Virgil on the JVM:

The language, its implementation, and insights for the JVM

Ben L. Titzer

Google

Background

- Virgil I and II for microcontrollers (2005-2007)
  - Objects, generics, delegates
  - No dynamic memory allocation
  - Space optimizations
- Virgil III: Time to grow up! (2008-)
  - Why not write Systems Software with it?
  - Bootstrap: new Virgil III compiler (written in Virgil III) running on old interpreter (written in Java)
- Not (!) official Google project
  - My 20% project for now

Language Evolution

- Virgil I - 2006
  - Interpreter/compiler (to C) written in Java
- Virgil II - 2007 [+generics]
  - Interpreter/compiler (to C) written in Java
- Virgil III-0 [+syntax, +tuples, +variance]
  - Interpreter written in Java
- Virgil III-1 [+syntax, +polymorphism]
  - Interpreter/compiler (to JVM) written in Virgil III-0
  - Bootstrap compiler (on JVM) in Fall 2010
- Virgil III-2 [+modules, +read-only arrays]
  - After bootstrap...

Motivation of this Talk

- Discuss unique language balance
  - Constraints of domain influencing feature set
  - Benefit of hindsight
- Share experience with JVM bootstrap
  - Unique challenges for Virgil, but
  - Likely to be indicative for new languages
- Feedback on language ideas
  - No one has written code in Virgil III besides me (research on Virgil II continues at UCLA)

Why Bootstrap on the JVM?

- Ubiquitous:
  - Can ship a JAR of the Virgil compiler
- Familiar:
  - Personal experience with JVM internals
  - Java bytecode and platform are well understood
- Debuggable:
  - Type safe, good bytecode verifier, debugging API
- It's supposed to be the best!
  - Best JIT, GC, and profiling tools

Virgil Basics

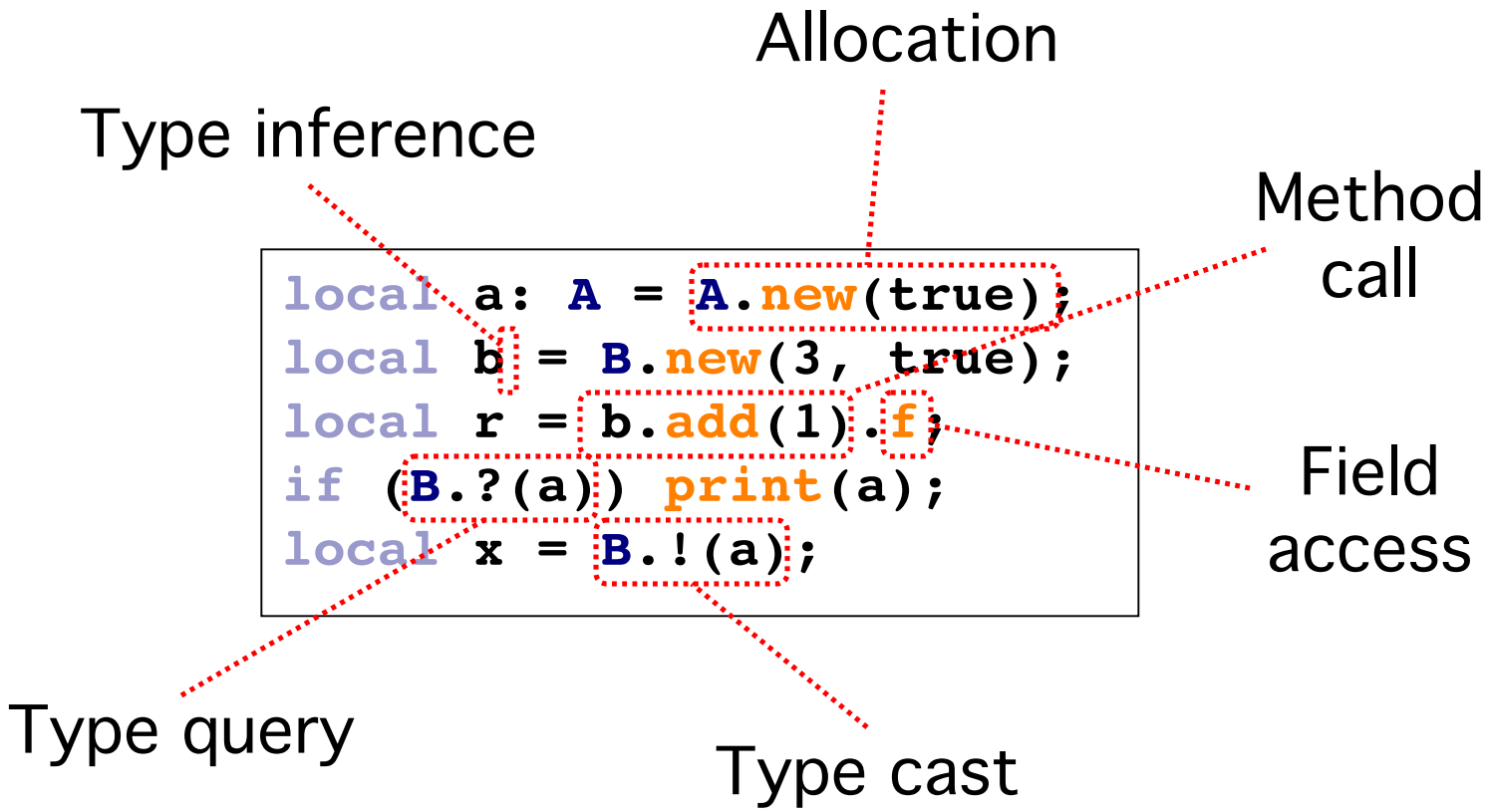
- Strong static type safety
- Familiar core language
  - `int`, `bool`, `char`
  - Bounds-checked arrays `Array<T>` for any `T`
- Blend of object-oriented and functional
  - Classes, objects, components (singletons)
  - Delegates (first class functions)
- Tuples of types `A`, `B`, ... as `(A, B, ...)`
- Full parametric polymorphism

Class Declarations

```
class A {
    field f: int;
    new(f) { }
    method add(a: int) -> A {
        f = f + a;
        return this;
    }
}
class B extends A {
    value g: bool;
    new(f: int, g) : super(f) { }
    method add(a: int) -> B {
        if (g) f = f + a;
        return this;
    }
}
```

- Single inheritance
- No implicit superclass
- No interfaces
- Automatic initialization of read-only fields
- Type variance in overriding (co-variant return types, contra-variant parameter types)

Using Classes



Creating Delegates

```
class A {
   field f: int;
   new(f) {}
   method add(a: int) -> A {}
}
class B extends A {
   new(f: int, g: bool) : super(f) {}
   method foo() {}
}

local b = B.new(10, false);
b.add          // int -> A
A.add          // (A, int) -> A
A.new          // int -> A
B.new          // (int, bool) -> B
B.!            // A -> B
B.?            // A -> bool
B.==           // (B, B) -> bool
```

- Any method or field of a class can be a first class function
- Syntax looks like method invocation without arguments
- New, query, cast, and compare are delegates
- Syntax precludes method overloading

Using Delegates

```
class B extends A {
   method getf() -> int -> int {}
}

B.new().getf()(0);
B.!(a).getf()(2);

int.+   // (int, int) -> int
int.==  // (int, int) -> bool
bool.!  // bool -> bool
bool.&& // (bool, bool) -> bool

method f(b: B) -> A;
method g(a: A) -> B;

local x: B -> A = f;
local y: B -> A = g;
```

- Standard invocation syntax chains with field and method accesses left to right
- All basic operators can be delegates
- Function types are co-variant in return type, contra-variant in parameter types

Tuples

```
local a: (int, int) = (1, 0);
local b = (true, 1, false);
local c = b.0;
local d = a.0 + a.1;
local e: (int, int) -> (int, int);
local f: (int, int, (bool, int));
local g: (int, void) = (0, ());

method add1(a: int, b: int) -> int {}
method add2(a: (int, int)) -> int {}

local x: (int, int) -> int = add1;
local y: (int, int) -> (int, int);

if ((0, 1) == (0, 1)) { }
```

- Can create tuples of arbitrary types, even void
- A tuple type can appear anywhere a regular type can
- Doesn't require allocating memory
- Useful for returning multiple values, but also models multi-argument methods
- Comparison is element-by-element

Type Parameters

```
class List<T> {
   value head: T, tail: List<T>;
   new(head, tail) {}
}

method cons<A>(h: A, l: List<A>)
         -> List<A> {
   return List<A>.new(h, l);
}

local x = cons<int>(0, null);

local y = cons(0, null);
local z = List.new(0, null);

local a: List<void>;
local b: List<(int, char)>;
local c: List<char -> int>;
local c: List<List<char> -> int>;
```

- Classes and methods can have type parameters (generics)
- Explicitly parameterized type in expression
- Type inference allows leaving them out nearly always
- *any* type can be used as a type argument

Advanced Type Parameters

```
class List<T> {
   value head: T, tail: List<T>;
   new(head, tail) {}
}
method map<A, B>(x: List<A>, f: A -> B) -> List<B> {
   if (list == null) return null;
   return List.new(f(x.head), map(list.tail, f));
}
method filter<A, B>(x: List<A>, s: A -> bool, f: A -> B) -> List<B>{
   if (list == null || !s(list.head)) return null;
   return List.new(f(x.head), filter(list.tail, s, f));
}

local cList: List<C>;
local dList1 = map(cList, D.!);
local dList2 = filter(cList, D.?, D.!);
```

It is 100%,
absolutely,
totally,
inescapably,
unavoidably,
completely
necessary
to have
FULL PARAMETRIC TYPES

Unfortunate Java Mistakes

- One afternoon I...
  - Traipsed into the JDK looking for blood
  - Found roughly 20,500 lines of code
  - That probably didn't need to exist...
  - Because of
- Partial Genericity
  - Can't be generic over primitives in Java
  - Examples:
    - java.util.Arrays.{sort, binarySearch, equals, fill, copyOf, copyOfRange}, java.lang.reflect.Array.{getBoolean, getByte, etc}, java.lang.reflect.Field.{getBoolean, getByte, etc}, java.util.StringBuilder{append, insert}, Unsafe API, IO, NIO, BigInteger, BigDecimal
  - And the list goes on....

Components by Example

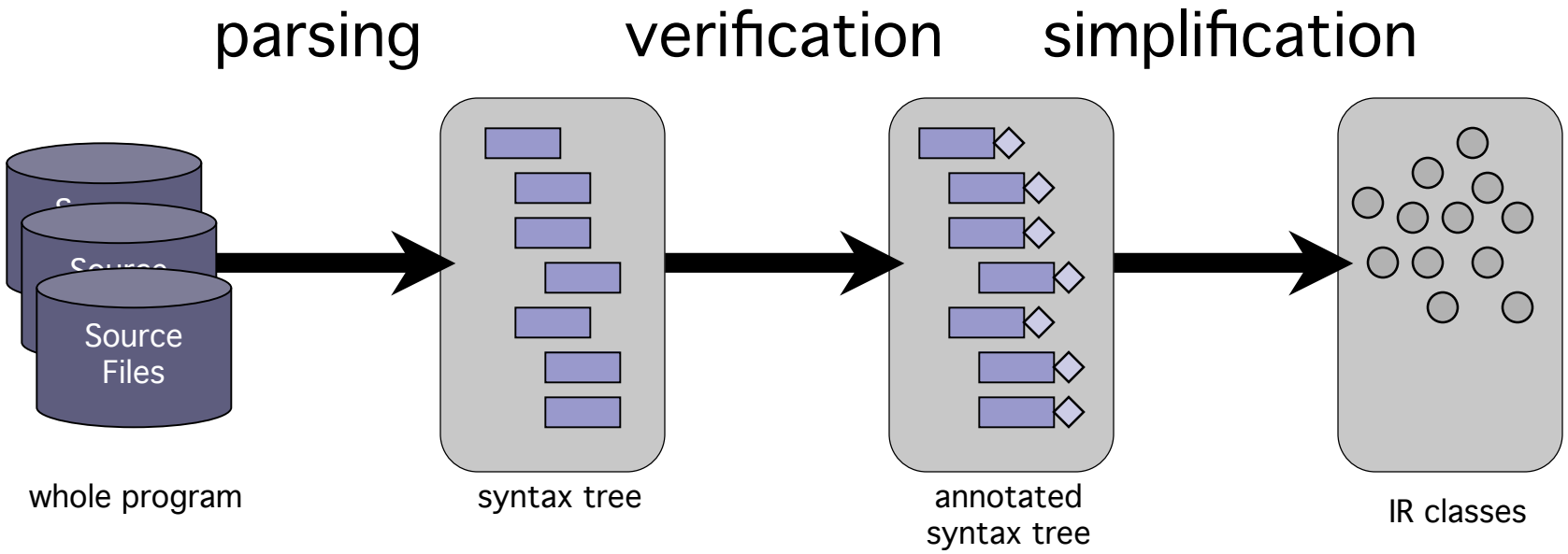
```
component C {
    value x: bool = true;
    field y: int;
    new() {
        // initialization code
        . . .
    }
    method add() {
        . . .
    }
}

local t = C.y;
local u = C.add();
local v = C.add;
```

- Classes cannot have static methods or fields
- Components serve as singletons
- Names globally available
- Initialization executed at compile time
- Fields serve as roots of saved object graph
- Methods available as delegates

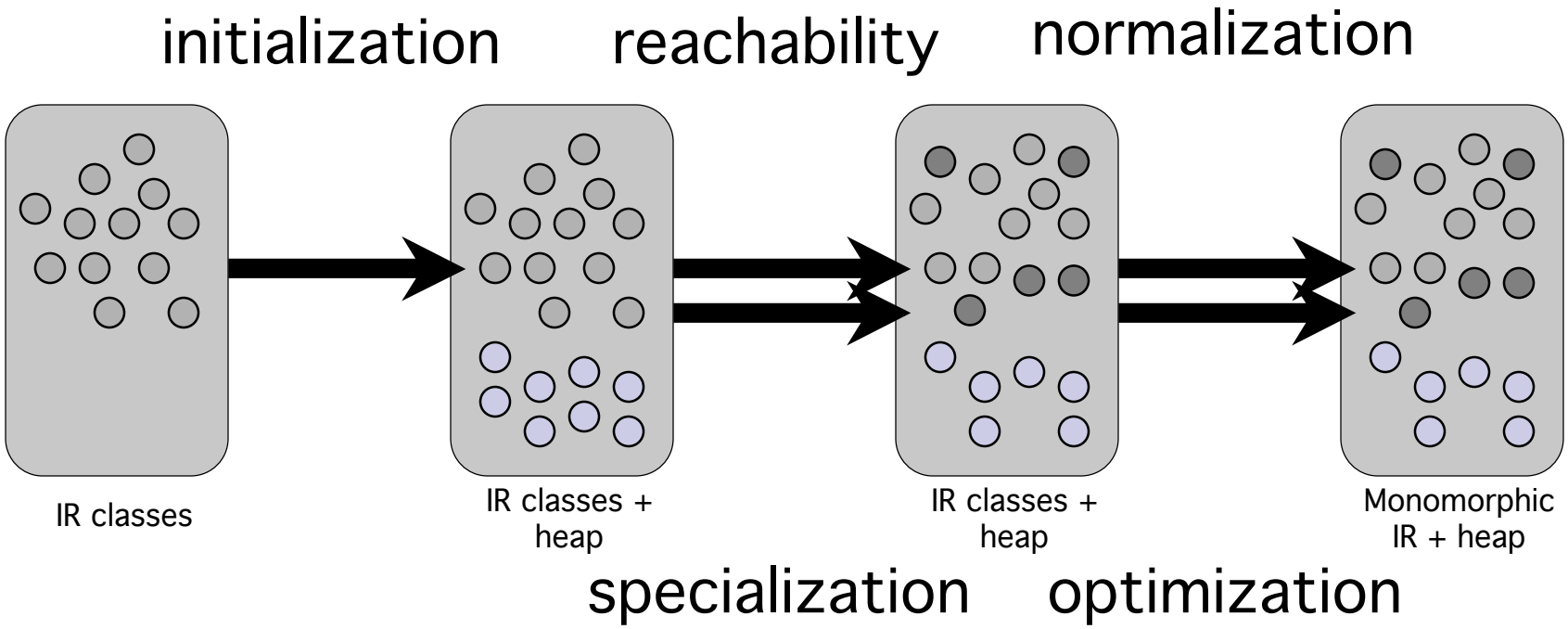
Compilation Model

- Static whole-program compilation
  - Microcontrollers, embedded systems, kernels
  - Whole-module compilation for libraries
- Parsing, typechecking, verification
- Component initialization
- Reachability
  - Polymorphic specialization
  - Tuple normalization
- Code emission
  - JAR or executable

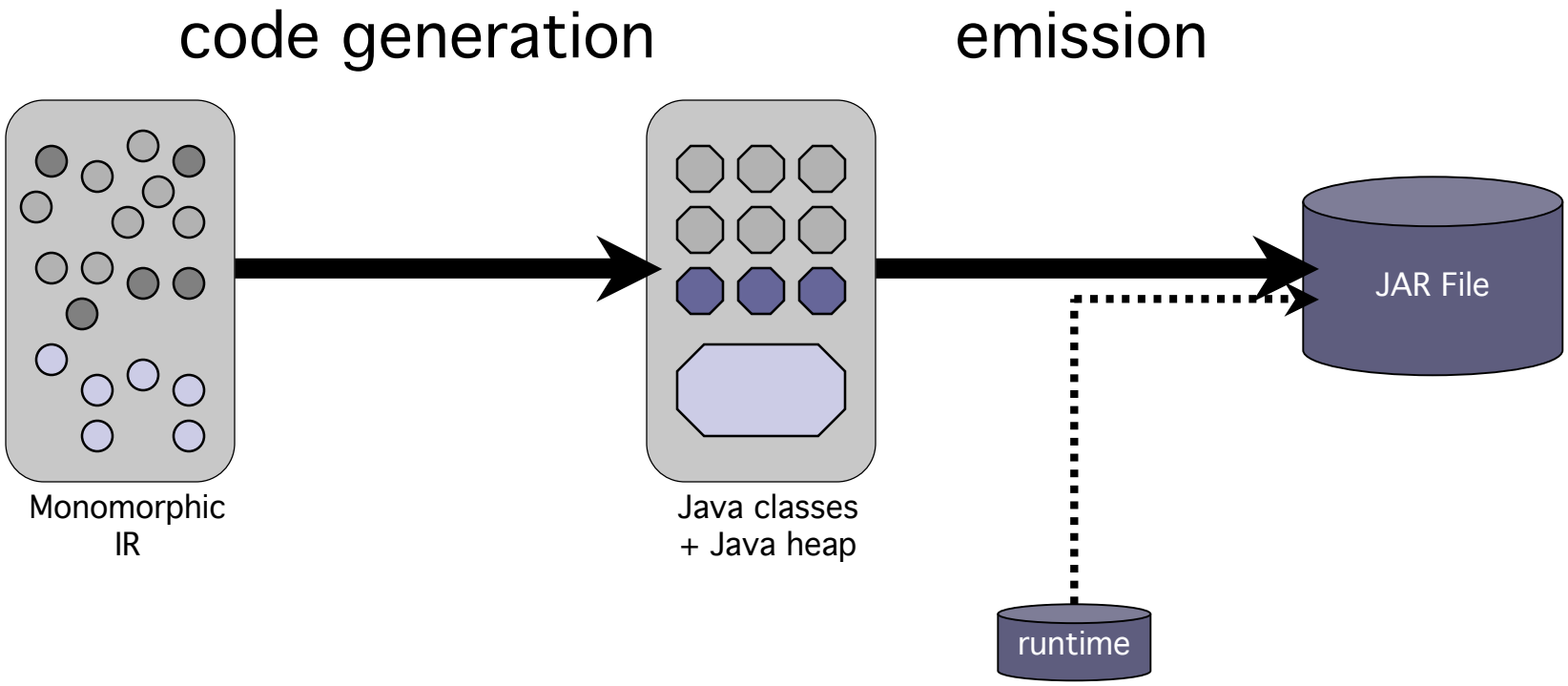
Compiler Frontend



Compiler Middle



Compiler Backend



Basic Implementation Strategy

- One JVM class per Virgil class
  - Parameterized classes are specialized as needed
- One JVM method per Virgil method
  - Parameterized methods are specialized as needed
  - Bridge methods generated for variant overriding
- Generate code to build initialized heap of JVM objects
  - Tricky for cyclic data structures, delegates
- RMA [OOPSLA '06] removes unreachable code and data
  - Only live fields, objects, methods are visited
  - Also used for devirtualization
  - Allows for sophisticated space optimizations [LCTES '07]

Filling in the corners

- What about that pesky void?
  - Arrays of void, parameters or fields of void
- Implementing Tuples
  - To box or not to box?
- Implementing Delegates
  - Do method handles help?
- Variance in function types
  - JVM requires exact signature matching

Array<void>

```
method map<A, B>(a: Array<A>, f: A -> B)
         -> Array<B> {
   local r = Array<B>.new(a.length);
   . . .
   return r;
}

method printFlower(f: Flower) { . . . }
local array: Array<Flower>;
map(array, printFlower); // Array<void>
```

- Holes in languages appear when you least expect
- Could work around this case, but what about the next one?
- Filling in the holes makes the language more “seamless” and usable
- Array<void> represented as java.lang.Integer on JVM

Tuple Normalization

```
class A {
   method add(a: int, b: int) -> int { }
}
class B extends A {
   method add(a: (int, int)) -> int { }
}

// if boxed, would generate
.class V3C_A
  .method add(II)I

.class V3C_B
  .method add(V3T_$Pint$Cint$Q)I

// but instead, after normalization
.class V3C_B
  .method add(II)I
```

- Tuple types used to represent multi-argument functions in the type system
- For maximum reuse, a single-tuple method and a multi-argument method are considered interchangeable
- Requires rewriting program to remove tuples when targetting JVM
- Must expand fields, parameters, locals, array elements

Delegate Implementation

```
class A {
   method add(a: int, b: int) -> int { }
}
local f = A.new().add;

// base class of all delegates
.class V3D
  .field methodId I;
  .field object Ljava/lang/Object;;
  .method equals(Ljava/lang/Object;)Z

// class for "(int, int) -> int"
.class V3F_$Pint$Cint$Q$Aint extends V3D
  .method invoke(II)I

// delegate for A.add
.class V3D_A$Dadd extends V3F_$Pint$Cint$Q$Aint
  .method invoke(II)I
```

- One base class for all delegates (to share object and method fields and comparison operator)
- One class per delegate signature
- One class per method used as a delegate
- Adapter delegates needed for type variance [explain]

Initial Heap

```
class A {
   field id: int = 100;
   method add(a: int, b: int) -> int { }
}
component C {
   field root: A = A.new();
}

.class V3K_C
   .static .field root LV3C_A;;
   .static .method <clinit>()V
       getstatic V3H_C.o1;
       putstatic root;

.class V3H_C
   .static .field o1 LV3C_A;;
   .static .method <clinit>()V
      . . . // code to initialize heap
```

- Compiler generates special "heap" class for entire program
- Components have <clinit> methods that initialize their fields
- Heap class has fields that reference the roots of the object graph
- Specialized code builds object graph in the <clinit> of the heap class

Compiler Status

- Frontend
  - Parser: 366 of 366 tests pass
  - Verifier: 859 / 875 tests pass
- Interpreter
  - Complete: 841 / 841 tests pass
- JVM Backend
  - Missing normalization: 769 / 841 tests pass
- Starting 32-bit (x86/ARM) backend
- Compiler bootstrap soon

JVM Weaknesses

- No value types (tuples)
  - Compensate by erasing all tuples (except return sites)
- No VM-level generics
  - Compensate by performing complete specialization (partial specialization in future?)
- Not compositional
  - Special case of void in several situations
- No function types
  - Boxing and delegation are required

JVM Startup Issues

- Initialization of basic “Java”
  - ~500 classes, 100k’s bytecodes, 260 ms
  - Almost none of which is relevant to Virgil
- Constructing initial heap
  - Currently done with specialized bytecode
    - One giant, straight-line <clinit> method
    - Size limit of 64k per method
  - Use serialization format or custom format?

Conclusion

- Initially appeared easy
  - One-to-one Virgil to Java breaks down
  - Many adapter classes / methods
  - Must erase all tuples, but
    - Boxing and unboxing tuples still required
  - Method handles to the rescue?
    - But what about older / embedded VMs
    - I really believe a new type constructor is necessary
- I wish it was easier!
  - The J in JVM is deeply entrenched

Suggested Questions

- How does Virgil relate to...
  - Go, Scala, Java, C#, etc
- Why not target...
  - LLVM, CLR, your own backend, etc
- How do you...
  - Do ad-hoc polymorphism?
  - Do printf?
  - Simulate raw types?
  - Define HashMaps?

HashMap Example

```
class HashMap<K, V> {
   value hash: K -> int;
   value equal: (K, K) -> bool;
   new(hash, equal) { }
   method get(k: K) -> V;
   method set(k: K, v: V);
}
local m1 = HashMap.new(int.!, int.==);
class MyKey {
   value hashCode: int = . . .;
   method equals(o: MyKey) -> bool;
}
local m2 = HashMap.new(
         MyKey.hashCode,
         MyKey.equals);
local m3 = HashMap.new(
         MyKey.hashCode,
         MyKey.==);
```

- Uses delegates instead of interface or methods on Object
- Can use any type as key or value
- Unbound delegates allow easy use of == or custom methods for equality

Ad-hoc Polymorphism

```
class StringBuffer {
   method append<T>(v: T) -> StringBuffer {
      if (int.?(v)) appendInt(int.!(v));
      if (char.?(v)) appendChar(char.!(v));
      if (bool.?(v)) appendBool(bool.!(v));
      . . .
      return this;
   }
   method appendInt(v: int) { }
   method appendChar(v: char) { }
   method appendBool(v: bool) { }
}

StringBuffer.new().append(0).append('1');
```

- Parameterize method instead of overloading
- Use dynamic checks and casts of types
- Method will be specialized with each instantiation
- Queries and casts will be statically folded when the method is specialized to its type parameters

Hiding Type Parameters

```
class Any {
   method as<T>() -> T {
      if (Some<T>.?(this))
         return Some<T>.!(this).val;
      }
      local none: T;
      return none;
   }
   method is<T>() {
      return Some<T>.?(this);
   }
}
class Some<T> extends Any {
   value val: T;
   new(val) { }
}
```

- Class inheritance provides for a form of existential types
- Superclass “hides” type parameter of subclass
- Having complete type information at runtime (through polymorphic specialization, e.g.) allows queries and casting

Interface-Adapter pattern

```
class I {
   method meth(i: int) {}
}
class I_Adapter<T> extends I {
   value receiver: T;
   value m: (T, int) -> void;
   new(receiver, m) { }
   method meth(i: int) {
      m(receiver, i);
   }
}

method m(x: I) {
   x.meth(0);
}

class X {
   method foo(i: int) { }
}

m(I_Adapter.new(x, X.foo));
```

- Type-parameter hiding and delegates allow a weird kind of flexible interface mechanism
- Target interface required by method
- Adapter interface implementation
- Source class
- Source class adapter uses unbound delegate to implement interface method

Other Questions?