Harmonizing Classes, Functions, Tuples, and Type Parameters in Virgil III

Ben L. Titzer
Google
titzer@google.com

Abstract

Languages are becoming increasingly multi-paradigm. Subtype polymorphism in statically-typed object-oriented languages is being supplemented with parametric polymorphism in the form of generics. Features like first-class functions and lambdas are appearing everywhere. Yet existing languages like Java, C#, C++, D, and Scala seem to accrete ever more complexity when they reach beyond their original paradigm into another; inevitably older features have some rough edges that lead to nonuniformity and pitfalls. Given a fresh start, a new language designer is faced with a daunting array of potential features. Where to start? What is important to get right first, and what can be added later? What features must work together, and what features are orthogonal? We report on our experience with Virgil III, a practical language with a careful balance of classes, functions, tuples and type parameters. Virgil intentionally lacks many advanced features, yet we find its core feature set enables new species of design patterns that bridge multiple paradigms and emulate features not directly supported such as interfaces, abstract data types, ad hoc polymorphism, and variant types. Surprisingly, we find variance for function types and tuple types often replaces the need for other kinds of type variance when libraries are designed in a more functional style.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Multi-paradigm languages; parametric types; object-oriented programming; functional programming; monomorphization; variance; closures; tuples; flattening; unboxing; static compilation

1. Introduction

Mainstream languages are becoming increasingly multi-paradigm and increasingly complex. Statically-typed object-oriented languages have now ventured beyond the subtype polymorphism of class and object inheritance and begun adding parametric polymorphism in various forms; erased generics in Java 5.0, reified generics in C# 2.0, C++ templates, and even more complex generics in Scala.

Functional programming constructs are now appearing in nearly all active languages, further adding to language complexity. Languages like Java first offered object-oriented emulation of functions in the form of single-method interfaces like `Runnable` and `Comparator`. Generics gave rise to even more complex emulations of functions such as `java.util.concurrent`'s `Callable` and Guava's `Function` [13] and `Predicate`. Now Java has proposals for syntax for function types and closures in JSR 335 [17]. Similar developments occurred in both C++ and C#, with libraries first evolving functional patterns before languages added syntactic support. C# first allowed *delegates*, which are closures bound to a method of an object, and then full lambdas. Scala set out to blend object-oriented and functional styles while still primarily targeting the JVM. Not to be outdone, C++11 has added support for lambdas and a host of other new features. Of course the relationship between functions and objects goes back even further in dynamically-typed languages. Smalltalk had blocks [12], popular dynamic languages such as JavaScript, Python and Ruby have first class functions, and newcomers such as Newspeak, Groovy and Dart also incorporate some kind of functional features.

A designer of a new programming language faces a daunting lineup of features from which to choose. What features work best? How does this feature enhance or hinder another feature? What feature should be implemented first? Should objects be primary, or functions be primary? What tools should programmers have to express polymorphism and reuse, model side-effects or state, and how can the constructs be implemented efficiently?

Of course, these are the tough design questions to which the whole of language research is directed. This paper explores just *one* potential combination of four language features and the implications of that combination. In seeking to find a smaller, more cohesive language core made from existing parts, and seeking to implement a practical language in which efficient systems can be written, we've found a number of surprising patterns arising from just a small set of features.

This paper finds that integrating four particular features in a statically-typed language achieves a surprising cohesiveness and simplicity. **Classes** provide data abstraction, encapsulation, and information hiding; first-class **functions** allow fine-grained reuse; **tuples** allow for the uniform treatment of multi-argument and multi-return function types and simplify composing values, and **type parameters** allow type abstraction and representation independence for more reuse of functions and data structures represented by objects. None of these features are novel, and all of them exhibit implementations in mainstream languages, but their seamless integration is key in Virgil's design. This integration yields surprising power; enough to build new design patterns that emulate other language features not in the language, including interfaces, abstract data types, ad hoc polymorphism, and variants. An interesting aside is Virgil's approach to type variance; a key design difficulty in recent statically typed languages with both subtyping and parametric polymorphism. Surprisingly, we find type variance for class types

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

to be far less important when libraries are designed in a more functional style because built-in variance for tuple and function type constructors is often sufficient.

2. Four Pieces of the Puzzle

Virgil I was originally designed [31] for programming severely resource-constrained devices with only a few hundred bytes of RAM. A staged computation model allows applications to execute initialization code during compilation, building a heap of objects that is compiled into a binary that executes directly on the hardware. Virgil I supported objects and functions but had no dynamic memory allocation, no type parameters, and no tuples, making it unsuitable for general-purpose programming. Virgil III is a clean redesign for a more general-purpose setting. It retains the staged compilation model of Virgil I but improves classes and functions, adds tuples, type parameters, dynamic memory allocation, and garbage collection.

Virgil includes primitive types such as `int`, `byte`, `bool`, and `void`[1]. All types support four basic operators: equality `==`, inequality `!=`, type cast `!`, and type query `?`. Control structures, basic statements and expression syntax are similar to most C language descendants, but declarations and classes look more like Scala.

2.1 Classes

Virgil provides classes that allow encapsulation, data abstraction and polymorphism through inheritance. Classes resemble those in Java and C# in that only single inheritance is allowed between classes and all methods are virtual except those declared private. All objects are created by instantiating classes and are always passed by reference, never by value. Unlike most object-oriented languages, Virgil has no universal superclass akin to `Object` from which all classes ultimately inherit. A class declared without a parent class begins a new hierarchy which is unrelated to other class hierarchies. This means that there is no unifying supertype for all objects, and there are no default methods available on all objects. Classes are further limited in that there is no concept of an *interface* which a class can implement. The example below illustrates the basics of Virgil classes.

```
(a1)  class A {
(a2)    var f: int;                    // mutable field
(a3)    def g: int;                    // immutable field
(a4)    new(f, g) { ... }              // constructor
(a5)    def m(a: byte) -> int { ... } // method
(a6)    def n(a: X, b: T) { ... }
(a7)  }
(a8)  class B extends A {              // subclass
(a9)    def m(a: byte) -> int { ... } // override
(a10) }
```

Pure object-oriented philosophy views all values as objects with some universal type and a default set of methods. The lack of a universal supertype in Virgil or any kind of interface mechanism at first seems crippling, but as we will see in this section, the other features of the language more than make up for the loss of expressiveness. Virgil does not hold the everything-is-an-object philosophy for several reasons. First, domains such as system software [9] [20] and scientific computing [21] demand efficient primitive types; modeling these primitives as objects can be cumbersome, performance can be unpredictable, and mapping certain object types onto hardware primitives requires significant extra-lingual and compiler support. Second, a universal type in object-oriented languages of the past was most often used to implement reuseable datastructures and provide default methods; these roles are better served with proper design of type parameters and use of first class functions, as we will see in the next section. Third, the possibility of subsumption to a universal type weakens confinement [34] properties and may reduce the precision of other static analyses [6][25][31].

[1] `void` has one value, `()`, which is always equal to itself.

2.2 First-class Functions

Virgil provides support for statically-typed first-class functions. Function types take the form T_p `->` T_r, where T_p is the parameter type and T_r is the return type. The type `void` becomes the parameter type of methods with no parameters and the return type of methods without a declared return type. Function types are unrelated to object types, forming a separate universe of values. Function types are co-variant in their return type and contra-variant in their parameter type. A function type can legally appear anywhere any other type can appear, such as the type of a class field, method parameter, local variable, or return type of a method. Unlike most traditional functional languages, tuples are used to model multi-argument functions instead of currying.

To bridge the object-oriented and functional worlds, any method of any object can be used as an *object method*, a function which is bound to the object as its closure. Similarly, *class methods* are not bound to an object but are functions that accept the receiver object as their first parameter followed by the method parameters. All of the basic primitive operators can be used as first-class functions as well, and all three kinds of functions can be used interchangeably.

Examples (b1-7) illustrate the basics, with the types of each expression given in comments. Continuing with the class `A` from (a1-6), we can (b1) create an object of that class and (b2) create a closure bound to the `m` method of that object. In (b3) we use the method `m` of class `A` as a first-class function that is not bound to an object, but accepts the receiver object as the first parameter. In (b4) we invoke `m` directly on the object, in (b5) apply the function `m1` to arguments, and in (b6) apply the function `m2`, supplying the object as the first argument. Notice that we can also use the `new` operator for a class `A` as a function as well (b7).

```
(b1) var a = A.new(0, 1); // A
(b2) var m1 = a.m;        // byte -> int
(b3) var m2 = A.m;        // (A, byte) -> int
(b4) var x = a.m('5');    // int
(b5) var y = m1('4');     // int
(b6) var z = m2(a, '6');  // int
(b7) var w = A.new;       // (int, int) -> A
```

The four basic operators `== != ! ?` of every type T are available as first class functions. Though normally written with infix notation, comparisons (b8-9) can be used as functions by referring to them as members of a type, as can primitive and integer arithmetic operators (b10-11).

```
(b8)  var z = byte.==; // (byte, byte) -> bool
(b9)  var w = A.!=;    // (A, A) -> bool
(b10) var p = int.+;   // (int, int) -> int
(b11) var m = int.-;   // (int, int) -> int
```

Type casts (b12) express all type conversions, including downcasts of objects and conversions between primitive values. Type queries (b13) evaluate to `true` if the value is of type T, `false` otherwise. Type casts and type queries of class types check the dynamic type of their input objects. Both operators each have a type parameter that abstracts the type of the input value. Parameterization allows dynamic casts and dynamic type queries between any two types, even polymorphic types. This is a blatant violation of the *parametricity* property of type parameters, but is an intentional tradeoff that allows dynamic casts to emulate other language features, as we will see later. In practice, the compiler rejects casts and queries between unrelated types wherever possible, such as between a function type and a primitive type or between unrelated

classes. Like the other operators, casts and queries can be used as functions (b14-15).

```
(b12) var x = A.!(...);   // A
(b13) var y = A.?(...);   // bool
(b14) var z = A.!<B>;     // B -> A
(b15) var w = A.?<B>;     // B -> bool
```

2.3 Tuples

Tuples are a lightweight mechanism to group multiple values into a single value. The resulting tuple value is not an object and has no identity; tuples with equivalent elements are always equal, no matter when or where in the program they are computed. No restrictions are placed on what types may appear as element types of a tuple; if `A` and `B` are valid types, then `(A, B)` is a valid tuple type. Inductively, this means that tuples can be inside of tuples, but not recursively, since tuple types have no names. Elements are ordered and accessed as if they were fields named as integer literals beginning at `0`. Tuples are not arrays; they cannot be indexed by expressions.

```
(c1) var x: (int, int) = (0, 1);
(c2) var y: (byte, bool) = ('a', true);
(c3) var z: ((int, int), (byte, bool)) = (x, y);
(c4) var w: (int) = x.0;
(c5) var u: byte = (z.1.0);
(c6) var v: () = ();
```

Tuple types with any number of element types are possible, with two degenerate cases: (c6) a tuple type $T =$ `()` with no elements is exactly the same as the `void` type, and (c4) a tuple type $T =$ `(A)` with just one element is exactly the same as the type `A`. Similarly, tuple values with any number of element values can be created, with (c6) a tuple value $V =$ `()` with no elements equivalent to the one `void` value, and (c5) a tuple $V =$ `(e)` is equivalent to `e`.

Immutability admits straightforward covariant type rules for tuples. $T = (T_0 ... T_m)$ is a subtype of $S = (S_0 ... S_n)$ if and only if $m = n$ and all elements T_i of T are subtypes of the corresponding S_i. [2]

Like all types in Virgil, every tuple type T has the four basic operators `== != ! ?`. Equality and inequality operators operate recursively on the elements of a tuple: two tuple values are equivalent if the positionally-correspondent elements are equivalent. Casts and queries are also defined recursively: a cast of a tuple value to a tuple type succeeds if a cast of each element to the positionally-corresponding element type succeeds, and a type query of a tuple value against a tuple type is `true` if all type queries of elements against the positionally-corresponding element type are `true`.

Syntactic choices for tuples give rise to familiar looking calls. For example, a typical method invocation such as `a.m(foo, bar)` parses as an application of the expression `a.m` to the tuple expression `(foo, bar)`. The expression `a.m` itself is an object method expression, but could be any expression that evaluates to a function of the appropriate type. Nested calls like `f(g(x))` compose nicely, even if the inner call has a tuple or `void` return type. This represents semantics similar to tuples in Haskell and ML, but with syntax that resembles Java and C#.

Tuple type syntax leads to familiar syntax for function types. `(A, B) -> C` denotes a function type with two parameters and `A -> (B, C)` denotes a function that returns two values. The degenerate rules for tuple types imply intuitive equivalences, such as `() -> () = void -> void` and `(A) -> (B) = A -> B`. This also allows the tuple type constructor `()` to be used as a grouping mechanism, either to emphasize the right-associative nature of `->` by writing `A -> (B -> C)`, or override it with `(A -> B) -> C`. For type rules, covariance of tuple types means that variance rules for function types operate exactly as one would expect when applied to functions that accept or return tuples.

2.4 Type Parameters

Classes, functions and tuples all coexist in Virgil, but an important abstraction mechanism is missing. Without abstraction over types, the nonoverlapping universe of primitive types, the separate class hierarchies of user classes, the system of tuple types, and the system of function types would be independent of each other forever. *Type parameters* provide this capability, allowing classes and methods to be parameterized over the types of values they manipulate, greatly increasing the expressive power of the language.

A type parameter T introduces a new name for a unknown type within the scope of its declaration. The unknown type is *universally quantified* in the sense that it could be instantiated to *any* type at a usage site. Within the scope of declaration, the new unknown type T is unrelated to all other types except itself. The compiler typechecks the body of the class or method without any information about T except that it supports the same four basic operators that all types support, namely `== != ! ?`. *Separate typechecking* means that the bodies of parameterized declarations are checked independently of any instantiation and type errors in the declaration do not generate type errors at the usage sites[3].

```
(d1) class List<T> {
(d2)   var head: T;
(d3)   var tail: List<T>;
(d4)   new(head, tail) { }
(d5) }
(d6) def apply<A>(list: List<A>, f: A -> void) {
(d7)   for (l = list; l != null; l = l.tail) f(l.head);
(d8) }
```

In (d1-8) we implement a cons list using a generic class. Objects of the class (d1) store a value (d2) of the generic type and a reference to the next link (d3) in the list. A generic convenience method (d6) provides a handy way of iterating over the elements of a list and applying a function to each element.

```
(d9)   def print(i: int) { ... }
(d10)  var a = List<int>.new(0, null);
(d11)  var b = List<(int, int)>.new((3, 4), null);
(d12)  apply<int>(a, print);
(d10') var c = List.new(0, null);
(d11') var d = List.new((3, 4), null);
(d12') apply(c, print);
(d13)  var e = List<bool>.?(a);
(d14)  var f = List<void>.?(a);
```

At every usage site of a parameterized class or method, *type arguments* are supplied for the type parameters, either explicitly by the programmer or inferred by the compiler. Explicit type arguments to a parameterized class or method can always be supplied with a `<...>` suffix (d10-12) immediately following the identifier, but the compiler can usually infer them (d10'-12'). Virgil uses a best-effort type inference algorithm[4] for type arguments to both classes and methods. Virgil does not perform type erasure; instead, enough information is always retained to recover the type argu-

[2] Subtyping rules for tuples could also allow a longer tuple to be a subtype of a shorter tuple, but too much static checking would be lost; basic errors such as passing more arguments to a function than it expects would go unnoticed.

[3] C++ templates are typechecked at instantiation time, leading to famously huge and inscrutable error messages where type arguments are already substituted for type parameters.

ments of any parameterized usage. That means, among other things, that polymorphic types can be distinguished at runtime (d13-14).

Any type can appear as a type argument. That means we can use our list class to create and manipulate homogeneous lists of any type, including primitives, objects, tuples, functions, and even `void`. Allowing unrestricted type arguments has subtle but important benefits. First, since there are no exceptions or special cases to remember, the language is actually simpler and more intuitive; any type can be a type argument, no exceptions. Secondly, reuse of data structures and functions is greatly enhanced by *composability*; we can easily create lists of functions, or lists of tuples of functions, or functions of arrays of lists of functions. Third, it promotes a programming style based on representation independence where no information about the representation of a type is necessary for the class or function to perform its role.

```
(e1) def time<A, B>(func: A -> B, a: A) -> (B, int) {
(e2)   var start = clock.ticks();
(e3)   return (func(a), clock.ticks() - start);
(e4) }
(e5) print(time(sqrt, 37).1);
```

In (e1-5) we make use of three language features to implement a method that times the execution of a function applied to some given arguments. First, our utility accepts a function as the routine to be timed. Second, it has type parameters that abstract the parameter and return type of that function, allowing it to be used with *any* function and arguments, including tuples or even `void`. Third, it uses a tuple to return both the time elapsed and function's result, even if the function returned `void` or another tuple.

2.5 Type Constructor Summary

Virgil's type system employs five kinds of type constructors. Four language-provided type constructors represent primitive types, arrays, tuples, and functions. The fifth kind of type constructor represents the types of class objects; a new class type constructor is created for each user-defined class. The table below summarizes the type constructors, their type parameters, and their syntax. In this table, the symbol $\ominus$ means *contravariant*, and $\oplus$ means *covariant*.

Typecon	Type Parameters	Syntax
Primitive		`void`|`int`|`byte`|`bool`
Array	T	`Array<`T`>`
Tuple	$\oplus T_0 \cdots \oplus T_n$	`(`$[T($`,`$T)*]$`)`
Function	$\ominus T_p \oplus T_r$	T `->` T
`class` X	$T_0 \cdots T_n$	`X[<`$T($`,`$T)*$`>]`

3. Patterns

Language designers face an unending series of tradeoffs when choosing which features to add, tradeoffs which become more complex and difficult as the language grows. Abstract data types, interfaces, ad hoc polymorphism, and variant types are useful, but how should they be prioritized relative to other features? Our experience with the design of Virgil suggests that these features can at least be postponed. Instead we found that the four principal language features were often powerful enough to *emulate* these other features with multi-paradigm design patterns made possible by the close collaboration of several features at once.

3.1 Interface Adapters

While other OO languages offer complex patterns of inheritance such as interfaces and traits, Virgil offers only single inheritance between classes. More complex patterns of inheritance and interface specification can be emulated with first class functions.

The most straightforward technique for emulating interfaces in Virgil is to define a class whose fields store first-class functions representing the methods of the interface, each with a name and a type. This class is essentially a dictionary of named interface methods. What would be somewhat awkward and verbose to define in other languages is actually quite easy in Virgil thanks to a compact syntax for declaring immutable public fields that are initialized in the constructor. In (f1-5) we see an example where the `DatastoreInterface` defines a dictionary with `create`, `load`, and `store` operations, each of which is a field which stores a function.

```
(f1) class DatastoreInterface(
(f2)   create: () -> Record,
(f3)   load: Key -> Record,
(f4)   store: Record -> ()) {
(f5) }
```

Virgil's ability to bridge the object and functional paradigms with class and object methods makes it easy for another unrelated class to simply construct an instance of the interface using its own methods. In (g6-7), the `DatastoreImpl` class *adapts* itself to the `DatastoreInterface` by instantiating an instance of the class, passing *object methods* bound to itself (i.e. the `this` object); invocations of those functions through the interface will receive the bound `this` as the receiver object.

```
(g1) class DatastoreImpl {
(g2)   def create() -> Record { ... }
(g3)   def load(k: Key) -> Record { ... }
(g4)   def store(r: Record) { ... }
(g5)   def adapt() -> DatastoreInterface {
(g6)     return DatastoreInterface.new(
(g7)       create, load, store);
(g8)   }
(g9) }
```

Tradeoffs. One advantage to this pattern is that an adapter object can be constructed from any set of functions, regardless of their names. A class can implement multiple interfaces, and the names of the methods it uses to implement the interface are immaterial, as long as the methods have the correct signature, which avoids name clashes. In fact, there is no requirement that an interface be constructed from class methods at all. An interface adapter object could as well be constructed from top-level functions, built-in operators, or any other functions at hand. Notice that in (g2-4), the methods could as well have had any names, since they are simply used as functions when constructing the interface adapter object (g6-7). Unfortunately, because Virgil has no implicit type conversions, object types do not subsume to the interface types they implement. Thus the primary disadvantage of this pattern is that to use an object as an instance of the interface always requires an extra step, usually the construction of another object. Another drawback is that the programmer must manually specify the mapping between existing functions to the interface by supplying them to the constructor of the interface object, as done in the `adapt` method in (g5).

3.2 Emulating Abstract Data Types

Abstract data types (ADTs) are useful when modeling a type that has unknown representation but has a set of associated operations with names and signatures. One way to model an ADT in Virgil is to parameterize an interface (h1-7). In this example we model a number that has an unknown representation but a known set of associated operations and some named values. The availability of the basic operators like `int.+` as first class functions makes it easy to adapt the basic primitive type `int` to the ADT interface (h8-9).

[4] Type inference with both subtyping and type parameters is tricky business; see for example [23]. Our compiler uses a bi-directional typechecking approach, but the details are beyond the scope of this paper.

In this pattern the interface is not bound to a particular value or object instance; instead, the values are external so that they can be manipulated by the client.

```
(h1) class NumberInterface<T>(
(h2)   add: (T, T) -> T,
(h3)   sub: (T, T) -> T,
(h4)   compare: (T, T) -> bool,
(h5)   one: T,
(h6)   zero: T) {
(h7) }
(h8) var IntInterface = NumberInterface.new(
(h9)   int.+, int.-, int.==, 1, 0);
```

A class interface is not always necessary, especially if the number of operations associated with the type is very small. In (i1-8) we see one way to design a hash table using the abstract data type pattern, but without the need to supply an interface object.

```
(i1) class HashMap<K, V> {
(i2)   def hash: K -> int;          // hash function
(i3)   def equals: (K, K) -> bool; // equality function
(i4)   new(hash, equals) { }
(i5)   def get(key: K) -> V { ... }
(i6)   def set(key: K, val: V) { ... }
(i7)   def apply(f: (K, V) -> void) { ... }
(i8) }
```

`HashMap` abstracts over the key and value types using type parameters and accepts the `hash` and `equals` functions as parameters to the constructor (i4). This allows this one `HashMap` implementation to be used with *any* key or value type. This is in contrast to the typical object-oriented approach, where every key type would necessarily be a class type and implement `hash` and `equals` methods with appropriate signatures. With unrestricted type parameters, even primitives (i15) and tuples (i18) can serve as the key, as long as they have the associated methods. Moreover, different `hash` and `equals` methods can be used on a per-instance basis (i13-14) with the help of *class methods* and first-class operators like `==` that bridge the gap between the object-oriented and functional worlds.

```
(i9)  class X {
(i10)   def deepEquals(x: X) -> bool { ... }
(i11)   def hash() -> int { ... }
(i12) }
(i13) HashMap<X, int>.new(X.hash, X.deepEquals);
(i14) HashMap<X, int>.new(X.hash, X.==);
(i15) HashMap<int, X>.new(int.!, int.==);
(i16) var h2: (int, int) -> int;
(i17) var e2: ((int, int), (int, int)) -> bool;
(i18) HashMap<(int, int), X>.new(h2, e2);
```

Tradeoffs. The primary disadvantage to using type parameters and functions to model abstract data types, rather than a module system, is that type parameters require the client code to be parameterized over the abstract data type. There must be a scope where the client code has declared a type parameter, whereas with a module system the type can be made available in the namespace of the client code. In the `HashMap` example, the client code instantiates the `HashMap` code with the types of the keys and values, but in another example, such as the `NumberInterface`, the code that provides the abstract data type must instantiate not only the number interface, but the code manipulating the numbers.

3.3 Emulating Ad-hoc Polymorphism

Virgil provides no direct support for ad-hoc polymorphism such as method overloading. While other languages make use of arguments' types at call sites to perform overload resolution, Virgil allows methods to be used in a first-class way where enough context may not be available. For example, suppose a method `m` has several overloaded variants but is used simply as `o.m`, lacking any arguments for overload resolution. The inherent ambiguity would require a manual resolution mechanism. Virgil chooses to disallow overloading altogether, requiring every method in the same class to have a unique name. It views overloaded methods as parameterized methods, with the type parameters being instantiated with the various overloadings at call sites. This approach admits an emulation of method overloading and still allows methods to be used in a first-class way without ambiguity.

Consider a use case like `print`. Remembering to call `printInt` versus `printBool` versus `printString` can become cumbersome. A somewhat cleaner solution is to use a design pattern that admits a small number of overloads, making use of type parameters and casts:

```
(j1) def print1<T>(fmt: string, a: T) {
(j2)   if (int.?(a)) printInt(fmt, int.!(a));
(j3)   if (bool.?(a)) printBool(fmt, bool.!(a));
(j4)   if (string.?(a)) printString(fmt, string.!(a));
(j5)   if (byte.?(a)) printByte(fmt, byte.!(a));
(j6) }
(j7) print1("Result: %1\n", 0);
(j8) print1("Boolean: %1\n", false);
(j9) print1("Hello %1!", name);
```

A parameterized method (j1) is the starting point of this pattern. The parameterized method dispatches to the appropriate `print*` method through a chain of dynamic type queries and type casts. The user of the library calls the parameterized method (j7-9), with the compiler inferring the type arguments. The chain of dynamic casts in (j2-5) will be optimized by the compiler. First, the compiler will specialize the parameterized method for each unique type argument, then optimize each version independently. The type queries and casts in each version can be decided statically, the chain of `if` statements will be folded away, and only a call to the corresponding version remains, which the compiler may then inline, resulting in code just as efficient as if the caller had called the appropriate `print*` method directly.

Tradeoffs. One clear disadvantage of this approach is that some static checking is lost since the parameterized method must necessarily accept any type for its type argument. Unless all possible types are covered by some case, the programmer will have to resort to producing a runtime error or defining some default behavior [5]. The benefit, aside from not needing to remember to call the appropriate unique method, is that there is no longer any ambiguity in first-class functions, and the parameterized method can be used in a first-class way like other methods, supplying explicit type arguments if necessary. Multi-argument overloads are possible with tuples, but type matching can become painfully ugly, so in practice one tends to write `print1`, `print2`, etc. For complex cases this is admittedly an unsatisfyingly clunky solution, but it works and is very efficient. It does not require boxing arguments in any situation, it optimizes away dynamic type tests, and does not require a *varargs* mechanism.

3.4 A Polymorphic Matcher

Subtyping provides a language with a form of information hiding where the exact type of an expression is intentionally forgotten when it is used in a context requiring only the supertype. This ability of subtyping can also hide information about type parameters. For example, declaring a base class `Any` and a subclass `Box<T> extends Any` allows any value to be boxed and used

[5] Our implementation of `print` accepts the standard primitive types and also functions of type `StringBuffer -> void`; we equip those classes that need to be printed with methods that render the object into a `StringBuffer`; we can then simply pass `o.render` to the `print` method.

wherever the type `Any` is accepted. Instead of language-provided classes for each primitive type such as `java.lang.Integer`, `java.lang.Double`, etc, a single pair of user-defined classes suffices.

```
(k1)  class Matcher {
(k2)    var matches: List<Any>;
(k3)    def add<T>(f: T -> void) {
(k4)      matches = List.new(Box.new(f), matches);
(k5)    }
(k6)    def dispatch<T>(v: T) {
(k7)      for (l = matches; l != null; l = l.tail) {
(k8)        var f = l.head;
(k9)        if (Box<T -> void>.?(f))
(k10)         return Box<T -> void>.!(f).unbox()(v);
(k11)     }
(k12)   }
(k13) }
```

Hiding information about type parameters using class subtyping allows us to improve upon our previous emulation of method overloading. In (k2-5), the matcher wraps each function in a `Box` and intentionally forgets its type, treating it as `Any`, in order to put it in a list. To dispatch on a polymorphic value, since Virgil does not erase type parameters but can in fact distinguish a `Box<int -> void>` from a `Box<bool -> void>`, the matcher can search the list for a `Box` that contains a function of the right type that can handle the polymorphic value. In (m1-8) we see examples of how we could use the `Matcher` to accomplish polymorphic dispatching for printing without having to write a series of type queries and casts. The `Matcher` class need only be written once, and it can be reused in whatever context is necessary. [6]

```
(m1) var m = Matcher.new();
(m2) m.add(printInt);
(m3) m.add(printBool);
(m4) m.add(printString);
(m5) ...
(m6) m.dispatch("Result: %1", 1);     // printInt
(m7) m.dispatch("Boolean: %1", true); // printBool
(m8) m.dispatch("Hello %1", name);    // printString
```

Tradeoffs. The polymorphic matcher pattern improves upon the manual type dispatching approach but still lacks static checking. Since the matcher's `dispatch` method can be instantiated with any type argument, it may fail at runtime if an appropriate method is not found in the list.

3.5 Emulating Variant types

The ability of subtyping to hide information can emulate a variety of ad-hoc polymorphism patterns, including the ability to have a number of variant types, each with the same operation.

```
(n1)  class Instr {
(n2)    def emit(buf: Buffer);
(n3)  }
(n4)  class InstrOf<T> extends Instr {
(n5)    var emitFunc: (Buffer, T) -> void;
(n6)    var val: T;
(n7)    new(emitFunc, val) { }
(n8)    def emit(buf: Buffer) {
(n9)      emitFunc(buf, val);
(n10)   }
(n11) }
(n12) var i = InstrOf.new(asm.add, (rax, rbx));
(n13) var j = InstrOf.new(asm.addi, (rax, -11));
(n14) var k = InstrOf.new(asm.neg, rax);
```

[6] Notice also that (m2-8) uses functions with multiple parameters, which works seamlessly due to the clean integration between functions and tuples.

The example above shows one approach to representing machine instructions in the backend of a compiler. Instead of manually defining classes that represent instructions with one operand, two operands, an operand and immediate, etc, the four main features of Virgil can be used to model all variants of instructions with just two class definitions. We first declare a base class for instructions (n1) with an abstract method (n2) that emits the instruction to some buffer. Then a parameterized subclass (n4) can be instantiated to each of the different variants, with the function that assembles the instruction and the operands themselves stored as fields (n5-6). The implementation of `emit` then simply calls the supplied function with the operands and the buffer (n9). Without writing any more classes, we can reuse methods from the assembler itself (n12-14) with various kinds of operands to create instruction variants. The `Instr` class in this case is like a super-closure: it not only closes over the unknown (variant) parts of the instruction, but can have more than one operation, such as iterating over the register operands of the instruction for register allocation.

If pattern matching is required on instructions, we can use a cascade of dynamic type casts and type queries (n15-20), or use a polymorphic matcher from (k1-13).

```
(n15) if (InstrOf<Reg>.?(i))
(n16)   printReg(InstrOf<Reg>.!(i));
(n17) if (InstrOf<(Reg, Reg)>.?(i))
(n18)   printRegReg(InstrOf<(Reg, Reg)>.!(i));
(n19) if (InstrOf<(Reg, int)>.?(i))
(n20)   printRegInt(InstrOf<Reg>.!(i));
```

This pattern requires all four features to work. Without classes and inheritance, we couldn't hide the actual representation type by having an unparameterized superclass, but would instead need some other form of existential quantification, a universal supertype, support for algebraic data types, etc. Without first-class functions, some other form of polymorphism would be required to supply the logic for assembling each instruction, e.g. subclassing the `Instr` class with a lot of redundant classes or using a switch or enumeration, all of which mean writing calls to assembler methods instead of just passing them around. Without tuples, any variant that consisted of more than one parameter would require an aggregate datastructure of some kind, such as another class. And finally, without type parameters, some universal representation would have been required to store all the different variants in some polymorphic location.

Tradeoffs. Emulating variants suffers from two main problems. The first problem is similar to emulating ad hoc polymorphism with dynamic casts; a loss of static checking in pattern matches (n15-20), where the programmer can too easily miss a case. The second problem is both an advantage and a disadvantage. This pattern allows an unbounded number of new variants to be created simply by instantiating the `InstrOf` class with new values and methods, which is an advantage for extensibility but is a disadvantage if the programmer wants to intentionally bound the set of variants that can be created.

3.6 Type Variance

Languages that contain both subtyping and type parameters face type variance problems. Java, C#, and Scala have various solutions that range from use-site annotations to declaration site annotations, wildcards, raw types, unsafe casts, or all of the above. Virgil only offers variance for tuples and function types; but surprisingly these two features reduce the need for other kinds of variance.

Returning to the cons list example (p1-p10), assume for the moment that the fields of this class are immutable, written only once in the constructor. Suppose we have types `Animal` and `Bat` with `Bat extends Animal`. We might intuitively consider `List` to be covariant, i.e. `List<Bat>` $<:$ `List<Animal>`. In the example

below (o2), we have a function `f` that performs an operation on every element in a list, and (o5) we have `b` of type `List<Bat>`. Virgil classes are invariant in their type parameters, so applying `f` to `b` is not well-typed (o6).

```
(o1) def g(a: Animal) { ... }
(o2) def f(list: List<Animal>) {
(o3)   for (l = list; l != null; l = l.tail) g(l.head);
(o4) }
(o5) var b: List<Bat>;
(o6) f(b);        // ERROR
(o7) apply(b, g); // OK
```

However, we can invert the control flow; instead we pass a function `g` of type `Animal -> void` which performs the operation for just one `Animal` to our utility function `apply` from (d6). Due to contra-variant function parameter types, `Animal -> void` $<:$ `Bat -> void`, and (o7) is well-typed.

As another example, consider again the `HashMap` class (h1-h8). If Virgil supported declaration-site variance annotations, the designer of this class might want to add a contra-variance annotation to `K` so that `HashMap<A, V>` $<:$ `HashMap<B, V>` for `B` $<:$ `A`. In another situation, a client might accept a `HashMap<K, V>` but only use the `get` method. In this case, it would be tempting to add variance at the use site, since the map could be considered co-variant in `V` and contra-variant in `K` at that use site. But notice that this is exactly the variance of the `get` method itself, and since function types *do* have variance, it would be better if the client simply accepted a function of type `K -> V`, and users could pass the `get` method of their `HashMap`, or some other function of their choosing. The client is thus more general, no longer dependent at all on the type of the object (if any) bound to the closure.

The prolific reuse of methods from objects radically simplifies libraries. For example, with help from tuples, functions, and variance, the call `a.apply(b.add)` copies the contents of `HashMap a` into `HashMap b`, without even writing a loop or burdening the library with another convenience method such as `addAll`. A more terse functional style transfers well to other kinds of datastructures, including arrays, sequences, maps, matrices, etc. All manner of familiar operators from functional libraries such as `map`, `fold`, `zip`, and `unzip` are easily defined, and surprisingly they don't require variance annotations.

Tradeoffs. The lack of variance for class types and the lack of either declaration or use site constraints on type parameters forces a different kind of programming style. This can make it hard to port code from existing languages that makes use of these features, necessitating both local refactorings and some redesign to rely more on a functional style. C# 2.0 debuted without variance annotations for generics, but they were later added. We believe this design choice was made because first-class functions were only timidly employed in their first inception as invariant, named delegate types. It may be that evolution of existing programs to a more functional style was deemed too radical at the time.

4. Implementation Issues

Language goals such as supporting system and low-level programming bring important efficiency considerations, posing challenges for implementing advanced features. This has driven Virgil's design throughout its earliest embodiment for microcontroller programming until today. The pressure to implement all of Virgil's constructs without undue reliance on implicitly allocating memory on the heap has been a substantial motivator for keeping its feature set conservative. Key in meeting these efficiency considerations is proper implementation of tuples and type parameters. This section describes the unique problems posed by tuples in Virgil and presents a solution based on a flattening strategy that guarantees that no implicit boxing or unboxing operations need to be inserted. In fact, Virgil's native implementation *never* allocates memory on the heap except when done explicitly by the programmer but still retains all the language flexibility described in previous sections.

4.1 Tuple Ambiguity

The uniform treatment of functions that accept multiple parameters as functions that accept a single tuple parameter is unfortunately not so uniform in implementation. The interaction between tuples and the other features of Virgil can give rise to implementation ambiguity in several different ways. In (p1-5) `f` and `g` are both of type `(int, int) -> void`, and the variable `x` could refer to either at runtime (p3). Two potential problems arise. The caller could legally pass either two integers (p4) or a tuple of two integers (p5). What calling convention would the compiler use at these call sites? What about within the body of `f` which expects two integers and `g` which expects a tuple? In this case, using tuple types to model multiple arguments creates a potential ambiguity because functions can be first-class.

```
(p1) def f(a: int, b: int) { ... }
(p2) def g(a: (int, int)) { ... }
(p3) var x = z ? f : g, t = (0, 1);
(p4) x(0, 1);   // ambiguous invocation
(p5) x(t);      // ambiguous invocation
(p6) def r<A>(a: A) { ... }
(p7) var y = z ? r<(int, int)> : f;
(p8) y(0, 2);   // ambiguous invocation
```

A parameterized function like the one defined in (p6) can also give rise to ambiguity. The method `r<(int, int)>` (p7) could be used anywhere a function of type `(int, int) -> void` is expected (p8). Ambiguity can also arise with simple method overriding. A class may declare a method (p11) with individual parameters that is overridden in a subclass (p14) with an implementation that has a single tuple parameter, leading to ambiguity at virtual method invocation sites (p17).

```
(p10) class A {
(p11)   def m(a: int, b: int) { ... }
(p12) }
(p13) class B extends A {
(p14)   def m(a: (int, int)) { ... }
(p15) }
(p16) var a = z ? A.new() : B.new();
(p17) a.m(1, 2); // ambiguous invocation
```

One solution is to place dynamic checks at invocation sites to determine whether the function to be invoked requires a tuple (i.e. boxed) or multiple scalars (i.e. unboxed) representation for its arguments. The Virgil interpreter uses this approach, but the checks are expensive. A small improvement can be made when compiling to a target machine by eliminating the dynamic checks at call sites where receiver methods can be statically determined. This is still suboptimal since too many such sites would give poor performance and might require allocating memory on the heap for tuples. Instead our compiler *normalizes* the program, rewriting all uses of tuples to eliminate such overhead. This ensures that all method calls pass scalar arguments regardless of whether the parameters were originally tuples or scalars, arrays and fields store dense scalar values, and operations on tuples never allocate on the heap, on both the JVM target and the native x86 targets.

4.2 Normalization

Tuples are ideal values: they are immutable, have no identity, and are not extensible. A program cannot discern implementation details such as whether a tuple is represented by a record or by individual scalars. *Normalization* is the process by which the Virgil

compiler converts all uses of tuples into uses of scalars, regardless of where they occur, including parameters, return values, local variables, array elements, fields, and elements inside other tuples. Variants of normalization (also known as scalar replacement of aggregates or flattening) have been employed in numerous functional language implementations, e.g. [3] and [36]. Reducing tuples to individual scalar values simplifies the program to a *normal form* where tuples no longer appear and exposes the individual values to classical compiler optimizations. It also eliminates the ambiguities demonstrated in the previous section; all normalized functions accept zero or more scalars and return zero or more scalars.

The compiler performs normalization on an internal representation of the program, but we can illustrate the basics of the algorithm with source-to-source translations. First, we see how the compiler treats local variables (q1) and parameters (q2) that are tuples, replacing them with multiple local variables (q1') or parameters (q2'). A use of a variable which was formerly a tuple (q3) is expanded to the normalized (q3') variables. An access of a tuple element (q4) is replaced with a reference to the corresponding normalized variable (q4'). Parameters (q6) and variables (q7) of type `void` are replaced with nothing (q6'-7') [7].

```
(q1) var b = ("hello", 15);
(q2) def m(a: (string, int)) { ... }
(q3) m(b);
(q4) m("goodbye", b.1);
(q5) m("cheers", (11, 22).0);
(q6) def f(v: void) { ... }
(q7) var t: void;
(q8) f(t);

  =>

(q1') var b0 = "hello", b1 = 15;
(q2') def m(a0: string, a1: int) { ... }
(q3') m(b0, b1);
(q4') m("goodbye", b1);
(q5') m("cheers", 11);
(q6') def f() { ... }
(q7')
(q8') f();
```

Normalization can potentially rewrite every expression in the program, whether it be a field access, array element access, array creation, method call, comparison, etc. Care is taken to avoid duplicating or reordering side-effects in expressions; this is facilitated by the compiler's internal SSA representation of the code. Fields are normalized by replacing with them with zero or more normalized fields; field accesses are normalized into multiple reads or writes as appropriate. Array element accesses require similar normalization. Depending on the compilation target, an array of tuples may be normalized to an array whose elements store tuple elements next to each other in memory, or to be multiple arrays, each of which stores one element of the tuple. Returns of a tuple become a return of multiple values, utilizing multiple return registers on native targets.

There are several corner cases that are not immediately obvious. Accesses to fields of type `void` are simply removed and replaced with null checks; this ensures that a null dereference always throws an exception, regardless of the field's type. Similarly, arrays of `void` require no storage for their elements, but accesses are dutifully bounds checked. The JVM does not support arrays of `void` so such arrays are represented with a `java.lang.Integer` object which distinguishes between `null` and an array with a length. On native targets a two-word array object that only stores the array length is sufficient. The JVM also does not support returning multiple values from a method call. In this case the Virgil compiler inserts a tuple creation at the return site and deconstructs the tuple return value after call sites.

Normalization is modular; normalizing a method's body does not require knowledge of the call sites, nor do call sites require knowledge of the methods which they call. However, normalization relies on knowing the types of all expressions and methods; this is only possible if type arguments have been substituted for type parameters, which is the subject of the next section.

Tradeoffs. Normalization removes all boxing related to tuples, achieving Virgil's implementation goal of avoiding implicit memory allocations. For small tuples, normalization has much better performance than boxing, but large tuples might actually perform better if allocated on the heap, depending on how the program uses them; in a program that uses large tuples, reading and writing pointers to objects on the heap may be cheaper than reading and writing the many tuple elements individually.

4.3 Monomorphization

In Virgil, runtime information about type arguments is needed in some operations, such as allocating or accessing an array of polymorphic type, performing a dynamic type cast or type query involving polymorphic types, or accessing a field of a polymorphic type. We saw dynamic type casts and queries were key in allowing several patterns such as the polymorphic matcher. In the Virgil interpreter, type arguments are passed as invisible arguments to polymorphic function calls and stored as type information within objects, arrays and closures. Even with lazy evaluation of the type expressions that represent type arguments, this exacts a considerable runtime cost.

The Virgil compiler instead employs *monomorphization*, where a specialized version of each polymorphic class or method is generated for each distinct assignment of type arguments to type parameters. Thus an object of type `List<(int, int)>` has a different representation than `List<byte>`, and similarly the method `id<int>` has a distinct representation from `id<byte>`. Once the representation of all classes and methods is obtained through specialization, no type parameters appear in the program. The resulting code is therefore monomorphic and can use the most efficient, dense representations. Monomorphization affords the opportunity for whole-program normalization which eliminates all tuples from the program, and therefore guarantees that programs can be compiled to a form where implicit memory allocations on the heap are not required. This is key for some systems programming problems where some code must be able to run without interruption from the garbage collector, and thus may not allocate memory [8].

Tradeoffs. The implementation of parametric polymorphism is full of tradeoffs and has been the subject of much research which we cannot fully summarize in the space alotted. The main drawback to monomorphization is that polymorphic code can be duplicated repeatedly, perhaps exponentially, even infinitely if the language allows polymorphic recursion, which Virgil disallows [9]. The result can be code explosion where polymorphic methods are duplicated repeatedly. In our experience, this has not been an issue in real programs. Other language implementations use monomorphization as well, including MLton [35] which employs whole-program monomorphization quite sucessfully for ML, and C++ does complete template expansion, which is similar to monomorphization. Some programming guidelines can reduce code explo-

[7] Normally programmers do not explicitly use variables of type `void`, but they often appear when expanding polymorphic code.

[8] This has been a constant source of problems in system code written in Java like Jikes RVM and Maxine VM, especially when writing the garbage collectors themselves in Java. We wrote our GC in Virgil without encountering these problems.

sion [33]. Another drawback is that monomorphization also requires the whole program to be available, or it must fall back on a universal representation when instantiations are not known across compilation unit boundaries.

One alternative to monomorphization is to remove type parameters from the program by simply *erasing* them [22]. This is only possible if all values have a universal representation (e.g. Scala), the compiler forces a universal representation through boxing (most functional language implementations), or by disallowing some types as type arguments (e.g. Java). With erasure, some polymorphic operations can only be implemented by passing a representation of the type arguments at runtime [15]. Both Scala and Java perform type erasure, but *do not* pass a runtime representation of type parameters, meaning that some polymorphic operations, such as allocating an array of a polymorphic type or casting a polymorphic type, are simply not allowed. Simply put, type erasure cannot work for Virgil. Also note that both Scala and Java target the JVM, which does not support type parameters; dynamic type casts must be inserted to satisfy the JVM's bytecode verifier. In contrast, monomorphization requires no casts to be inserted.

To be fair, Scala recently attempted to address the problem of missing type information with *manifests*, which are indeed a representation of type parameters, but at first they were manual and not fully integrated with the rest of the language. Scala 2.10 uses type tags which can be passed as implicit parameters, subsuming the role of manifests. However, they require the programmer to request a type tag be passed to a class or method in order to perform some polymorphic operations. Scala also recently experimented with annotations to allow programmers to manually instruct the compiler to perform specialization [7].

In Java, a programmer can explicitly pass instances of `java.lang.Class` along with type parameters to act as the missing type information, but only one level deep: class objects do not contain information about type arguments to parameterized classes.

C# coined the term *reified generics*; the semantics are closest to Virgil's type parameters. C# mainly targets the CLR which provides support for type parameters in bytecode, and the virtual machine performs specialization on demand [18], sharing identical machine code for some specializations. When one can assume a VM with a dynamic compiler, late specialization has strong advantages: it avoids the need for the whole program at compile time and reduces code explosion. However it is not an option for compiling Virgil statically. Other differences are that C# has no tuples, does not allow `void` as a type argument, and does not allow casts between, e.g., a type parameter and a concrete type.

Most functional language implementations use a mix of specialization and boxing, where the overhead of boxing is removed wherever the compiler has full knowledge of the representation of values [35], even for tuples [36]. There are more complicated schemes for implementing parametric polymorphism, such as intensional type analysis [15] which relies on representation passing. In some sense, this is the implementation adopted by the Virgil interpreter as discussed before.

5. Experience

The balance of features in Virgil has proven to be quite practical in writing small to medium-scale programs. Our experience in writing over 100,000 lines of Virgil code has motivated continual refinement of the language. A self-hosted, fully-bootstrapped compiler written in Virgil is the primary implementation. It features compilation to both the JVM and native x86 targets, includes a standard suite of intraprocedural compiler optimizations, array bounds check elimination, sophisticated instruction selection, a linear-scan register allocator, whole-program optimizations such as monomorphization, as well as sophisticated dead code and dead data elimination [31]. On native targets Virgil provides a precise semi-space garbage collector (also written in Virgil), a runtime system (written in Virgil) and direct access to kernel system calls. The compiler also offers ancillary tools such as a full interpreter, a profiler, and a code coverage tool. Despite its small size (just 25,000 lines of code), the Virgil compiler generates decent quality machine code and compiles very fast. Virgil III is open source and freely available at `http://code.google.com/p/virgil`.

We also wrote a small number of applications, ported several benchmarks, and created a vast battery of test programs. We found, like many Scala programmers, that the free intermixing of functions with objects brings a new kind of expressiveness beyond pure object-oriented and pure functional styles. Objects work well for encapsulating state and related methods, while functions excel at small-scale reuse, such as that found in operations like `map`, `fold`, and `sort`. Our experience has been that unrestricted type arguments and cheap access to tuples have markedly increased expressiveness beyond simply throwing objects and functions together into the same language. For example, the ability to quickly define a list of tuples and then sort them by, say, the first element, has been very convenient for rapid prototyping.

Every pattern described in this paper has made it into practical use. The emulation of abstract data types has direct applicability in the design of reusable data structures like maps; the emulation of ad hoc polymorphism has been useful in printing and logging; and the emulation of variant types is used the Virgil compiler backend to represent and manipulate machine instructions without requiring a complete duplication of the x86 assembler's interface, nor the definition of many small variant classes.

6. Conclusion

Virgil brings a new kind of harmony between classes, functions, tuples, and type parameters in a statically-typed setting. The whole is more than the sum of its parts; these features complement each other in subtle and powerful ways that reduce the need for more language features, including complex patterns of inheritance, type variance annotations, and a universal super type. Perhaps most interestingly, multi-paradigm design patterns can be built that allow emulation of features not directly supported in the language, including abstract data types, ad-hoc polymorphism, and variant types. Each emulation presents some tradeoffs, and balancing the inclusion of new features versus those tradeoffs is ongoing challenge in the art of language design; a challenge which we by no means claim to have conquered. Moreover, we do not argue that any of the emulated features should necessarily be excluded from this or any other language; rather that the use of patterns can give designers more time to prioritize new language features and ensure they compose well.

6.1 Future Work

Virgil lacks many advanced features and there are still rough edges. For example, we find our own emulation of `printf`-like cases still somewhat cumbersome; a well-developed pattern matching system coupled with a kind of *varargs* mechanism could greatly increase expressive power without upsetting the existing design. Our experience programming in the language suggests that enumerated types are of high priority, and certainly other languages offer excellent examples of potential design.

Virgil lacks a module system, which can offer representation independence and separate compilation units for modularity on a larger scale. Higher-rank polymorphism (i.e. polymorphism over type constructors) allows more forms of generic programming [24]. We believe both extensions to be important in the future, and like

[9] Virgil disallows polymorphic recursion but it is not currently enforced.

the other features mentioned, Virgil's core design has done nothing to preclude them.

We mentioned issues with full monomorphization in section 4.3. We continually track the amount of code expansion due to specialization for the compiler and applications we have already built. We consider it an important goal to avoid code explosion, and a hybrid approach to monomorphization is the subject of current research. See for example [29].

Acknowledgments

Thanks to Alex Warth, Jan-Willem Maessen, Marek Gilbert, and Jens Palsberg for comments on early drafts of this paper.

References

[1] E. Allen and R. Cartwright. The case for run-time types in generic Java. *In Proceedings of the 1st Conference on the Principles and Practice of Programming in Java (PPPJ '02)*. Dublin, Ireland. Jun 2002.

[2] C. Baker-Finch, K. Glynn, and S. P. Jones. Constructed product result analysis for Haskell. *In Journal of Functional Programming (JFP)*, Volume 14, Issue 2. Mar 2004.

[3] L. Bergstrom and J. Reppy. Arity raising in Manticore. *In Proceedings of the 21st International Conference on Implementation and Application of Functional Languages (IFL '09)*. South Orange, NJ. Sep 2009.

[4] N. H. Cohen. Type-extension type tests can be performed in constant time. *In ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 13, Issue 4. Oct 1991.

[5] J. Dean, D. Grove and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Aarhus, Denmark. Aug 1995.

[6] A. Diwan, K. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *In ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 23, Issue 1. Jan 2001.

[7] I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. *In Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. Genova, Italy. Jul 2009.

[8] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C# generics. *In Proceedings of the 20th Annual European Conference on Object-Oriented Programming (ECOOP '06)*. Nantes, France. Jul 2006.

[9] D. Frampton, S. M. Blackburn, P. Cheng , R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. *In Proceedings of the International Conference on Virtual Execution Environments (VEE '09)*. Washington, DC. Mar 2009.

[10] K. Faxén. Representation analysis for coercion placement. *In Proceedings of the 9th International Symposium on Static Analysis (SAS '02)*. Madrid, Spain. Sep 2002.

[11] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. Boston, MA. 1995.

[12] A. Goldberg and D. Robson. Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc. Boston, MA. 1983.

[13] Guava: Google Core Libraries for Java 1.5+ http://code.google.com/p/guava-libraries/

[14] J. J. Hallett, V. Luchangco, S. Ryu and G. L. Steele Jr. Integrating coercion with subtyping and multiple dispatch. *In Science of Computer Programming*, Volume 75, Issue 9. 2010.

[15] R. Harper and G. Morrisett. Compiling polymorphism with intensional type analysis. *In Proceedings of the 22nd Symposium on Principles of Programming Languages (POPL '95)*. San Francisco, CA. Jan 1995.

[16] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. *In Proceedings of the 16th Annual European Conference on Object-Oriented Programming (ECOOP '02)*. Malaga, Spain. June 2002.

[17] JSR 335: Lambda expressions for the Java(TM) programming language. http://jcp.org/en/jsr/detail?id=335

[18] A. J. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. *In Proceedings of the 23rd Conference on Programming Language Design and Implementation (PLDI 2001)*. Snowbird, UT. Jun 2001.

[19] A. J. Kennedy and D. Syme. Combining generics, pre-compilation and sharing between software-based processes. Jan 2004.

[20] The Maxine Virtual Machine. http://labs.oracle.com/projects/maxine/

[21] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *In IBM Systems Journal*, Volume 39, Issue 1. Jan 2000.

[22] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. *In Proceedings of the 24th Symposium on Principles of Programming Languages (POPL '97)*. Paris, France. Jan 1997.

[23] M. Odersky, M. Sulzmann and M. Wehr. Type inference with constrained types. *In Proceedigns of the 4th International Workshop on Foundations of Object-Oriented Programming (FOOL '97)*. Paris, France. Jan 1997.

[24] B. Oliveira and J. Gibbons. Scala for generic programmers. *In Proceedings of the Workshop on Generic Programming (WGP '08)*. Victoria, BC. Sep 2008.

[25] J. Palsberg. Type-based analysis and applications. *In Proceedings of the Workshop on Program Analysis for Software Tools (PASTE 01)*. Snowbird, UT. Jun 2001.

[26] B. C. Pierce. Advanced topics in types and programming languages. MIT Press. 2004.

[27] B. C. Pierce and D. N. Turner. Local type inference. *In Proceedings of the 25th Symposium on Principles of Programming Languages (POPL '98)*. San Diego, California. Jan 1998.

[28] C. van Reeuwijk and H. J. Sips. Adding tuples to Java: a study in lightweight data structures. *In Proceedings of the Joint Java Grande/ISCOPE Conference (JGI '02)*. Seattle, Washington. Nov 2002.

[29] O. Sallenave and R. Ducournau. Lightweight generics in embedded systems through static analysis. *In Languages, Compilers, Tools and Theory for Embedded Systems (LCTES '12)*. Beijing, China. Jun 2012.

[30] D. Smith and R. Cartwright. Java type inference is broken: can we fix it? *In Proceedings of the 23rd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '08)*. Nashville, Tennesee. Oct 2008.

[31] B. L. Titzer. Virgil: Objects on the head of a pin. *In Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. Portland, Oregon. Oct 2006.

[32] B. L. Titzer and J. Palsberg. Vertical object layout and compression for fixed heaps. *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '07)*. Salzburg, Austria. Oct 2007.

[33] D. Tsafrir, R. W. Wisniewski, D. F. Bacon, and B. Stroustrup. Minimizing dependencies within generic classes for faster and smaller programs. *In Proceedings of the 24th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '09)*. Orlando, FL. Oct 2009.

[34] J. Vitek and B. Bokowski. Confined types. *In Proceedings of the 14th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. Denver, CO. Oct 1999.

[35] S. Weeks. Whole-program compilation in MLton. *In Proceedings of the 2006 workshop on ML*. Portland, OR. Sep 2006. Slides available: http://mlton.org/pages/References/attachments/060916-mlton.pdf

[36] L. Ziarek, S. Weeks and S. Jagannathan. Flattening tuples in an SSA intermediate representation. *In Journal of Higher-Order and Symbolic Computation*, Volume 21, Issue 3. Sept 2008.