Type Inference

Boo is a statically typed language.

Static typing is about the ability to type check a program for type correctness.

Static typing is about being able to deliver better runtime performance.

Static typing is not about putting the burden of declaring types on the programmer as most mainstream languages do.

The mechanism that frees the programmer from having to babysit the compiler is called **type inference**.

Type inference means you don't have to worry about declaring types everywhere just to make the compiler happy. Type inference means you can be productive without giving up the safety net of the type system nor sacrificing performance.

Boo's type inference kicks in for newly declared variables and fields, properties, arrays, for statement variables, overriden methods, method return types and generators.

Variables

Assignments can be used to introduce new variables in the current scope. The type for the new variable will be inferred from the expression on the right.

```
s1 = "foo" # declare new variable s1
s2 = s1.Replace("f", "b") # s1 is a string so Replace is cool
```

Only the first assignment to a variable is taken into account by the type inference mechanism.
The following program is illegal:

```
s = "I'm a string" # s is bound with type string
s = 42 # and although 42 is a really cool number s can only hold strings
```

Fields

```
class Customer:

    _name = ""
```

Declare the new field _name and initialize it with an empty string. The type of the field will be string.

Properties

When a property does not declare a type it is inferred from its getter.

```
class BigBrain:
      Answer:
           get: return 42
```

In this case the type of the Answer property will be inferred as int.

Arrays

The type of an array is inferred as the least generic type that could safely hold all of its enclosing elements.

```
a = (1, 2) # a is of type (int)

b = (1L, 2) # b is of type (long)

c = ("foo", 2) # c is of type (object)
```

For statement variables

```
names = (" John ",
         " Eric",
         "  Graham",
         "TerryG  ",
         " TerryJ",
         " Michael")

for name in names:
    # name is typed string since we are iterating a string array
    print name.Trim()  # Trim is cool, name is a string
```

This works even when with **unpacking**:

```
a = ( (1, 2), (3, 4) )

for i, j in a:
     print i+j  # + is cool since i and j are typed int
```

Overriden methods

When overriding a method, it is not necessary to declare its return type since it can be safely inferred from its super method.

```
class Customer:

    override def ToString():
         pass
```

Method return types

The return type of a method will the most generic type among the types of the expressions used in return statements.

```
def spam():
    return "spam!"

print spam()*3
# multiply operator is cool since spam() is inferred to return a string
# and strings can be multiplied by integer values
```

```
def ltuae(returnString as bool):
    return "42" if returnString
    return 42

# ltuae is inferred to return object
print ltuae(false)*3  # ERROR! don't know the meaning of the * operator
```

When a method does not declare a return type and includes no return statements it will be typed System.Void.

Generators

```
g = i*2 for i in range(3)

# g is inferred to generate ints

for i in g:
    print i*2  # * operator is cool since i is inferred to be int

# works with arrays too
a = array(g) # a is inferred to be (int) since g delivers ints

print a[0] + a[-1]  # int sum
```

⚠️ A Word of Caution About Interfaces

When implementing interfaces it's important to explicitliy declare the signature of a method, property or event. The compiler will look only for exact matches.

In the example below the class will be considered abstract since it does not provide an implementation with the correct signature:

```
namespace AllThroughTheDay

interface IMeMineIMeMineIMeMine:

 def AllThroughTheNight(iMeMine, iMeMine2, iMeMine3 as int)

class EvenThoseTears(IMeMineIMeMineIMeMine):

 def AllThroughTheNight(iMeMine, iMeMine2, iMeMine3):
  pass

e = EvenThoseTears()
```

Ok. So where do I have to declare types then?

Let's say when.

- when the compiler as it exists today can't do it for you. Ex: parameter types, recursive and mutually recursive method/property/field definitions, return for abstract and interface methods, for untyped containers, properties with a only set defined

```
abstract def Method(param /* as object */, i as int) as string:
    pass

def fat([required(value >= 0)] value as int) as int:
    return 1 if value < 2
    return value*fat(value-1)

for i as int in [1, 2, 3]: # list is not typed
    print i*2
```

- when you don't want to express what the compiler thinks you do:

```
def foo() as object:  # I want the return type to be object not string
    # a common scenario is interface implementation
    return "a string"

if bar:
    a = 3 # a will be typed int
else:
    a = "42" # uh, oh
```

- when you want to access a member not exposed by the type assigned to an expression:

```
f as System.IDisposable = foo()
f.Dispose()
```

- when you want to use Duck Typing:

```
def CreateInstance(progid):
    type = System.Type.GetTypeFromProgID(progid)
    return type()
```

-
```
ie as duck = CreateInstance("InternetExplorer.Application")
ie.Visible = true
ie.Navigate("http://boo.codehaus.org/Type+Inference")
```