

The Indie Programming Language

Bret Taylor and Jim Norris

{btaylor,jcn}@cs.stanford.edu

1 Introduction

The Indie programming language is designed to be a safe alternative to C and C++. It is designed to have the “feel” of Java, but with features that make it easy to analyze and compile to native code efficiently. Indie’s unique type system makes it easy to create generic, reliable components without incurring significant runtime type safety costs.

The most prominent features of Indie are:

- Indie is compiled to native code. It does not have a language runtime like the .NET languages and Java.
- Indie is completely type safe, and this is enforced statically by the Indie compiler. This means you will not necessarily incur a runtime cost for type safety as you do in Java and the .NET languages.
- Indie has a novel type system that eliminates the need for class casts and `null` values. This has a side effect of eliminating the need for `ClassCastException` and `NullPointerException`.
- Indie supports generic types. Indie’s generic types are similar in spirit to C++ templates, but Indie generic types are type safe.
- Indie uses garbage collection. Since Indie does not have a language runtime, garbage collection is integrated with compiled Indie programs.
- Indie enforces object-oriented design patterns by integrating some of the most common design patterns into the language itself. For example, `singleton` is a keyword in Indie.

```

singleton class HelloApplication extends Application {
    method start(args : string[]) : int
    {
        Console.println("Hello, world");
        return 0;
    }
}

```

Figure 1: ‘Hello, world’ in Indie.

The most distinctive part of Indie is its type system. Indie’s type system motivates a slightly new style of programming that may be strange to Java and C++ programmers, but we argue that it makes Indie programs both more efficient and less error prone.

Section 2 describes the basic syntax of the language and gives some remedial examples. Section 3 specifies Indie’s unique type system. Section 4 describes inheritance and method calls and how they relate to Indie’s type system. Section 5 describes Indie’s mechanism for generic types. Finally, Section 6 describes some of the remaining features of the language, like Indie’s ability to integrate with native C code.

Section 7 describes how our compiler implements the Indie type system in practice, making use of full-program analysis for performance optimizations.

2 Basic Syntax and Semantics

An introduction to any language must include the canonical “Hello, world” example program. Indie’s “Hello, world” program is in Figure 1.

There are a few interesting Indie concepts in this small program. First, you may have noticed the unusual keyword `singleton` in the first line. Indie does not contain the `static` keyword, so there is no such thing as a `static` method in Indie. Instead, classes can either be “normal” or “singleton.” Normal classes must be created with the new keyword, but singleton classes have a single, global instance instantiated before the program begins. In this case, there is exactly one instance of our `Application` object, `HelloApplication`. Every Indie application has exactly one singleton `Application` object whose `start` method is called to start the program.

Second, the syntax of the `args` parameter on the third line is quite

```

class Adder
{
    field n : int;

    new(x : int) {
        n = x;
    }

    method add(x : int) : int {
        return n + x;
    }

    method increment() {
        n++;
    }

    property operand : int {
        get {
            return n;
        }
    }
}

```

Figure 2: A simple Indie class.

different than C++ and Java. Unlike C++ and Java, the type of variables comes after the variable name rather than before it, and the type is separated from the variable or method definition by a colon. If you have ever used ML, this syntax may look familiar.

2.1 Classes, Fields, Methods, and Properties

Indie classes are similar in structure to Java classes, with a few exceptional keywords. The `Adder` class in Figure 2 illustrates the basic structure of an Indie class.

The third line contains a field declaration. Class fields are declared with the following syntax:

```
field name : type ;
```

Fields are either private or protected, and are private by default. They must be instantiated in the constructor; it is a compile-time error if they are not instantiated. There are no public fields in Indie.

The fifth line contains a constructor for the `Adder` class. Constructors are declared with the following syntax:

```
new ( parameters ) { body }
```

The `Adder` constructor takes a single argument, which is used to instantiate the `n` field. Unlike Java and C++, Indie constructors have a different syntax than normal methods, using the `new` keyword rather than the `method` keyword.

The ninth line contains a method declaration. Methods are declared with the following syntax:

```
method name ( parameters ) : type { body }
```

Methods that do not return a value (“`void`” in Java and C++) simply do not have a type specifier in their declaration. Notice that the `increment` method does not have a type specifier and therefore does not return a value.

Finally, the seventeenth line of the class definition contains a property. Properties are declared with the following syntax:

```
property name : type {  
  get { body }  
  set ( parameter ) { body }  
}
```

A property is accessed like a public field, but explicitly defines getters and setters. The `operand` property is “read-only” because it only defines a getter. Properties can be public, private, or protected. They are public by default.

3 Types and Type Conversion

3.1 No More null

`null` is not a keyword in Indie. Every variable of type `string` must contain a pointer to a valid instance of a `string`. For example, the code snippet in Figure 3 would produce an error by the compiler.

```

class InvalidClass
{
    method foo() : string
    {
        // Invalid; null is not a string
        return null;
    }
}

```

Figure 3: This code would produce a compiler error.

```

class ValidClass
{
    method foo() : null string
    {
        return null;
    }
}

```

Figure 4: This code is valid because of the modified method signature.

To make the code in Figure 3 valid, the type of `foo()` must be “either a `null` or a `string`”. What does it mean to return “a `null`”? Well, in Indie, `null` is simply a singleton class defined by the system. The modified method signature in Figure 4 means that `foo()` will return an instance of a `string` or the single, global instance of the `null` class.

The return type of `foo()` in Figure 4 is actually a list of types, in this case `null` and `string`. Indie generalizes this concept to any number of classes in the form of *union types*. Union types are described in detail in Section 3.3.

3.2 No More Casts

Callers to the method `foo()` defined in Figure 4 need to check the value returned by `foo()` before they can use the value as a `string`. In many languages, this would be accomplished by a *type cast*, in which the programmer forces the compiler to re-interpret the `null string` as a normal `string`. In a language like Java, this conversion throws an exception at runtime if the conversion is not valid at runtime. In a language like C++, this conversion

```

class ValidClass
{
    method bar() : string
    {
        var s : null string;
        s = foo();
        if (s is null) {
            return "null";
        } else {
            return s;
        }
    }

    method foo() : null string
    {
        return null;
    }
}

```

Figure 5: `bar()` must check the return value of `foo()`.

happens at compile time, and, if it is invalid at runtime, its behavior is unpredictable.

Indie does not have type casts. Instead, Indie makes use of a new `is` operator to guarantee that type conversions will be safe at runtime. This concept is demonstrated by the code in Figure 5. In the example, the method `bar()` returns a `string`, so it must check the return value of `foo()` before returning it. In this case, the `is` operator behaves like `instanceof` in Java; it returns `true` if the instance on the left hand side of the operator is a subtype of the class on the right hand side of the operator.

The definition of `bar()` is unusual because it returns `s` on the `else` path even though `s` has a declared type of `null string`. This is allowed because the `is` test in the `if` ensures that `s` is not `null` on the `else` path. This concept of *path-sensitive types* is explored in detail in Section 3.4.

3.3 Union Types

In Indie, every type is a set, or *union*, of atomic types, where an atomic type is a single class like `string` or `object`. Although we use set notation

```

class Foo
{
    method bar(n : int) : string int Foo
    {
        if (n < 0) return "Less than 0";
        else if (n > 0) return 1;
        else return new Foo();
    }
}

```

Figure 6: This method returns an elaborate union type.

throughout this paper, we use the term *union* because an instance has exactly one atomic type at runtime. The syntax of an Indie type is described by the following grammar:

$$Type : Atomic \mid Type \ Atomic$$

Normally, types are a single class, which corresponds to a union of size 1. For a type T and a class A , we say $A \in T$ if A is a member of the union T . For types T_1 and T_2 , $T_1 \leq T_2$ means that T_1 is a subtype of T_2 . For classes A and B , $A \leq B$ means that $A = B$ or A is a subclass of B . Finally, $\{A, B\}$ denotes the union type containing A and B .

Types are restricted in that if $A \in T$ and $B \in T$, then neither $A \leq B$ nor $B \leq A$. Informally, no two classes in a type can be relatives. For example, `string null` is a valid type in Indie, but `string object` is not. This is not restrictive because if $T \leq \text{string}$ implies $T \leq \text{object}$.

We define subtyping on union types as follows:

$$T_1 \leq T_2 \text{ iff } \forall A \in T_1 : \exists B \in T_2 : A \leq B$$

Informally, T_1 is a subtype of T_2 if every class in T_1 is a descendent of some class in T_2 . It follows from this definition that for every type T , we have $T \leq \text{object}$.

Figure 6 illustrates the use of a more elaborate union type.

3.4 Path-Sensitive Types

The format of a path-sensitive “cast” in Indie is quite restrictive. The basic form is

if (*var* is *A*) { *true-path* } else { *false-path* }

var must be either a local variable or a method parameter, and *A* must be an atomic type. These restrictions greatly simplify the compiler, although they require a few extra lines of code in some cases.

Let *var* have type *T*. Then the type of *var* on *true-path* becomes T_{true} , and the type of *var* on *false-path* becomes T_{false} , where

$$\begin{aligned} T_{true} &= A \\ T_{false} &= \{B \in T \mid B \not\leq A\} \end{aligned}$$

As one can see from our definition of subtyping above, $T_{false} = \emptyset$ corresponds to the case when $T \leq \{A\}$. To prevent this exceptional case, the compiler reports an error if $T \leq \{A\}$ because the `is` test is unnecessary. Likewise, if $\{A\} \not\leq T$, the compiler reports an error because the test can never possibly be true.

Figure 7 illustrates some of these type rules. Most of the example is pretty straight forward, although it takes a little while to get used to inferring types mentally. Line 16 of the example is probably the most interesting part. In this line, we test whether `i is Bar`. The test is valid because `Bar` is a subtype of `Foo`, and thus `Bar` is also a subtype of `int string Foo`. If the test evaluates to false, then the only thing we know about `i` is that it is not a `Bar`, but it still may be a `Foo`. Therefore, the type of `i` on the false path is still `int string Foo`; $T_{false} = T$ in this case, even though the test in the `if` was perfectly valid.

4 Inheritance and Method Dispatch

4.1 Method Subtyping

Indie defines subtyping between method signatures similarly to many C++ compilers.

Let *m* and *n* be methods such that the the return types of *m* and *n* are R^m and R^n , respectively, and the parameter types of *m* and *n* are P_1^m, P_2^m, \dots and P_1^n, P_2^n, \dots , respectively. Then *m* is a subtype of a method *n* iff

1. *m* has the same name as *n*.
2. *m* has the same number of parameters as *n*.
3. $R^m \leq R^n$.

```

class Foo
{
    method bar(i : int string Foo)
    {
        if (i is int) {
            i = i + 1;
        } else {
            // i is either a string or a Foo
            if (i is string) {
                System.out.println(i);
            } else {
                // i is definitely a Foo
                i.bar(1);
            }
        }

        if (i is Bar) {
            i.bar(1);
        } else {
            // The type of i is still int string Foo
            // because i still may be a Foo
            if (i is Foo) {
                i.bar(1);
            }
        }
    }
}

class Bar extends Foo
{
}

```

Figure 7: This method sample illustrates path-sensitive typing.

$$4. \forall i : P_i^m = P_i^n.$$

For the sake of simplicity, if m does not return a value, we say $R^m = \top$ where $A \leq \top$ for every type A .

This definition is used in checking consistency in class signatures. For example, if an **abstract** class B defines an **abstract** method n , then a non-**abstract** subclass A must define a non-**abstract** method m such that m is a subtype of n . Likewise, no class a can define two methods m and n such that m is a subtype of n .

Note that this is a stricter definition of method subtyping than is necessary. Specifically, the relationship $P_i^m = P_i^n$ could be changed to $P_i^n \leq P_i^m$, and we could still retain a valid subtype relationship between subclasses. However, this looser definition creates a variety of unnecessary ambiguities, and, as discussed below, it is confusing and inconsistent relative to Indie’s method dispatch mechanism.

In languages like Java, if A is a subclass of B , and A contains a method m that is a subtype of some method n in B , then we say m *overrides* n . We say this because if an instance i has a runtime type of A , we can never directly call n on i .

Indie also exhibits this behavior, but it does so through a more general mechanism. *Overriding* is a side-effect of the general method dispatch definitions in subsequent sections.

4.2 Method Dispatch

Dynamic dispatch is a characteristic of all object-oriented languages. With dynamic dispatch, a method call depends not only on the static types of the method parameters, but on the runtime type of one or more of the parameters.

In many languages, like Java and C++, methods can be dispatched on a single parameter: the instance on which the method is called (or the “**this**” parameter). For example, the Java code in Figure 8 produces the output

```
Method 1
Method 3
Method 1
Method 3
```

Indie dispatches on the the runtime type every method parameter. The Indie code in Figure 9 is analogous to the Java code in Figure 8, but the Indie code produces the following output:

```

class A
{
    public void foo(A a) {
        System.out.println("Method 1");
    }
    public void foo(B b) {
        System.out.println("Method 2");
    }
}

class B extends A
{
    public void foo(A a) {
        System.out.println("Method 3");
    }
    public void foo(B b) {
        System.out.println("Method 4");
    }
}

public class JavaDispatch
{
    public static void main(String[] args) {
        A i1 = new A();
        A i2 = new B();
        i1.foo(i1);
        i2.foo(i1);
        i1.foo(i2);
        i2.foo(i2);
    }
}

```

Figure 8: Java method dispatch.

Method 1
 Method 3
 Method 2
 Method 4

Dispatching on all method parameters has some useful properties, but it also has some unusual side-effects. We will define Indie’s dispatch mechanism in detail before we discuss these properties.

4.3 Multiple Parameter Dispatch

For subsequent definitions, let φ be an Indie method call on an instance i with compile-time type I . The arguments to this call have compile-time types A_1, A_2, \dots, A_k .

First, we define which methods can possibly be called from φ at runtime. Since we do not restrict ourselves to compile-time types, this set of *compatible* methods is loosely defined, and it can be quite large depending on the call.

A method m defined in a class B^m is *compatible* with C iff

1. m has the same name the method called in C .
2. m has k arguments.
3. $B^m \leq I$ or $I \leq B^m$.
4. $\forall i : P_i^m \leq A_i$ or $A_i \leq P_i^m$.

Let C_φ be the set of all compatible methods for φ in the program P . We define whether a call is *valid* or not using this set.

A method call φ is *valid* iff there is some $m \in C_\varphi$ such that $I \leq B^m$ and $A_i \leq P_i^m$ for every argument i .

The above definition ensures that there is at least one method m in the set of compatible methods such that m is guaranteed to be compatible with the runtime signature of φ . Equivalently, one can think of this definition as saying “a call φ is valid in Indie iff it would be valid in a language without multiple parameter dispatch.” If a call is not valid, it is rejected by the Indie compiler.

Let the runtime signature of φ be I^r and A_1^r, \dots, A_k^r . We choose which method to call for φ from the set C_φ as follows:

A method $m \in C_\varphi$ is called for φ iff

```

class A
{
    method foo(a : A) {
        Console.out.println("Method 1");
    }
    method foo(b : B) {
        Console.out.println("Method 2");
    }
}

class B extends A
{
    method foo(a : A) {
        Console.out.println("Method 3");
    }
    method foo(b : B) {
        Console.out.println("Method 4");
    }
}

singleton class IndieDispatch extends Application
{
    method start(args : string[]) {
        var i1 : A;
        var i2 : A;
        i1 = new A();
        i2 = new B();
        i1.foo(i1);
        i2.foo(i1);
        i1.foo(i2);
        i2.foo(i2);
    }
}

```

Figure 9: Indie method dispatch.

1. $A_i^r \leq P_i^m$ for every argument i .
2. $I^r \leq B^m$.
3. For every method $n \in C_\varphi$, if $n \neq m$ and n satisfies (1) and (2), then either
 - (i) There is some $j > 0$ such that $P_j^n \not\leq P_j^m$, and for every $1 \leq i \leq j$ we have $P_i^m \leq P_i^n$.
 - or
 - (ii) $P_i^m = P_i^n$ for every parameter i and $B^n \not\leq B^m$.

The definition above simply establishes the order which methods are dispatched; Indie calls are dispatched on the types of the parameters in order, and then are dispatched on the type of the callee instance. It is easy to see that the definition above therefore specifies a single method $m \in C_\varphi$ because, if no method is “singled out” by condition (i), then one will surely be singled out by condition (ii). This follows from the fact φ is valid, and no two methods with the same signature can exist in the same class definition.

Informally, the definition above finds the compatible method with the “tightest” bound on the runtime types of the method parameters.

This strict parameter ordering is necessary to disambiguate the condition illustrated by the code in Figure 10. In this example, Method 1 and Method 2 are both compatible with the runtime signature of the call, and are equally “tight” depending on which parameter you dispatch first. In this case, the rule above guarantees that this program will print

Method 2

4.4 Method Call Examples

Dynamic dispatch on multiple method parameters motivates an interesting object-oriented programming style. We explore some of the interesting cases we have come across in this section.

4.4.1 The equals Method

In Java, `Object` has an `equals` method, and most classes override the `equals` method to accurately distinguish between members of each particular subclass. An example Java `equals` implementation is in Figure 11.

Most of the code in Figure 11 involves testing and converting between types. In Indie, this can all happen automatically with dynamic dispatch,

```

class A
{
    method foo(a : A, b : B) {
        Console.out.println("Method 1");
    }
    method foo(b : B, a : A) {
        Console.out.println("Method 2");
    }
}

class B extends A
{
}

singleton class IndieDispatch extends Application
{
    method start(args : string[]) {
        var a : A;
        a = new A();
        a.foo(new B(), new B());
    }
}

```

Figure 10: Method dispatch takes into account parameter order.

```

class Foo
{
    public boolean equals(Object other) {
        if (object instanceof Foo) {
            Foo otherFoo = (Foo) other;
            return (this.n == otherFoo.n);
        }
        return false;
    }

    public int n;
}

```

Figure 11: A Java equals implementation.

```

class Foo
{
    method equals(other : Foo) : boolean {
        return (this.n == other.n);
    }

    field n : int;
}

```

Figure 12: An Indie `equals` implementation.

so the Indie definition would look something like Figure 12. It is interesting because the definition of `equals` in `Foo` does not override the definition of `equals` in `Object` as it does in the Java version. Rather, it adds a new definition of `equals` that is compatible with a certain class of calls.

If a caller calls `equals` on an instance of type `Foo`, Indie’s method dispatch will execute `Foo`’s definition of `equals` iff the runtime type of `other` is a subtype of `Foo`. Otherwise, it will call the default definition in `object`. This greatly reduces the “housekeeping” code needed for such a simple function, and lets the programmer only define the cases in which he or she is interested.

4.4.2 The Visitor Design Pattern

One of the most annoying side-effects of the VTable-based dispatch in Java and C++ is the necessity of the Visitor design pattern. See Gamma et al. for more information on this design pattern. The Visitor design pattern entails adding a `accept` method to every class in a hierarchy to ensure that dispatching on method parameters is handled correctly. Clearly this is not necessary in Indie.

As a canonical example, we explore the code necessary to traverse a compiler’s abstract syntax tree (AST) data structure. In many compilers, including the reference Indie compiler, the AST is implemented as a hierarchy of classes. For example, in the Indie compiler, `AstBinaryExpression` is a subclass of `AstExpression`, which is in turn a subclass of a general `Ast` class.

An `AstBinaryExpression` is an expression of the form

left-operand operator right-operand

To traverse such an expression, we must traverse both the left and right operands, which are of type `AstExpression`. Since these can be arbitrarily complex expressions, we must call the appropriate method based on the runtime type of each operand, or we will not traverse the expressions completely.

The Java code to accomplish this is in Figure 13. In this case, the `accept` method is defined in each of the non-abstract AST classes. This method calls the appropriate `visit` method in the `AstVisitor` class to achieve the proper runtime dispatch.

In Indie, there is no need to define extra methods in each of the AST classes because the `visit` methods dispatch on the runtime types of the parameters. This is illustrated in Figure 14.

4.4.3 Ambiguous Return Values

Indie's dynamic dispatch can lead to some confusing situations if developers add methods in a haphazard manner. For example, consider the method call in Figure 15. In this example, Method 1 and Method 2 are both compatible with the method call, but their return types are unrelated. What is the return type of this call?

Indie works around this problem using its flexible union type definition. Specifically, the return type of a call φ is the union of all of the return types in C_φ . In this case, the return type of the call is `string int`. If one of the methods in C_φ does not return a value, then the call is not an R-Value.

We do not recommend this as a programming style. In general, if two methods in a class hierarchy have the same name and same number of parameters, they should return related types.

5 Generic Types

5.1 Basic Syntax

Indie supports *generic* classes, or classes that take other types as parameters. The syntax of Indie's generic classes is similar to C++ with a few added elements to ensure type safety.

Consider the example `List` class in Figure 16. This class takes a single parameter `Type` which must be a subtype of `object`. This restriction is indicated by the first line of the class declaration: `<Type : object>`. Since every type in Indie is a subtype of `object`, we can use this class to create a list of any type.

```

class AstVisitor {
    ...
    public void visitIntegerConstant(AstIntegerConstant e) {
        // Process integer constant
    }

    public void visitBinaryExpression(AstBinaryExpression e) {
        // Process binary expression
        e.getLeftOperand().accept(this);
        e.getRightOperand().accept(this);
    }
    ...
}

class AstBinaryExpression extends AstExpression
{
    ...
    public void accept(AstVisitor visitor) {
        visitor.visitBinaryExpression(this);
    }
    ...
}

class AstIntegerConstant extends AstExpression
{
    ...
    public void accept(AstVisitor visitor) {
        visitor.visitIntegerConstant(this);
    }
    ...
}

```

Figure 13: The Visitor design pattern in Java.

```

class AstVisitor
{
    ...
    method visit(e : AstIntegerConstant) {
        // Process integer constant
    }

    method visit(e : AstBinaryExpression) {
        // Process binary expression
        visit(e.leftOperand);
        visit(e.rightOperand);
    }
    ...
}

```

Figure 14: No design pattern is necessary in Indie.

```

class A
{
    method foo(a : A) : int {
        return 1;
    }
    method foo(o : object) : string {
        return "1";
    }

    method caller(o : object) {
        // What is the return type?
        foo(o);
    }
}

```

Figure 15: Indie's dynamic dispatch can ambiguate return types.

```

class List<Type : object> {
    method new(item : Type, next : null List<Type>) {
        field.item = item;
        field.next = next;
    }

    property field item : Type {
        get {
            return field.item;
        }
    }
    property field next : null List<Type> {
        get {
            return field.next;
        }
    }
}

```

Figure 16: A generic linked list class.

We say a type is *concrete* if it is not generic. You cannot use a generic type directly – you must make it concrete by *instantiating* it with concrete type arguments. The program in Figure 17 uses the concrete type `List<int>`, which is based on the generic type `List`.

5.2 Basic Semantics

Unlike C++, Indie’s generic classes are type checked independently of the concrete instantiations in the program. For example, the program in Figure 18 is illegal in Indie even though the equivalent would be legal in C++. Even though the only instantiation of the `Incrementer` class is `Incrementer<int>`, the addition operator is incompatible with `object`, which is the base type of the generic parameter `T`.

Also, a generic class cannot have a generic type parameter as its superclass. This restriction is necessary because clients can instantiate generic types with union types of arbitrary size, and classes can only be a subclass of a single class.

```

singleton class ListTester extends Application {
  method start(args : string[]) : int
  {
    // Construct the list
    var list : List<int>;
    list = new List<int>(3, new List<int>(4, null));

    // Print the list to the console
    while (true) {
      Console.println(list.item.toString());

      // Get the next element with path-sensitive types
      var next : null List<int>;
      next = list.next;
      if (next is null) {
        break;
      } else {
        list = next;
      }
    }

    return 0;
  }
}

```

Figure 17: A program that uses our generic linked list class.

```

// This class is invalid!
class Incrementer<T : object> {
    T increment(i : T) {
        return (i + 1);
    }
}

singleton class IncrementTester extends Application {
    method start(args : string[]) : int
    {
        var inc : Incrementer<int>;
        inc = new Incrementer<int>;
        inc.increment(1);
        return 0;
    }
}

```

Figure 18: The generic type `Incrementer` will not compile.

5.3 Type Checking and Inheritance

Indie’s type system interacts with generic types in unusual ways. For example, the set of compatible methods for a particular call can change depending based on the concrete instantiation of a generic type. Consider the program in Figure 19.

In this example, the first `print` method is only compatible with the call in the `go` method if $T \leq \text{Bar}$, which is only true for some instantiated versions of `Foo`. Likewise, the concrete type `Foo<Bar>` is invalid because the two `print` methods would have the same signature.

Indie ensures that the proper method is called for every instantiated version of `Foo` and disallows instantiations that violate Indie’s basic type rules. This ensures that Indie’s type rules are applied in the same way to every type in the system, but it represents a conceptual leap from non-generic types.

```

class Foo<T : object> {
    method go(o : Bar) {
        print(o);
    }

    method print(o : T) {
        Console.println("T");
    }

    method print(o : Bar) {
        Console.println("Bar");
    }
}

```

Figure 19: The runtime behavior of the method call in `bar` depends on the type of `T`.

6 Other Features

6.1 Singleton Classes

As discussed in the “Hello, world” example, the `singleton` keyword plays an important role in Indie. Singleton classes have a single, global instance instantiated before the main method is called. This design pattern is useful for a variety of classes besides `Application`, including logging classes, caches, and other centralized classes that are typically defined as `static` in other languages. Singleton classes are accessed by their name, as if a global variable of the same name were declared in the namespace of the class.

`static` is not a keyword in Indie.

6.2 Native Code

It is easy to integrate existing libraries into Indie programs using the `native` keyword. With the `native` keyword, you can embed C code directly into an Indie method. While we will not document the details in this paper, the excerpt from the `Console` class in Figure 20 illustrates the concept.

```

singleton class Console {
    method println(message : string) {
        native {
            fprintf(stdout, L"%s\n", GetStringValue(p1));
        }
    }
}

```

Figure 20: An example of the `native` keyword.

6.3 Boxing and Unboxing of Basic Types

Every class is a subtype of `object` in Indie, including the basic types `int`, `byte`, `char`, and `boolean`. To achieve efficiency, basic types are stored just as they are in C, and the arithmetic operations on them are as efficient as they are in C. However, when basic types are passed to methods that require an `object` or other normal type, they are *boxed* in a real class instance so that the class can perform normal operations on them.

There are a few general cases where boxing or unboxing must occur:

- Passing basic types to a method that expects a normal (class instance) type as a parameter. In this case, the basic type will be boxed “on the way in.”
- Assigning a basic type to a variable that is a normal type. The basic type will be automatically boxed in the assignment.
- Using basic types on type-conditional paths. There are some cases where a normal type will need to be unboxed based on the format of the type test. Consider the example in Figure 21. In this example, the parameter is a normal object instance, but the `if` path of the type test requires an `int`. If `o` is an `int` at runtime, the integer value of `o` is unboxed on the `if` path and “re-boxed” when the program leaves the `if` path.

7 Implementation of the Indie Type System

One of our primary goals in our design of the Indie language is to make it suitable for low-level software systems. Therefore, it is critical that we

```

class Foo
{
    method m(o : object) : object {
        if (o is int) {
            // o is unboxed here
            o++;
        }
        // o is "re-boxed" here
        return o;
    }
}

```

Figure 21: Unboxing basic types on type-conditional paths.

support Indie’s high-level features at a minimal runtime cost. Having the complete set of classes available at compile time affords us the ability to optimize away much of the overhead that could potentially arise from Indie’s type system.

7.1 Dynamic Dispatch

Allowing dynamic dispatch based on the types of multiple parameters prevents us from using a straightforward vtable-based approach to polymorphic functions like C++ and Java. Instead, the compiler uses its information about the compile-time types of each method parameter at the call site and the knowledge of the entire class hierarchy to figure out which methods could be named by that call site. This closely resembles the use of Class Hierarchy Analysis to optimize away virtual function calls in more traditional languages, but its importance is even greater in Indie due to the additional flexibility afforded by the language.

Once the compiler finds the set of methods that a given call site could invoke, it generates code to choose one of those methods based on the runtime type of each parameter. This is currently implemented as a tree of switch statements that narrows the set of possible methods iteratively for each parameter, where the leaf nodes are the most tightly bound methods that are compatible for a given set of runtime parameter types. One significant optimization would be to use multilevel vttables to do direct method lookups rather than using switch statements. However, this would have to come at the cost of a vtable memory footprint.

7.2 Generic Types

There have traditionally been two ways of implementing generic types: by using a uniform data representation, as in ML and proposals for Java and C#; and by compiling separate code for each concrete instance of a generic type, as in C++ templates. Our original goal was to use our preexisting uniform data representation as a mechanism for handling generic types, but we quickly realized that this was not possible.

To perform the aforementioned optimizations, the compiler depends on compile-time type information to emit method calls, but this information is not available for method calls inside generic types. For example, the call to `print` in Figure 19 cannot be emitted using our traditional optimizations.

One possible approach for resolving this problem is the use of “trampoline” functions. Instead of emitting the dynamic dispatch code at the call site, the call dispatch code can be put in an auxiliary member function that is specialized per generic-class instantiation and looked up via that class’s vtable. This approach appears to be a promising area for future development.

For the current implementation of the Indie compiler, we decided to take the second approach, that of C++-style templates. This approach optimizes for execution speed at the expense of code size, a tradeoff that is not always reasonable; many people eschew C++ template libraries such as STL because of code-size blowup. However, there are many cases where the template-style approach is superior as well. In the end, ease (or tractability) of implementation became the dominant factor influencing our decision to go with the code specialization approach.

8 Availability

The Indie language, the Indie compiler, and this documentation is available under the terms of the MIT open-source license. The reference Indie compiler is under constant development, and is available for download at <http://indie.sourceforge.net/>.

9 Acknowledgements

We would like to thank Professor John Mitchell of Stanford University for giving us the opportunity to develop Indie, and for his many wonderful courses on programming languages in the Stanford Computer Science de-

partment. We would also like to thank SourceForge.net for providing free web hosting, CVS, and bug tracking software to our (open source) project.