The Design of Monty: a Programming/Scripting Language

David A. Watt [1]

Department of Computing Science
University of Glasgow
Glasgow, Scotland, UK

Abstract

This paper describes the design of Monty, a language intended to be equally suitable for both scripting and conventional programming. Monty features an unusually flexible type system in which all values are viewed as objects in a single-inheritance class hierarchy, static and dynamic typing are smoothly integrated, and both non-variant and covariant generic classes are supported. An interesting byproduct of the design of Monty has been the light it has shed on the power of mutability as a linguistic concept. Among other things, it turns out that the type-soundness of a covariant generic class is closely related to the class's mutability.

Key words: Programming language, scripting language, static typing, dynamic typing, single inheritance, mutability, inclusion polymorphism, parametric polymorphism, generic class, covariant type parameterization.

1 Introduction

Programming languages and scripting languages have followed very different historical development paths. Indeed, some authors [4,15] have argued that programs and scripts are fundamentally different:

- Scripts are very high-level, while programs are relatively low-level.
- Scripts are dynamically typed, while programs are statically typed.
- Scripts are fast to develop (with concise syntax and a lightweight edit–run cycle), while programs are slow to develop (with verbose syntax and a heavyweight edit–compile–link–run cycle).
- Scripts are slow-running (being interpreted and containing dynamic type checks), while programs are fast-running (being compiled into native code).

[1] Email: `daw@dcs.gla.ac.uk`

In reality, the boundary between programming and scripting languages is blurred:

- Programming languages range from low-level (C) to high-level (Haskell, Prolog).
- Scripting languages are very high-level only in specialized areas, such as string pattern matching.
- Some scripting languages support a measure of static typing. Some programming languages are dynamically typed (Lisp, Prolog, Smalltalk).
- While scripts are not *explicitly* compiled, they are usually (Perl, Python) compiled behind the scenes. Some programming language IDEs have compile-link-and-run commands that achieve the same effect.
- Not only most scripting languages (Perl, Python) but also some programming languages (Prolog, Java) are compiled into interpretive code.

Python [17] is a language that comes close to straddling the boundary between programming and scripting. Among its features are built-in tuples, arrays, lists, and dictionaries; string pattern matching using full regular-expression notation; iteration over tuples and lists; exceptions; list comprehensions; first-class functions; and a (crude) object model. These features betray the influence of languages as diverse as Perl and Miranda. Python has a clean concise syntax, and very good feature integration. Its core language is small but powerful, and its rich module library provides very high-level functionality such as string pattern matching.

To illustrate scripting, let us consider the following easily-stated problem. An HTML document contains headings at different levels, e.g.:

```
<H1>Scripting Languages</H1>
...
<H2>Perl</H2>
...
<H2>Python</H2>
...
```

It is required to generate a "table of contents", in which each heading's title is indented according to the heading's level:

```
Scripting Languages
   Perl
   Python
```

Fig. 1 shows a Python script that solves this problem. This script is extraordinarily concise and readable. It contains no type information at all, since Python is dynamically typed.

By contrast, a Java program to solve the same problem would be rather less concise. It would contain a lot of type information, some of it redundant. It would be rather clumsy in its string pattern matching (even using the `Pattern`

```
def print_contents (filename) :
   file = open(filename)
   html = file.read()
   pattern = re.compile("<(H[1-9])>(.*?)</\\1>")
   headings = pattern.findall(html)
   for (tag, title) in headings :
      level = eval(tag[1])
      print "\t" * (level-1) + title
```

Fig. 1. "Table of contents" script in Python

library class). Finally, it would need auxiliary methods.

Despite these differences, it is clear that programming and scripting languages are converging in most respects. Modern scripting languages such as Python are absorbing programming language principles such as readable syntax, rich control structures, rich data types, data abstraction, and rich libraries. On the other hand, scripting languages still tend to favor dynamic typing and implicit variable declarations.

Convergence between programming and scripting languages is desirable because it combines the advantages of both. Maintainability is important for (non-throwaway) scripts as well as programs. Rapid development is desirable for programs as well as scripts. Very high-level functionality is important for both programs and scripts, although it need not necessarily be built into the core language.

The main obstacle to complete convergence seems to be the issue of static *vs* dynamic typing. Static typing enables compilers to certify absence of type errors and to generate efficient object code. Dynamic typing is costly in run-time checks. However, dynamic typing provides flexibility that is essential in many scripting applications. Scripts must be able to process heterogeneous data, often derived from web forms or databases.

There have been few attempts explicitly to combine static and dynamic typing in a single programming language. Programming language designers tend to choose one or the other (nearly always static typing), and to impose their preference on all programmers.

But consider an object-oriented language with a single-inheritance class hierarchy:

- If a variable's declared type is a class C with no subclasses, the variable will always contain an object of that class C.[2] These objects may be provided with class tags, but these tags need never be inspected.
- If the variable's declared type is a class C with one or more subclasses, the variable may contain an object of class C or any of its subclasses. These objects *must* be provided with class tags, and these tags are tested by certain

[2] More precisely, the variable will contain a *reference* to an object.

operations to avoid run-time type errors.

- If the variable's declared type is the class `Object` at the top of the hierarchy, then the variable may contain an object of *any* class.

Thus object-oriented languages actually support a measure of dynamic typing. In fact, their type system combines static and dynamic typing rather smoothly. The programmer is able to choose the degree of dynamic typing, by locating each individual variable at a particular point in the class hierarchy. Thus static and dynamic typing need not be mutually exclusive: they can be seen as opposite ends of a spectrum.

If a language integrates static and dynamic typing, the compiler can still type-check each part of the program, but now there are three possible answers: *well-typed* (will not fail with a type error), *ill-typed* (will fail with a type error), or *dynamic-typed* (might fail with a type error, depending on the run-time state).

The rest of this paper is structured as follows. Section 2 is an overview of the Monty language. Section 3 introduces mutability as a linguistic concept. Section 4 shows how Monty supports generic classes. Section 5 studies Monty's type system, focusing on generic classes. Section 6 briefly considers key implementation problems for Monty, namely data representation, static *vs* dynamic typing, and implementation of generic classes. Section 7 is an overview of related work. Section 8 concludes by sketching the possible future evolution of Monty.

2 Overview of Monty

If a single language is to be suitable for both scripting and conventional programming, it should meet the following requirements:

- The language should have a concise (but *not* cryptic) syntax. This is to facilitate both rapid development and maintenance of code.
- The language should have a simple and uniform semantics. This is to facilitate learning – particularly important for inexperienced script-writers.
- The language should smoothly integrate static and dynamic typing. This is to enable programmers to choose between the security and efficiency of static typing and the flexibility of dynamic typing.
- The language should have a rich easy-to-use library. This is to support the very high-level functionality, such as string pattern matching, that is so useful in scripting applications.
- The language should enable the programmer to choose either fast compilation into portable interpretive code or (possibly optimized) compilation into fast native code. The former is to support rapid development of programs and scripts, the latter to support development of efficient application programs.

```
proc printContents (filename: String) :
   val file := IO.open(filename)
   val html := file.readAll()
   val pattern := Pattern("<(H[19])>(.*?)</\\1>")
   val headings := pattern.findAll(html)
   for hdg in headings :
      val tag := hdg[0]
      val title := hdg[1]
      val level := Int.parse(tag.charAt(1))
      IO.print("\t" * (level  1) + title)
```

Fig. 2. "Table of contents" script in Monty

```
func least (vals) :
   var min := vals.get(0)
   for i in 1 .. vals.size() - 1 :
   val elem := vals.get(i)
      if elem < min :
         min := elem
   return min
```

Fig. 3. Dynamically typed function in Monty

The programming/scripting language Monty [3] has been designed to meet these exacting requirements. It has concise but readable syntax. Declarations are compulsory, but the types of variables, parameters, and functions may be omitted (their default type being `Object`). Monty provides a rich variety of object types, including collections. It provides a rich set of control structures, including iteration over collections. It will have a rich class library, supporting very high-level functionality such as string pattern matching.

Fig. 2 shows a Monty script that solves the "table of contents" problem of Section 1. The expression "`Pattern(...)`" constructs a `Pattern` object from a regular expression. The expression "`pattern.findAll(html)`" uses that `Pattern` object to compute all substrings of `html` that match the regular expression; it yields a list of arrays of strings, where each array in the list contains a tag (e.g., "H1") and the corresponding title (e.g., "Scripting Languages"). The loop "`for hdg in headings : ...`" iterates over the list. This Monty script is slightly less concise than Python (Fig. 1); just as concise as Perl (but much more readable!); and much more concise than Java.

Monty has been most strongly influenced by Java and Python. Monty's syntactic style is similar to Python's: the extent of a loop, method, or class body is defined by indentation. The syntax of Monty is summarized in the appendix.

[3] Monty is named after *Monty Python*, not *The Full Monty*!

```
immutable class Date :
   private val y: Int
   private val m: Int
   private val d: Int

   public cons (y: Int, m: Int, d: Int) :
      this.y := y
      this.m := m
      this.d := d

   public virtual func toString (): String :
      return ... + "/" + (this.y % 100).toString()
```

Fig. 4. `Date` class in Monty

The script of Fig. 2 contains little explicit type information. Each `val` declaration declares a read-only variable, whose type can be inferred from its initializer. Similarly, the type of the loop control variable `hdg` can be inferred from the type of the collection over which it iterates.

Monty has a single-inheritance class hierarchy, in which the top class is `Object`. *All* values are (notionally) objects, including `Bool`, `Char`, `Int`, and `Real` values. An expression like "`m+n`" is just syntactic sugar for a method call, in which the method is named "`+`", the receiver object is the value of `m`, and the argument is the value of `n`. This design keeps the language's semantics simple and uniform, and avoids a troublesome Java-like dichotomy between primitive and object types. Of course, efficient data representation is important; this will be discussed in Section 6.

Fig. 3 illustrates dynamic typing in Monty. The `least` function assumes that its parameter `vals` is an array, list, or other integer-indexed collection. More precisely, it assumes that `vals` is equipped with a `size` method, and with a `get` method that uses its `Int` argument to select an element of the collection. Moreover, the `least` function assumes that the collection's elements are equipped with a "`<`" method. The `least` function will throw a `NoSuchMethodError` exception if either of these assumptions proves to be false.

Figs. 4, 5, and 6 show examples of class declarations in Monty. In most respects these resemble Java class declarations. The keyword `val` declares an instance variable that can be initialized by a constructor but cannot subsequently be updated. The keyword `var` declares an instance variable that can be updated at any time. The keyword `cons` declares a constructor. The keyword `func` declares a method that returns a value. The keyword `proc` declares a method that returns no value.

Fig. 7 shows part of the Monty class hierarchy, including some built-in classes (`Object`, `Exception`, `String`, `Number`, `Int`, `Real`, `Array<X>`) and some

```
immutable class Person extends Object :
   private val name: String
   private val dob: Date      ~ date of birth

   public cons (name: String, dob: Date) :
      this.name := name
      this.dob := dob

   public virtual func toString (): String :
      return this.name + " DOB: " + this.dob.toString()
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
immutable class Employee extends Person :
   private val eid: Int   ~ employee id
   private val pay: Int   ~ annual pay

   public proc print () :
      ...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
class Student extends Person :
   private val sid: Int      ~ student id
   private var deg: String   ~ degree programme

   public cons (name: String, dob: Date, sid: Int) :
      ...

   public virtual func toString (): String :
      ...

   public proc enrol (deg: String) :
      this.deg := deg

   public proc print () :
      ...
```

Fig. 5. `Person` class and subclasses in Monty

declared classes.

3 Mutability

Mutability is an important practical concept in object-oriented programming. Whether a class is to be mutable strongly influences not only the design of the class but also the data representation. The Python specification [17] distinguishes between immutable values (such as primitive values and tuples thereof) and mutable values (such as arrays), and insists that the keys in a dictionary

```
class Stack :       ~ heterogeneous bounded stacks
   private var depth: Int
   private var elems: Array<Object>

   public cons () :
      this.depth := 0
      this.elems := Array<Object>(10)

   public proc push (x) :
      this.elems[this.depth] := x
      this.depth +:= 1

   public func pop () :
      this.depth -:=1
      return this.elems[this.depth]
```

Fig. 6. Heterogeneous `Stack` class in Monty

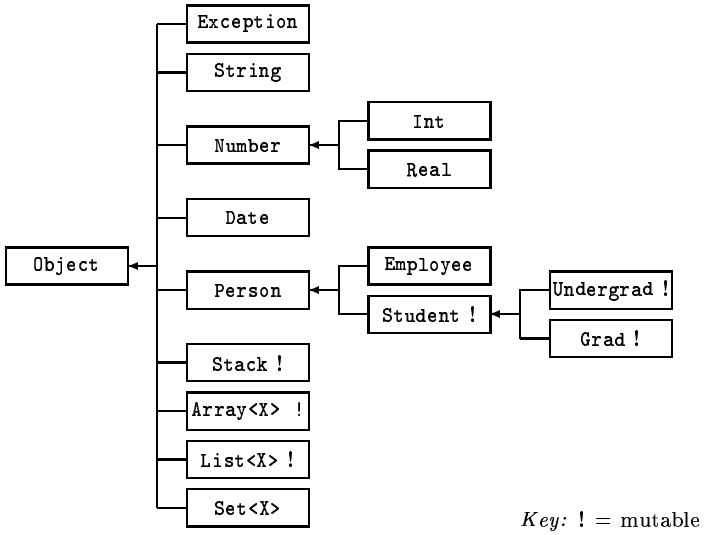

Fig. 7. Part of the Monty class hierarchy

are immutable. In the Java API specification[4], the contracts for the `Map` classes similarly insist that keys in a map are immutable. However, neither Python nor Java enforces such immutability.

In my view, mutability should be recognized as a *linguistic* concept. During the design of Monty, the usefulness of the mutability concept has cropped up repeatedly, sometimes unexpectedly.

Many of Monty's built-in classes are immutable, including `Bool`, `Char`, `Int`, `Real`, `String`, and `Object` itself. A Monty class declaration may annotate the class as `immutable`; this asserts that no object of the class can ever change

[4] java.sun.com/j2se/1.4.2/docs/api/

```
class List <X> :
   private var size: Int
   private var elems: Array<X>

   public cons (maxsize: Int) :
      this.size := 0
      this.elems := Array<X>(maxsize)

   public proc add (x: X) :
      if this.size = this.elems.size() :
         val elems2 := Array<X>(2 * this.size)
         elems2.copy(0, this.elems)
         this.elems := elems2
      this.elems[this.size] := x
      this.size +:= 1
```

Fig. 8. Generic `List<X>` class in Monty

state once constructed. [5]

The compiler should attempt to verify such a class's immutability. A sufficient condition for immutability is that the class has only `val` instance variables (which prevents update by assignment) whose own types are immutable (which prevents update by method calls). Note that mutability of a class is inherited by all its subclasses. In Fig. 7, mutable classes are marked.

The importance of mutability will become clear in Sections 5 and 6.

4 Generics

Early versions of Java did not support generics, and this was widely recognized as a serious weakness. If a program needed a homogeneous set (say) with components all of the same class, the programmer had to settle for a heterogeneous set, of class `Set`, with arbitrary objects as components. In other words, the programmer was forced to use dynamic typing, with all of its disadvantages but none of its advantages.

Java 5 has largely eliminated this weakness by introducing generic classes. Our programmer can now achieve a homogeneous set by declaring a generic class `Set<X>`, and then instantiating it as in `Set<Person>` or `Set<Date>`. Unfortunately, Java 5 generic classes can be instantiated only at object types; an instantiation at a primitive type, such as `Set<int>`, would be illegal.

Monty is required to *enable* but not to *force* the use of dynamic typing. If a program must be efficient and type-safe, it should be possible to make it completely statically typed. Therefore Monty also supports generic classes.

[5] Monty insists on *deep* immutability: an immutable object cannot contain a mutable object.

```
immutable class Set <X> :

   private ...

   public cons () :
      ...   ~ construct an empty set

   public func contains (x: X): Bool :
      ...   ~ return true iff x is a member of this set

   public func choose (): X :
      ...   ~ return an arbitrary member of this set

   public func plus (x: X): Set<X> :
      ...   ~ return the union of this set and {x}

   public func union (s: Set<X>): Set<X> :
      ...   ~ return the union of this set and s
```

Fig. 9. Generic `Set<X>` class in Monty

Examples are the built-in class `Array<X>`, and the classes `List<X>` and `Set<X>` whose declarations are outlined in Figs. 8 and 9.

In the declaration of `Set<X>`, the *type parameter* `X` denotes the unknown type of the set members. This generic class may be instantiated at *any* type, so `Set<Person>`, `Set<Date>`, and `Set<Int>` are all legal.

An instantiated generic class may be used to specify the type of a variable or to construct an object, as in:

```
var dict: Set<String>
dict := Set<String>()
```

An object construction may use a generic class instantiated with any type, including a type parameter. For example:

```
s := Set<X>()
```

constructs a `Set<X>` object, where `X` is replaced by the actual type that it denotes. An object construction may not use a naked type parameter, so "`X(...)`" is illegal. This does not seem to be a serious restriction in practice.

Monty treats `Array<X>` exactly like any other generic class, so an array object construction such as "`Array<X>(8)`" is perfectly legal. (By contrast, the object construction "`new X[8]`" is not fully supported in Java 5.)

5 Type System

In this section we will use the following notation for the type of an instance method m:

$$m : T \to (T_1 \times \ldots \times T_n) \to T_0$$

where T is the type of the receiver object (denoted by `this`); T_1, ..., and T_n are the parameter types; and T_0 is the result type. For example, in class `Stack`:

```
push : Stack → Object → ()
 pop : Stack → () → Object
```

and in class `Set<X>`:

```
contains : Set<X> → X → Bool
   union : Set<X> → Set<X> → Set<X>
```

We will also use the following notation for type equivalence and subtyping:

- $T_1 \equiv T_2$ means that T_1 is equivalent to T_2 (every T_1 object is a T_2 object, and *vice versa*)
- $T_1 \subseteq T_2$ means that T_1 is a subtype of T_2 (every T_1 object is a T_2 object)

The fundamental theorems of object-oriented programming are as follows. For every variable V of static type T_V we have the following invariant: V always contains null or an object of type $T_O \subseteq T_V$. If an expression E has static type T_E, then evaluation of E (if it terminates) always yields either null or an object of type $T_O \subseteq T_E$. In the assignment "V `:=` E", the compiler checks that $T_E \subseteq T_V$. It follows that $T_O \subseteq T_V$, so upholding V's invariant.

A language's type system is *sound* if no well-typed program will ever fail due to a run-time type error (other than a failed downcast).

Subtype relationships between non-generic classes are straightforwardly determined by the class hierarchy. For example, `Int` $\subseteq$ `Number`, since `Int` is a subclass of `Number`.

Nonvariant and Covariant Generic Classes

When we consider a generic class such as `Set<X>`, what (if any) type relationship exists between the instantiations `Set<`T_1`>` and `Set<`T_2`>`?

In Java 5, there may be a subtype relationship between instantiations of `X[]` (which in some respects resembles a built-in generic class), but not between instantiations of declared generic classes. For example:

(1) `Integer[]` $\subseteq$ `Number[]`

(2) `Set<Integer>` $\not\subseteq$ `Set<Number>`

The subtype relationship (1) is a legacy from Java 1. Unfortunately it is not type-sound. A program could assign an `Integer[]` array to a `Number[]` variable, and subsequently attempt to store a `Float` object in the `Integer[]`

array. Of course this would be a type error, but (unlike all other type errors in Java) it would be detected by a *run-time* type check.

In order to ensure that there is no similar unsoundness for declared generic classes, Java 5 insists that `Set<`T_1`>` and `Set<`T_2`>` are incomparable unless $T_1 \equiv T_2$. In other words, Java 5 insists that all declared generic classes $G\langle X\rangle$ are nonvariant, not covariant:

- If $G\langle X\rangle$ is *nonvariant*, then $G\langle T\rangle \subseteq G\langle T'\rangle$ if and only if $T \equiv T'$.
- If $G\langle X\rangle$ is *covariant*, then $G\langle T\rangle \subseteq G\langle T'\rangle$ if and only if $T \subseteq T'$.

Although Java 5's insistence on nonvariance can be justified on type-theoretic grounds, intuitively it seems unnatural. The subtype relationships (1) and (2) are surely the wrong way round. A particular `Set<Number>` object could be populated entirely with `Integer` members, so it seems perfectly reasonable to treat a `Set<Integer>` object as if it were a `Set<Number>` object. In other words, we would like to allow some generic classes, such as `Set<X>`, to be covariant.

Perhaps the most important innovation in Monty is that it treats all *immutable* generic classes as covariant. Recall that `Array<X>` is mutable while `Set<X>` is immutable. Thus we have the following subtype relationships in Monty:

(3) `Array<Int>` $\not\subseteq$ `Array<Number>`

(4) `Set<Int>` $\subseteq$ `Set<Number>`

A couple of examples should clarify the reason for making a distinction between mutable and immutable generic classes. Assume the following simplified API for `Array<X>`:

```
class Array<X> :
   public func get (i: Int): X :
      ...    ~ return this[i]
   public proc set (i: Int, x: X) :
      ...    ~ store x in this[i]
```

Consider the following application code:

```
var ints: Array<Int> := ...
var nums: Array<Number>
var r: Real := ...
nums := ints       ~ ill-typed
... nums.get(7)    ~ well-typed; type is Number
nums.set(7, r)     ~ well-typed
```

The assignment "`nums := ints`" is ill-typed since `Array<Int>` $\not\subseteq$ `Array<Number>`. It would be unsound to treat it as well-typed just because `Int` $\subseteq$ `Number`. The subsequent method call "`nums.set(7, r)`" would store a `Real` object in an `Array<Int>` array.

Now assume the declaration of `Set<X>` in Fig. 9, and consider the following

application code:

```
var ints: Set<Int> := ...
var nums: Set<Number>
var r: Real := ...
nums := ints            ~ well-typed
... nums.contains(r)    ~ well-typed; type is Bool
... nums.plus(r)        ~ well-typed; type is Set<Number>
```

Here the assignment "`nums := ints`" is well-typed since `Set<Int>` $\subseteq$ `Set<Number>`. Since `Set<X>` is an immutable class, no method call can change a `Set` object's state, although a method call can return a *new* `Set` object. The method call "`nums.plus(r)`" returns a new `Set<Number>` object, although its receiver object happens to be a `Set<Int>` object, so it does not matter that `Real` $\not\subseteq$ `Int`. To make this work, however, we must define the semantics of method calls with care.

Typing and semantic rules for FGJ (Featherweight Generic Java, a tiny subset of Java 5) are given in [8]. Consider the method call "`nums.plus(r)`", and note the type of the `plus` method in class `Set<X>`:

(5) `plus` : `Set<X>` $\rightarrow$ `X` $\rightarrow$ `Set<X>`

According to FGJ's *typing rules*, `X` in (5) would be statically instantiated to `Number`, consistent with the fact that `nums` has static type `Set<Number>`, so the method call's static type would be `Set<Number>`. According to FGJ's *semantic rules*, however, `X` would be dynamically instantiated to `Int`, since the receiver object turns out to have dynamic type `Set<Int>`. Fortunately, this inconsistency cannot arise in FGJ (or Java 5) because all generic classes such as `Set<X>` are nonvariant, so `nums` can never contain a `Set<Int>` object.

Monty's typing and semantic rules are similar to FGJ's, except for a subtle difference in the semantic rule for method calls. In the method call "`nums.plus(r)`", `X` in (5) is both statically *and dynamically* instantiated to `Number`, giving `Set<Number>` $\rightarrow$ `Number` $\rightarrow$ `Set<Number>`. This is consistent with the receiver object's dynamic type `Set<Int>` (since `Set<Int>` $\subseteq$ `Set<Number>`), with the argument's dynamic type `Real` (since `Real` $\subseteq$ `Number`), and with the statically-determined result type `Set<Number>`.

Soundness

Consider a generic class $G\langle X\rangle$ equipped with a method m with a parameter of type X:

```
class G⟨X⟩ :
   private var v:X
   ...
   public proc m (..., x:X, ...)  :
      ...
```

Focus on the parameter x of type X. The method m *could* use the corresponding argument to change the receiver object's state (e.g., by "`this.`v `:=` x"). The type of method m is:

$$m : G\langle X \rangle \to (\ldots \times X \times \ldots) \to \ldots$$

Now consider the method call:

O`.`m`(...,` A`, ...)`

where O is an expression yielding the receiver object, and A is an expression yielding the argument corresponding to x. This method call will be treated as follows: [6]

- At *compile-time*: Suppose that O's static type is $G\langle T \rangle$. The instantiation of method m's type is then $G\langle T \rangle \to (\ldots \times T \times \ldots) \to \ldots$. So A's static type should be a subtype of T.
- At *run-time*: The argument's dynamic type could be any type $T_a \subseteq T$. The receiver object's dynamic type could be any type $G\langle T_e \rangle \subseteq G\langle T \rangle$. If $G\langle X \rangle$ is *nonvariant*, $G\langle T_e \rangle \subseteq G\langle T \rangle$ only if $T_e \equiv T$. Here there is no problem, since $T_a \subseteq T_e$. But if $G\langle X \rangle$ is *covariant*, $G\langle T_e \rangle \subseteq G\langle T \rangle$ if $T_e \subseteq T$. Here there is a potential problem, since $T_a \not\subseteq T_e$. The potential problem becomes a real problem if m stores its argument in an instance variable such as v. Since the receiver object's dynamic type is $G\langle T_e \rangle$, the instance variable `this.`v has type T_e, and it would be unsound to allow a value of type $T_a \not\subseteq T_e$ to be stored in that variable. However, this problem cannot arise if $G\langle T \rangle$ is immutable.

This informal argument suggests that covariant generic classes are type-sound provided that they are immutable. However, this conjecture remains to be proved (see Section 8).

Bounds on Type Parameters

A Monty type parameter may be bounded. For example:

```
class Group <P extends Person> :
    ...
```

Inside this class body, we can assume that `P` $\subseteq$ `Person`, so any call to a `Person` method with a receiver object of type `P` is well-typed. `Group<P>` may be instantiated only at class `Person` and subclasses thereof.

By default, a type parameter's bound is `Object`. For example, the `Set` class of Fig. 9 could have been written:

```
class Set <X extends Object> :
    ...
```

[6] Here we assume static typing, for the sake of simplicity.

In general, Monty follows Java 5 in permitting a type parameter to be bounded by an ordinary class or by an instantiation of a generic class (but not by a naked type parameter).

6 Implementation

This paper is concerned primarily with the language design, but it is appropriate here to mention briefly some implementation issues that have influenced the design. The implementation of Monty will be explored more fully in a future paper.

Representation of Objects

The canonical representation of an object is a reference to a heap-allocated record comprising the object's class tag and its instance variables. This boxed tagged representation allows objects of different class to be used interchangeably.

Of course, the tagged boxed representation is extremely unwieldy for primitive objects such as `Int`s. Fortunately we can do better. For example, a variable of type `Int` will always contain an `Int` object, and a collection of type `Array<Int>` will contain only `Int` objects, so these `Int` objects need no class tags. In general, an unboxed untagged representation is suitable for such objects.

It is well known [10,11,16] that an unboxed untagged representation is possible for *primitive* values (and values of certain other built-in types). However, we can generalize this observation: the compiler can choose an unboxed untagged representation for any *immutable final* class whose objects are small and fixed in size. Examples of such classes are Monty's built-in classes `Int` and `Real`, and also declared classes such as `Date` (Fig. 4). On the other hand, `Person` and `Student` objects must have a boxed tagged representation (since `Person` is not final and `Student` is mutable).

The use of an unboxed representation for class C entails an implicit *boxing* operation whenever an object of class C is upcast to a superclass, and an implicit *unboxing* operation whenever an object is downcast to class C.

The use of unboxed representations for some classes implies a change from reference semantics to copy semantics for these classes. This is justified because reference semantics and copy semantics are essentially indistiguishable for immutable classes.

Static vs *Dynamic Typing*

In order to facilitate dynamic typing, Monty allows downcasts as well as upcasts to be implicit. The compiler can easily insert downcasts where required.

The critical implementation problem is a method call in which the inferred type of the receiver object is a class that, whilst not equipped with the named

method, has one or more subclasses that are so equipped.

For example, consider the `Person` class and subclasses of Fig. 5 and the following application code:

```
p: Person := ...
p.enrol("BA")
```

The `Person` class does not have an `enrol` method, but its `Student` subclass (alone) does have such a method. This method call can be implemented simply by inserting a downcast (in effect):

```
((Student)p).enrol("BA")
```

Now consider the following application code:

```
p: Person := ...
p.print()
```

The `Person` class does not have a `print` method, but both its `Employee` and `Student` subclasses do have such methods. This method call can be implemented by translating it (in effect) into a type-case command:

```
typecase p :
when e: Employee :
   e.print()
when s: Student :
   s.print()
else :
   throw NoSuchMethodError()
```

Implentation of Generic Classes

Generic classes can be implementated in two very different ways: [7]

- The *type-erasing translation* generates a single class file that will be shared by all possible instantiations of the generic class.
- The *specializing translation* generates a specialized class file for each distinct instantiation of the generic class.

The type-erasing translation in effect translates a generic class such as `Set<X>` to a non-generic class `Set`. At run-time, all objects of type `Set<Date>`, `Set<String>`, and so on are represented as objects of class `Set`, and are tagged accordingly. No information about type arguments exists at run-time.

The specializing translation in effect translates each *instantiation* of a generic class to a distinct non-generic class. For example `Set<Date>` would be translated to `Set$Date`, `Set<String>` to `Set$String`, and so on. Information about type arguments is encoded in the names of the generated classes.

[7] In [13] these implementations are (somewhat confusingly) called the *homogeneous translation* and *heterogeneous translation*, respectively.

Monty has been designed in such a way as to allow the compiler to choose freely between these implementations. For this reason Monty imposes some restrictions on the use of type parameters. A type test or cast may name only an uninstantiated class. For example:

```
if t is Set : ...
```

tests whether the value of `t` is an object of some class `Set<...>`. And:

```
s := (Set) t
```

checks that the value of `t` is an object of some class `Set<...>`, before assigning that object to `s`. A cast or class test may not use a type parameter, so "`(X)x`", and "`(Set<X>)t`" "`x is X`", "`t is Set<X>`", are all illegal. These restrictions are necessary to enable the type-erasing translation to work.

7 Related Work

Amber [1,6] is one of the few explicit attempts to combine static and dynamic typing in a single programming language. While being generally statically typed, Amber also provides a special type `dynamic`. A value of any ordinary type can be injected into `dynamic`, when it acquires a type tag; later the value can be projected out in a type-safe manner. This feature is useful for persistent storage of values of different types, but it does not achieve a smooth integration of static and dynamic typing.

Python [17] might yet evolve into a true programming/scripting language. Its data model has already evolved to the stage where all values are classified as objects. However, Python's object model is just too dynamic for conventional programming: any object can gain and lose instance variables at any time. Python still lacks the option of static typing, which is important for conventional programming. The Python website[8] includes much discussion about how static typing might be retrofitted (not an easy task), but at the time of writing this has not actually happened.

There have been numerous proposals for extending Java with generic classes [2,3,5,7,9,12,13]. In the end, the Generic Java proposal [5] was adapted slightly and incorporated into Java 5.

The language NextGen [7] was the first proposed extension of Java to support covariance. In NextGen each individual type parameter of a generic class may be annotated as either nonvariant or covariant.[9] However, there is a restriction: if a type parameter `X` is covariant, none of the generic class's methods may have a parameter of type `X`. In practice, this restriction is very severe, ruling out generic container classes such as `Set<X>`. In this paper we have argued that such generic classes can be allowed without loss of type-

[8] www.python.org/

[9] *Contravariant* type parameters were also considered as a possible future extension to NextGen. However, contravariance would be difficult to implement using JVM technology.

soundness.

A more radical idea is to allow *type arguments* in generic class instantiations (as opposed to type parameters in generic class declarations) to be annotated as nonvariant or covariant. This idea was first presented in [9], and has found its way into Java 5 in the form of *wildcards*. Consider the following mutable `Mset` class:

```
class Mset<X> :
   public proc add (x: X) :
      ...   ~ make x a member of this set
   public func contains (x: X): Bool :
      ...   ~ return true iff x is a member of this set
```

An example of nonvariant instantiation is:

```
var nums1: Mset<Number>
```

The variable `nums1` may be assigned only a `Mset<Number>` object. An example of covariant instantiation is:

```
var nums2: Mset<? extends Number>
```

The variable `nums2` may be assigned a `Mset<Number>`, `Mset<Int>`, or `Mset<Real>` object. There is, however, a restriction on method calls in the context of such a covariant instantiation. The method call "`nums2.add(r)`" would be considered ill-typed, since the `add` method has a parameter of type `X`. This restriction is necessary, otherwise the `add` method might store its argument (whose dynamic type could be `Real`) in some component of the receiver object (whose dynamic type could be `Mset<Int>`). Unfortunately, the method call "`nums2.contains(r)`" would be considered ill-typed for the same reason, since the `contains` method also has a parameter of type `X`. But this method does not update the receiver object. This restriction on method calls in the context of a covariant instantiation is sufficient but not necessary to ensure type-soundness.

The Java 5 collections library uses workarounds like the following:

```
class Set<X> {
   public boolean contains (Object x)        {...}
   public X choose ()                        {...}
   public Set<X> plus (X x)                  {...}
   public Set<X> union (Set<? extends X> s)  {...}
}
```

The `contains` method has a parameter of type `Object` (instead of the more natural "X"), allowing its argument to be *any* object. The `union` method has a parameter of type `Set<? extends X>` (instead of the more natural `Set<X>`), allowing its argument to be a set whose elements are of a subtype of `X`. We have seen that the Monty `Set<X>` class of Fig. 9 allows application code a similar degree of flexibility rather more naturally.

Scala [14] foreshadows some of Monty's key ideas. In Scala all values are objects; the difference is that the Scala class hierarchy is explicitly partitioned into *value classes* (which have unboxed representations) and *reference classes* (which have boxed representations). In [14] it is observed that covariance of immutable generic classes is type-sound. However, mutability is not a linguistic concept in Scala, which restricts covariance in much the same way as NextGen.

So Scala (like Java 5) needs a workaround like the following:

```
class Set[X] {
   public def contains (x: Object): boolean   = ...
   public def choose (): X                    = ...
   public def plus (x: X): Set[X]             = ...
   public def union (s: Set[Y <: X]): Set[X]  = ...
}
```

to allow application code a similar degree of flexibility to that which arises naturally with the Monty `Set<X>` class of Fig. 9. ("`<:`" is Scala's notation for "is a subtype of".)

8 Conclusion

The design of Monty is work in progress. At present Monty lacks some important features that are commonly found in more mature object-oriented languages, such as interfaces, packages, overloading, and reflection.

Two experimental implementations of the Monty core language have been initiated. The first (largely complete) translates Monty into Java, using the type-erasing translation. The other (just started) will translate Monty into JVM code, using the specializing translation.

The type system and semantics of the Monty core language are currently being formalized. A particular challenge will be to prove the soundness of Monty's type system. For this purpose a tiny subset of Monty will be used, akin to FGJ [8], but including assignment in order to explore the implications of mutability.

This paper has identified mutability as a useful linguistic concept. However, more work needs to be done on the role of mutability and covariance in Monty. It is striking that Monty allows a generic class to be covariant only if it is immutable, whereas NextGen [7] and Scala [14] allow a generic class's type parameter `X` to be covariant only if the class has no method with a parameter of type `X`. Since it seems that each of these restrictions is sufficient to ensure type-soundness, both of them must be stronger than necessary.

I believe that a weaker condition would be sufficient to ensure type-soundness: a generic class's type parameter `X` may be covariant only if the class has no *mutator* method with a parameter of type `X`. If this conjecture proves to be justified, it will strongly influence the design of the next version of Monty.

Java 5 and Scala are also likely to be major influences.

Acknowledgement

I am grateful to Simon Gay for reading an earlier version of this paper. I am also happy to acknowledge numerous comments and suggestions, from colleagues and LDTA participants, on the ideas presented in this paper.

References

[1] Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. (1991) Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13, 2, 237–268.

[2] Agesen, O., Freund, S., and Mitchell, J. (1997) Adding type parameterization to the Java language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages* (OOPSLA '97). ACM Press, New York, NY, 49–65.

[3] Allen, E., Bannet, J., and Cartwright, R. (2003) A first-class approach to genericity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages* (OOPSLA '03). ACM Press, New York, NY, 96–114.

[4] Barron, D.W. (2000) *The World of Scripting Languages*. Wiley, Chichester, England.

[5] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998) Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages* (OOPSLA '98). ACM Press, New York, NY, 183–199.

[6] Cardelli, L. (1986) Amber. In *Combinators and Functional Programming Languages* (ed. G. Cousineau, P. Curien, and B. Robinet), LNCS 242, Springer, Berlin, Germany, 21–47.

[7] Cartwright, R., and Steele, G.L. (1998) Compatible genericity with run-time types for the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages* (OOPSLA '98). ACM Press, New York, NY, 201–215.

[8] Igarishi, A., Pierce, B., and Wadler, P. (1999) Featherweight Java: a minimal core calculus for Java and GJ. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages* (OOPSLA '99). ACM Press, New York, NY, 132–146.

[9] Igarishi, A., and Viroli, M. (2002) On variance-based subtyping for parametric types. In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP '02). LNCS 2374, Springer, Berlin, Germany, 441–469.

[10] Leroy, X. (1992) Unboxed objects and polymorphic typing. In *Proceedings of Principles of Programming Languages* (POPL '92). ACM Press, New York, NY, 177–188.

[11] Morrison, R., Dearle, A., Connor, R., and Brown, A. (1991) An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13, 3, 342–370.

[12] Myers, A., Bank, J., and Liskov, B. (1997) Parameterized types for Java. In *Proceedings of the ACM Conference on Principles of Programming Languages* (POPL '97). ACM Press, New York, NY, 132–145.

[13] Odersky, M., and Wadler, P. (1997) Pizza into Java: translating theory into practice. In *Proceedings of the ACM Conference on Principles of Programming Languages* (POPL '97). ACM Press, New York, NY, 146–159.

[14] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004) An overview of the Scala programming language, scala.epfl.ch/docu/.

[15] Ousterhout, J. (1998) Scripting: higher-level programming for the 21st century. *IEEE Computer*, 31, 3, 23–30.

[16] Peyton Jones, S., and Launchbury, J. (1991) Unboxed values as first-class citizens. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture* (FPCA '91). LNCS 523, Springer, Berlin, Germany, 636–666.

[17] van Rossum, G., and Drake, F. (2003) *The Python Language Reference Manual*, Network Theory Ltd, Bristol, England.

Appendix: Monty Syntax

This appendix summarizes the syntax of the Monty programming/scripting language. The language consists of a core plus extensions.

Note that an identifier (Id) starts with a lowercase letter, while a type-identifier (TyId) starts with an uppercase letter.

Indentation is significant: line terminators (shown here as ↓), indents (→), and outdents (←) are language tokens.

Programs

Prog ::= (Com | VarDec | MethDec | ClassDec)*

Commands (core language)

```
Com    ::=  skip ↓
          |  exit ↓
          |  return Exp? ↓
          |  throw Exp ↓
          |  Assign ↓
          |  if Exp : ↓ Block ( else if Exp : ↓ Block )*
               ( else : ↓ Block )?
          |  repeat : ↓ Block
          |  while Exp : ↓ Block
          |  try : ↓ Block ( when Id : TyId : ↓ Block )+
               ( else : ↓ Block )?

Block  ::=  → ( Com | VarDec )* ←
```

Expressions (core language)

```
Exp    ::=  AExp                    atomic expression
          |  Type Args              object construction
          |  Exp . Id               instance variable access
          |  TyId . Id              static variable access
          |  Exp . Id Args          instance method call
          |  TyId . Id Args         static method call
          |  Id Args                global method call
          |  Exp is TyId                      class test
          |  ( TyId ) Exp                        cast
          |  ( Exp )

AExp   ::=  Lit                            literal
          |  null                     null reference
          |  this                     receiver object
          |  Id       local/global variable access

Args   ::=  ( Exps? )
          |  AExp

Exps   ::=  Exp ( , Exp )*

Assign ::=  Exp
          |  Exp := Assign
```

Declarations (core language)

```
ClassDec   ::=  final? immutable? class TyId ( < TyPars > )?
                  ( extends Type )? : ↓ ClassBody

TyPars     ::=  TyPar ( , TyPar )*

TyPar      ::=  immutable? TyId ( extends Type )?

ClassBody  ::=  → CompDec* ←

CompDec    ::=  VisSpec static? VarDec
             |  VisSpec ConsDec
             |  VisSpec ( static | virtual )? MethDec
             |  VisSpec ClassDec

VisSpec    ::=  private | protected | public
```

VarDec ::= (`val` | `var`) Id TySpec (`:=` Exp)$^{?}$ ↓

ConsDec ::= `cons` `(` Pars$^{?}$ `)` `:` ↓ Block

MethDec ::= `proc` Id `(` Pars$^{?}$ `)` `:` ↓ Block
| `func` Id `(` Pars$^{?}$ `)` TySpec `:` ↓ Block

Pars ::= Par (`,` Par)*

Par ::= Id TySpec

Types

Type ::= TyId (`<` TyArgs `>`)$^{?}$

TyArgs ::= Type (`,` Type)*

TySpec ::= (`:` Type)$^{?}$

Commands (extended language)

Com ::= ...
| `case` Exp `:` ↓ (`when` Lit `:` ↓ Block)$^{+}$ (`else` `:` ↓ Block)$^{?}$
| `typecase` Exp `:` ↓ (`when` Id `:` TyId `:` ↓ Block)$^{+}$
(`else` `:` ↓ Block)$^{?}$
| `for` Id TySpec `in` (Exp | Exp `..` Exp) `:` ↓ Block

Expressions (extended language)

Exp ::= ...
| (Op | \ ) Args — prefix operator call
| Exp \$^{?}$ Op Args — infix operator call
| Exp `[` Exp `]` — collection indexing
| Type `{` Exps$^{?}$ `}` — collection construction
| Type `{` Qual^{+} Exp `}` — comprehension

Qual ::= `for` Id TySpec `in` Exp `:`
| `if` Exp `:`

Assign ::= ...
| Exp Op `:=` Assign

Declarations (extended language)

MethDec ::= ...
| `func` Op `(` Pars$^{?}$ `)` TySpec `:` ↓ Block