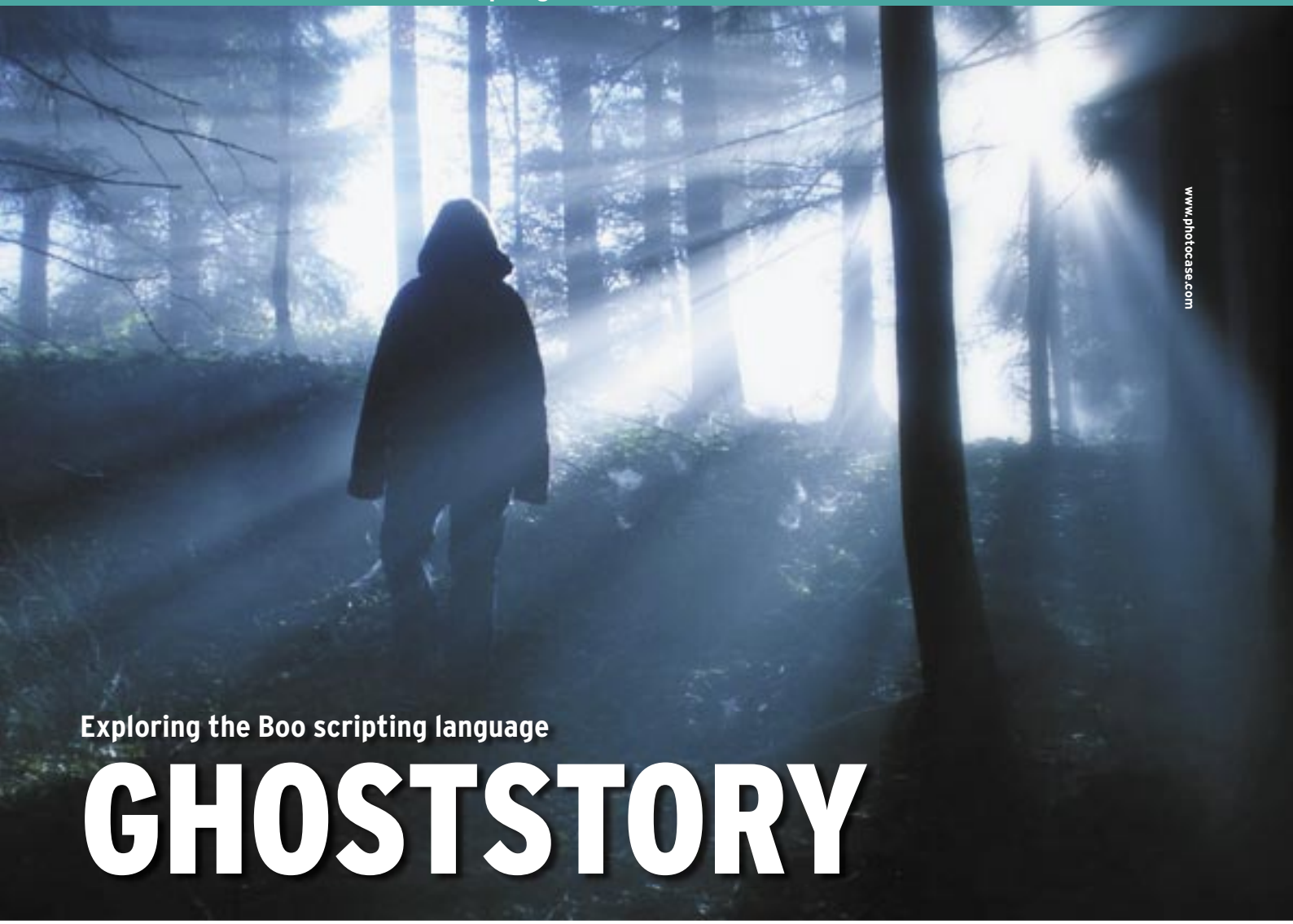

Exploring the Boo scripting language

GHOSTSTORY

Boo is a scripting language tailor-made for Mono and .NET. This haunting mixture of Python and C# may be just what you need to get started with the .NET framework. **BY TIM SCHÜRMANN**

Rodrigo Barreto de Oliveira was frustrated; he couldn't find a programming language that was suitable for his next project. Python didn't have the kind of static type checking he needed, and he would have preferred better .NET integration. C# was well integrated with .NET, but it required too much typing. These disappointments led Rodrigo to develop his own scripting language. His new language would be Python-based, and it would rely on the Common Language Infrastructure (or CLI) and the Dotnet framework. He spiced the results with a couple of C# and Ruby design principles and finally came up with the object-oriented Boo [1] language, which is available for Linux thanks to the Mono environment.

Saying Hello

Because the ubiquitous "Hello World" with its *print("Hello World")* line is almost too trivial, Listing 1 gives you a GTK# variant on the theme.

Boo doesn't simply enclose contiguous blocks of code in curly brackets, but uses indenting, just like Python. In fact, you aren't even allowed to type *end* to finish a block. Boo does away with semicolons at the end of every line, and you should only use them to resolve ambiguity. On the other hand, Boo does support both Python comments introduced by a pound sign, #, and typical C and C++ variants with /* and //.

Variables

In contrast to Python, Boo uses static typing; that is, you are required to state a variable's type before using the variable for the first time. One advantage of this approach is that the compiler will detect incorrect assignments at build time and complain about them. To avoid the need for programmers to juggle with data types, the clever Boo compiler automatically derives the correct type in a process known as type inference:

```
a as int // a is an integer
a=3
```

Listing 1: Hello GTK#

```
01 import Gtk
02
03 Application.Init()
04 window = Window("Hello World",
   DefaultWidth: 300,
   DefaultHeight: 200)
05
06 window.DeleteEvent += def():
07     Application.Quit ()
08
09 window.ShowAll()
10 Application.Run()
```

Figure 1: The Boo project homepage provides code snippets and a useful introduction to the Boo language.

```
b=a
print b
```

It is quite clear that *b* must be a number. In fact, the first line in this example is redundant, as the compiler has all the information it needs from the *3*. Programmers only need to declare variables explicitly where inference will not work. Lists, arrays, and hash tables are all easy to handle. To fill a list, *a*, with the numbers 1 through 5, you either implicitly create a list object:

```
a = List(range(1,5))
```

Or use the shorthand notation:

```
a= [1,2,3,4, 'five']
print(a[:3])
```

range() is an internal function that supplies values from a given range. As the example demonstrates, you can even use mixed types in a list. In Boo-speak, the shorthand in the *print* command is referred to as a slicing operation; in this case, it returns the first three values from *a*. As an additional goody, Boo supports regular expressions, including Perl's *Match* operator (=~):

```
exampletext = "This is a test"
if exampletext =~ @/te.t/:
    print("contained")
```

This example searches *exampletext* for a string of *test*, where the *s* between the *e* and the *t* could be any other letter. The regular expression is escaped in *@/.../*.

Functions

Python fans should feel at home with function definitions:

```
def Hello(name as string):
    return "Your name: ${name}"

print Hello("Joe")
```

Boo handles functions as first class objects, a concept that originated with functional programming languages. The functions support anything the programming language supports, so you can drop them into variables or pass them in to other functions as arguments. And, of course, you can return them as results:

```
def function1(item as int):
    return item // do nothing
def function2():
    return function1
```

Closed Today

Boo will even let you define a function within another function:

```
def outer():
   inner = def():
      print("inner has ➥
      been called")
   inner()

outer()
```

This technique is just a short step away from closures. Again, Boo lifts this concept from functional programming. A closure, block, or function terminator is a piece of code or a function stem that local variables in the enclosing function can see and use.

Boo supports two different syntactical forms: block-based syntax, as in the example we have just seen, or brace-based syntax, as shown in Listing 2.

In this case, the closures are surrounded by braces, and their stems use the *a* variable from the enclosing function. You can even call closures if the environment, and thus the variable, have actually ceased to exist.

Spiced

Now, if Boo supports the concept of returning functions, this means that brain taxing expressions like the following are possible:

```
power = { x as int | return ➥
{ y as int | return x ** y }} ➥
// ** represents a power
```

Listing 2: Block Syntax

```
01 def function():
02      a = 0 # new variable
03      inc = { a+=1 }
04      show = {return a}
05      return inc, show
06
07 i,s=function()
08 print(s())
09 i()
10 print(s())
```

power expects a closure function which, in turn, returns a new function. However, the function this churns out also expects some input. You could actually call this duo like this:

```
power(5)(2)
```

As you can see, the *5* is first passed into the *power* function which, in turn, returns a function (the function { *y as int* | *return 5 ** y* } to be precise). The *2* is then passed in to the new function, which returns the result. The bottom line is that, instead of having a single function with two arguments, we now have two functions with one argument apiece. The software's architects refer to this as currying.

Generators

Besides closures, Boo also supports generators: functions that produce a series of results. However, they do not simply generate and return a complete list. Instead, the next element in the list is not calculated unless it is actually needed. Boo refers to these functions as generator methods (Listing 3).

The *MyGenerator* generator method runs until the next *yield*, then it outputs the value of the variable it finds there and stops. When the calling function (*List* in our example) needs the next value, the generator picks up exactly where it left off. As the next value is only generated if needed, this approach needs less memory under ideal conditions – after all, a single number will occupy less space than a complete list. This is particularly apparent in iteration controls, such as in for loops. Besides generator methods, Boo also has generator expressions, which work exactly like their functional counterparts, but comprise a single *for* loop. This removes the need for an explicit function definition,

Listing 3: Generators

```
01 def MyGenerator():
02     i = 1
03     yield i
04     for x in range(10):
05         i += 2
06         yield i
07
08 print(List(MyGenerator()))
```

Compilation

You have three options for running Boo scripts. Whichever approach you choose, you will need to install the Mono environment and unpack the Boo package from [1] before you can start. First of all, you can use the compiler, *booc.exe*, to compile a script:

```
mono bin/booc.exe -out:➥
hello.exe hello.boo
```

The binary built from the *hello.boo* script can then be run by giving *mono hello.exe* command. The *-r* option will include any additional DLLs you need:

```
mono bin\booc.exe -r:➥
gtk-sharp -out:hallogtk.exe ➥
hallogtk.boo
```

When you distribute programs like this, you just have to include the *Boo.Lang.dll* library from the *bin* directory with the distribution package.

Besides the Boo compiler, the package also includes the *booi.exe* interpreter, which can run Boo scripts directly, as in *mono bin\booi.exe hallo.boo*. In addition to this, there is *booish.exe*, an interactive shell that supports direct command input in a style similar to Python.

Finally, you can use the Boo API to run Boo code. The Dotnet objects the API provides will even let you run one Boo script from within another (Listing 5).

providing useful support for compact definitions of quantities such as "all uneven numbers between 1 and 10 subsequently multiplied by 2":

```
uneventimestwo = i*2 for i ➥
in range(10) if i%2
for x in uneventimestwo :
        print x
```

Again, Boo only produces the next number from the hat after testing for the terminating condition in the second *for* loop. Putting together a list of generator expressions gives you a list generator, a fast way of filling a list with selected objects:

```
uneventimestwolist = ➥
[i*2 for i in range(10) if i%2]
```

Classes

In contrast to C#, Boo can completely do without classes if necessary. You can use imperative programming or opt for functions, like in C, however, Boo is completely object-oriented under the hood. You can regard any function as an object. Saying *print("Hello World")* is the same as saying *print.Invoke("Hello World")*. And replacing *.Invoke* with *.BeginInvoke*, launches the function stem in a thread of its own:

```
def Calculation():
    for x in range(10):
        longcalculation(x)

Calculation.BeginInvoke()
```

This will bring tears to many a C programmer's eyes. Multithreading has never been so easy.

Attributes and Fields

In an object oriented language, every object has its own methods and attributes. To access attributes, programmers typically need to provide a method to set and get the attribute values. This frequently recurring task can be a nuisance, so Boo adopted the idea of properties from C#. Boo distinguishes between fields and properties, where fields are legacy variables that implement the attributes of an object. Properties are an interesting syntactical alternative to set and get methods (Listing 4).

Access to the *Color* is handled intuitively by a simple assignment. Behind the scenes, this example actually calls the get method and stores the passed value in the *_color* field. All of this is hidden from the object user. To save typing, Boo even has shorthand:

```
class Chair:
    [Property(Color)]
    _color as string
```

Listing 4: Properties

```
01 class Chair:
02     Color as string:
03         get:
04             return _color
05         set:
06             _color = value
07
08     _color as string
09
10 woodenchair = Chair()
11 woodenchair.Color = 'Brown'
```

Boo automatically generates the required get and set methods in the background.

Duck Typing

Boo lifted the concept of duck typing from Ruby. Duck typing involves disabling static type checking for a variable, and thus gaining the power to assign arbitrary objects to the variable:

```
d as duck // d can be anything➥
 from here on in
d = 5 // d is an integer ➥
from here...
d = "Hello" // ... and a ➥
string from here
```

So if it quacks like a duck, looks like a duck, and walks like a duck, you can treat it like a duck.

Extensible

Boo was designed as an extensible language. For example, you can add custom macros. Boo comes with a handful of useful helpers, such as *assert*, which checks a condition and throws an exception if the test is negative:

```
assert 1>5, ➥
"One is not bigger than 5"
```

Hiding behind the macro is a simple Dotnet object that follows a specific signature, so Boo doesn't care what language the macro is written in. But there is more to extensibility: the Boo compiler was designed as a modular pipeline. If you need to, you can latch into the build process and perform actions of your own.

Conclusions

Boo provides the CLI with an easy-to-learn scripting language that turns out pleasingly quick binaries. This makes Boo suitable for both prototype development and Dotnet and Mono integration. The syntactical proximity to Python also makes life easy on programmers thinking of a change. Support for Dotnet 2.0 is currently in progress. ■

Listing 5: Using the API to Run Boo Code

```
01 import Boo.Lang.Compiler
02 import Boo.Lang.Compiler.IO
03 import Boo.Lang.Compiler.Pipelines
04
05 compiler = BooCompiler()
06 compiler.Parameters.Input.Add(StringInput("<script>",
   "print('Hello!')"))
07 compiler.Parameters.Pipeline = Run()
08 compiler.Run()
```

INFO

[1] Boo homepage:
http://boo.codehaus.org