scala. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification. LNCS, vol. 6418, pp. 51–57. Springer, Berlin (2010)
21. Odersky, M.: The Scala language specification. Tech. rep., Programming Methods Laboratory, EPFL (2011)
22. Papi, M.M., Ali, M., Correa, Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: ISSTA (2008)
23. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In: GPCE. pp. 127–136. New York, NY, USA (2010)
24. Ross, D.T.: Uniform referents: An essential property for a software engineering language. In: Tou, J.T. (ed.) Software Engineering, vol. 1, pp. 91–101. Academic Press (1970)
25. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: PLDI (2011)
26. Voelter, M.: Embedded software development with projectional language workbenches. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MoDELS. LNCS, vol. 6395, pp. 32–46. Springer-Verlag, Berlin, Heidelberg (2010)
27. Wirth, N.: The programming language PASCAL. Acta Informatica 1(1) (1971)

A Artifact Description

Authors of the artifact. Core developer: Michael Homer. Dialect case studies: Michael Homer, Timothy Jones, James Noble.

Summary. The artifact is based on Minigrace, a prototype compiler for Grace implemented by the first author. Minigrace has been extended to include dialects, language variants that extend or restrict the language available to the programmer. The artifact includes several case studies exploring different areas of the dialect space, including both extensional and restrictive dialects.

Content. The artifact package includes:

- a version of Minigrace including the dialect system described in the paper;
- twelve case study dialects: the six described in the paper (turtle graphics, design-by-contract, dialect dialect, mandatory type annotations, structural subtyping, and requiring literal blocks) and six others;
- detailed instructions for using the artifact, for rebuilding it from scratch, and for obtaining the newest source code, provided as an `index.html` file.

To simplify experimenting with our case studies, we provide a VirtualBox disk image containing our prototype fully installed and with all case study dialects immediately available. The image contains Ubuntu 13.10, logs the user in by default, and includes the `minigrace` tool in the path with all case studies in the initial directory. All dependencies are preinstalled and the tool is ready to run.

We also include a tarball of the complete source code of the newest version of Minigrace, which includes our dialect changes. Minigrace compiles to both C and JavaScript, and some dialects function only on one backend or the other; to that end, we include a fully set-up version of the JavaScript backend including all case studies and instructions for accessing it.

Getting the artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink. The latest version of our code is available from the Grace language website, http://gracelang.org.

Tested platforms. The virtual machine is known to work on any platform running VirtualBox version 4 with at least 8 GB or free space on disk and at least 1 GB of free space in RAM. Minigrace is known to work on most POSIX-compatible systems, including Linux and Mac OS X. Installation instructions are included in the source tarballs. The JavaScript interface of Minigrace is known to work on all current major desktop browsers, including Firefox, Chrome, Safari, and Internet Explorer.

License. GPL 3 or later (https://www.gnu.org/licenses/gpl-3.0.html)

MD5 sum of the artifact. 1995f3ef018c83de31dfe445c9cafd4b

Size of the artifact. 1489950046 bytes (1.4 GB)